Monitoring Distributed Consensus In The Cloud

Lochlann O'Regan[1], Zhidong Fang[2], Shaocheng Li[3] Srushti Jain [4], Guijia Zhu [5] and Dexiang Chen [6]

***Abstract*—Distributed systems offer greater fault tolerance, scalability and performance over centralized systems. Consensus algorithms [13], [16] manage the interaction between nodes in a distributed system to ensure the operations that take place are coordinated. The cost of these benefits is dramatically increased complexity when monitoring a distributed system. We ask the following question, how can one monitor a consensus algorithm in a deployed environment? Furthermore, we question how can this data be presented to develop intuition about the behaviour of the deployed system?**

This project conducts a review of the industry standard consensus algorithms, the Raft [16] consensus algorithm is selected as its emphasis on being understandable is well suited to the monitoring needs of this project. The application of this consensus algorithm is for the very demanding task of a distributed data store. We have successfully implemented the Raft consensus algorithm, a scalable distributed key-value data store and a monitoring suite that answers the aforementioned questions.

We have conducted a rigorous evaluation to ensure the implemented distributed data store is reliable reported on the results. We have also found that through monitoring the overall consensus statistics on each node and the in-depth Remote Procedure Call (RPC) messages of the consensus algorithm we are able to provide clear, concrete and consistent feedback about the operation of the distributed system.

***Index Terms*—distributed systems, distributed consensus, Raft, data storage, cloud monitoring**

I. Introduction

An essential part of a distributed data store system is the mechanism with which data operations are coordinated across the nodes in the cluster. Consensus algorithms define how this communication takes place and as a result are responsible for the core functionality of the distributed system. As we have discovered in this project, the implementation of a consensus algorithm is nuanced, every line of code has an impact on the numerous states that the cluster can enter. This situation becomes even more complex when an application is developed on top of the consensus algorithm. Further states, communication messages and logs are introduced.

Therefore, in order to have confidence that the system in functioning correctly, is is crucial to monitor how the system is behaving. During the early development phase, unit tests are important to verify the correct functioning of individual parts of the system. As the systems matures, detailed logging from each node with different error levels allows developers to reproduce issues and work to identify the root cause using debugging techniques. The approach of adding print statements and rebuilding the system to reproduce issues is suitable for developers as they are very knowledgeable about the system and have the freedom of a non-production environment.

However, in a production environment, it is often the task of a site reliability engineer to monitor the behaviour of the system. The massive quantity of logging outputs can be overwhelming if one is not intimately familiar with the latest version of the system. Another concern is that logging outputs can become obsolete and misleading, their output too abstracted from the true functioning of the system. In this paper, we believe log files are a vital part of a distributed system, however, we suggest that monitoring the underlying consensus algorithm is an additional approach which can give greater context to the log files on each node. Monitoring the consensus algorithm becomes a useful sanity check as to how the system is performing as a whole.

This all leads to defining the the central challenge of this paper, how does one monitor the functioning of a consensus algorithm such that a user can develop intuition about its behaviour? The first step is to have a system to monitor. The consensus algorithm is at the heart of a distributed system but it is the application that runs on top which provides the utility. A distributed data store demands reliability and consistency in order to function properly and is a real-world application where monitoring the consensus algorithm can greatly increase confidence about the functioning of the system.

Choosing an existing implementation of a distributed data store was ruled out for two reasons. The first reason is the consensus algorithm, the main focus of the project, would be deeply abstracted in the project and having direct access to its internals is crucial to the success of this project. The second reason is that an existing distributed data store would have an array of complex features and restrictions that would unnecessarily complicate the development of this project for no tangible gain. It was for this reason that we accepted the challenge of not only developing a monitoring suite but also implementing a consensus algorithm and a data store.

Having discussed the central challenge and the core technical challenges to be overcome it is necessary to add greater specificity by identifying what is to be gained by this project and by whom which is discussed in the next section.

[1]L. O'Regan is with the School of Computer Science, University College Dublin lochlann.oregan@ucdconnect.ie

[2]Z. Fang is with the School of Computer Science, University College Dublin zhidong.fang@ucdconnect.ie

[3]S. Li is with the School of Computer Science, University College Dublin shaocheng.li@ucdconnect.ie

[4]S. Jain is with the School of Computer Science, University College Dublin srushti@ucdconnect.ie

[5]G. Zhu is with the School of Computer Science, University College Dublin guijia.zhu@ucdconnect.ie

[6]D. Chen is with the School of Computer Science, University College Dublin dexiang.chen@ucdconnect.ie

II. User Scenarios

In the realm of monitoring and understanding, the most important question is who are the users? In this paper there are two user scenarios that guided the development of this project, user scenario A and user scenario B.

A. Distributed Data Store

Scenario A is to design and implement a distributed data store including the underlying consensus algorithm. The goal is to provide the same core features of a distributed data store such as Redis [9] to meet the needs of a developer while having the opportunity to open the consensus algorithm to monitoring. As developers and having discussed with both University College Dublin (UCD) module team and Microsoft, we identified the following three areas that are most important to developers.

1) Functionality
2) Reliability
3) Scalability

Functionality defines what operations the distributed data store is to support. It was identified that basic Create Read Update Delete (CRUD) operations on the data store were desired. In terms of the data type, key-value pairs were the format that was desired from our discussions with Microsoft. Supporting these actions through Application Programming Interfaces (APIs) completed the functionality for users of the data store.

Reliability is important as Microsoft stated that it was vital that the data added via the APIs is consistent across the nodes of the data store. It is also desired that the system can continue functioning if there are up to n failures where the number of servers is 2n + 1. Therefore, as will be discussed later in the implementation Section IV-A and the evaluation section V it was identified early in the project that reliability was crucial to meeting the needs of this user scenario.

Scalability was also important as a system with a fixed node configuration it not a modern distributed system. It was highlighted by the UCD team that in-order for the benefits of a distributed system to be realised, at its core is the ability to configure the size of the cluster and have the system continue functioning.

B. Monitoring Suite

Scenario B is to monitor the deployed distributed system by retrieving, organising and visualising the operations of the consensus algorithm to meet the needs of users monitoring the system in deployment. This functionality is useful for developers but especially so for Site Reliability Engineers (SRE) who are responsible for the deployed system and may not be as intimately familiar with the behaviour of the system as a developer. The understanding within the team of the needs of this user was developed through discussions with our Microsoft and our assumptions were tested through user evaluations as discussed in Section V-C. We identified two key needs that we aim to meet in our implementation.

1) High-level frequently updated monitoring data
2) Real-time snapshots of the behaviour of the consensus algorithm

The first need of high-level monitoring data was important as it is necessary to orient an SRE using concepts that are familiar to them. In this case, the overview provides a view of all the nodes in the system, their current hostname and the most recent operations they have performed. This birds-eye view is necessary to build intuition about how the structure of the system appears when it is behaving normally. The complexity and scale of modern distributed systems means without this it would be extremely difficult to narrow focus to problem areas of the system.

The second need explores the following question; what does real-time mean in a system that operates in milliseconds and nanoseconds? A consensus algorithm is always communicating with others nodes in the cluster to ensure it can rapidly adapt to changes in the cluster configuration and to ensure the appropriate operations take place on each node. This question is explored by considering the previous work of monitoring in section III-B and the question is answered in the implementation section IV-B.

III. Related Work

There were two key areas to explore to inform the direction of this paper. The first was the choice of consensus algorithm. The second was to identify existing solutions proposed for monitoring distributed systems and consider their approaches while developing the idea in this paper further.

A. Consensus Algorithms

Consensus is often a topic in the research area of distributed systems. During the ACM Symposium on Principles of Distributed Computing in July 2022 [1], 5 of the 40 papers alone mentioned consensus in the title of the paper. Distributed consensus algorithms are a crucial component to any distributed system. Both algorithms enable the state machine replication approach to fault-tolerant services [1], this is where the same commands are executed on each node in the same order on the cluster. This is what allows a series of disparate nodes to come together as a cohesive whole.

There are two distributed consensus algorithms which are most commonly used in production systems. The first is Paxos [2], [7], [13] which is the standard choice across the industry for a consensus algorithm since its publication in 1998 [7], [13], notable it is used in many of Microsoft's systems. The second is Raft [3], [16] which is a more recent publication from 2013 and is increasingly being adopted, an example is the Kubernetes etcd component which stores the state of a cluster across different nodes. In this section we discuss both algorithms and justify the decision to choose one of them.

The Paxos paper [7], [13] is inspired by a system of governance of the Paxon civilisation whose approach can

[1] https://www.podc.org/podc2022/accepted-papers/

be mapped onto the distributed systems problem of fault-tolerant services. The government of the Paxon people was changeable in that members of parliament would come and go in short succession. Having identified that the main role of the parliament was to pass certain laws, they compensated for this behaviour by defining a set of rules for members of the government. Each member was in possession of a ledger, the ledger contained numerical blank entries that increased monotonically. Entries in this ledger were permanent and only took place in the event of a successful ballot for a law among the members of the government. The overall aim was to use consistency between ledger entries as the source of truth for which laws were to be applied. This allowed continuation of governance as the members of the government changed. Similarly, this can be used for a fault-tolerant service where entries in the ledger are client requests to be executed and each member of the government is a server with their own digital ledger.

Paxos and Raft have been compared in publications [3], [4], [11] which highlights the similarities and differences between the two algorithms. A weakness of the Paxos paper is its description of the rules for the governance process is nuanced [2] and an algorithm is not defined which makes the implementation of Paxos a challenging task with many different interpretations of the same paper. The result is that Paxos is more of a family of algorithms than a single algorithm.

The main priority of the Raft algorithm [16] was to make its process understandable. How did Raft go about making the process more understandable? Raft uses two approaches, the first is problem decomposition where each sub-problem is considered individually and the second is reducing the state space i.e. the number of scenarios the algorithm can enter. One of those sub-problems is leader election, which is differs between Paxos and Raft. In Raft, leader election involves navigating the three possible states by each server, a leader, a follower or a candidate. Normally there is a sole leader which handles all requests from clients, a follower only responds to requests from leaders and candidates, and candidates who on receiving a majority of votes can become the sole leader. Term numbers are used to represent the passage of time in the distributed system and within each term an election is held to elect the leader. Raft only allows a node whose log entries are fully up-to-date to become the leader. This is different in Paxos where any server can become leader, even if its logs are out of date. In Paxos this situation causes the state space to be greater as the logs must be updated within the concept of the leader election whereas this scenario doesn't exist in Raft. This is an example where the state space has been reduced in Raft.

The outcome of this exploration was Raft being selected as the consensus algorithm to be used for this distributed data store. This is due to its clear algorithm description for implementation, its focus on aiding understanding and its increasing popularity in modern distributed systems. These three characteristics make it extremely desirable for building both a reliable distributed data store and exposing metrics that will be meaningful to users.

B. Monitoring Distributed Systems

The following publications in the areas of monitoring distributed systems are a subset identified to be most appropriate for this project [5], [6], [8], [10], [12], [14], [15], [17], [18].

We begin by exploring an industrial survey of professional engineers about fault analysis and the debugging of microservices [6]. This survey discusses twenty-two common faults in deployed microservices and their solutions. It was found all participants depend on log analysis for resolving the faults. The three levels of log analysis were reading of logs, visual log analysis where they are structured logically and finally visual trace analysis where the traces of system execution are collected using the tool and can be visualised. The great strength of this survey was they built their own large scale microservice test system called TrainTicket and reproduced faults reported from the industry survey. They were then able to get several graduate students using one of the three methods of log analysis to solve certain faults. The sample size of six graduate students was a weakness due to the size but the overall impression of the paper was that monitoring consensus in deployment was a unique and useful additional tool.

Having narrowed focus through the review, the focus is now on a tool classified as a visual trace analysis tool [8] which is the most advanced according to the previously discussed industrial survey [6]. The tool comes in two components, one part is ShiViz [2] which accepts logs and can visualise and interact with the timeline of execution. The second part is ShiVector and it enables the logs to be visualised by ShiViz by adding appropriate timing information. The tool was functional and very understandable at taking a complex log file and visualising the chain of execution. However, the level of granularity was that of a logging file so it does not provide an overview of the entire system.

To conclude, reviewing the related work was an extremely important process to better understand what was valuable to our users and what experimental work had already been completed. The need for more monitoring tools is clear from the surveys [6], [18]. Finally, the mixture of high-level overview of the whole cluster yet also the lowest-level communication messages offered by monitoring the consensus algorithm appeared to be a novel additional approach to the area of distributed debugging.

IV. Implementation

This project demanded a sizable and technically challenging implementation to answer the following question; how does one monitor the consensus algorithm of a distributed data store? The overall implementation can be broken down into three key components; the Raft key-value store, the Raft visualization and the deployment architecture. In each section the choice of technology is justified and the structure of the solution is described with the aid of Figure 1.

[2]https://bestchai.bitbucket.io/shiviz/

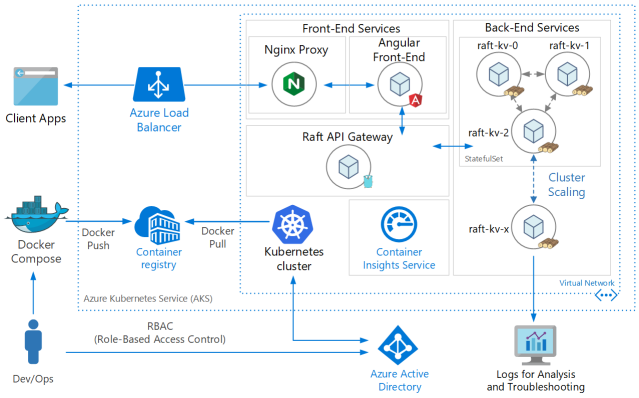

Fig. 1. Distributed Data Store Implementation Diagram

A. Raft Key-Value Store

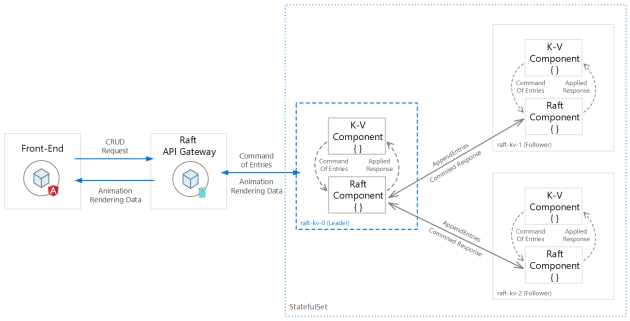

Fig. 2. Raft Key-Value Implementation Diagram

Having selected the Raft consensus algorithm to be the coordinator for distributed consensus, it was necessary to identify the most suitable language for the task. Three key features of the language were highlighted as being crucial to successfully implement this component.

1) Concurrency
2) Memory Management
3) Networking

The first feature was concurrency, the Raft Key-Value store requires concurrent execution as many Raft processes must execute periodically. Thus, syntactically compact code for enabling concurrency on a method would greatly reduce the development time. Java was identified as a counterexample, the syntax was deemed very heavy weight for the demands of this implementation. It was at this point the Go language was identified as being a solution, its currency features include a concept called a go routine. This construct is syntactically lightweight, a method can be made into a go routine i.e. a thread by simple adding ”go” before the method name.

The second feature was memory management. Languages such as C offer great performance but the necessity of low-level memory management would greatly increase the risk of memory leaks in the code for this implementation. Therefore, a garbage collector was desirable and would not inhibit the pace of development. Building on top of the first area, the Go language provides garbage collection which makes it more appropriate for this project.

The final feature to consider was networking support. This was necessary as the communication between Raft K-V instances takes place via Remote Procedure Calls (RPC). Furthermore, the front-end needs to be able to communicate with each Raft K-V instance via REpresentational State Transfer (REST) APIs. It was at this point Go was chosen as the language for the Raft K-V Store, it offered the language library of ”net/rpc” for RPC communication and a widely-used library called ”Gin” for creating REST APIs.

Having discussed why Go was chosen, it is now necessary to discuss how Go was used to implement a Raft K-V data store. This is explained with the aid of Figure 1, specifically the section annotated by ”Back-End Services”.

The Raft algorithm operates according to two RPCs, the first is RequestVote RPC and the second is AppendEntries RPC. These RPC calls work together to enable two key functions, the first is leader election and the second is log replication. It is worth noting the distinction between the Raft consensus algorithm and the application for which it is used. The Raft consensus algorithm is a detailed description of how ”Logs” are to be kept in sync across a number of instances running the Raft algorithm. In this application, a ”Log” contains a command e.g. PUT along with some data to be stored. The idea is that once a ”Log” on an instance is known to be on a majority of other instances it can then be committed. Committed means the command in the ”Log” can be applied to the state machine which in this case is a key-value data store. In summary, the Raft algorithm drives the operation of the distributed data store but it is separate from the data store.

The process followed for implementation can be split into two major milestones. The first goal of development was stable leader elections which could withstand n failures according to the 2n + 1 requirement. This involved extensive testing and a test suite using mocked RPC connections was used to simulate network disconnections to evaluate the behaviour of the implementation. Once leader election was considered stable, the Go ”net/rpc” library was added to enable communication between separate machines running this Raft implementation. This constituted the first milestone for the interim demonstration. The second stage was developing Log replication which allowed the Raft implementation to begin accepting Logs and distributing them across the application. The Key-Value data store could then be developed as there were now Logs for it to accept from the Raft algorithm. The Key-Value data store could then apply the operation e.g. PUT and store the data. A hash map was used as the underlying data structure for data. Once the core functionality was complete it was necessary to add scalability to the data store. This involved communicating with the Kubernetes API via the Go Kubernetes client to enable service discovery and a mechanism to adjust the number of replicas of the Raft Key-Value system. This constituted the second and final milestone of the final demonstration.

The above features were exposed to the front-end via ten

REST Endpoints which not only offer the ability to monitor the Raft consensus algorithm in deployment but also enable a scalable distributed data-store.

The main technical challenge encountered was managing shared memory and concurrency. These situations are encountered often as the Raft implementation executes many operations periodically, an example is every one hundred milliseconds the AppendEntries RPC is send by the leader. The Go language as discussed previous provides a high level of support for writing concurrent programs. The mutex (mutual exclusion) feature was used, condition variables which allowed multiple goroutines to wait for a shared resources and finally go channels while allow communication between go routines to coordinate behaviour. The included race detector in Go also allowed detailed testing to identify situations where race conditions were created. It was crucial to confront the challenge of shared memory and concurrency is it occurred throughout the development of the core Raft functionality and during the development of the service discovery features.

B. Raft Visualization Interface

The goal of this component is central to answering the question of this paper, how can the behaviour of a consensus algorithm be monitored? The Raft Visualization Interface directly meets the second need of users in Section II-B by providing a snapshot of the real-time actions of the consensus algorithm. The real-time snapshot is achieved by the collection and ordering of RPC messages by the Raft Key-Value Store which is then returned to the Front-end. This data is formatted as a JSON response which is then received by the Raft Visualization which is embedded in the Front-end at which point it works to describe the behaviour of the consensus algorithm via an interactive interface.

The implementation of the Raft Visualization interface is based on an educational demonstration project for Raft [3]. The ethos throughout the project has been to prioritise understanding the technologies used over using a pre-built library for ease of development. This drove the decision to implement the Raft Key-Value Store in Section IV-A from scratch using the most appropriate language as the existing Raft implementations were not appropriate for monitoring. However, the research of Raft visualisations found there were several educational versions available and the decision was made to develop the above project into the monitoring Raft Visualization Interface. This required translating existing JavaScript code to TypeScript, adapting the data structures to accept data returned from the Raft K-V Store and a series of user interface changes following our user evaluation in Section V-C.

The Raft Visualization is shown in Figure 3 and is used to track the AppendEntries RPC request and response messages that are sent when performing a Put operation when there are seven nodes in the cluster.

[3]https://github.com/ongardie/raftscope

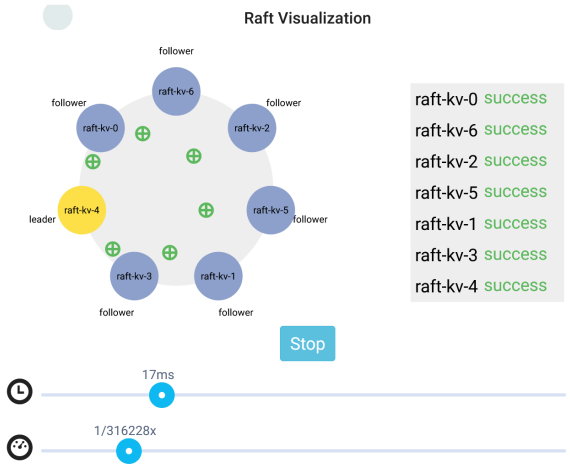

Fig. 3. Raft Visualization Interface

C. Deployment

The first challenge of designing the deployment architecture in Figure 1 was choosing the orchestration platform. Containerisation was the most appropriate solution for a distributed system due to the ease of testing, deployment and portability across different architectures. Docker was selected as the container runtime and Kubernetes as the orchestration platform. We were fortunate to have support from Microsoft who provided access to Azure, more specifically the Azure Kubernetes Service (AKS). The configuration for the Kubernetes needed to deploy the Raft Key-Value component, the front-end component and the raft-proxy component. The standard Kubernetes stateless configuration of a LoadBalancer service and then a Deployment configuration was perfectly adequate for the front-end and the raft-proxy.

However, for the Raft Key-Value store this configuration was not appropriate as the data store is stateful not stateless. As the front-end needed to connect to specific instances of the Raft Key-Value store in order to monitor their behaviour, it was necessary to use a Kubernetes headless service. A headless service acts as a DNS lookup for the instances associated with an application running Kubernetes. A StatefulSet was needed to run the Raft Key-Value code as that allowed the number of replicas to be adjusted in a predictable manner. This means for a single replica the instance would always be accessible at the hostname ”raft-kv-0”.

The greatest technical challenge in the deployment section was the following; how does one allow the system to scale and discover instances on Azure? After much research and testing, the Kubernetes API 2.0 was identified as being the most performant way of retrieving the number of the Raft Key-Value instances that were running. There exists an official Kubernetes Go Client library which allows the client to control the Kubernetes orchestrator via the API. This was integrated into the Raft Key-Value component, authentication was granted by Azure Role-Based Authentication Control (RBAC) and several

REST APIs were written and then exposed to the Front-end. The REST APIs written include /scale-cluster, /delete-pod and /get-raft-instances. The service discovery implementation queries the current number of instances of the Raft Key-Value component via the Kubernetes 2.0 API every four hundred milliseconds. If there is a change, a go routine updates the RPC connections of the Raft Key-Value component.

V. Evaluation

At the beginning of the project, clear evaluation criteria under several headings were established to comprehensively measure the quality of the final system. This is an area that was a priority of the team to ensure the quality of the system was tested and proven through a variety of tools including surveys, statistical plots and simulated scenarios. There are three sections; reliability, performance and user evaluations which provide an overall picture of the system.

A. Reliability

The reliability section was of particular importance for the distributed data store user scenario in Section II-A. In our evaluation plan for reliability we defined our experiment parameters as follows.

1) Number of Raft Key-Value Instances
2) Number of concurrent requests

The number of Raft Key-Value Instances is the number of replicas configured within Kubernetes for the Raft Key-Value component. This is different from the underlying compute servers that Kubernetes is responsible for managing. In this experiment we are using AKS on Azure and the system is deployed on three compute servers, each server has two virtual Central Processing Units (CPUs) and seven GigaBytes (GB) of Random Access Memory (RAM) for a total of 6 vCPUs and 21 GB of RAM. Kubernetes uses a service called kube-scheduler to assign the instances of the Raft Key-Value component to the optimal compute server. The implementation of the Raft Key-Value component in Section IV-A is designed to function correctly no matter how many underlying computer servers Kubernetes is managing.

The number of concurrent requests is in this instance used solely to demonstrate the scaling speed of system. A single Put request is made every second to the distributed data store and the number of failed requests are tracked. In the event of a server failure when scaling the cluster, the Put request will be unable to contact the leader and thus the request will fail. The number of failed Put requests will be incremented and show that the system did not scale effectively.

What is the hypothesis being tested in Figure 4? The hypothesis is that the system is able to scale rapidly between 2 and 7 servers without losing consistency according to the 2n + 1 servers where the system can continue function up to n failures. This involves handling failures when the cluster is decreased in size or in the event of increased instances they must be introduced into the cluster and updated to the current log values. The scaling speed is tested by the user constantly sending Put requests, each second there must be a leader elected who is able to accept the request, reach consensus and return an acceptable response or else the number of failed requests will increase.

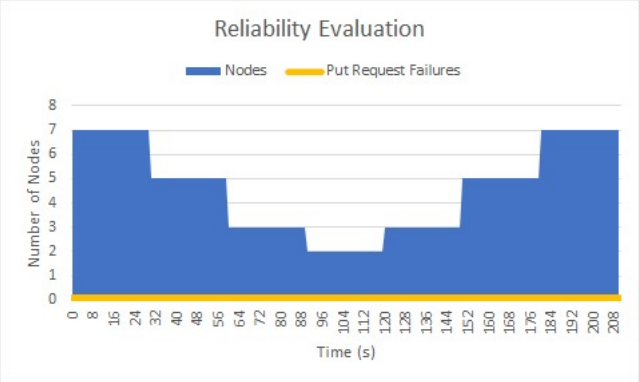

Fig. 4. Evaluating reliability of system

This experiment is tracked in Figure 4 and performs the scaling scenario as described above without any ”Put Request Failures”. The outcome of this experiment is to increases our confidence in our hypothesis that the system is reliable and follows the 2n + 1 requirement criteria from 2 to 7 servers as the data was retained.

B. Performance

The performance of the system is a lesser priority in contrast to the reliability according to the distributed data store user scenario in Section II-A and thus profiling the code to optimise performance has not taken place in this project. However, the performance of the system can impact the reliability and thus the goal is to verify the stability of the system for a concurrent load of 10 requests and a response time of less than 400 ms. This defined response time target is a target calculated using the Raft paper [16] and the configured heartbeat and election timeout values for the Raft Key-Value implementation. The following experiment parameters are used for the tests.

1) Number of Raft Key-Value Instances
2) Number of concurrent requests
3) Duration of evaluation

What is the hypothesis being tested in Figure 5? The hypothesis is that the distributed data store API can handle at least 100 concurrent requests without an increase in mean response time greater than 400 ms. Furthermore, the load is increased to identify at what point the mean response time moves outside that range. The goal is that after the conducting this test, we can have greater confidence in its ability to perform under load. The number of Raft Key-Value Instances is fixed in this experiment at 3 instances. The underlying Kubernetes compute is a single compute server running locally. This decision was made to control the variables, the response time to Azure East US servers is not under test in the experiment, only the response times of the system. It was found that for this experiment configuring the number of instances between 2 and 7 instances had no impact on this performance evaluation and

thus both experiments are fixed at 3 instances. The number of concurrent requests starts at 1 request and steadily increases to 300 concurrent requests. The data used for the Put request is the following data set [4].

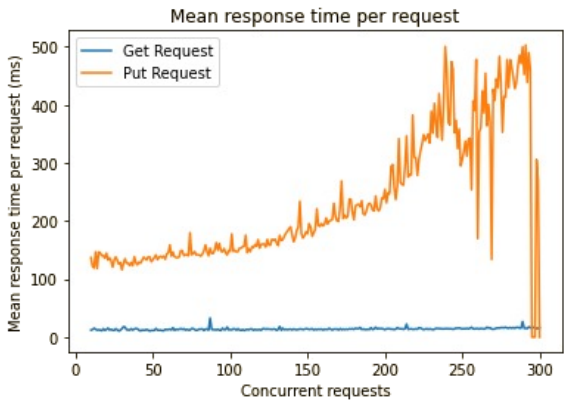

Fig. 5. Current performance limits of the system where nodes = 3

The result of the experiment in Figure 5 is that the mean response time remains within the 400 ms limit at 100 concurrent requests. A limit of the system is identified at 225 concurrent requests where the mean response time begins to move outside the limit. The outcome of this experiment is increased confidence in the stability of the system at a peak load of 100 concurrent requests. The graph also shows a dramatic decrease in performance around 225 concurrent requests. Our metrics running locally identified the Raft-Proxy as a bottleneck in this situation. Thus to further increase our performance it would be necessary to increase the number of instances of the Raft-Proxy.

What is the hypothesis being tested in Figure 6? Having established the mean response time for 100 concurrent requests we want to better understand the distribution of response times for a sustained load using a box plot. The number of Raft Key-Value instances is fixed as in the previous experiment at 3 instances. The underlying Kubernetes computer is a single compute server running locally as before. The number of concurrent requests is fixed at 10 concurrent requests which is 10 percent of the peak load which was tested in the previous experiment and is a realistic consistent load. The 400 ms is now applied for every request, not the mean request response time. The duration of the evaluation is 60 seconds. The goal of this experiment is to further test our hypothesis that the system is reliable by using the statistical tool of a box plot to show the complete distribution of response times.

The result of the experiment in Figure 6 is that the Get request median is below 25 ms and the Put request median is below 300 ms. The upper and lower quartile different for the Put request is acceptable but the upper and lower whisker values is significant. The difference in maximum and minimum values is related to the rate of the AppendEntries

[4]https://datahub.io/core/geoip2-ipv4

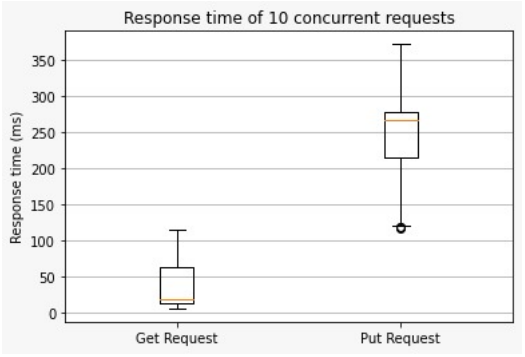

Fig. 6. Performance of sustained load of system where nodes = 3

RPC reaching consensus across the different nodes varying depending on the state of the cluster at the point in time the Put request is received.

C. User Evaluation

The monitoring suite user scenario in Section II-B defines that it is crucial that the user's understanding be enhanced by the visualizations rather than obfuscating the operation of the system further. As a team, we developed a survey that focused on testing the assumptions we had made and collected as much feedback as possible from our fellow students. In this evaluation section the focus is on stating our prior assumptions about the Raft Visualization Interface and how they changed as a result of the user evaluation. Our assumptions about the visualization in Figure 7 at the time of user evaluations were as follows.

Our sole prior assumption was that the animation was useful but lacked greater detail about the Raft log entries as shown on the grey table on the right of the animation. It was expected that the grey table would be criticised for not showing the actual data within the AppendEntries rather than the demo 1 value which represented that data had been replicated on a specific node i.e. S0 in the Figure 7. We therefore had a plan to further enhance the data represented in the grey table.

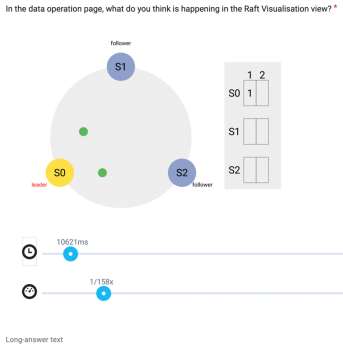

Fig. 7. User evaluation survey question

The feedback from our user evaluation contrasted greatly without our prior assumptions. The sole prior assumption that greater detail was needed on the Raft log entry table

conflicted with feedback from the users. Feedback included not understanding what the "up-right table" represented and questions about what the 1 and 2 integers represented. As a result, in the final version as you can see in Figure 3 the grey table has been removed and a green success text was found to be far more user friendly.

There was also much feedback about areas that we had not anticipated before the user evaluation. This included confusion about how to use the controls to play and stop the animation. As a result a far clearer "Start" and "Stop" button was added to the final version as shown in Figure 3. A final piece of unexpected feedback was that none of the users expected the Raft Visualization to appear after adding a piece of data through the Monitoring Suite. Therefore, a pop-up message was added to query the user as to whether they wanted to see a snapshot of the real-time Put operation on the system. This approach to testing our assumptions greatly improved our final visualization with positive feedback from Microsoft mentors about the intuitive design.

VI. Conclusion

How could the system be improved with more time? There is great potential for improvements that could further enhance the usefulness of the monitoring suite. The Raft Visualization could be extended to represent more complex common failure scenarios such as certain nodes slowing down over time as discussed in the industrial survey [6]. There are also optimisations that can be made to the Raft implementation to improve its performance on a greater scale. To conclude, it is worth considering the opening question of this paper. The central challenge asked at the beginning of the paper was the following; how does one monitor the functioning of a consensus algorithm such that a user can build an intuition about its behaviour? Having successfully developed a distributed data store and a monitoring suite, the objectives established at the start of this project have been achieved.

The success of this project has been achieved by confronting uncertainty at all points of the project. If the decision had not been made to implement our own reference implementation of the Raft algorithm it would have been extremely difficult to explain it in any meaningful way. How can someone explain something that they themselves do not understand? It was through a thorough understanding of the core parts of the Raft algorithm [16] that the monitoring suite actually became a tool we used ourselves to ensure our implementation was behaving correctly. Following our extensive review of other publications in this area in Section III this approach of using the distributed consensus as a logging tool provides a novel way of sanity checking increasingly complex distributed systems.

Acknowledgment

Thank you very much to Simon Caton, Dimitris Chatzopoulos and Pavel Gladyshev for their advice, support and encouragement throughout this project. We would also like to thank our Microsoft mentors; John Twomey, Vahan Hovhannisyan and Matt Wilkinson for their helpful advice and feedback over the past eight weeks.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990, ISSN: 0360-0300.

[2] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[3] H. Howard, "Arc: Analysis of raft consensus," University of Cambridge, Computer Laboratory, Report, 2014.

[4] H. Howard, M. Schwarzkopf, A. Madhavapeddy, and J. Crowcroft, "Raft refloated: Do we have consensus?" *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 12–21, 2015, ISSN: 0163-5980.

[5] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Communications of the ACM*, vol. 59, no. 8, pp. 32–37, 2016, ISSN: 0001-0782.

[6] X. Zhou, X. Peng, T. Xie, *et al.*, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018, ISSN: 0098-5589.

[7] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*. 2019, pp. 277–317.

[8] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 2, pp. 1–38, 2020, ISSN: 1049-331X.

[9] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards scalable and reliable in-memory storage system: A case study with redis," in *2016 IEEE Trustcom/BigDataSE/ISPA*, IEEE, pp. 1660–1667, ISBN: 1509032053.

[10] E. Garduno, S. P. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "Theia: Visual signatures for problem diagnosis in large hadoop clusters," in *26th Large Installation System Administration Conference (LISA 12)*, pp. 33–42.

[11] 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*, pp. 1–9.

[12] J. Klein and I. Gorton, "Runtime performance challenges in big data systems," in *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development*, pp. 17–22.

[13] B. W. Lampson, "How to build a highly available system using consensus," in *International Workshop on Distributed Algorithms*, Springer, pp. 1–17.

[14] X. Liu, Z. Guo, X. Wang, *et al.*, "D3s: Debugging deployed distributed systems," in *NSDI*.

[15] F. Neves and N. Machado, "Falcon: A practical log-based analysis tool for distributed systems," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 534–541, ISBN: 1538655969.

[16] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pp. 305–319, ISBN: 1931971102.

[17] J. R. Wilcox, D. Woos, P. Panchekha, *et al.*, "Verdi: A framework for implementing and formally verifying distributed systems," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 357–368.

[18] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik, "An interview study of how developers use execution logs in embedded software engineering," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, pp. 61–70, ISBN: 166543869X.