Kafkaesque

CSCI-6421 Distributed Systems

Submitted to: Prof. Timothy Wood, Prof. Roozbeh Haghnazar

Team Members:

Aarohi Agarwal
Biyas Basak
Chinmay Patil
Dilip Varma Sagi

Problem statement

- This project emulates a distributed message queuing system for collecting and distributing real-time log streaming data from multiple sources.
- For this, a publish-subscribe based model is implemented, where producers and consumers communicate by sending-receiving of messages on a topic channel from a message queue.
- In order to understand how the logs are distributed in a publish-subscribe based distributed system, existing systems like Apache Kafka and RabbitMQ were referenced.
- One major bottleneck of kafka is its dependency on Apache Zookeeper cluster to keep track of kafka nodes, topics and partitions.
- We implemented raft algorithm at both broker and partition level that gives us similar benefits without the drawback of zookeeper.

Related Work

- The existing systems referred by this implementation are: -

1. **Apache Kafka** - It is a publish-subscribe system written in Scala, that uses Zookeeper to handle scalability of messages and fault-tolerance issues between different servers.
2. **Apache Pulsar** - Pulsar is a horizontally scalable distributed system that uses bookeeper and Zookeeper to handle bandwidth of messages between producers and consumers.
3. **NATS** - This distributed queuing system implemented in Go, uses streaming service mechanism, providing an API to communicate between NATS server and clients.

Distributed System challenges addressed

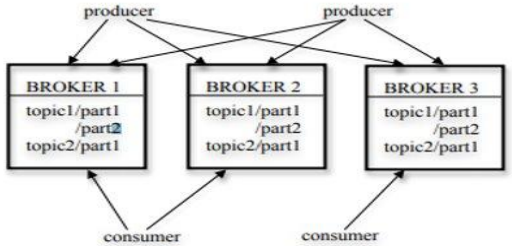
- **Openness**
- **Scalability**
- **Fault Tolerance**
- **Transparency**
- **Concurrency**

Distributed System challenges addressed

- **Openness**
  - Inspired by NATS client protocol, we designed our own protocol over TCP that supports operations which clients can use to communicate with the Kafkaesque server.
  - This allows clients to create libraries in any language of their choice and implement higher level optimizations like batching.
  - Some of the operations currently supported are:
    CRTP (create topic)
    PUBS (publish to a topic)
    SUBS (subscribe to a topic)
    CONS(consume from a topic)
  - The readme file has described all the operations in much greater detail with examples.

Distributed System challenges addressed

- **Scalability**
  - Our system offers horizontal scalability like Kafka where topics are further divided into partitions which are replicated over multiple brokers.
  - Each broker will have a few partitions that are leaders to handle read and write from clients because of raft leader election implementation at partition level, and our system offers load balancing requests between the leader partitions.
  - The brokers can be horizontally scaled as leaders are distributed across multiple brokers.



Kafka: a Distributed Messaging System for Log Processing (Kreps, Narkhede and Rao, 2011)

Distributed System challenges addressed

- **Fault Tolerance**
  - This implementation uses Raft Consensus algorithm to do leader election to avoid single-point of failures and replicates data across multiple brokers , which is backup into the local file system using file manager which makes the system less prone to data loss.
- **Transparency**
  - In this project, transparency is achieved by hiding the internal mechanism of how messages are sent to different partitions and how they are received by the consumers.
- **Concurrency**
  - Each client is handled in a separate goroutine and whenever they have to access any shared resource we use mutex to synchronize.
  - Go channels are used to communicate between Raft Server and brokers.

Key Design choices

- Our system gives the guarantee of message ordering in a single partition. We achieved this by two key design choices.
    - Giving clients a choice of assigning a key to their messages so ordered messages are always sent to the same partition. That way events are always consumed in the same order. In absence of key, the message is added to the partition in a round robin fashion
    - Multiple consumer instances belonging to the consumer group(cluster) can never be assigned the same partition. This way we guarantee that the same consumer is going to consumer ordered messages.
- The consumers form a consumer group and provide an offset which refers to the location in file to from which messages will be received.
- The messages are stored in the local storage system using a file manager, which creates two files, an index file to store offset and log file that contains the actual messages.
- The log data is replicated across multiple brokers using Raft's state machine replication and single-point of failures for brokers addressed by Raft leader election mechanism.

Code Snippets

```
type partition struct {
    mutex          sync.Mutex
    id             int
    name           string
    indexFile      string
    logFile        string
    messages       []Message
    subscribers    []*subscriber
    offset         string
    raft           *raft.Raft
    commitChan     chan raft.Commit
    taskCompChan   chan bool
    segment        int
    activeSegment int
}
```

```
// NewPartition constructor
func newPartition(id int, topic string) *partition {
    activeSegment := 0
    indexFile := path.Join(topic+"-"+strconv.Itoa(id), "index"+strconv.Itoa(activeSegment)+".txt")
    logFile := path.Join(topic+"-"+strconv.Itoa(id), "log"+strconv.Itoa(activeSegment)+".txt")

    fm := filemanager.GetFileManager()
    // create physical partitions on disk
    var wg sync.WaitGroup
    wg.Add(2)
    go fm.AddFile(indexFile, &wg)
    go fm.AddFile(logFile, &wg)
    wg.Wait()
    // partition instance
    p := partition{
        id:            id,
        name:          topic,
        indexFile:     indexFile,
        logFile:       logFile,
        subscribers:   []*subscriber{},
        messages:      []Message{},
        offset:        "",
        activeSegment: activeSegment,
    }

    startElection := make(chan bool)

    serviceName := "partition:" + strconv.Itoa(id)

    // get the raft rpc Server
    raftServer := raft.GetServerInstance()

    // channel where all the commits are going to be available
    commitChan := make(chan raft.Commit)
    taskCompChan := make(chan bool)

    raftObj := raft.NewRaft(strconv.Itoa(id), serviceName, raftServer, commitChan, startElection)

    // register the partition service on rpc
    raft.RegisterService(serviceName, raftObj)

    p.raft = raftObj
    p.commitChan = commitChan
    p.taskCompChan = taskCompChan
    go p.listenForCommits()

    startElection <- true
```

Demo

Impact of design

- We did not test our system with any metric so we don't have any hard numbers.
- But in testing, our system works well in responding to users while maintaining its guarantees
- One big downside of using traditional raft algorithm at partition granularity is the chattiness that comes with it. Each partition is now communicating with all its replicas
- Future upgrades can be applied to overcome this downside. CockroachDB has successfully applied its own raft algorithm which they call MultiRaft which overcomes this issue
- The controller(leader) broker currently only supports one administrative task that is creating partitions and replicating it. This can be extended to removing partitions, persisting subscriptions and various partition configurations to disk etc.
- Adding Serf or a similar library that can be used to dynamically discover broker nodes

Concluding statements:

- In this project, we successfully implemented a distributed messaging system which emulates a publish-subscribe model
- The key difference and selling point of current implementation as compared to existing systems is that it implements raft algorithm at both broker and partition level which gives enhanced scalability and fault-tolerance.
- The messages are published to a partition by hashing a key, which can be provided either by client at runtime or using a round-robin algorithm.
- Finally, a file manager system ensures that backup of all the log messages is replicated across the topics in all the brokers to handle node failures in the system.