International Research Journal of Education and Technology
Peer Reviewed Journal
ISSN 2581-7795

# SCALABLE PROCESSING USING KAFKA ALONG WITH AKKA ACTOR

[1]Balasubramaniam P
*Computer Science And Engineering*
*Bannari Amman Institute Of Technology*
Erode, Tamil Nadu
balasubramaniam.cs19@bitsathy.ac.in

[2]Kiruthikashree K
*Computer Science And Engineering*
*Bannari Amman Institute Of Technology*
Erode, Tamil Nadu
kiruthikashree.cs19@bitsathy.ac.in

[3]Thamaraikannan M
*Computer Science And Engineering*
*Bannari Amman Institute Of Technology*
Erode, Tamil Nadu
thamaraikannan.cs19@bitsathy.ac.in

***Abstract*** **- An organisation manages its relationships with consumers using a system called customer relationship management, which often involves studying a lot of data through data analysis. For that, Changes done in one module of a system has to be updated in all other modules and has to be processed by many other custom functions. Here we will first store the changes made in any module of the system as a message in kafka Brokers and then fetch and process using consumers. But in consumers, each message is processed by single thread which is very much time consuming. In order to overcome this problem, we will be using Akka Actor, which is actually a multithreading environment.Akka is a open-source framework. Applications that are concurrent, distributed, and fault-tolerant are developed using it. An actor is a thing that exchanges messages with other actors. Actors each have unique states and behaviours. Using Actor, we can process a single message using multiple threads so that we can process messages faster. We have made some changes in available framework and made it customizable to our need.**

***Keywords- Apache kafka, Akka Actor, Scalability.***

## I. INTRODUCTION

At any significant internet company, a lot of data is generated. This information typically consists of two types of information: (1) operational metrics such as service call stack, call latency, and errors; and (2) user activity events such as logins, page views, clicks, "likes," sharing, comments, and search queries on each computer. Log data has long been a part of analytics used to monitor indicators like user engagement and system utilisation. Activity data, however, is now a component of the production data pipeline used directly in site features thanks to recent developments in internet applications. Among them are: (1) search relevancy; (2) recommendations that may be influenced by item popularity or co-occurrence in the activity stream; (3) ad targeting; and (4) other uses.

Because the volume of this creation and real-time utilisation of log data is orders of magnitude greater than the volume of "actual" data, it presents new difficulties for data systems. For instance, computers is frequently used for search, suggestions, and advertising. Granular click-through rates, which produce log entries for dozens of items on each page that are not clicked in addition to every user click. China Mobile collects between 5 and 8 terabytes (TB) per day of call records, and Facebook collects about 6 TB per day of various user activity events.

A distributed, divided, replicated commit log service is Kafka. With a distinctive look, it offers messaging system features. Even with TBs of communications saved, it could handle hundreds of thousands of messages every second. Kafka performs consistently even on very basic hardware by appending messages to files using on-disk structures. Akka Actor is integrated with kafka for faster processing of messages. The actor model provides a high-level abstraction for writing scalable and concurrent systems, making it easier to process things faster. If we integrates Akka Actor with kafka, we can process messages more efficiently and quickly.

## II. LITERATURE SURVEY

It is a centralised service that handles group services, distributed synchronisation, configuration information maintenance, and naming. Akka Actors was inspired by the actor model, a mathematical theory of concurrency that was first introduced in the 1970s. The actor model provides a powerful and flexible way to model and implement concurrent systems, and Akka Actors builds on this foundation to provide a practical, scalable, and production-ready implementation.

Scalability of the actor model has been a subject of research and the Akka Actors framework has been noted as one of the implementations of the actor model that offers scalability.Actors in Akka communicate by sending messages to each other, which eliminates the need for locks and other synchronization mechanisms that can cause scalability bottlenecks. Akka Actors provides an implementation of the actor model, which has been widely adopted in industry and academia for building concurrent, scalable, and fault-tolerant systems.

ZooKeeper is simple**.** It allows distributed processes to coordinate with each other through a shared hierarchal namespace which is organized similarly to a standard file system. The name space consists of data registers - called znodes, in it parlance - and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means it can achieve high throughput and low latency numbers.

ZooKeeper is ordered. It stamps each update with a number that reflects the order of all ZooKeeper transactions. Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives.

ZooKeeper is fast. It is especially fast in "read-dominant" workloads. Its applications run on thousands of machines, and it performs best where reads are more common than writes. Apache ZooKeeper is a project under the Apache Software Foundation which is open source.

The results of the ZooKeeper's development team at Yahoo! Research indicate that Zookeeper can perform very well as

long as reads > writes. This is because writes involve synchronizing the state of all servers (Reads outnumbering writes is typically the case for a coordination service).

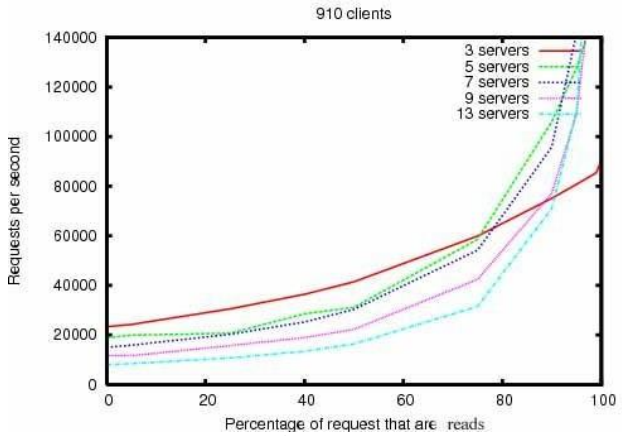


Figure 1 Throughput of ZooKeeper by Yahoo Research

Apache ZooKeeper is an integral part of Apache Kafka. Zookeeper is a cluster management tool. It is used by Kafka, as Kafka is a distributed application and it needs ZooKeeper to manage the cluster on which Kafka runs on.

In addition, actors can be distributed across multiple nodes, enabling applications to scale horizontally by adding more nodes to the system.

There have been several studies that have compared the performance of Akka Actors to other frameworks and technologies, and the results have generally shown that Akka Actors provides good performance and low latency.

Proposed Approach

### 3.1 Introduction to Kafka:

A distributed, divided, replicated commit log service is Kafka. With a distinctive look, it offers messaging system features.

- Topic-based message feeds are maintained by Kafka.
- Producers are processes that post messages to Kafka topics.
- Consumers are programmes that process the published message feed and subscribe to topics.
- Kafka is operated as a group of servers, referred as brokers.

As seen in figure 2, producers transmit data to Kafka cluster through the server groups, which then distributes them to consumers.

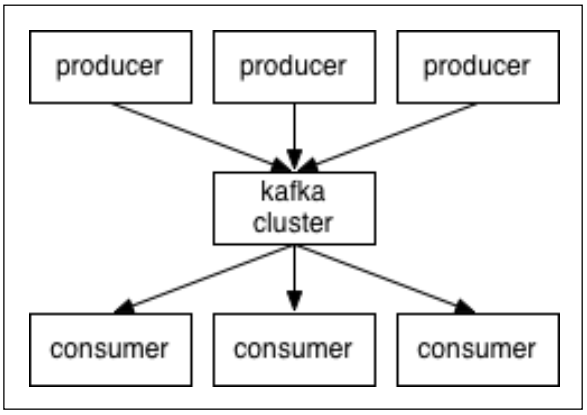


Figure 2. Basic Kafka Design

### 3.2 Topics and Logs:

A topic is a name for a category or feed where messages are posted. The Kafka cluster keeps a partitioned log for each topic that looks like this:

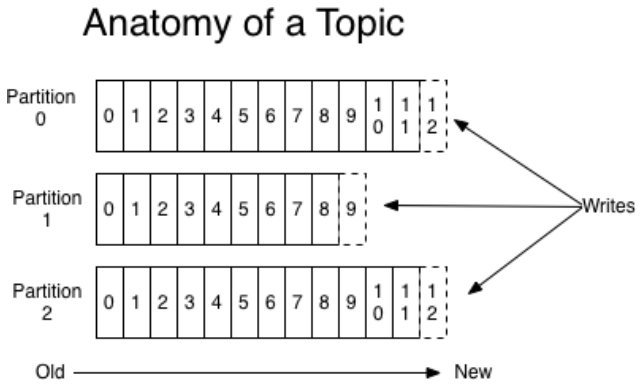


Figure 1. Anatomy of a Topic

Each partition is a commit log that is continuously expanded with an ordered, immutable sequence of messages. The offset is a sequential id number that is given to each message in a partition to identify the messages uniquely within the partition .

Whether or not they have been consumed, all published messages are kept in the Kafka cluster for a configurable amount of time. For instance, if the log retention is set to two days, then a message is available for consumption for two days after it is published before being deleted to free up space. Performance by Kafka is essentially constant.

In actuality, the only metadata that is saved for each consumer is their offset, or place in the log. Normally, a consumer advances its offset linearly as it reads messages, but in reality, the consumer controls the position and is free to consume messages in any sequence they like. For instance, a consumer can revert to a previous offset to process again.

In the beginning, they enable the log to scale beyond the limit of what can fit on a individual server. A topic may have several partitions so it may hold unlimited amount of data, but each partition must fit on the servers that host it.

### 3.3 Replication:

The Kafka servers or brokers disseminate the log's partitions among themselves. Other brokers for a partition are followers of the one leader broker for that partition.

In essence, all information travels to and from the leader. Following that, information is internally duplicated from the leader to every follower. Kafka does this in an asynchronous manner.

This is being done out of caution. If the partition's leader broker fails, one among the followers takes over as leader. This is achieved through a Kafka-used method called leader election.

### 3.4 Producers:

Data is published by producers on issues of their choosing. The producer is in charge of deciding which message is

appropriate for each topic split. To balance the load, this is achieved through round-robin style, or using a semantic partition function.

### 3.5 Consumers:

Traditional messaging strategies include publish-subscribe and queuing. Each data from a broker is sent to a specific consumer in a queue, whereas in publish-subscribe, the message is broadcast to all consumers. Kafka provides the consumer group as a single consumer abstraction that unifies both of these.

Each message published to a certain topic is delivered to one consumer instance inside each subscribing consumer group. Consumers identify themselves using a consumer group name. Consumer instances may run in different hardware or processes.

This operates exactly like a conventional queue balancing load over the consumers if all the consumer instances belong to the same consumer group.

When different consumer groups are represented in each consumer instance, this operates similarly to publish-subscribe.

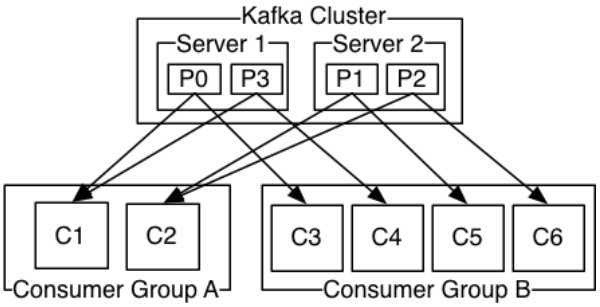


Figure 2. Example of Kafka sending messages to two consumer groups

Compared to a conventional messaging system, Kafka offers higher ordering guarantees. When many consumers use a typical queue, the server keeps messages in the queue's original order and distributes them to the consumers in that order. Messages may arrive out of order on various consumers even when the server distributes them in order. This is due to messages are delivered asynchronously to consumers. This effectively means that when there is parallel consumption, the messages' order is lost. Here parallel processing is not involved, although messaging systems frequently get around this by allowing just one process to consume from a queue under the concept of "exclusive consumer".

With the help of a pool of consumer processes, Kafka may offer load balancing and ordering guarantees. To do this, the topic's partitions are assigned to the consumer group's members in such a way that each partition is consumed by precisely one member of the group.

By doing this, we make sure that the consumer reads that partition only once and consumes the data sequentially. Since there are numerous partitions, the load is still distributed among numerous customer instances. However, the sequence of messages cannot be preserved if there are more consumer instances than partitions.

Additionally, Kafka does not offer a total order of messages across partitions in a topic, only over messages inside a single partition.

### 3.6 Introduction to Akka Actor:

Akka is an open-source toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. The Actor Model is a fundamental concept in Akka and is used to build scalable and resilient systems. An Actor is an object that encapsulates state and behavior, communicates with other Actors by exchanging messages, and can be used to model a wide range of concurrent and distributed systems.

## III. DESIGN AND ARCHITECTURE

### 4.1 Distributed Coordination:

Each producer can choose whether to publish a data to a particular partition that is chosen at random or one that is decided by a partitioning function and key based on the semantics. We'll concentrate on the interactions between brokers and consumers.

It was the first to propose the idea of group of consumers. Every group of consumers consists of one or more users who jointly consume a variety of subscribed-to subjects, a message is only sent to one user within the group. Coordination between consumer groups is not required as each group of customers receives the entire collection of subscription messages separately.

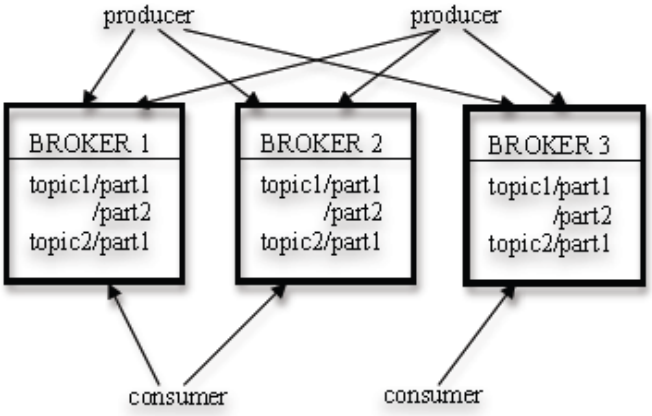


Figure 3. Kafka Architecture

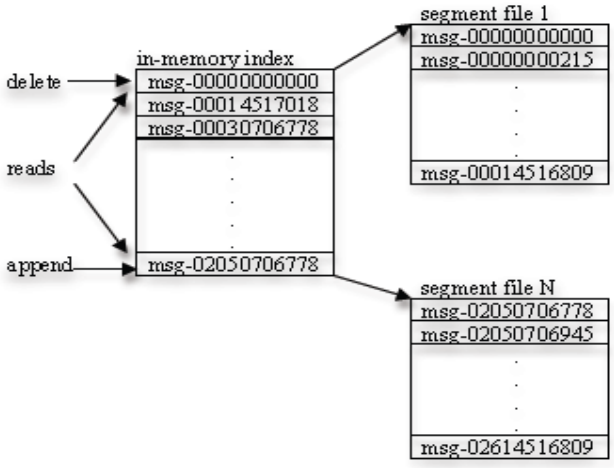


Figure 4. Kafka Log

This design has a significant side advantage. A user may purposefully consume data again by going back to an earlier offset. This goes against the queue's fundamental tenets, but many customers find it to be indispensable. For instance, the application may be able to replay specific messages after fixing an issue in the consumer's application logic.
The same consumer group may take part in various procedures or employ various devices. The objective is to equally distribute among the clients the communications kept in the brokers without placing an unnecessary coordinating strain.

### 4.2 Concurrent Processing:

An Actor has its own mailbox where incoming messages are placed and processed one at a time in the order they were received. This provides a clear separation of concerns between the Actors and helps to manage concurrency in a more controlled and scalable way. The Akka Actor Model also provides a hierarchical structure, where Actors can be organized into parent-child relationships. This allows for the creation of complex systems with more structure and reduces the need for manual coordination between Actors. In summary, Akka Actors provide a way to build scalable, concurrent, and resilient systems by modeling the objects in the system as Actors that communicate with each other through message passing.

## IV. DESIGN AND ARCHITECTURE

### 5.1 Running Kafka:

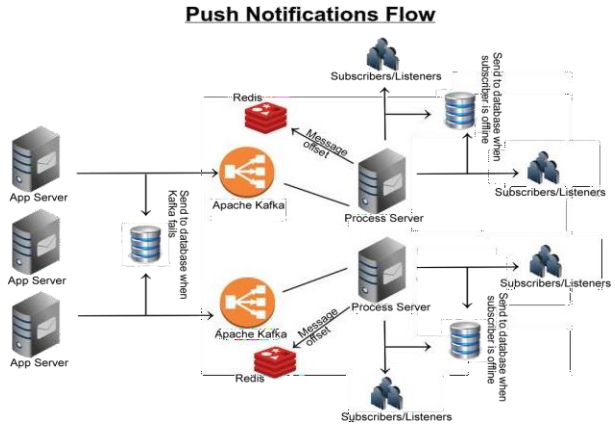


Figure 7. System Design

The above figure represents the design which I have implemented. App servers are the servers which produce data to be produced to the subscribers/listeners. Process Servers are those servers which consume data from topic and figures out to which subscriber to send to.
Redis is an open-source, networked, in-memory, key-value data store with optional durability. The name Redis means REmote DIctionary Server. It is used because of high performance for read and writes of small amount of data. When a data is consumed, the Process Server receives the message offsets, which are then stored in this location.
**Data:** The data delivered consists of headers used to specify which subscribers the data should be sent. Typically, it is a list of all the names of subscribers.

**Flow:** Both the data and headers used to identify which subscribers the data should be sent to make up the delivered data. Usually, it is a list of every subscriber's name.

### 5.2 Executing Akka Actor:

Consume messages from Kafka topic and forward it to actor for execution. In the actor's receive method, you can process the incoming messages from the topic and pass it to the appropriate actors for further processing. Create instances of the Actors for processing the consumed message and wire them together to form a processing pipeline. The Consumer Actor then pass the received messages to the appropriate actors for processing.

## V. IMPLEMENTATION DETAILS

### 6.1 System Requirements:

- Operating System – Ubuntu 12.04 64 bit
- RAM installed – 8 GB DDR4 SDRAM
- CPU – i5 9$^{th}$ Generation 3.5 GHz
- Kafka – version 2.8
- Hard Disk size – 1 TB (minimum required due to Kafka's method of saving messages on disk
- Ethernet – Gigabit Ethernet
- AKKA – version 2.4

### 6.2 Kafka Implementation:

#### 6.2.1 Producer APIs:

Two low-level producers are wrapped by the Producer API, they are:

- kafka.producer.SyncProducer
- kafka.producer.async.AsyncProducer.

#### 6.2.2 Consumer APIs:

- The "basic" low-level just one broker keeps an API connection and closely resembles the network queries made to the broker. The offset is provided in with every call to this stateless API, allowing the user to manage this metadata anyway they see fit.
- The high-level API enables consumption off the cluster of machines without taking into account the underlying topology while hiding the specifics of brokers from the client. Additionally, it preserves the condition of what has been consumed.

### 6.3 Akka Actor Implementation:

- **Create an ActorSystem:** This is the entry point to use Akka and provides a shared environment for Actors to run in.
- **Define an Actor:** In Akka, Actors are defined by creating a subclass of the AbstractActor class and implementing the createReceive method. The createReceive method returns a PartialFunction that defines the behavior of the Actor.
- **Create an Actor instance:** To create an instance of an Actor, you need to use the ActorRef reference obtained from the ActorSystem.
- **Send messages to the Actor:** To send messages to an Actor, you use the tell method on the ActorRef reference.

## VI. CONCLUSION

In-depth research has been done on Apache Kafka and Akka Actor, and the system design has been successfully deployed. Kafka has a pull-based consumption approach, similar to a messaging system, which enables an application to consume data at its own tempo and rewind the consumption as necessary. Kafka outperforms traditional messaging systems in terms of throughput by concentrating on log processing applications. It can grow out and has integrated distributed support. Additionally, Akka Actors have been shown to perform well in comparison to other similar technologies, and can handle high amounts of incoming messages.

## VII. LIMITATIONS & FUTURE SCOPE

Developing a queuing system that can manage and process data from modules with high data rate.

- To implement Kafka as a system used with Hadoop rather than just to transfer messages as it is now.
- To integrate Akka Actor for all the kafka consumers which takes large time for execution of consumed messages.

## VIII. REFERENCES

[1] Kafka: A Distributed Messaging System for Log Processing" by Jay Kreps, Neha Narkhede, and Jun Rao (2010).

[2] "Design and Deployment of Large-Scale Messaging Systems using Apache Kafka" by Gwen Shapnick, Kurt Greaves, and Jun Rao (2015).

[3] "Kafka at LinkedIn: Scale and Performance" by Samarth Jain, Jay Kreps, and Neha Narkhede (2012).

[4] "The Log: What every software engineer should know about real-time data's unifying abstraction" by Jay Kreps (2014).

[5] "An Evaluation of Akka as a Distributed System Framework" by Arne Roennqvist and Thomas Arts (2015) .

[6] "Building Scalable and Resilient Applications with Akka Actors" by Roland Kuhn (2015)

[7] "A Study of Scalability and Resilience Patterns for Akka Actors" by Bilgin Ibryam and Davide D. Longo (2016).

[8] "Akka in the Large: Building Scalable and Resilient Applications with Actors" by Konrad Malawski and Igor Wojda (2016)

[9] "A Practical Guide to Developing Reactive Applications with Akka Actors" by Viktor Klang (2017)

[10] "Using Akka Actors for Scalable and Resilient Microservices" by Adib Saikali and Pascal Vijayakumaran (2018)

[11] "Akka Actors in Action: Building Scalable, Resilient, and Efficient Applications" by Hector Veiga Ortiz (2020).