Big Data (UE21CS343AB2) - YAKt (Yet Another KRaft)

Introduction

- Yet Another KRaft (YAKRaft) is one of projects as a part of the Big Data Course (UE21CS343AB) at PES University
- This project involves the students developing the KRaft protocol from scratch, managing metadata for the Kafka system.

About KRaft

- KRaft is a event based, distributed metadata management system that was written to replace Zookeeper in the ecosystem of Kafka.
- It uses Raft as an underlying consensus algorithm to do log replication and manage consistency of state.

Project Objectives and Outcomes

Gain a deeper understanding of

- The Raft Consensus algorithm and concepts of Leader Election, Log Replication, Fault Tolerance
- ZAB Consensus algorithm
- Kraft and Zookeeper Architecture and its nuances
- Event driven Architecture
- Kafka Architecture

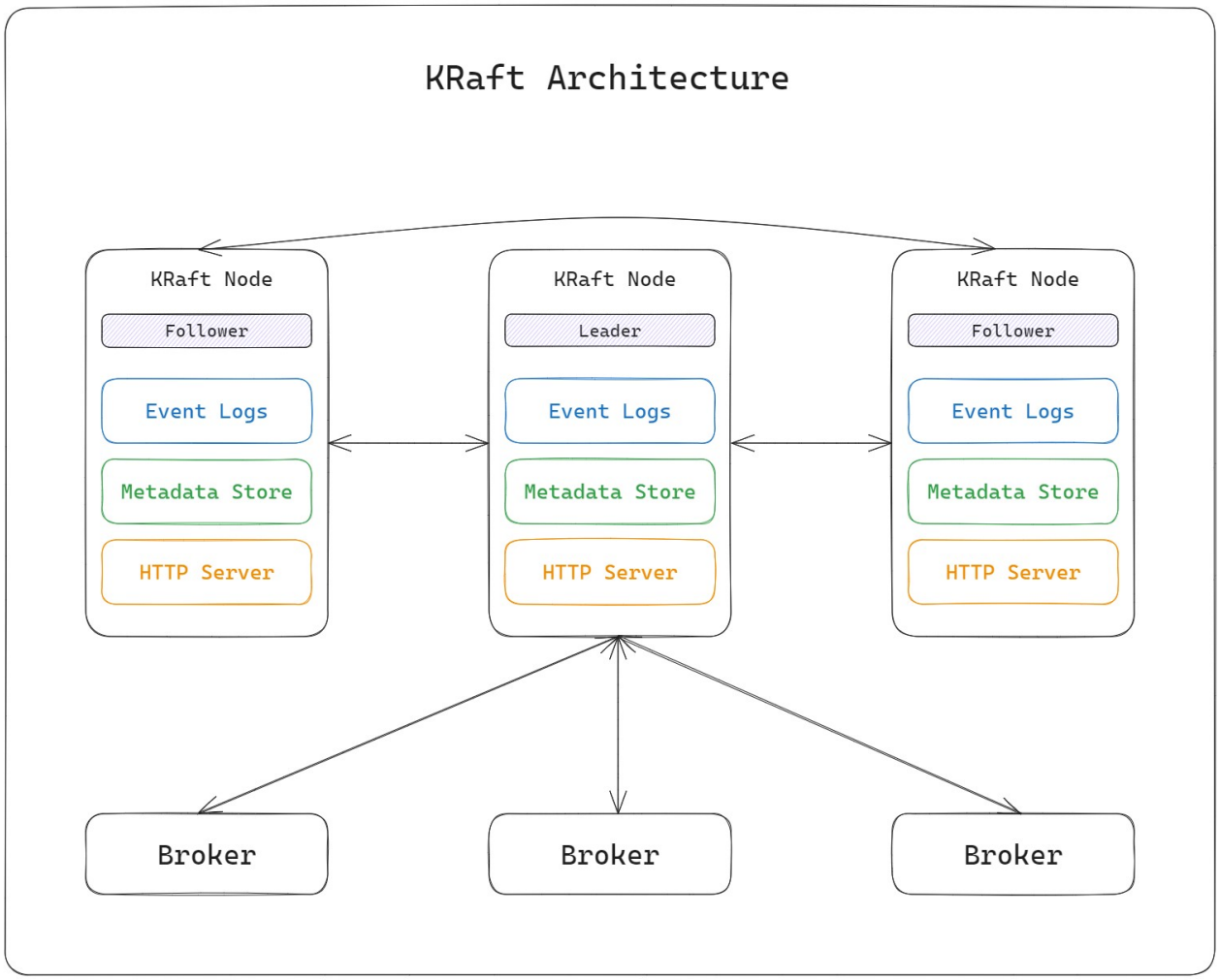
Outcomes

- A robust system that mimics the working of KRaft
- A project with a real world use-case

Technologies/Languages to be used

- You are free to use any language such as Python, Go, Java, among others.
- Ensure that the chosen language supports all the required functionality.
- You are allowed to use any external libraries or APIs if required.
- NOTE : You cannot demo KRaft (Original) as your project

Project Specification



Raft Node

Must handle the following (going above and beyond these are upto the developer)

- Leader election : selecting the leader for the KRaft cluster
- Event driven architecture
- Eventual Consistency
- Failover management : must be able to provide standard raft failover guarantees (3 node cluster can handle single failure; 5 node cluster can handle 2 failures;)
- Maintaining event log : event log of all changes being made (this can be used to reconstruct the metadata store)
- Snapshotting :
  - Creation and retrieval of the snapshots
  - Take periodic snapshots of the event log at the leader to be able to provide a level of fault tolerance

NOTE : Most of the above are inherently supported by raft as a consensus algorithm;

Metadata Storage

Records Types

For the scope of this project add some limitations/modifications to the original records (as seen in reference),

- Record types
- Record structure

Ref : KRaft Metadata Records (https://cwiki.apache.org/confluence/display/KAFKA/KIP-746%3A+Revise+KRaft+Metadata+Records)

1. RegisterBrokerRecord

```
{
        "type": "metadata",
        "name": "RegisterBrokerRecord",
        "fields": {
                "internalUUID": "", // type: string; set by the server
                "brokerId": 0,  // type: int; given by client, through config
                "brokerHost": "", // type: string; given by client, through config
                "brokerPort": "", // type: string; given by client, through config
                "securityProtocol": "", // type: string; given by client, through config
                "brokerStatus": "", // type: string; set by server, can be set to defaul
                "rackId": "", // type: string; given by client, through config
                "epoch": 0 // type: int; epoch number given by the quorum controller; ir
        },
    "timestamp": ""
}
```

2. TopicRecord

```
{
        "type": "metadata",
        "name": "TopicRecord",
        "fields": {
                "topicUUID": "",  // type: string; must be set by server;
                "name": "" // type: string; given by client, through config
        },
        "timestamp": "" // type: timestamp
}
```

3. PartitionRecord

```
{
        "type": "metadata",
        "name": "PartitionRecord",
        "fields": {
                "partitionId": 0, // type: int; given by client
                "topicUUID": "",  // type: string; given by client
                "replicas": [], // type: []int; list of broker IDs with replicas; given
                "ISR": [], // type: []int; list of insync broker ids; given by client, t
                "removingReplicas": [], // type: []int; list of replicas in process of r
                "addingReplicas": [], // type: []int; list of replicas in the process of
                "leader": "", // type: string; uuid of broker who is leader for partitic
                "partitionEpoch": 0 // type: int; number that incrementatlly changes wit
        },
        "timestamp": "" // type: timestamp
}
```

4. ProducerIdsRecord

```
{
        "type": "metadata",
        "name": "ProducerIdsRecord",
        "fields": {
                "brokerId": "", // type : string/int; uuid of requesting broker; given b
                "brokerEpoch": 0, // type : int; the epoch at which broker requested; se
                "producerId": 0 // type : int; producer id requested; given by client
        },
        "timestamp": "" // type: timestamp
}
```

5. BrokerRegistrationChangeBrokerRecord

```
{
        "type": "metadata",
        "name": "RegistrationChangeBrokerRecord",
        "fields": {
                "brokerId": "", // type: string; given by client
                "brokerHost": "", // type: string; given by client
                "brokerPort": "", // type: string; given by client
                "securityProtocol": "", // type: string; given by client
                "brokerStatus": "", // type: string; given by client
                "epoch": 0 // type: int; set to broker epoch after update
        },
        "timestamp": "" // type: timestamp
}
```

Broker Statuses :

- INIT
- FENCED
- UNFENCED
- ALIVE

- CLOSED

Storage Data Structure

- Implementation of how these records are stored is upto the developers; as long as it can do the below functions

Example/Suggested (key-value store) :

```
{
        "RegisterBrokerRecords": {
                "records": [], // type : []RegisterBrokerRecord
                "timestamp": "", // type : timestamp; updated time
                ... // feel free to add any auxillary fields to help with the functional
        },
        ...
}
```

HTTP Server

CRD API

- Must include Create, Read, Delete APIs for the Record Type mentioned above
- Create : creation of records, returning unique ID of the entity if available
- Read : return single/array of records
- Delete : deletion; returns unique ID of entity (only for brokers); update status as required;

Broker Management API

Requests made by the Brokers

API Spec

RegisterBroker API (POST Request)

- The first time the broker comes online, it registers itself with KRaft
- Add required MetadataRecords Accordingly

RegisterBrokerChange API (POST Request)

- Any changes to be made to the Broker info is sent to this endpoint

MetadataFetch API (POST Request)

- Acts as the heartbeat mechanism for a broker

- Broker sends the last offset of metadata that the broker is aware of (Last offset can be considered as timestamp)
- Server must reply back with
    - All changes to the metadata since last offset
    - If offset is too far behind latest, send the entire snapshot (most recent snapshot) of metadata

Client Management API

Requests made by the Consumers and Producers

API Spec

RegisterProducer API (POST Request)

- Register the Producer

MetadataFetch API (GET Request)

- Fetch Topics, Partition and Broker Info Records

Endpoints

These are the endpoints that are to be made. Going above and beyond these is upto the developer

- RegisterBrokerRecord
    - register
    - get all active brokers
    - get broker by id
- TopicRecord
    - create topic
    - get topic by name
- PartitionRecord
    - create partition
- ProducerIdsRecord
    - register a producer from a broker
- BrokerRegistrationChangeBrokerRecord
    - updates to broker; increment the epoch on changes;
    - unregister a broker
- BrokerMgmt
    - a route takes previous offset/timestamp and returns metadata updates since then / if later than 10 minutes send entire snapshot; send diff of all metadata that has been updated
- ClientMgmt
    - a route takes previous offset/timestamp and returns metadata updates since then / if

later than 10 minutes send entire snapshot; send only topics, partitions and broker info

Tips

1. To demo multiple nodes the following approaches can be done (not an exhaustive list)

- Using multiple processes on different computers
- Using multiple processes on same computer
- Spawn multiple nodes on different VMs

2. Can use an implementation of raft in the language of choice to back your KRaft implementation

Weekly Guidelines

This is merely an ideal/suggested timeline for the project.

Week 1

- Read the raft paper and understand the nuances
- Read about KRaft and Zookeeper
- Pick a language
- Setting up project
- Setup a simple HTTP Server
- Figure out the technologies/libraries/frameworks to be used

Week 2

- Setup raft node
- Chalk out the endpoints required

Week 3

- Setup the HTTP endpoints as per requirements
- Test

References

KIP-500 Replace Zookeeper with KRaft (https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum)

Working of Zookeeper (https://zookeeper.apache.org/doc/r3.5.4-beta/zookeeperOver.html)

KRaft Metadata Records (https://cwiki.apache.org/confluence/display/KAFKA/KIP-746%3A+Revise+KRaft+Metadata+Records#KIP746:ReviseKRaftMetadataRecords-BrokerRegistrationChangeRecord)

Confluent : Intro to KRaft (Suggested Read) (https://www.slideshare.net/HostedbyConfluent/introducing-kraft-kafka-without-zookeeper-with-colin-mccabe-current-2022)

Hussein Nasser : How does Kafka work? (https://www.youtube.com/watch?v=LN_HcJVbySw)

What is Zookeeper and how is it working with Apache Kafka? (https://www.youtube.com/watch?v=t0FDmj4kaIg)

Martin Kleppman : Consensus (https://www.youtube.com/watch?v=rN6ma561tak&pp=ygUNemFiIGNvbnNlbnN1cw%3D%3D)

Zookeeper Explained (https://www.youtube.com/watch?v=gZj16chk0Ss)

Raft Visualization (https://thesecretlivesofdata.com/raft/)

Raft Paper & More (https://raft.github.io/)

Core Dump : Understanding Raft (https://www.youtube.com/watch?v=lujMVjKvWP4&pp=ygUNemFiIGNvbnNlbnN1cw%3D%3D)