Project Kafkaesque

A publish-subscribe based distributed communication system

Aarohi Agarwal

Computer Science

The George Washington University

Washington, DC, 20052

aagarwal14@gwu.edu

Biyas Basak

Computer Science

The George Washington University

Washington, DC, 20052

biyas@gwu.edu

Chinmay Patil

Computer Science

The George Washington University

Washington, DC, 20052

patchinmay@gwu.edu

Dilip Varma Sagi

Computer Science

The George Washington University

Washington, DC, 20052

dilipvarma7@gwu.edu

Introduction

We are in the 21st century and almost everything is available on our fingertips. Numerous amounts of software and different applications are running on our hand and generating a lot of data. With the usefulness and ease of availability of these applications and software, there is a great architecture responsible for smooth functioning. These modern days applications or software are usually built on the principles of distributed systems which makes them to function smoothly. One of the important features of any system is to data flow and communication between one component to another. There are various ways in which data can be moved in distributed systems such as using ETL, messaging systems, etc. Along with the communication and data flow the system also needs to perform flawlessly. Fault tolerance is one of the major concerns while building any distributed system (Aung, T., Min, H. Y., & Maw, A. H, 2019). Along with the fault-tolerance a good distributed system should also be easily scalable and can be easily decoupled.

Considering these points, in this project we will be implementing a messaging system similar to already existing systems such as Apache Kafka and RabbitMQ. According to Krep, J. and et al., Apache kafka is a messaging-based log aggregator where a producer publishes the data to topics and a subscriber subscribes to the topics and gets the data from those topics (Kreps, J., Narkhede, N., & Rao, J., 2011). Basically, it uses a publish subscribe messaging model to send the messages which significantly reduces the latency which is otherwise caused in systems which are using RPC's.

Challenges

To design and implement any distributed system it should be fault-tolerant, easily scalable, transparent, should also be decoupled easily, concurrent, etc. In this project we will be mainly focusing on challenges like how the system can be fault tolerant, how we can scale easily, good throughput, and how we can implement the decoupling of services in this system.

- **Decoupling of services**

Decoupling the publishers and subscribers is arguably the most fundamental functionality of a pub/sub system (Dobbelaere, P., & Sheykh Esmaili, K. 2019). The pub-sub messaging framework provides a broker that both producer and consumer services can connect to without knowing the actual address of the destination service. The consumers or subscribers subscribe to something called a topic in case of kafka, and are notified each time data is available for processing in the queue. Whenever the consumers are free, they read the message from the queue, read it and then remove it from the queue. Multiple consumers can read the same topic at their own pace. This leads to decoupling of services - such that the system becomes more robust and failure of one of the nodes in any of the consumer groups will not affect the performance of other nodes processing the data.

- **Fault Tolerance and High Availability**

In distributed message queuing systems like Kafka, topics are divided into partitions which can be written by multiple producers at the same time. The subscriber clients form consumer groups to read messages from the partitions. Within a consumer group, each consumer reads messages which form the subset of partitions. These partitions are in turn replicated to multiple brokers (message queues) to ensure fault tolerance. The replication factor indicates how many copies of partition are maintained for each topic. Further, the message ordering is handled within replicated queues using leader-follower pattern. Each partition has a single leader to handle read and write requests and a configurable number of followers to replicate writes to leaders in the background. To avoid a single point of failure, we will be implementing a resource manager instead of a single master node to initiate a new leader election in case of broker hosting the leader fails. This information is then propagated to all the clients subscribed to a topic in that partition.

- **Scalability**

The scalability goal necessitates that the user message load is distributed across servers and represents a departure from previous metadata-private systems (Tyagi, N., Gilad, Y., Leung, D., Zaharia, M., & Zeldovich, N., 2017). The messages are stored in topics which in turn are divided into partitions. As partitions form independent units within a topic, these allow brokers to scale out horizontally and manage load within the cluster. The Cluster capacity can be increased by adding new message queues and partitions. This idea is from Kafka which implements topics and partitions as a way of managing and scaling messages delivery.

Approaches

There are two models by which we can build a dynamic distributed messaging system. Those are:

1. Publish/Subscribe Model.

2. Messaging Queue Model.

1. Publish/Subscribe Model:

It is a mechanism where the publisher posts the data/streams of data to the server. The data is grouped into various categories/classes. The subscriber/consumer subscribes to various classes and thus retrieves the data. The broker makes sure that the subscriber subscribed to a particular class/category of data will receive data at any time. Popular distributed messaging systems which rely on the public/subscribe paradigm are Apache Kafka, Apache pulsar etc. The core features of publish/subscribe paradigm are:

1. Time Decoupling.
2. Space Decoupling.
3. Synchronization Decoupling.

Apache Kafka:

This design was introduced by LinkedIn. It is based on the publish/subscribe model written in Scala language. It consists of consumers, brokers, topics, clusters and producers. The producers can publish messages to a topic. A topic is a stream of messages categorized by a particular type. Next, the topic is posted to one or more servers often called as brokers. This kind of model is called a push-based model where the streams of a message are pushed on one of the available brokers. A consumer can retrieve a stream of messages from the broker where the message is stored. This kind of model is called a pull-based model where the message is retrieved from the broker. The broker does not have any state i.e. the broker does not keep track of how many consumers consumed the information, which information is retrieved etc. Apache kafka is distributed in nature and therefore it has multiple brokers which can store the data/stream of messages. Zookeeper maintains all the services that are required to maintain the communication between the producer and consumer(Aung, T., Min, H. Y., & Maw, A. H. ,2018). The whole event can be done either synchronously or asynchronously.

Below is the sample algorithm for producer and consumer as per the authors (Kreps, J., Narkhede, N., & Rao, J., 2011).

Sample producer code:

```
producer = new Producer(...);
message = new Message("test message str".getBytes());
set = new MessageSet(message);
producer.send("topic1", set);
```

Sample consumer code:

```
streams[] = Consumer.createMessageStreams("topic1", 1)
for (message : streams[0])
{
bytes = message.payload();
 // do something with the bytes
}
```

Apache pulsar:

Apache pulsar is one of the popular distributed messaging systems. It was developed by yahoo in 2014. Pulsar is horizontally scalable. A pulsar cluster handles the bandwidth of messages between producer and consumer. Pulsar uses zookeeper and in addition to that it also uses bookkeeper (Intorruk, S., & Numnonda, T. 2019). Pulsar provides end-end latency.

2. Messaging Queue Model:

It is majorly used for inter-process communication. In this kind of architecture, the publisher delivers the message to a message queue. If a consumer wants to consume the message, the consumer connects to the message queue and retrieves the message. The messages are kept live on the message queue until a consumer consumes the message. Each Message is processed only once. The entire event is done in an asynchronous way i.e. producer and consumer are not required to interact simultaneously with the message queue. Messaging systems like RabbitMQ, IBM MQ are based on the messaging queue paradigm.

RabbitMQ:

RabbitMQ is an open-sourced framework that implements Advanced Messaging Queuing Protocol(AMQP). It enables asynchronous message based communication, by enabling loose coupling between server and clients in a cluster, i.e. they need not run at same time. Additionally, this framework can be implemented in any language because of its language agonistic nature. In RabbitMQ, the messages are transported over TCP connections (Sharvari, T., & Sowmya Nag K., 2019). It consists of a Publisher, Consumers, Exchange and Queues. Every message contains a payload containing the message and a routine key which determines the queue in which the message will be placed.

NATS streaming model:

Additionally, there are some frameworks such as NATS Streaming which support both the publish/subscribe and message queuing systems. This is a lightweight, open-source, cloud-based messaging service implemented in Golang and is maintained by Synadia Group. In the publish/subscribe based model, publishers send the message to the NAT subjects, which are subscribed by the consumers. The consumers listen for any data published to these NAT subjects in order to receive messages. Scalability and fault tolerance in case of node failures are handled by master server known as "**gnastsd server**" by cutting off subscriptions that extend a certain timeout value.

The NATS server in turn is embedded within NATS streaming service, which provides an API to communicate with NATS server. The go channels act as subjects in this case, where producers publish messages and subscribers receive messages. This service guarantees at least once delivery of messages using google protocol buffers as well as maintains persistence of messages using durable subscriptions, where in case of network failures, the server starts again with the earliest message unsubscribed by the consumer.

Project Proposal:

This project will emulate a publish subscribe communication model for distributed systems such as Apache Kafka and RabbitMQ for collecting and distributing real-time data generated from multiple sources. As a part of implementation, a publisher-subscriber model for distributing messages will be created in which multiple consumers (VM's, servers, etc.) subscribe to a message queue which consists of data generated by the producer machines. This project also explores as to how to partition the messages across different segments and data replication to achieve fault tolerance.

The producer and subscribers will communicate to each other by subscribing to topics which in turn are divided into partitions. The clients will form a consumer group within partition and will subscribe to topics related to that consumer group. The main focus of the project will be to ensure that failure of nodes within one of consumer groups, will not prevent other clients across other groups, or within the same group in different partitions from receiving the messages from the message queues. Also, in order to maintain high availability of the system, mechanism to recover from master failure using leader election has to be handled. This system guarantees at least once delivery of messages to the clients.

References:

Kreps, J., Narkhede, N., & Rao, J. (2011, June). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (Vol. 11, pp. 1-7).

Tyagi, N., Gilad, Y., Leung, D., Zaharia, M., & Zeldovich, N. (2017, October). Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 423-440).

Intorruk, S., & Numnonda, T. (2019, July). A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data. In *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (pp. 102-107). IEEE.

Sharvari, T., & Sowmya Nag K. (2019). A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming. *CoRR*.

Aung, T., Min, H. Y., & Maw, A. H. (2018, October). Performance Evaluation for Real-Time Messaging System in Big Data Pipeline Architecture. In *2018 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* (pp. 198-1986). IEEE.

Aung, T., Min, H. Y., & Maw, A. H. (2019, November). Coordinate Checkpoint Mechanism on Real-Time Messaging System in Kafka Pipeline Architecture. In *2019 International Conference on Advanced Information Technologies (ICAIT)* (pp. 37-42). IEEE.

Dobbelaere, P., & Sheykh Esmaili, K. (2019). Industry Paper: Kafka versus RabbitMQ. *A comparative study of two industry reference publish/subscribe implementations*.