Implementing Distributed Key Value Store

By: Shatakshi Gupta, Shefali Sharma, Sanskar Vij, Saransh Malik
Guided By: Mr. Vijay, CSE Department, Meerut Institute of Engineering and Technology, Meerut
shatakshi.gupta85@gmail.com
rockingshefali99@gmail.com
sanskarvij007.sv@gmail.com
Saransh10malik@gmail.com
vijay.movfrose@gmail.com

DOI:10.48047/ecb/2023.12.si4.730

Abstract

This paper describes how we can design and implement a Distributed Key Value Store using a programming language specifically the Golang programming language. Distributed Key Value Store has become majorly essential these days with the large amount of user data being generated on a day to day basis. This data needs to be organized and stored in a DataBase that is easy to query and manipulate. This approach is required when the DataBase usually involves frequent retrieval requests like Read, Write and Delete. Prior to the introduction of Distributed KVs, Single Storage Key Value pair would be used but one node store could not keep up with issues like sudden increase in demand of certain resource and would mostly be center of single point of failure where if the server shuts down abruptly it would lead to major customer loss and business as a whole. Due to this, Distributed KVs came up with the idea of distributing keys and values on multiple servers and instances so that reliability and overall performance of the system could be improved. The Golang programming language was chosen for its simplicity, readability, and performance, which makes it an ideal choice for developing a Distributed Key Values Store.

Keywords- Distributed Computing, Distributed Key-Value Stores, Raft

Introduction

Distributed Key Value Store can be described as a modern solution for most high-data oriented organizations and companies where top priority is user satisfaction by fulfilling their needs and demands. Whether it is launching a new product or providing services to users, it is very much important to keep user requirements in mind in order to present the best possible experience of the product to users. To meet this condition the first thing that comes is user data of any form likes, dislikes, profile customisation and much more. As the data and content increases with more users accessing a platform, the complexity of handling such a huge amount of data also increases. Hence there was a need to develop a system that could take control of big data without any loss.
Initially, many methods were introduced that are discussed below:

Distributed computing was developed for easy access, process, and analyzing data using technologies like Apache Hadoop, Apache Spark where data processing is performed across multiple nodes and servers. This leads to efficient and faster processing of large volumes of data. **NoSQL databases** are designed for handling unstructured and semi-structured data. Some examples include MongoDB, Cassandra etc. **In-memory computing** method suggests storing data in direct RAM instead of Disk which can also help in increasing processing speed. **Data-streaming** method involves processing and analyzing data in real-time rather than storing it in batches and then performing operations. This is helpful to significantly improve user interaction by reducing time. **Machine learning** involves using algorithms and programs to analyze and predict patterns in data so that more user friendly products and services can be built. Some commonly used applications for this purpose are TensorFlow, PyTorch etc.

Of the approaches discussed above, **NoSQL databases** which have the capability of storing and managing **unstructured** and semi-structured data stood out in terms of flexibility, efficiency and complexity while dealing with huge amounts of data that is big data. And it is inferred that this should be chosen for further operations. NoSQL databases could offer many approaches like document and collection based approach in MongoDB, **key-value** storage method for **Cassandra**, **Redis** etc.

After many application testing, and operations it was found that using key-value storage methods in NoSQL databases would be much helpful considering the evolution of **big data** and **cloud computing**. Key-value store refers to the concept of storing data in the form of key-value pairs in which each data is represented by a unique key and data is accessed and manipulated using this key. Key-value is introduced so that operations like **indexing** and **searching** of data could be queried and done in minimum possible time. Here keys are usually strings or integers type while values can be of any type like texts, numbers or binary data.

At first a single node or server was used to implement and store data of key-value form for accessing and manipulating data. But with the sudden boost in big data and cloud computing , data started increasing with new users signing up and registering for products and services. **Single node** method would fail at this point because it was not built to handle such enormous data. Hence distributed key-value storage method is introduced to manage **scalability** issues that could not be handled in a single server method.

A **distributed key-value** store is a type of database storage system specifically designed to store and manage data as key-value pairs that are distributed across multiple networked servers and nodes. In a distributed key-value store, data is **partitioned** and **replicated** across multiple nodes in the cluster to provide **fault tolerance** and high **availability**. Some important features provided by a distributed key-value are described below:

The very first feature is **Scalability** in which Distributed key-value stores are designed to be horizontally scalable which is implemented using a method termed **sharding** in which data is held and distributed on the basis of tuples and not columns. Next being **Fault tolerance** implies that Distributed key-value must replicate data across multiple nodes to ensure that data remains available even in the event of node failures. The following is **High availability** to confirm Distributed key-value stores provide high availability by ensuring that data is accessible even in the case of nodes unavailability or failure. **Data partitioning** means Distributed key-value stores partition data across multiple nodes to ensure that data is distributed evenly and efficiently across the cluster. The last is **Data replication** so that Distributed key-value stores replicate data across multiple nodes to provide fault tolerance and high availability.

Overall, distributed key-value stores are built to be scalable, fault-tolerant and highly-available. These are designed to handle large volumes of data spread across multiple networked nodes.

Consensus Problem

Single Server architecture could not handle large numbers of requests and traffic. Modern society requires **highly reliable** and **available** systems in order to execute and meet their needs and demands. **Single machine** concept was not able to keep up with the expectations as these are more susceptible to crashes and **downtime** with increase in traffic and network.

Developing multiple machines that could perform the same tasks and provide response in seconds with minimal hassle for users was the need of the hour. For multiple machines to coordinate they all must have a strong **network connection**, high **availability** and **resilient** to any kind of system failure.

Consensus is a fundamental part of building **replicated** systems and getting such machines to work together. For multiple machines to work together there needs to be some kind of **agreement protocol** that all machines in the system can follow to achieve a desired result. Consensus problem is described as the process in which multiple nodes or servers try to agree on a single value to fulfill a common goal. A coordination process is required so all nodes can follow the same steps to get a value.

To obtain correct result, a consensus algorithm must follow the properties of Agreement, Validity and Termination. In **Agreement,** every non-faulty node or server must agree on the same value. **Validity** lays emphasis that the value to be agreed on must have been offered by another process as well so that majority can be counted and decision can be made. **Termination** includes the method in which every correct process eventually decides on a value and replicates the decision to execute it.

What is a **non-faulty node**? In practical situations, a node that crashes abruptly by sending no response to a message sent by another node, acts strangely like not agreeing to a common value, or gets hacked are termed to be faulty and are undesirable while making critical decisions in consensus problems as these might lead to wrong output.

To create a software application that is highly reliable and available, it becomes crucial for the program to be active all the time so that each client request is served. To ensure this condition, it is important to keep the application code on several machines or servers so that a high number of client requests can be served even if one or more nodes suffer any downtime or network failure. This process of replicating the program code on several machines and nodes also helps to distribute requests load to different servers thereby reducing overall workload of single node as in case of single server systems.

Raft Algorithm

When we want to make our computer program **reliable** and believe every server is acting and serving requests it is important to have strong network connection between these servers in order to ensure **consistency** and availability of data among all servers.

Raft Consensus Algorithm was introduced to deal with this concept. It is often referred to as a distributed consensus algorithm. In this algorithm a **leader** is chosen that ensures that each and every node in the system agrees to **common value** and acts in accordance with that agreed value. It solves the problem of getting multiple servers to agree on a shared state even in the face of failures. The shared status is usually a data structure supported by a **replicated log**.

Raft initiates by electing a leader in the cluster. The leader is responsible for accepting client requests, interactions and managing the logs. The data flows only in one direction: from leader to other servers. All servers except Leader is a **follower** that responds to commands issued by the Leader through the client. **Candidate** refers to a temporary state that is used by servers during the **Leader election** process to depict their potential candidacy.

Raft divides consensus into three smaller issues:

The first process of **Leader Election** involves Election of a new leader in the event of the situation of current Leader failure. The next step is **Log replication** in which By replication, the leader must keep all servers' logs up to date with its own. The last process revolves around **State Machine Safety** such that No other server may apply a different log entry for a given index if one of the servers has already committed a log entry at that index.

Importance of Raft

Reliability, Consistency and Safety are the main components due to which Raft Algorithm seems appropriate to deal with the situation of replication of a computer program to multiple servers. These components are managed and presented through this algorithm in the following ways:

Safety means there should only be one and only one leader at any point in time by ensuring only one vote is sent out by every server and further refusing to accept requests of other candidates. And persist this data on disk to avoid crash failure etc.

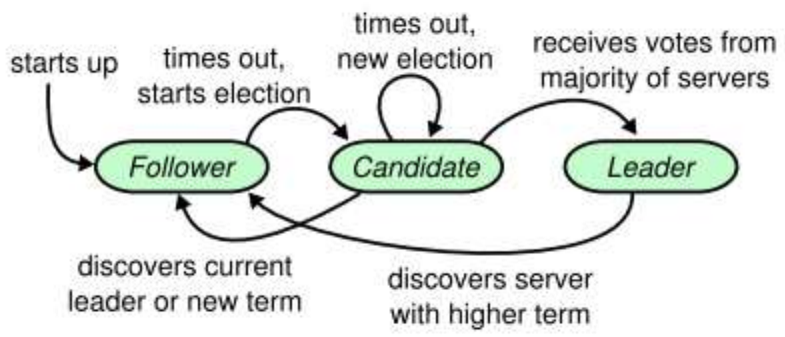
Liveness means there must not be any situation where the system is running out of leader server. Every server must start an **election timeout** randomly to avoid election collision with another server. In this condition, whichever server has lesser T value will start before than any other server and eventually gain votes and win the elections.

The above mentioned features further enhance the Raft Algorithm to succeed in meeting with crucial conditions like Fault Tolerance and availability of the entire system.
Compared to other algorithms like Paxos, Raft is easier to understand, and implement. Some other reasons why Raft should be considered are: **Simplicity** as Raft Algorithm is the most easily understandable consensus algorithm which makes it easier to implement as compared to Paxos. **Availability** is implemented in Raft and is designed in a way to be highly available even in situations like server downtime and node failure. **Safety** features in this algorithm helps ensure there is always a safety mechanism up and running in the system like the Leader Re-election process if no leader is elected in a certain term, sending heartBeat messages etc.

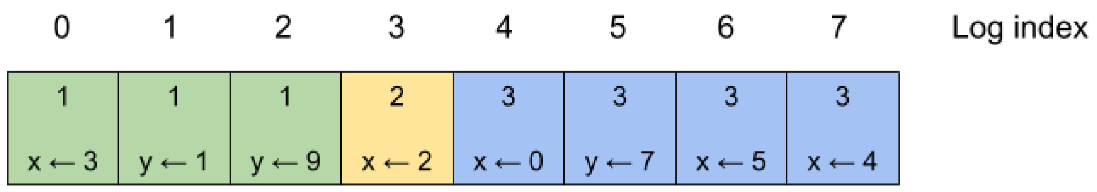
Design and Implementation of Proposed Systems

Raft uses a state machine to ensure that each node in the cluster agrees upon the same series of state transformations. Each node in the cluster has three possible states: Follower, Candidate or Leader. The Follower node accepts write operations from the leader. Candidate node has the same responsibilities as the follower node but it is also a potential leader. The Leader node is responsible for accepting requests from the clients and controlling other nodes.

In this algorithm, Time is divided into terms. Each term consists of two situations-leader election and operations. In case no leader is elected, time automatically proceeds to the next term that elects a leader. Each server maintains the term number in order to not contain any outdated information. Each server starts as a follower. For followers to ensure that there exists an active leader with these followers, the leader of the system is expected to send out heartbeat messages or remote procedure/function call that are basically empty RPCs on a regular basis. a certain amount of time is allotted, that is election timeOut then followers believe there's no leader among them and a new election process has to take place.


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Raft synchronizes changes with the help of replicated logs. Every change submitted to the Raft cluster is a log entry that gets stored and replicated to the followers in the cluster. The log is constructed as a linear sequence of commands that should be applied to the state machine. If a node in the Raft cluster crashes then it is brought back up by replaying all the commands in log through the state machine.

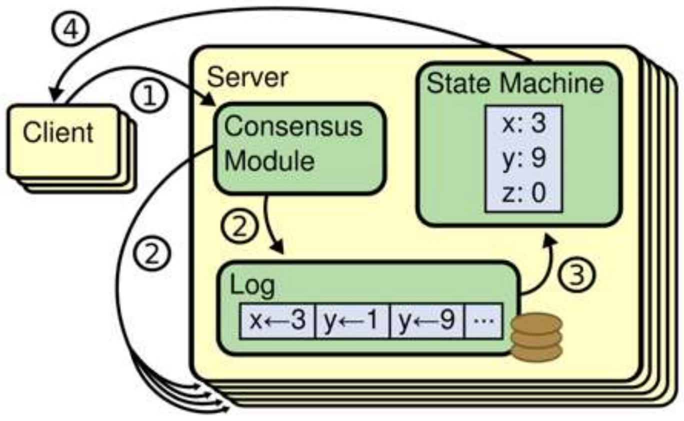


0	1	2	3	4	5	6	7	Log index
1	1	1	2	3	3	3	3	
x ← 3	y ← 1	y ← 9	x ← 2	x ← 0	y ← 7	x ← 5	x ← 4	

For implementing our distributed key value store, we have made use of hashicorp raft and raft-boltdb golang packages. Hashicorp raft is a library that is used for managing replicated log and replicated state machines. Each node in the cluster will need a permanent storage to store the append-only log. We shall be using raft-boltdb package to implement that storage. This package exports the BoltStore which is an implementation of both a LogStore and StableStore. We have implemented get, set and delete methods for the values in our key-value store. We have exposed these methods to the user with the help of HTTP APIs written in go.

As defined before, logs are append only sequence of records. They play a very crucial role in the design of distributed applications. They allow us to implement algorithms like version control, replication and consensus very easily. A log entry consists of the following components: index, term and command. Index is an increasing sequence number for each entry in the log. Term is basically a logical clock, a counter value indicating the current term when the leader receives the entry from the client. Command is the instruction given by the client. These logs are replicated across all the nodes for maintaining consistency in the system. When a client sends a write request to the Raft cluster, the leader node appends that entry to its log and sends an AppendEntries RPC to all the follower nodes. The follower nodes then append that log entry to their own logs.

Raft Finite State Machine is responsible for executing commands from the replicated log. The state machines ensure that the commands are executed from the replicated log in a deterministic manner across all the nodes in the cluster. The state machine on each node applies the command to its local state in the same order as it was executed on the leader's state machine. It has the following three methods, **Apply** is invoked after commiting a log entry. **Snapshot** is periodically called to snapshot the state of the FSM. **Restore** is used for restoring an FSM from the snapshot.



Sharding is a frequently used technique in Databases. This method is helpful when there is a need to spread and distribute data on multiple servers. This situation is often observed in databases when the data being generated is huge and a single server fails to keep up with the requirements and When there is a sudden burst in the number of queries being made to the database which leads to increasing traffic due to high rise in demand of users and clients.

This condition gives rise to CPU exhaustion and memory running out of space thereby reducing the overall efficiency of the system. Sharding utilizes the idea of horizontal partitioning to evenly distribute the data across several existing servers so that the huge load of queries and requests could be handled. In the concept of horizontal partition, rows are held separately and are stored on database servers. This method helps in fast retrieval of search queries as the indexing of each row is greatly improved. This is the base technique which is most commonly used in the process of load balancing and managing increase in traffic. In this method, a large dataset is broken into smaller pieces called shards and are distributed across servers. Then these database servers and nodes are responsible for locating and serving the client requests. Sharding uses the method of consistent hashing to get an index and store the desired shard on a particular server and evenly distribute data on all servers.

Consistent hashing is a technique used to map keys to nodes in a distributed key-value store. It ensures that the load is evenly distributed across the nodes. Consistent hashing also ensures that the system can scale dynamically as nodes are added or removed.

Conclusion

In this paper we present how we can implement distributed key-value stores to design highly scalable and available distributed systems. We highlight the need for key-value stores in large scale distributed systems and the methodology to implement them in golang with the help of the Raft protocol.

References

[1] Boost the Performance of Deep Learning Model Based on Real-Time Medical Images." Journal of Sensors 2023 (2023).

[2] Babu, S. Z., et al. "Abridgement of Business Data Drilling with the Natural Selection and Recasting Breakthrough: Drill Data With GA." Authors Profile Tarun Danti Dey is doing Bachelor in LAW from Chittagong Independent University, Bangladesh. Her research discipline is business intelligence, LAW, and Computational thinking. She has done 3 (2020).

[3] NARAYAN, VIPUL, A. K. Daniel, and Pooja Chaturvedi. "FGWOA: An Efficient Heuristic for Cluster Head Selection in WSN using Fuzzy based Grey Wolf Optimization Algorithm." (2022).

[4] Faiz, Mohammad, et al. "IMPROVED HOMOMORPHIC ENCRYPTION FOR SECURITY IN CLOUD USING PARTICLE SWARM OPTIMIZATION." Journal of Pharmaceutical Negative Results (2022): 4761-4771.

[5] Narayan, Vipul, A. K. Daniel, and Pooja Chaturvedi. "E-FEERP: Enhanced Fuzzy based Energy Efficient Routing Protocol for Wireless Sensor Network." Wireless Personal Communications (2023): 1-28.

[6] Tyagi, Lalit Kumar, et al. "Energy Efficient Routing Protocol Using Next Cluster Head Selection Process In Two-Level Hierarchy For Wireless Sensor Network." Journal of Pharmaceutical Negative Results (2023): 665-676.

[7] Paricherla, Mutyalaiah, et al. "Towards Development of Machine Learning Framework for Enhancing Security in Internet of Things." Security and Communication Networks 2022 (2022).

[8] Sawhney, Rahul, et al. "A comparative assessment of artificial intelligence models used for early prediction and evaluation of chronic kidney disease." Decision Analytics Journal 6 (2023): 100169.

[9] Srivastava, Swapnita, et al. "An Ensemble Learning Approach For Chronic Kidney Disease Classification." Journal of Pharmaceutical Negative Results (2022): 2401-2409.

[10 Mall, Pawan Kumar, et al. "FuzzyNet-Based Modelling Smart Traffic System in Smart Cities Using Deep Learning Models." Handbook of Research on Data-Driven Mathematical Modeling in Smart Cities. IGI Global, 2023. 76-95.

[11] Mall, Pawan Kumar, et al. "Early Warning Signs Of Parkinson's Disease Prediction Using Machine Learning Technique." Journal of Pharmaceutical Negative Results (2022): 4784-4792.

[12] Pramanik, Sabyasachi, et al. "A novel approach using steganography and cryptography in business intelligence." Integration Challenges for Analytics, Business Intelligence, and Data Mining. IGI Global, 2021. 192-217.

[13] Narayan, Vipul, et al. "Deep Learning Approaches for Human Gait Recognition: A Review." 2023 International Conference on Artificial Intelligence and Smart Communication (AISC). IEEE, 2023.

[14] Narayan, Vipul, et al. "FuzzyNet: Medical Image Classification based on GLCM Texture Feature." 2023 International Conference on Artificial Intelligence and Smart Communication (AISC). IEEE, 2023

[15] Mahadani, Asim Kumar, et al. "Indel-K2P: a modified Kimura 2 Parameters (K2P) model to incorporate insertion and deletion (Indel) information in phylogenetic analysis." Cyber-Physical Systems 8.1 (2022): 32-44.

[16] Singh, Mahesh Kumar, et al. "Classification and Comparison of Web Recommendation Systems used in Online Business." 2020 International Conference on Computation, Automation and Knowledge Management (ICCAKM). IEEE, 2020.

[17] Awasthi, Shashank, Naresh Kumar, and Pramod Kumar Srivastava. "A study of epidemic approach for worm propagation in wireless sensor network." Intelligent Computing in Engineering: Select Proceedings of RICE 2019. Springer Singapore, 2020.

[18] Srivastava, Arun Pratap, et al. "Stability analysis of SIDR model for worm propagation in wireless sensor network." Indian J. Sci. Technol 9.31 (2016): 1-5.

[19] Ojha, Rudra Pratap, et al. "Global stability of dynamic model for worm propagation in wireless sensor network." Proceeding of International Conference on Intelligent Communication, Control and Devices: ICICCD 2016. Springer Singapore, 2017.

[20] Shashank, Awasthi, et al. "Stability analysis of SITR model and non linear dynamics in wireless sensor network." Indian Journal of Science and Technology 9.28 (2016)