**Abstract.** The Raft Consensus Protocol [] is widely used consensus algorithm like Paxos [], ZAB [] that uses a leader-based approach to achieve consistency in a distributed system. However, in certain scenarios where the leader node is centralized, the performance of the cluster is compromised in terms of fault tolerance and throughput. To address these limitations, we implemented a raft cluster with a virtual dissemination of multi-level tree network topology []. We compare the performance two Raft clusters, both following the standards of the raft consensus protocol. One cluster uses the single level tree topology, while the other employs the multi-level tree topology. The evaluation is based on the average response time taken by the clusters to process the multiple client requests with different loads and analyze their behaviors when they encounter bottleneck. The clusters are deployed AWS [] public cloud infrastructure to perform the analysis that closely simulates to real-world production constraints.

**Declaration.** I declare that this dissertation represents my own work except where otherwise explicitly stated.

**1. Introduction**

In the field of distributed system, making a system fault tolerant and maintain data consistency is a potential challenge. To address these issues consensus algorithms have played a vital role. RAFT consensus algorithm is a well know distributed consensus protocol which is used to ensure fault tolerance and consistency in distributed systems. It was developed by Diego Ongaro and John Ousterhout in 2013. It provides a simple straightforward approach to solve the consensus problems as compared to Paxos algorithm. Raft is adopted by many large-scale distributed systems to manage their replicated logs and achieve fault tolerance such as etcd [], cockroachDB [], consul [], RQLite [], etc. The strength of this algorithm lies on creating multiple raft nodes across different servers, forming a replicated cluster thereby enhancing the robustness of the service. To achieve consensus in a raft cluster, if more than half of the nodes in a cluster are operational then the entire cluster can continue to function effectively ensuring high availability and fault tolerance.

Raft nodes within a distributed system can assume one of the three states: leader, follower and candidate. The consensus is achieved by ensuring that these state transitions take place in a controlled manner. The Leader node is responsible for receiving and processing the client requests as well as maintaining the logs of the distributed system. The log replication is then sent to the follower nodes to maintain consistency. When the current leader is down, and a follower node wants to become the leader it transitions to the candidate state and requests for votes from other nodes in the cluster. If the node receives the majority of votes that is more than half of the total number of nodes in the cluster, it becomes the new leader.

Raft consensus protocol is implemented to solve many problems such as maintaining data consistency across the cluster and ensuring that the other nodes apart from the leader have reached the same state, log replication, fault tolerance, etc. However, when a raft consensus protocol is implemented such that only one node is eligible to become a leader, it increases the chances of failure as it undermines the fault tolerance and availability benefits of using consensus algorithm. In this dissertation we will try to resolve this issue by creating a raft cluster with a virtual dissemination of multi-level tree network topology and observe how it increases the fault tolerance of a distributed system when compared to a single level tree topology where only the root node of a cluster can act as the leader.

We will be using pysyncobj [] a python [] package to create a basic raft cluster by using python flask framework []. We will then use this flask application to create raft clusters with desired topology and observe their behavior by implementing various optimization strategies like batching [], log compaction [] using pysyncobjconf [] python package. To create a real-world production ready distributed system, we will deploy the application in AWS EC2 [] instances where these instances will serve as raft nodes, when the instances are launched it will pull the application from a GITHUB [] repository. To process the data sent by client to the leader node efficiently we will integrate Redis [] as a caching mechanism.

* 1. **Aim**

This research has focuses on doing a thorough study on raft consensus protocol and implementing it in a distributed system where the raft nodes are connected firstly in star network communication pattern and secondly in hierarchical star network communication pattern and confirm with the help of thorough analysis that how the latter is better than the former in terms of performance with respect to fault tolerance. This research will also compare the impact on latency and throughput for both the clusters by sending number of client requests to the leader node of the clusters.

* 1. **Objectives**
  2. **Motivation**
  3. **Project Scope**
  4. **Project Deliverables**

1. **Background Research**

**2.1 Distributed Systems and consensus**

A distributed system is a complex network of multiple interconnected nodes that collaboratively strive to achieve a common objective. Unlike traditional centralized systems, where a single entity controls all actions, distributed systems distribute tasks and data across various nodes to enhance performance, scalability, and fault tolerance. However, this distribution introduces challenges related to synchronization, communication, and decision-making. Consensus, a pivotal concept in distributed systems, addresses these challenges by facilitating a harmonious agreement among nodes, even in the face of failures, network disruptions, and other adversities.  
  
In the realm of distributed systems, the necessity for consensus arises from the inherent unpredictability of node behaviour and the likelihood of failures. Nodes can experience crashes, delays, or erroneous outputs due to hardware issues, software bugs, or network glitches. Consensus protocols are mechanisms that enable nodes to collaborate effectively, mitigating the impact of these issues and ensuring that the system remains functional and dependable. By establishing a unified decision-making process, consensus protocols foster coherence and reliability within the distributed system.

Consensus protocols operate under different models, such as synchronous and asynchronous, each reflecting the assumptions about the network's behaviour. In the synchronous model, where message delays have a known upper bound, achieving consensus is relatively straightforward. However, in the asynchronous model, where message delays and node failures can occur without warning, designing a robust consensus protocol becomes more intricate. The famous Byzantine Fault Tolerance (BFT) consensus algorithms, like Practical Byzantine Fault Tolerance (PBFT), are tailored to handle such scenarios, ensuring agreement even when malicious nodes attempt

to disrupt the process.

The process of consensus involves several stages, typically beginning with a node proposing a value or decision to the network. Subsequently, other nodes evaluate the proposal, provide feedback, and eventually acknowledge their acceptance or rejection of the proposal. The iterative nature of this process allows nodes to converge toward a single decision. Notably, the consensus protocols' design emphasizes properties like safety and liveness: safety guarantees that all nodes agree on the same value, while liveness ensures that a decision is

eventually reached, even if nodes are faulty.

Consensus protocols' deployment extends to various distributed systems applications. In blockchain technology, which underpins cryptocurrencies, consensus mechanisms like Proof of Work (PoW) [] and Proof of Stake (PoS) [] enable nodes to agree on the state of the blockchain without requiring a central authority. Cloud computing platforms also employ consensus for resource allocation and load balancing among virtual machines. Distributed databases and file systems use consensus to maintain data consistency across multiple nodes.

**2.2 Tools and Technologies**

The implementation of raft consensus protocol is done in many ways based on its use cases. For instance, hashicorp/raft [], etcd-io/raft [], go-raft [], etc. Similarly, python [] has a well-defined package pysyncobj which can be used to implement raft consensus protocol.

**2.2.1 Python**

Python is an interpreted programming language. It provides various web-application frameworks such as Flask [] and Django [] which can be used to build microservices with ease. Python provides a lot of packages that can be used to integrate your application with cloud providers like AWS, Azure [], GCP []. For instance, it has a separate package called Boto3 [] which is an AWS Software Development Kit (SDK) that can be used to interact with many AWS services.

**2.2.2 Flask**

Flask is a web-based application framework provided by python. It is built over Werkzeug toolkit [] and follows the Web Server Gateway Interface (WSGI) standards. With its ease to support not only RESTful APIs creation by providing tools for handling API endpoints and HTTP methods, but also managing data sterilization and deserialization [], it has gained a lot of popularity.

**2.2.3 Amazon Web Services**

Amazon Web Services (AWS) is a public cloud platform provided by amazon which can be used to test and deploy your application in the cloud. It provides many essential services to create a robust distributed system. Some of the commonly used AWS, services are:

* AWS Virtual Private Cloud (VPC): Amazon VPC [] is networking service that is used to create isolated private network environments in AWS cloud. It provisions a logically isolated section on AWS account where the user can create all the AWS resources. VPCs are region specific and every AWS region has various availability zones (AZs), which represents data centers that are separated physically.
* AWS Elastic Compute Cloud (EC2) instances: It is a service that can be used to create and manage virtual machines. Developers can set their own configurations like size and type of operating system based on their requirements and create an ec2 instance. EC2 instances plays a vital role in creating a distributed system where the nodes an application are spread over different subnets [].
* AWS Security Groups: Security Groups are used to restrict the inbound and outbound traffic to create a secure large scale distributed system. A security group by default, denies all the inbound and outbound traffic. User needs define security group rules to allow traffic to a specific ip-address or a range of ip-address along with ports.
* AWS Identity and Access Management (IAM) Roles: IAM roles [] are another security feature provided by AWS to give specific permissions to AWS services. It is used to follow the principle of least privilege [] which intern enhances the security of a system as a whole.

**2.2.4 Redis Database**

Redis [] is an open-source in-memory store that is used for high performance caching and data storage. It can be used to handle various types of data which can be as simple as a key-value pair as complex as data structures like hash, sets, sorted sets, lists. Due to its flexibility, low latency responses and speed, Redis is employed in many applications and one of them is caching.

**2.2.5 GitHub**

GitHub is a platform that is widely used for version control, code management and collaboration. The software projects are stored in GitHub repositories. It uses Git which is distributed version control system and therefore it can keep track of the changes made in a particular code. It can be a very useful to make the workflow of a distributed system easier. For instance AWS EC2 instances can pull the required application code and use it effectively. GitHub has become an essential tool for an individual developer as well as for an organisation.

**2.3 Working of RAFT consensus protocol.**

In Raft, the distributed system consists of a cluster of nodes communicating with each other to agree on a common log that represents the state of the system. It has three main components namely leader election, log replication, heartbeat mechanism. The figure below displays the working of raft consensus protocol in a distributed system.

**2.3.1 Leader Election**

Leader election [] is a crucial stage in raft consensus protocol that allows a node of distributed system to establish a new leader when the current leader fails, or a new node joins the cluster. Initially all the nodes in a cluster starts operating in the Follower state and recognizes the current leader’s authority. However, if a node starts up or loses communication with leader, it transitions to Candidate state by incrementing its term which signifies a new election round. The node then broadcasts requests for votes to other nodes to become the new leader. As soon as the node receives vote from the majority of the nodes, n/2 + 1 (where n is the number of nodes) in the cluster, it becomes the new leader. If no receives the minimum number of votes in a term, a new election is triggered in the next term. This well-defined leader election process guarantees that a leader is managing the cluster consistently thereby enabling effective consensus among the nodes of the cluster.

**2.3.2 Log replication**

When a node is elected as the leader of the raft cluster successfully, it assumes the responsibility of accepting requests from clients and guarantees the dissemination of changes throughout the distributed system. This is achieved through the process of log replication where the leader maintains a sequential log of commands, recording consequential state changes within the raft cluster. A log entry consists of a term and an index which is uniquely identified by its position in the log. When a client sends a request to the leader, it appends a new command to its log and then sends the updated log to all the followers in the cluster. The follower nodes upon receiving the log entries, each follower node sends acknowledgement (ACK) or append entries response back to the leader. When the leader receives ACK from majority of follower nodes, it considers that log entry as committed. This mechanism of log replication ensures the synchronization of all the nodes in the cluster. Figure demonstrates the process of log replication.

**2.3.3 Heartbeat mechanism**

Raft consensus algorithm also has a heartbeat mechanism where the leader node sends heartbeat periodically to the follower nodes to maintain its status as a leader to avoid frequent leader elections.

**2.4 Use of PySyncObj to create a RAFT cluster**

The figure shows workflow of pysyncobj python package and how it can be utilised to create a cluster of nodes that perform raft consensus algorithm by inheriting the SyncObj class. SyncObj class consists of the key functionalities required to manage the complex communication between nodes in a cluster.

A screenshot of a computer

Description automatically generated

**2.3.1 SyncObj class**

The SyncObj class comprises of a methods that help the nodes in a cluster to achieve consensus and maintain its stability with the help of leader election, log replication and sending append entries periodically. When a raft cluster is created leader election starts and as soon as the leader is elected \_\_onBecomeLeader() private method is called which sets the state of the node to LEADER and updates the value of nextIndex and matching index of each of the follower node. The next index value is used for keeping track of which is the next index will the follower nodes of the cluster will receive in the process of log replication. Matching index on the other hand is responsible for keeping track of all indices that have been replicated successfully in the follower nodes. If the match index count is 0 it means that none of the log entries have been replicated to the followers. The figure shows code for \_\_onBecomeLeader() function.

A computer screen shot of a code

Description automatically generated

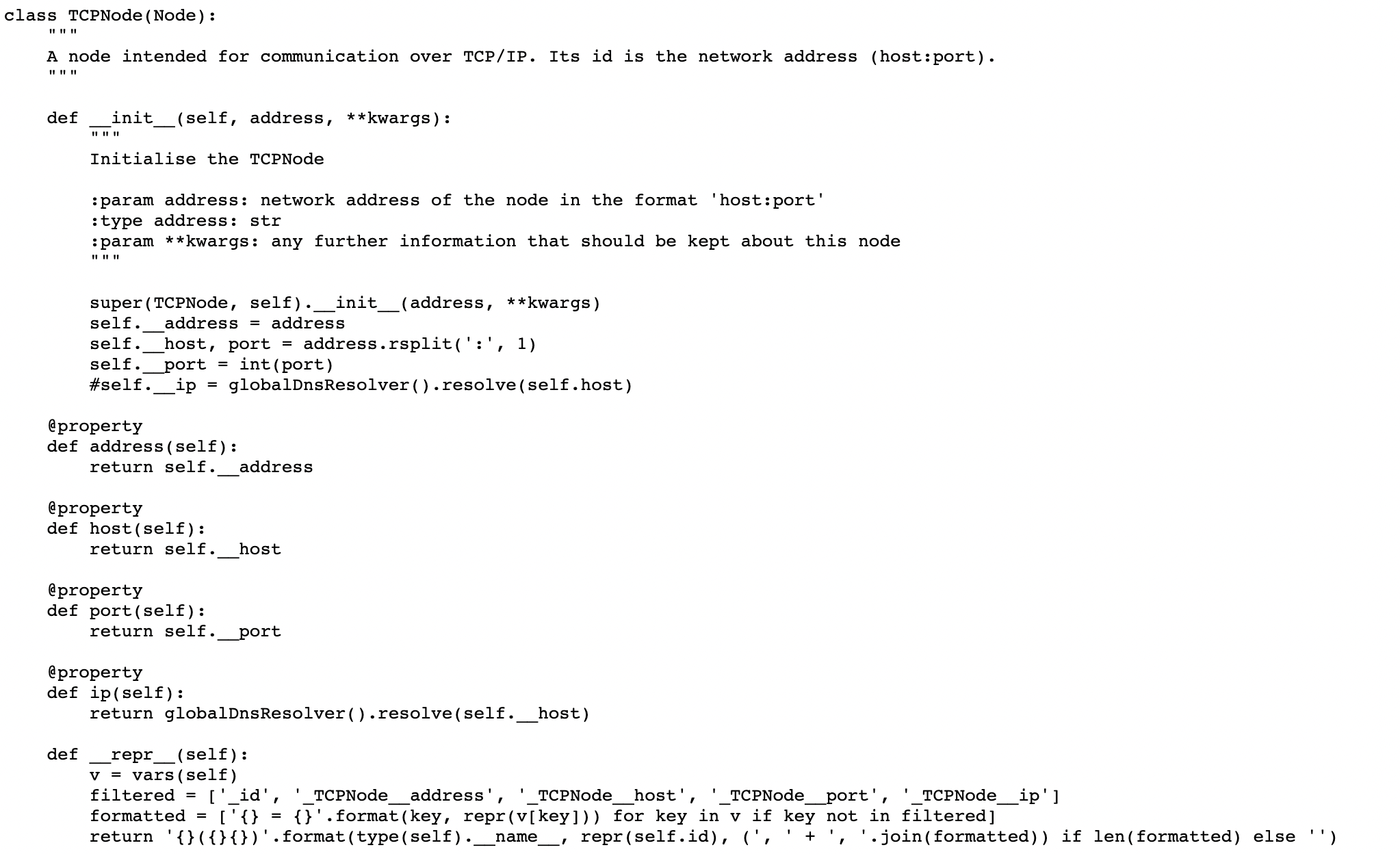
The \_\_onBecomeLeader() method calls the \_\_sendAppendEntries() method to send the heartbeats to the followers. The figure shows the code for \_\_sendAppendEntires method (). The log entries sent by the leader node consists of information such as current term of the leader which signifies the term that leader of the raft cluster operating, commit index indicating the highest log entry committed at present, actual data of the log entry, previous log index and previous log term. The figure shows the message sent by the \_\_sendAppendEntries() method.

A computer code with text

Description automatically generated with medium confidence

**2.3.2 Node Class and TCPNode class**

The node class is designed to create a raft node in a network. It has a subclass called TCPNode which handles the communication of nodes in a cluster over TCP/IP []. Every node in the cluster has its unique id as its representation. Every node is initialized with the network address that consists of hostname and port. Once the communication between the nodes of the cluster is established it becomes the part of the quorum of the raft cluster. Figure and figure show the code for Node class and TCPNode class.



A screenshot of a computer code

Description automatically generated

**2.3.4 DnsCachingResolver class**

The dnsCachingResolver class and shown in the figure takes responsibility to resolve domain names to ip-addresses using caching. By using caching mechanism, it minimizes the need to query DNS servers [] repeatedly for same hostnames which improves the performance of the distributed system. The globalDnsResolver uses this class and acts as the central point to make sure that same instance is used throughout the application for efficient DNS resolution. Both Node and TCPNode class uses utilizes the globalDnsResolver to resolve host names. This is a crucial step to establish a stable network connection between the raft nodes. Figure shows the code for DnsCachingResolver class.

A screenshot of a computer program

Description automatically generated

**2.3.5 Transport class**

Transport class adds an abstraction layer for different types of network transports that enables raft nodes to exchange messages, maintaining connections along with managing communication events. It is not only defining many significant methods and callbacks for initializing, connecting, sending and handling messages, but also engages in managing the state of nodes. It has functionality to provide support for both regular and read-only raft nodes. It is inherited by TCPTransport class which helps a crucial role in creating a stable raft cluster by making sure that nodes can communicate effectively, exchange messages and coordinate consensus operations.

When the raft cluster is created, the TCPtransport class is utilized in the following manner:

* Instantiation and configuration: TCPTransport class instantiate with a reference to SyncObj class as shown in figure. It takes parameters such as selfNode and otherNode which implies the raft’s host node and its partner nodes.
* Server Management: When it comes to regular nodes TCPTransport class creates a TCP server to accept the incoming connection requests and handles the management of outgoing connection to other nodes. A dictionary \_connection is used to keep track of all the connections.
* Exchange of messages: TCPTransport class has the functionality to ensure whether messages are sent and received successfully by handling retries and connection state management.

Figure shows a detailed uml class diagram for transport class.

A screenshot of a computer

Description automatically generatedA screenshot of a computer error

Description automatically generated

**2.4 Network Communication Patterns in a Distributed System**

Networks topology [] refers to the arrangement of nodes in a system that enables exchange of information through communication links along with coordination of tasks in a distributed environment. It serves as a blueprint for addressing challenges like data consistency, availability etc. Based on the selection of network topology, a system’s performance a can have a major impact in terms scalability, fault tolerance, latency, and throughput. Star communication pattern [] and hierarchical star communication pattern [] are two well know network topologies.

**2.4.1 Star Network Communication Pattern**

In this type of network communication, all the nodes (servers) are connected directly to single node of a distributed system which will be responsible for handling tasks and maintaining the stability of the system. It provides a centralized management which makes it easier to monitor traffic and control data flow. Addition of new nodes does not affect the working of system as long as the central node is able to handle the workload. However, if the central node is down, it will have a major impact on the cluster in terms of fault tolerance as it is the single point of failure. Therefore, the central node of the cluster in this communication pattern, becomes the performance bottleneck. Figure shows the representation of a star network communication pattern with five nodes.

**2.4.2 Hierarchical Star Network Communication Pattern**

This communication pattern is an enhancement of star network communication pattern. Here the nodes of a distributed system cluster are connected such that all the nodes are not connected directly to a single node. Instead, the connection between the nodes is such that it has different levels communication. This network pattern increases the fault tolerance of the system as all the nodes are not relying on a single node for distributing the messages in the entire cluster. Figure shows the representation of this network communication pattern with five nodes.

1. **Methodology**

This section illustrates the implementation of raft consensus protocol by creating a flask framework-based application that uses pysyncobj python package to create a raft cluster of desired number of nodes. The logic to create a flask application to implement raft consensus protocol using pysyncobj package and run it in local environment has already been implemented []. We will be discussing the workflow of the application and how we can create a raft cluster with star network communication pattern and hierarchical star network communication pattern between raft nodes in a distributed system created in AWS public cloud.

* 1. **Application Overview**

**References**

[] Moisieienkov, V. (2021). flask-pysyncobj-raft-replication-demo. [Online]. 6 December 2021. Available at: <https://github.com/VMois/flask-pysyncobj-raft-replication-demo/tree/main> (*Accessed: 11 June 2023*).