Building Resilient Distributed Systems: Challenges and Solutions in a Raft-based Leaderboard Application

Introduction

The ability to withstand disruptions and failures is a crucial quality for any distributed system that aims to deliver dependable and uninterrupted service. In today's world, distributed systems have become a fundamental aspect of our daily lives. Massive amounts of data are managed by systems like telecommunication networks, power grids, and financial systems that provide vital services. Any downtime or malfunction can result in serious consequences.

To ensure uninterrupted service in adverse conditions, it's imperative to construct robust distributed systems that can handle failures, recover promptly, and continue delivering services. One of the widely adopted approaches to attain resilience in distributed systems is by utilising a distributed consensus protocol, which enables multiple nodes to reach a consensus on a single value despite the presence of failures.

The Raft consensus algorithm has gained popularity as an effective solution for constructing resilient distributed systems. It offers a fault-tolerant mechanism for ensuring consistency within a group of nodes in a distributed system. The Raft algorithm guarantees that the cluster can operate seamlessly, even if some of the nodes fail or are not available.

This report aims to examine the obstacles in constructing a robust distributed system using the Raft consensus algorithm. We will focus on a leaderboard application as an example to demonstrate the difficulties and resolutions in developing a fault-tolerant system capable of managing failures and recuperating rapidly.

Background and concepts

A distributed system consists of several autonomous nodes that collaborate to accomplish a shared objective. These nodes communicate with each other through message passing and coordination mechanisms to synchronise their operations and ensure coherence throughout the system.

Nodes within a distributed system can be either physical machines or virtual instances operating on a cloud platform. Each node in the system has its designated tasks to execute and can serve a distinct purpose, such as a client, server, or leader.

Nodes in a distributed system utilise message passing as a means of exchanging information and synchronising their actions. Messages can be transmitted either synchronously or asynchronously and can carry various forms of content, including data, commands, or requests.

In a distributed system, coordination refers to the process by which nodes collaborate to attain a shared objective. This necessitates guaranteeing that all nodes are informed of one another's state and that they are striving towards a shared goal. Coordination is critical for preserving coherence and avoiding conflicts within the system.

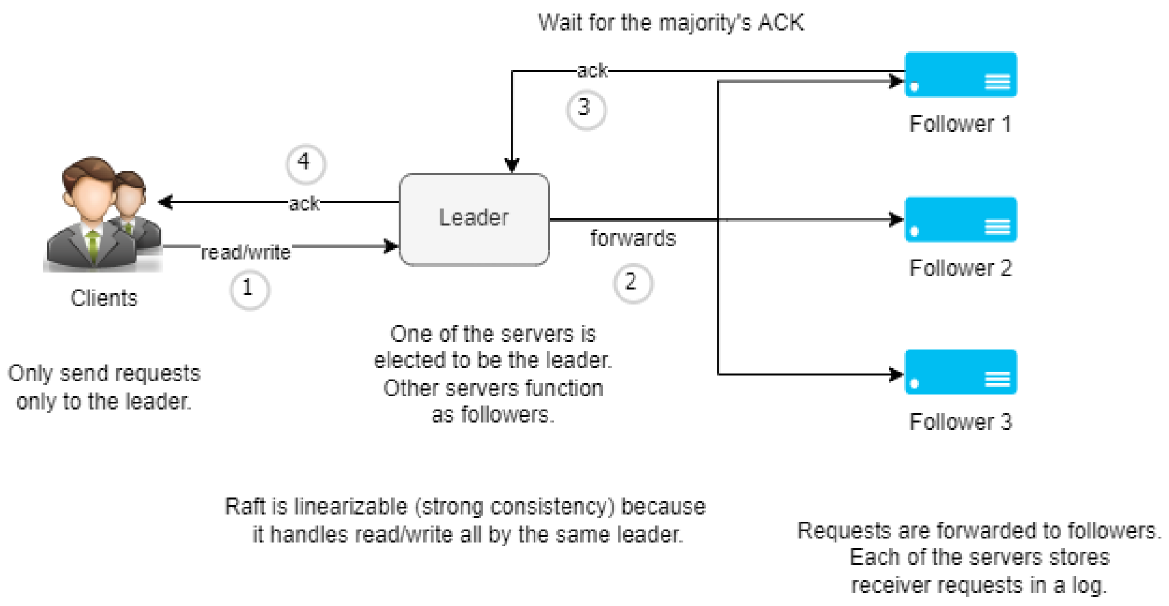
Distributed systems frequently face the issue of node crashes, which occur when one or more nodes within the system do not respond or shut down unexpectedly. Such crashes can result in data inconsistencies as the failed node may have contained crucial data that is no longer accessible. Additionally, network partitions pose a challenge, where the network connections among nodes are separated into two or more disjoint sets, rendering communication among them impossible. This can cause inconsistencies and conflicts within the system as different sets of nodes may be working with varying data.

Raft consensus algorithm has the following key concepts of consensus: leader election and log replication.

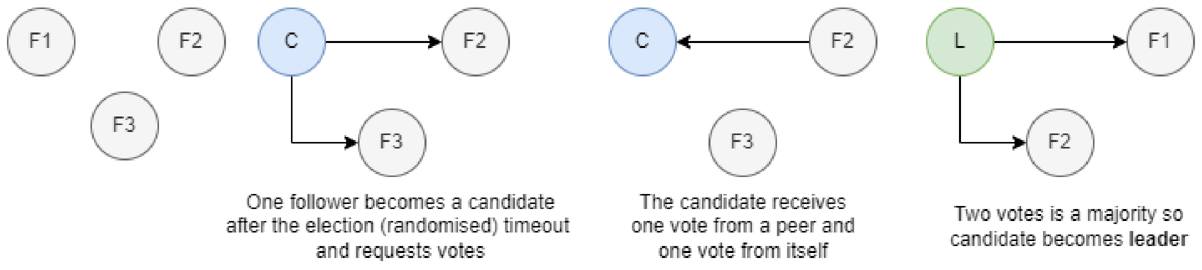
- **Leader Election**

In the event of a leader crash, a new election is initiated, resulting in the selection of a new leader. The election occurs among all member nodes of the cluster when creating a new Raft cluster or when a leader is not available. Hence, nodes can exist in any of the following states at any given time: Follower, Candidate, or Leader.

Upon receiving a command from a client, the leader node in the Raft cluster processes all incoming commands and subsequently disseminates them to each follower. Once a majority of nodes respond successfully, the command is committed, and the client receives confirmation. The data flow direction is unidirectional, i.e., from the leader to other servers. Conversely, the follower nodes are passive and solely respond to requests from the leader or candidates. In the absence of a leader, any follower node can become a candidate and initiate an election to obtain votes from other members in the cluster.


Wait for the majority's ACK
ack
3
4
ack
Leader
read/write
1
forwards
2
Follower 1
Follower 2
Follower 3
Clients
Only send requests only to the leader.
One of the servers is elected to be the leader. Other servers function as followers.
Raft is linearizable (strong consistency) because it handles read/write all by the same leader.
Requests are forwarded to followers. Each of the servers stores receiver requests in a log.

The Raft consensus algorithm employs a heartbeat-based RPC mechanism to determine when to initiate a new election. During normal operation, the leader node sends periodic heartbeat messages to all active follower nodes. If a follower node has not received any heartbeat messages for a specific duration, it triggers an election timeout, which leads to a new election to select a new leader.


F1
F2
F3
C
F2
F3
One follower becomes a candidate after the election (randomised) timeout and requests votes
C
F2
F3
The candidate receives one vote from a peer and one vote from itself
L
F1
F2
Two votes is a majority so candidate becomes leader

In our application, all servers start as followers, the first server will transition to the candidate state and will request votes from the other available followers. A follower will grant its vote to a candidate that has a log at least as up to date as its own. If the candidate receives the majority of votes, it will become a leader.

- **Log Replication**

This is the process by which the leader node ensures that the logs of all nodes in the cluster are consistent with its own. When the leader receives a request, it adds an entry for it to its durable local log. The leader then replicates this log to all of its follower nodes or replicas. After a majority of replicas have acknowledged that they have persisted the log, the leader applies the entry and instructs the replicas to do the same.

Techniques and solutions

Raft consensus algorithm enables nodes to choose a leader that synchronises the operations of all nodes in a distributed system. The algorithm ensures consistency across the system by copying the leader's state to all nodes and requiring a majority agreement before any decision can be made. Using the Raft algorithm in our leaderboard application, nodes communicate via message passing and coordinate their actions to keep the leaderboard consistent throughout the system.

To ensure fault tolerance and resilience in our distributed leaderboard system, we replicate the leaderboard data across multiple nodes and utilise the Raft consensus algorithm to elect a leader that coordinates the actions of all nodes. In case of node failure, the system can continue functioning by electing a new leader and replicating the data to other nodes. If a network partition occurs, the leader can detect it and ensure that all nodes work with the same data by waiting for a majority of nodes to agree on a decision before committing it.

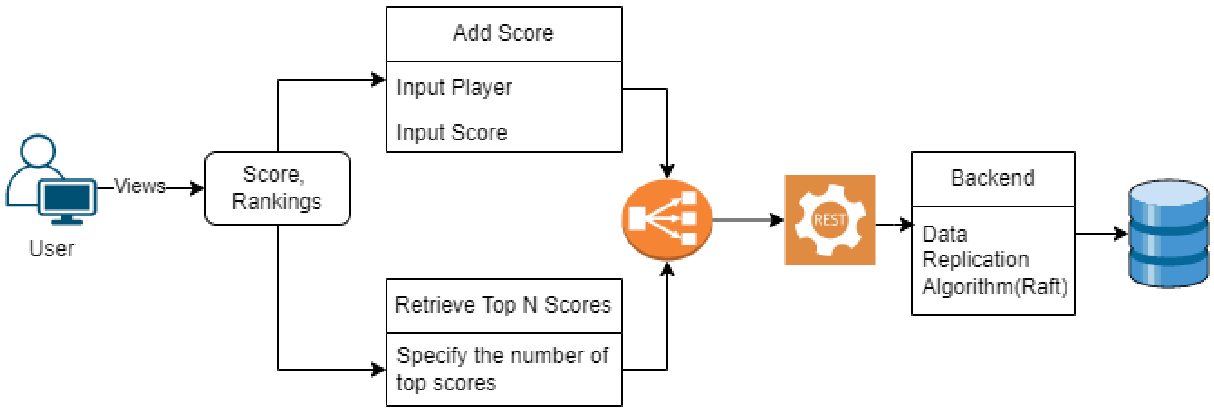
The load balancer is a critical component in our proposed application, as it distributes incoming requests among the servers that form the system. It ensures that requests are directed to the current leader server in the Raft cluster, which is responsible for coordinating data replication across all servers in the system.

In the absence of a load balancer, incoming requests would require the specification of the address of the current leader server, which can change as a result of leader elections in the Raft cluster. This can make the system more complex and less scalable.

Mini-application proposal

Distributed Leaderboard System:

Implement data replication and a consensus algorithm (Raft) for consistency and availability. A simple web application with RESTful API endpoints for adding scores, retrieving top N scores, and displaying player rankings. Store data in memory.



1. The user interacts with the application through the frontend, which is a web page.
2. The frontend communicates with the backend through RESTful API endpoints, sending requests to add scores, retrieve top N scores, and display player rankings.
3. A load balancer is added before the REST API layer. The load balancer distributes incoming requests to multiple instances of the REST API running on different nodes. This helps to evenly distribute the load and improve the overall performance and scalability of the system.
4. The backend receives the requests and processes them using the data replication algorithm and persistence layer, which ensure consistency and availability of the data.
5. The backend stores the data in memory.

Implementation and demonstration

- Implement the proposed mini-application using appropriate tools, libraries, or platforms.

The steps that follow were executed:

1. Implement the web application with RESTful API endpoints for: /add_vote, /top_n_scores and /player_rankings.
2. Implement the Raft consensus algorithm:
    a. Integrate an existing Raft library, such as “pySyncObj” for Python in the application;
    b. Configure the Raft library with the addresses of the our 3 servers;
    c. Using the library, replicate the logs across all the servers.
3. Set up and configure a load balancer for the managed instance group. (Initially, we have a list with the addresses of the our 3 servers) (loadbalancer.py)
    - We basically build a load balancer using the Round Robin algorithm and assign requests equally to each available node (step by steep, Ex: if we have only 2 nodes, node 1 with request 1, node 2 with request 2 and then we start from the beginning with the node 1 with request 3, like a circular list).
    - Also the same file acts as a proxy, taking requests from the front-end and communicating them further to the available nodes and finally sending the response received from the nodes to the front-end. If we don’t receive a response from a node, we move on to the next node in the list of available nodes. (if a follower node receives a specific request from a client, it does not have the authority to process the request on its own. Hence, it forwards the request to the leader node to handle.

- Explain the implementation process and any challenges or lessons learned during development.

  Challenges encountered during the implementation process:

1. **Choosing the right data replication algorithm**

Various algorithms exist for implementing data replication, including Raft, Paxos, and Zab. In order to fulfil our homework requirements, we conducted research on data replication algorithms, comprehended the needs of our application, and selected the most suitable approach, which is the Raft consensus algorithm.

2. **Scaling the system**

To ensure fault tolerance and graceful handling of failures, we integrated a load balancer into our system.

Conclusion

In conclusion, building a resilient distributed system is a challenging task that requires careful consideration of various factors such as fault tolerance, consistency, scalability and performance. The Raft consensus algorithm has emerged as a popular solution for building such systems due to its simplicity, fault tolerance, and ease of implementation.

In this report, we have presented a Raft-based leaderboard application as a case study to highlight the challenges and solutions involved in building resilient distributed systems. We discussed how the Raft algorithm helps maintain consistency and fault tolerance by electing a leader and replicating the leaders' log across all nodes in the systems.

We also discussed several design choices that can impact the performance and scalability of the system, such as the choice of data storage, communication protocol, and load balancing strategy.

References

[1] JUNJIE HU, KE LIU (2020). Raft consensus mechanism and the applications.
[2] GEORGIOS BITZES (2017). Distributed consensus and fault tolerance.
[3] EMANUELE GALLONE (2020). Data-plane consensus: applying Raft consensus at network level, 29-35.
[4] JOAKIM ANTUS (2016). Reaching Consensus Using Raft
[5] https://towardsdatascience.com/raft-algorithm-explained-a7c856529f40
[6]https://codeburst.io/making-sense-of-the-raft-distributed-consensus-algorithm-part-1-3ecf90b0b361
[7] https://www.geeksforgeeks.org/raft-consensus-algorithm/