M1 Informatics Project

Securing an event streaming platform for IoT - Apache Kafka

Faculty of Science and Technology
University of Limoges

presented by

Nguyen Van Tien
Nguyen Thi Mai Phuong
Doan Thi Van Thao

Master 1 CRYPTIST

Supervisors:

Prof. Emmanuel Conchon
Dr. Mathieu Klingler

January 11, 2021

Contents

List of Figures

Introduction

In modern cloud architecture, applications are decoupled into independent building blocks, where each block is flexible, robust, composable and complete. This way of architect offers a unique kind of modularization; they make big solutions easier, increase productivity, offer flexibility in choosing technologies and are great for distributed teams. However, like any architectural approach, it has its own disadvantages. The main drawback is using different languages, libraries, frameworks and data storage technologies which can lead to intimidate and paralyze for organizations. To resolve this problem, the Publish Subscribe model (or pub/sub) was born in order to broadcast communication to different parts of a system asynchronously. To broadcast a message, a component called a *producer* simply pushes a message to the topic. All components that subscribe called a *customer* to the topic will receive every message that is broadcast.

There are several projects that follow Publish Subscribe architectural design popularly used in organizations; however, Kafka drew our attention by its prominent features. Kafka was built by LinkeIn at the time when the company was struggling with a tracking-user-activity system. This was an HTTP service that front-end servers would connect to periodically and publish a batch of messages (in XML format) to the HTTP service. Unluckily, this system had many faults. The XML formatting was inconsistent, and parsing it was computationally expensive. In additionally, monitoring and user-activity tracking could not use the same back-end service. The monitoring service was too clunky and the data format was not oriented for activity tracking. To work this thing out, Jay Kreps, a principal software engineer of development team in LinkedIn set out to create a messaging system that could meet the needs of both the monitoring and tracking systems, as well as scale for the future. Kafka is a combination with Apache Avro project for message serialization, which is effective for handling both metrics and user-activity tracking at a scale of billions of messages per day. The scalability of Kafka has helped LinkedIn's usage grow in excess of one trillion messages produced (as of August 2015) and over a petabyte of data consumed daily [1].

Being driven by Kafka's interests, in the scope of the project, we will present our understandings on Apache Kafka as well as its related aspects and how to secure this platform. The first chapter is about structures, functions and securities of Kafka. The second part starts with Zookeeper, which keeps a control role of communication between Kafka servers and their clients. Last but not least, the third chapter is saved for Encrypted matching scheme. Talking about any distributed system, it is crucial to discuss security aspect. Therefore, the last part concerns the safety and information security when working with

untrusted storage servers. Besides, this chapter also analyses several reputed articles related to our subjects.

Chapter 1

Apache Kafka

Apache Kafka is a derivation of the publish / subscribe architecture, used in particular for communication between different objects connected to each other. A sender will publish data in the form of topic or message, where the "topic" is the identifier of the channel. Apache Kafka was born to solve the problem of data persistence within the messaging system to allow multiple *consumers* and *producers*.

1.1 Kafka's presentation

According to Neha Narkhede, Gwen Shapira and Todd Palino, the authors of "Kafka-The definitive Guide" [1], publish/subscribe messaging is a pattern that is characterized by the producer (or *publisher*) of a piece of data (message) not specifically being directed to a receiver. Instead, the publisher classifies the message somehow, and that consumer (or *subscriber*) subscribes to receive certain classes of messages. In the other words, publish/subscribe is a messaging structure where senders do not program the messages to be sent directly to specific receivers, but instead categorize published messages into classes without any knowledge of receivers.

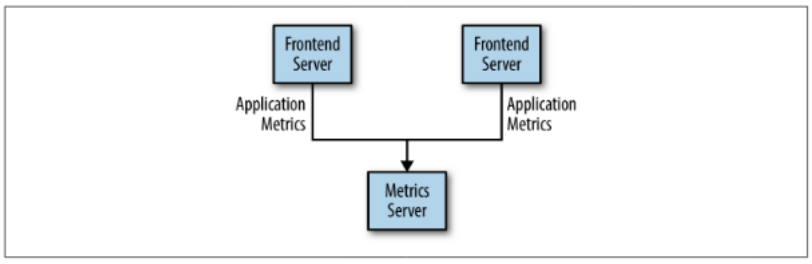

Figure 1.1: A single, direct metrics publisher.

For a simple problem of communicating between a single publisher, the easy solution works with monitoring, no need of complicated analysing or data storing. However, nowadays, with a large number of big-sized companies, the system needs to handle thousands of concurrent connections. It will receive messages from and send messages to users

all over the world. In addition, the system needs to be capable of handling high volume and global geographical spread of users. Thus, now a proposed solution given is introducing multiple processing modules, whose architecture can look much like Figure 1.2 [1]. This has the effect of splitting the system horizontally.

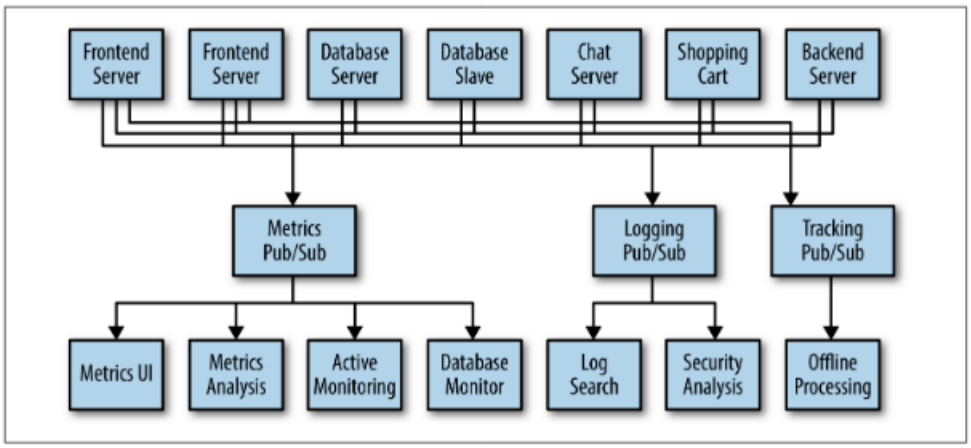

Figure 1.2: Multiple publish/subscribe systems.

Nonetheless, this increases routing complexity and connections are even harder to trace. The input modules must now route the messages to the correct processing module. Under such circumstances, the design of message passing from one module to the next begs a radical rethink. When utilizing point-to-point connections does not work, Apache Kafka is a publish/subscribe messaging system designed to solve this problem.

1.2 Kafka's structure

For a ideal publish-subscribe system, a producer knows who is his subscriber and finds the way to deliver the data to him. However, this way of approach has a risk of making the direct connections complicated and difficult to expand. Kafka's architecture inherits the benefits of this ideal system, but there are deviations to overcome its limitations. The following provides a list of definitions of the most important concepts in Kafka's structure:

- **Broker**: A broker in other words is a Kafka server which stores messages sent by producers and sent to consumers;
- **Producer**: A producer creates records and publishes them into topic's partitions;
- **Consumer**: A consumer is an external process that receives topic streams from a Kafka server or cluster;
- **Topic**: A topic can be considered as a table in database, which includes a queue of messages written by one or more producers and read by one or more consumers.
- **Record**: A record is a publish-subscribe message.
- **Partition**: A topic can be divided into many partitions, thought of as a subset of all the records for a topic.

1.2.1 Brokers

A host can run multiple Kafka servers, each of which is called a *broker*. Brokers are the bridge to help producers and consumers communicate to each other (see Figure 1.3).

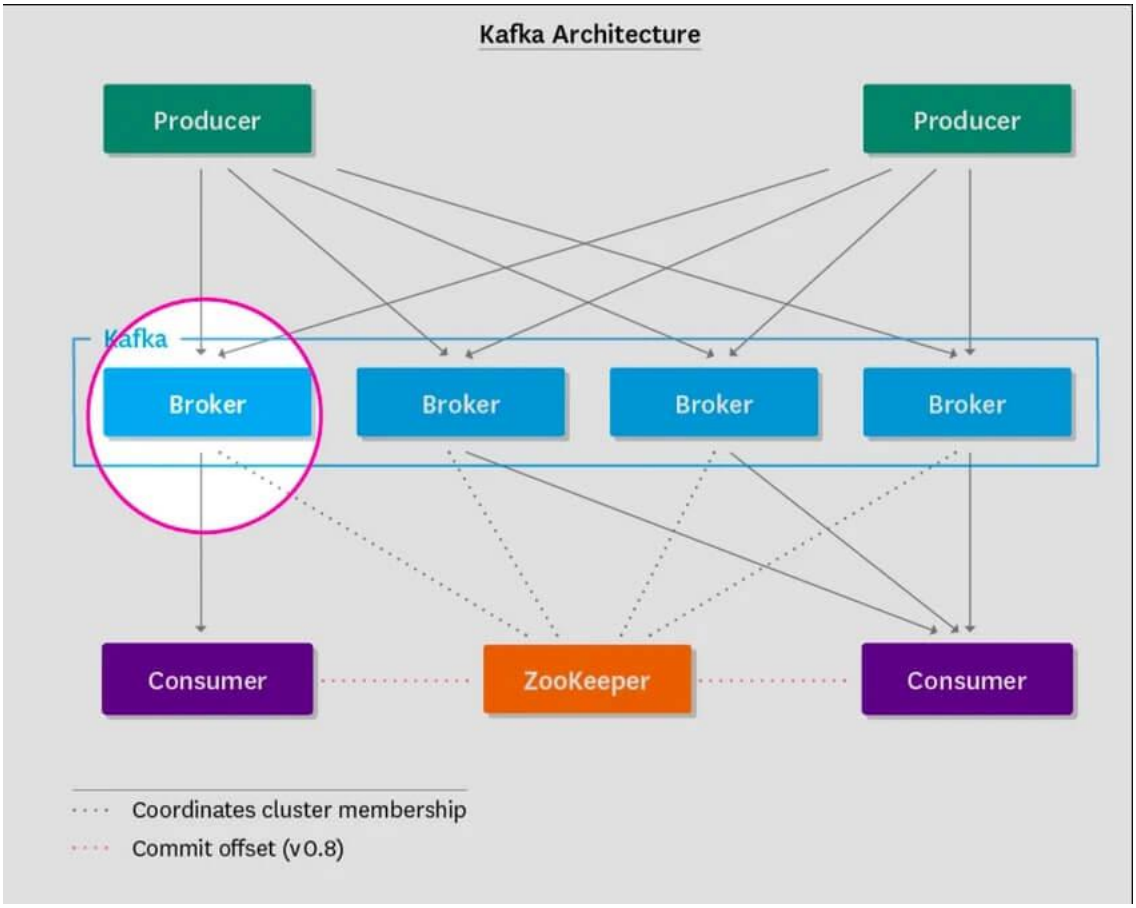

Figure 1.3: Brokers in Kafka.

Broker receives messages from producers, assigns offsets to them and stores them on disk. It also serves the comsumers, responding to partition retrieval and returning messages committed on disk. Depending on the specific hardware and its performance characteristics, a broker can process thousands of partitions and millions of messages per second. From the view of architecture, each broker is such a node. When these nodes point to the same zookeeper, they are a *cluster* of brokers. The job of brokers' management in the cluster is performed by Zookeeper.

Kafka brokers all talk to Zookeeper for distributed coordination, which also plays a key role in achieving the "Unlimited Scaling" goal from the ideal system. Furthermore, there is one broker that is responsible for coordinating the cluster. That broker is called the *controller*.

Another responsibility of Kafka brokers is to store partitions. Each broker can contain multiple partitions (see more on subsection 1.2.4). As illustrated on Figure 1.3, to reduce the complexity of management, all consumers and producers now communicably focus

on *topics* stored on brokers. The producers do not need to find ways to reach customers, but publish messages on topics, and do the consumers.

1.2.2 Producers

Producers are the components that create messages and push them to the topic partitions in brokers. By default, the producer does not care what partition the message is on. Depending on whether or not to specify which partition to write to, the producer will either send the topic's partitions or distribute it evenly across partitions.

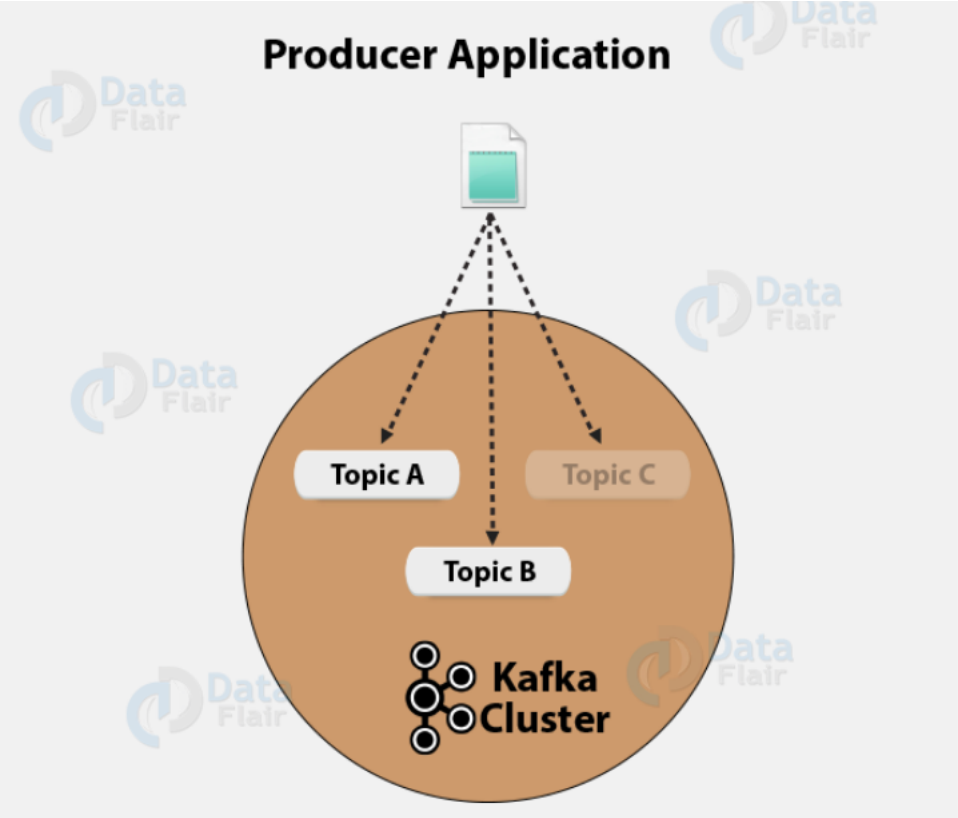

Figure 1.4: Producers in Kafka.

As shown by DataFlair [9] on Figure 1.4, in some cases, the producer will send messages to specific partitions. This is usually done using the key to ensures that all messages generated with a given key will be written to the same single partition.

In order to send messages asynchronously to one or more topics, the Kafka Producer API permits to an application. The KafkaProducer class provides "send" method, which is:

Listing 1 `send` method in KafkaProducer class

```
producer.send(new ProducerRecord<byte[],byte[]>(topic,
partition, key1, value1) , callback);
```

where `ProducerRecord` is a buffer of records waiting to be sent by producers and `Callback` is a user-supplied callback to execute when the record has been acknowledged by the server. The `ProducerRecord` class constructor is defined as:

Listing 2 `ProducerRecord` class constructor.

```
1 public ProducerRecord (string topic, int partition, k key, v value)
```

where

- Topic: user-defined topic name that will append to record.
- Partition: partition count.
- Key: The key that will be included in the record.
- Value: Record contents.

As stated above, we can add a *key* to a message. Basically, the aim is to get ensured that all these messages (with the same key) will end up in the same partition if a producer publishes a message with a key (see more in subsection 1.2.4). Due to this feature, Kafka offers message sequencing guarantee. Though, unless a key is added to it, data is written to partitions randomly.

1.2.3 Consumers

While Kafka producers write to topics, Kafka consumers read from topics. The consumer subscribes to one or more topics and reads the messages in the order in which they were produced. Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset.

Each new message in the partition gets an *Id* which is one more than the previous Id number. This Id number is also called as the Offset. So, the first message is at 'offset' 0, the second message is at offset 1 and so on. These offset Id's are always incremented from the previous value. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages.

Consumers work as part of a *consumer group*, which is one or more consumers that work together to consume a topic. One consumer group might be responsible for delivering records to high-speed, in-memory microservices while another consumer group is streaming those same records to Hadoop.

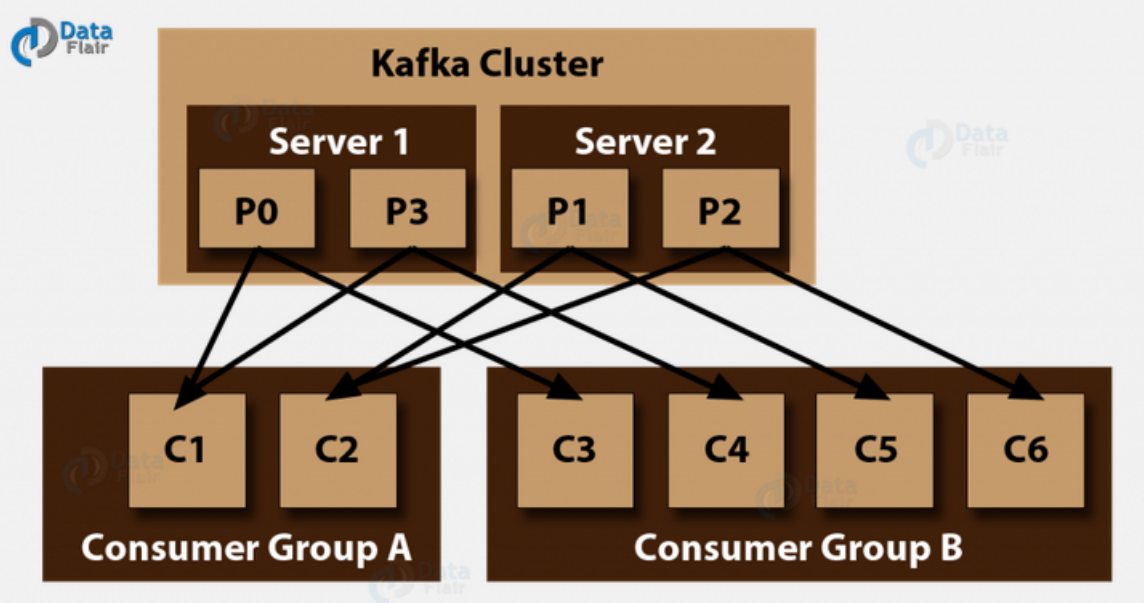

Figure 1.5: Consumers in Kafka.

As same as *offset*, a consumer group has a unique *id* to identify them from other consumer groups. Each group is a subscriber to one or more Kafka topics. The group of consumers is a solution for a problem of having multiple subscribers. A notice needs to be made is that consumers in one group work together but a record gets delivered to only one consumer in a consumer group, which helps to load balance record processing among a group. In case a new consumers join a consumer group, it gets a share of partitions. If a consumer dies, its partitions are split among the remaining live consumers in the consumer group. As a consequence, each consumer in the consumer group is an exclusive consumer of a "fair share" of partitions.

1.2.4 Topics and Partitions

Topic is a concept introduced with the aim of easily matching between producers and consumers. Topics are written by one or more producers and read by one or more consumers. In fact, a topic is such a table in an SQL database, which contains the message.

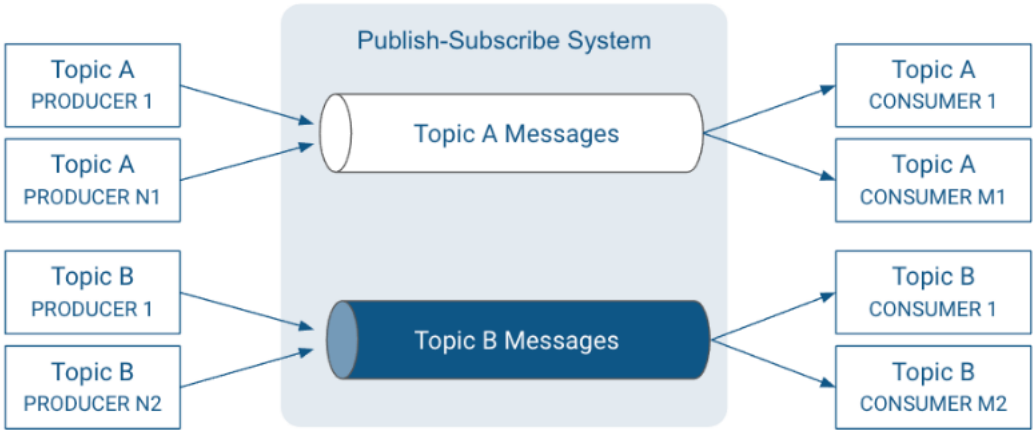

Figure 1.6: Topics in Kafka.

According to Cloudera [7] on Figure 1.6, Messages in Kafka are categorized into topics, which are object-oriented. Basically, Kafka broker stores topics. However, a topic in Apache Kafka is broken up into several *partitions*. And, further, Kafka spreads those log's partitions across multiple servers or disks. The most interesting of data management in Kafka is about *partitions*.

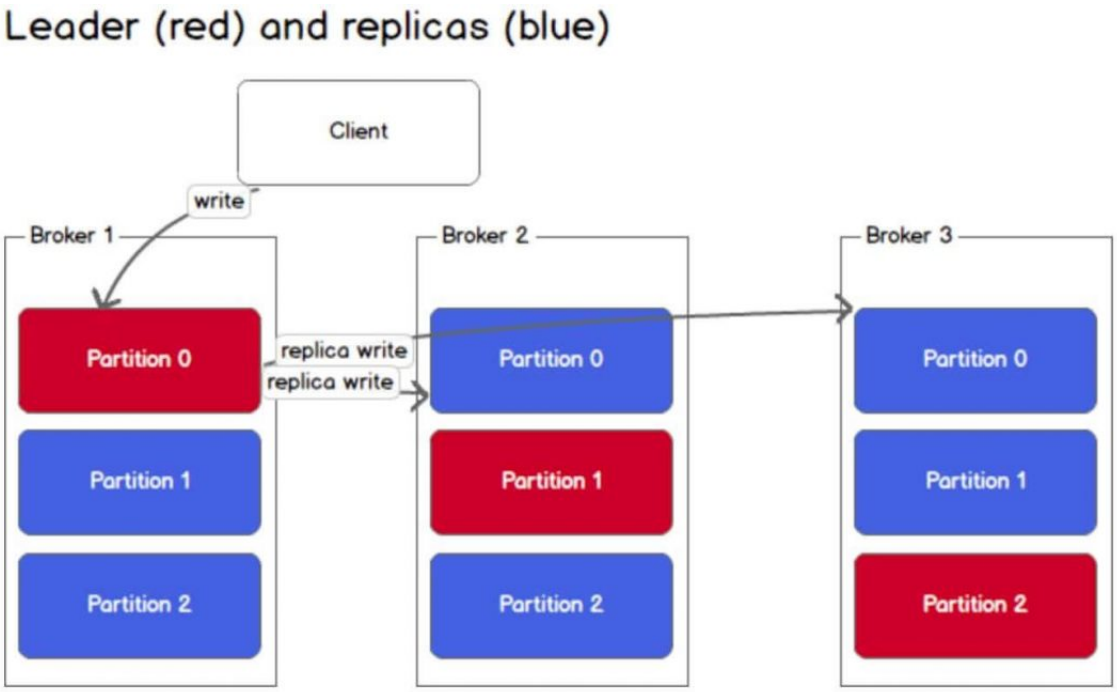

Figure 1.7: Partitions of topics.

When creating a topic, it is needed to configure two properties:

- Partition count: The number of partitions that records for this topic will be spread among.
- Replication factor: The number of copies of a partition that are maintained to ensure consumers always have access to the queue of records for a given topic.

The more the partition, the faster the read and write parallel work. The messages in the partition are stored in an immutable order (offset). Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data.

One Kafka client (a producer or consumer) communicates only with the leader partition for data. If a leader fails, a replica takes over as the new leader. The number of replicas is always smaller than the number of brokers. In the Figure 1.7, as described by Kenvin Sookocheff [10], the producer is writing to partition 0 of the topic and partition 0 replicates that write to the available replicas. Ideally, the follower partitions have an exact copy of the contents of the leader.Therefore, it is clearly to say that partitions play a crucial role for keeping good record throughput.

1.2.5 Records

As mentioned above, a record is a publish-subscribe message. A record is so-called the unit of data within Kafka. Simply put, a record is such a line in the "table" topic. As shown on 2, a producer sends key/value pair to Kafka that consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value. The key is not required, but can be used to identify messages from the same data source. In fact, the key is also a byte array and, as with the message, has no specific meaning to Kafka.

For efficiency, messages are written into Kafka in *batches*. A batch is simply a group of records, all of which are being produced to the same topic and partition. Kafka batches the data into chunks which helps in reducing the network calls and converting most of the random writes to sequential ones. It's more efficient to compress a batch of data as compared to compressing individual messages. However, of course, this is a trade-off between latency and throughput: the larger the batches, the more messages that can be handled per unit of time, but the longer it takes an individual message to propagate [1].

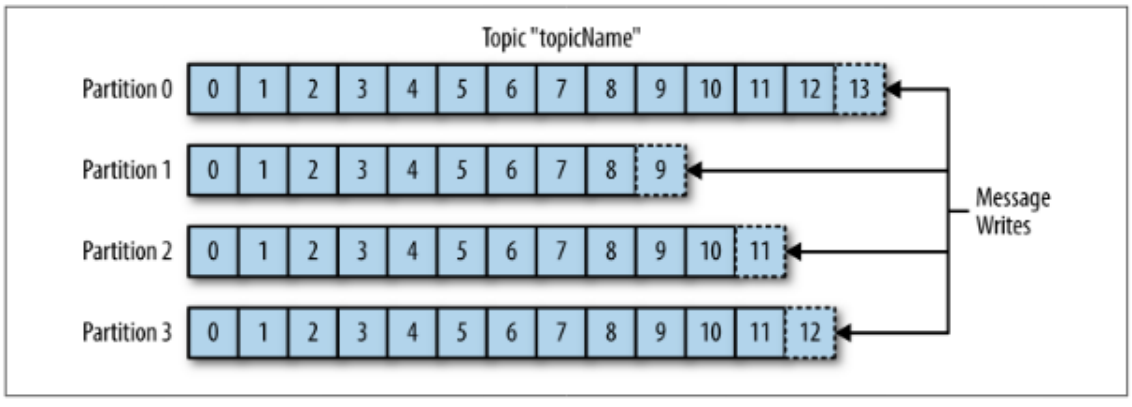

Figure 1.8: Log structure.

Another important definition related to records is *log* structure on Figure 1.8. In *log*, records are appended to the end of the log, and reads proceed left-to-right. In other words, log is an append-only structure. Each entry is assigned a unique sequential log entry number. The ordering of records defines a notion of "time" since entries to the left are defined to be older then entries to the right. The log entry number can be thought of as the "timestamp" of the entry. So a log is not all that different from a file, where the records are sorted by time.

On Figure 1.8, each partition is a totally ordered log, but there is no global ordering between partitions. According to the page *enginering.linkedin* [11], although lack of a global order across partitions is a limitation, but it has not found it to be a major one. Indeed, interaction with the log typically comes from hundreds or thousands of distinct processes so it is not meaningful to talk about a total order over their behavior. Instead, the guarantees that we provide are that each partition is order preserving, and Kafka guarantees that appends to a particular partition from a single sender will be delivered in the order they are sent.

1.3 Kafka's functions

Apache Kafka is a high performance, highly available, and redundant streaming message platform. Kafka functions much like a publish/subscribe messaging system, but with better throughput, built-inpartitioning, replication, and fault tolerance. Kafka is a good solution for large scale message processing applications. It is often used in tandem with Apache Hadoop, and Spark Streaming [7].

The original use case for Kafka was for Website Activity Tracking [8]. This means whetever user behaviors on websites are, all are saved and published to central topics with one topic per activity type. This is the reason why on the New-Feeds of Facebook or Youtube, there are available offers or advertisements which are related to users' interests. In the other words, all site activities such as page views, searches, or other actions users may take are being analysed, real-time processed, and real-time monitored.

Moreover, Kafka is also a smart choice for log aggregation, with low latency and convenient support for multiple data sources. In a nutshell, it is convinced to say that Kafka can provides the following:

- **Scalability**: Kafka is a distributed system, which is able to be scaled quickly and easily without incurring any downtime. All producers, consumers, and brokers can all be scaled out to handle very large message streams with ease.
- **High performance**: Kafka delivers high throughput and supporting hundreds of thousands of messages per second, even with modest hardware.
- **Durability**: Kafka supports partitioning messages and intra-cluster replication over Kafka servers managed by Zoomkeeper. This makes for a highly durable messaging system.
- **Fault Tolerance**: The Kafka cluster can handle failures with the masters and databases.

- **Extensibility**: There are as many ways by which applications can plug in and make use of Kafka. In addition, offers ways by which to write new connectors as needed.
- **Disk-Based Retention**: Messages are committed todisk, and will be stored with configurable retention rules. If a consumer falls behind, either due to slow processing or a burst in traffic, there is no danger of losing data.

1.4 Kafka's security

As presented, any user or application can write any messages to any registered topic, as well as read data from any topics. However, when a company moves towards a shared tenancy model where multiple teams have different roles and can not reveal some critical and confidential information to each other, security needs to be implemented.

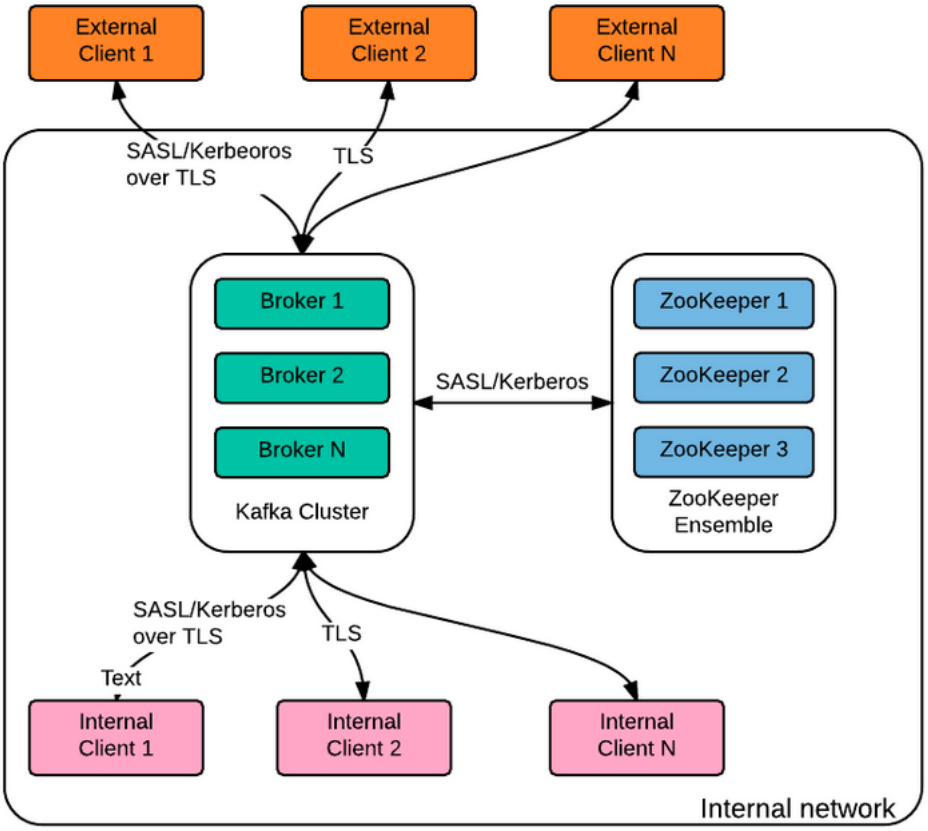

Figure 1.9: Kafka's security.

Being noticed of the security problem, Kafka Security has three components:

1. **TLS Keys and Certificates**
   TLS or Transport Layer Security is considered as a client authentication. At first, a TLS key and certificate for each broker and client in the cluster are generated. At this point, each broker has a public-private key pair and an unsigned certificate

to identify itself. To prevent forged certificates, it is important for each certificate to be signed by a certificate authority (CA). As long as the CA is a genuine and trusted authority, the clients have high assurance that they are connecting to authentic brokers. This way of security is secure method allowing producers and consumers to authenticate to Kafka cluster, which verifies their identity.

2. **Authorization using ACLs**
   ALCs is Access Control Lists. Once clients (producers, consumers, admin) are authenticated, Kafka brokers can run them against ACL to determine whether or not a particular client would be authorised to write or read to some topic. Kafka ACLs are defined in the general format of "Principal P is [Allowed/Denied] Operation O From Host H On Resource R". The operations available are both for clients and inter-broker operations of a cluster.

3. **Data encryption**

   Data is encrypted and securely transmitted to the brokers. Here, by encryption, data is kept secret between producers and Kafka as well as consumers and Kafka (see more in chapter 3).

Chapter 2

ZooKeeper

2.1 ZooKeeper's Presentation

The broker, topic, and partition information in Kafka are maintained in Zookeeper. Therefore, this part will focus on ZooKeeper and its roles related to Kafka. As known, ZooKeeper is a distributed, open-source and high-performance coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

2.1.1 Design Goals

As other coordination distributed system, ZooKeeper was designed with following goals:

Simplicity: ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The namespace consists of data registers - called znodes, in ZooKeeper 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 ZooKeeper can achieve high throughput and low latency numbers.

Reliability: The ZooKeeper implementation puts a premium on high performance, highly available, strictly ordered access. The performance aspects of ZooKeeper means it can be used in large, distributed systems. The reliability aspects keep it from being a single point of failure. The strict ordering means that sophisticated synchronization primitives can be implemented at the client.

Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a set of hosts called an ensemble.

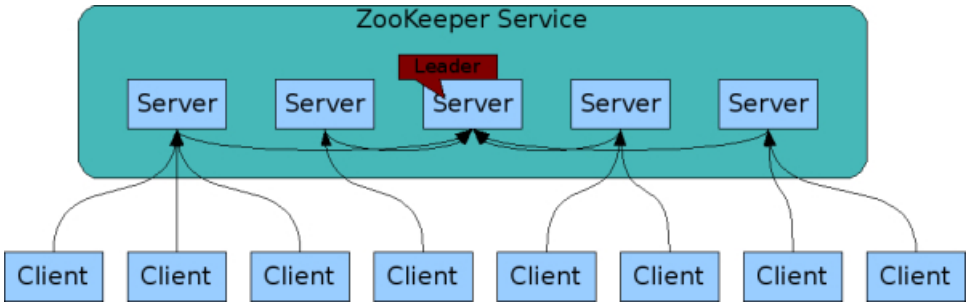

Figure 2.1: Replicated ZooKeeper service.

Scalibility: Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a set of hosts called an ensemble. The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

ZooKeeper 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. [13]

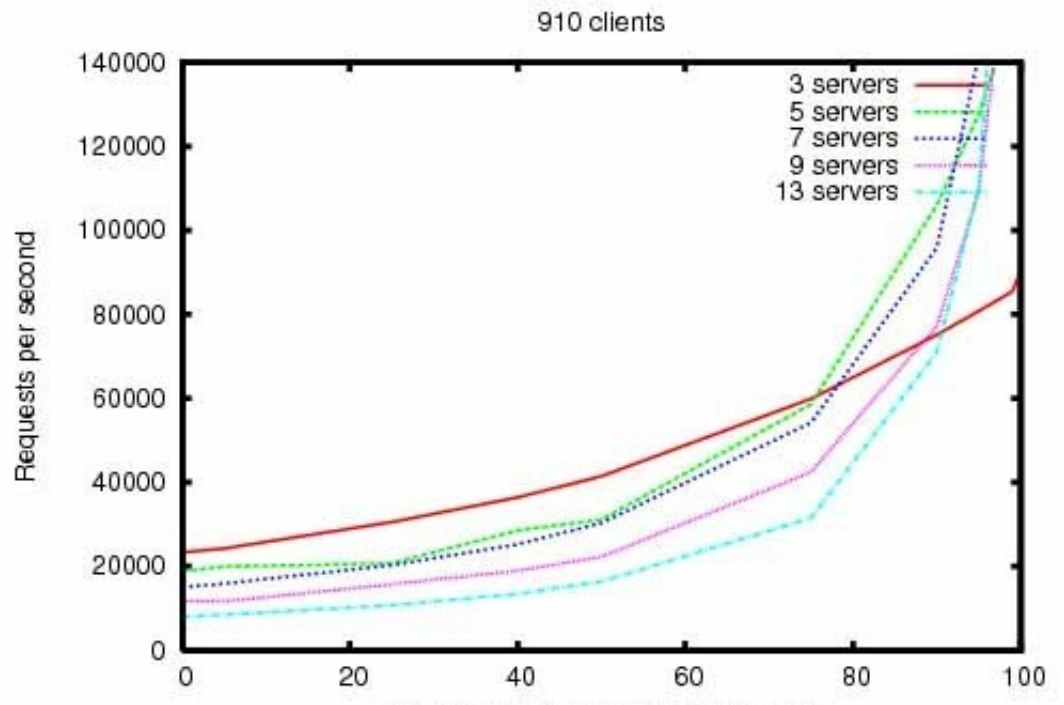

Figure 2.2: ZooKeeper Performance

Speed: It is especially fast in "read-dominant" workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.

2.1.2 Provided Services

ZooKeeper is used by companies including Yelp, Rackspace, Yahoo! [16], Odnoklassniki, Reddit, NetApp SolidFire, Facebook [17], Twitter [18] and eBay as well as open source enterprise search systems like Solr [19]. By providing a greate deal of services, ZooKeeper is becoming more and more useful.

Name service — A name service is a service that maps a name to some information associated with that name. A telephone directory is a name service that maps the name of a person to his/her telephone number. In the same way, a DNS service is a name service that maps a domain name to an IP address. In our distributed system, we may want to keep a track of which servers or services are up and running and look up their status by name.

Locking — To allow for serialized access to a shared resource in our distributed system, we may need to implement distributed mutexes. ZooKeeper provides for an easy way for you to implement them.

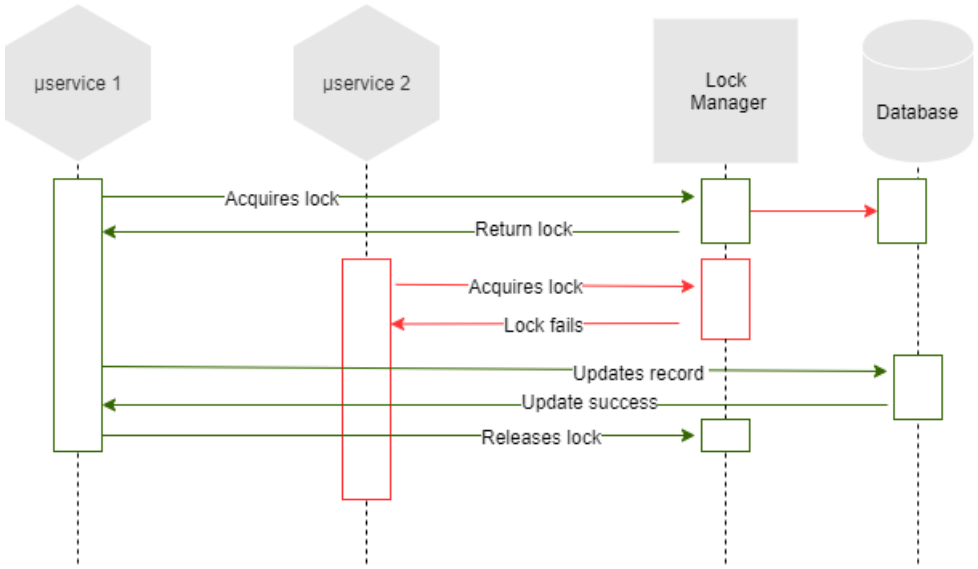

Figure 2.3: Locking Mechanism

Synchronization — Hand in hand with distributed mutexes is the need for synchronizing access to shared resources. ZooKeeper provides a simple interface to implement that.

Configuration management — We can use ZooKeeper to centrally store and manage the configuration of your distributed system. This means that any new nodes joining will pick up the up-to-date centralized configuration from ZooKeeper as soon as they join the

system. This also allows you to centrally change the state of your distributed system by changing the centralized configuration through one of the ZooKeeper clients.

Leader election — Our distributed system may have to deal with the problem of nodes going down, and you may want to implement an automatic fail-over strategy. ZooKeeper provides off-the-shelf support for doing so via leader election [14].

2.2 ZooKeeper Structure

2.2.1 Architecture

The Architecture of Apache Zookeeper is categorized into 5 different components as follows [21]:

- Ensemble: It is basically the collection of all the Server nodes in the Zookeeper ecosystem. The Ensemble requires a minimum of three nodes to get itself set up.
- Server: It is one among-st the other servers present in the Zookeeper Ensemble whose objective is to provide all sorts of services to its clients. It sends its alive status to its client in order to inform its clients about its availability.
- Server Leader: Ensemble Leader is elected at the service startup. It has access to recover the data from any of the failed nodes and performs automatic data recovery for clients.
- Follower: A follower is one of the servers in the Ensemble. Its duty is to follow the orders passed by the Leader.
- Client: Clients are the nodes that request service from the server. Similar to servers, the client also sends signals to servers regarding their availability. In case if the server fails to respond, then they automatically redirect themselves to the next available server.

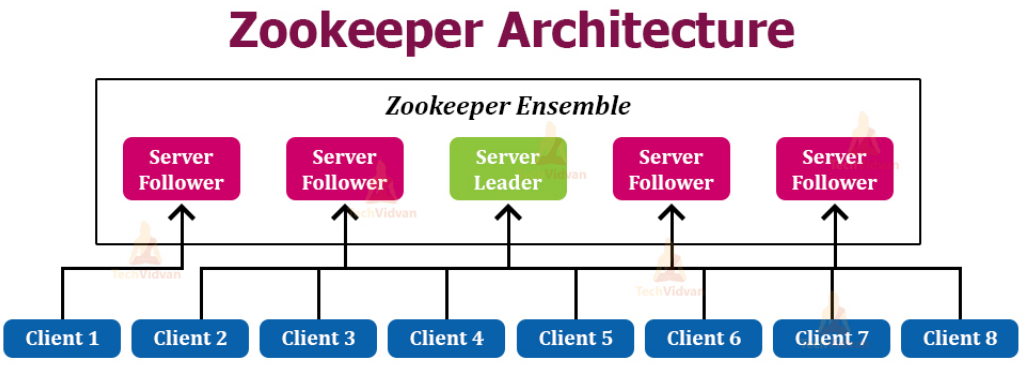

Figure 2.4: Zookeeper Architecture

2.2.2 Data Model

As same as a standard file system, the namespace provided by ZooKeeper. Basically, a sequence of path elements which separates by a slash (/) is what we call a name. In ZooKeeper's namespace, a path identifies every node. Moreover, in a ZooKeeper namespace, each node can have data associated with it and its children. As same as a file-system which permits a file to also be a directory. [13]

We use the term znode to make it clear that we are talking about ZooKeeper data nodes.

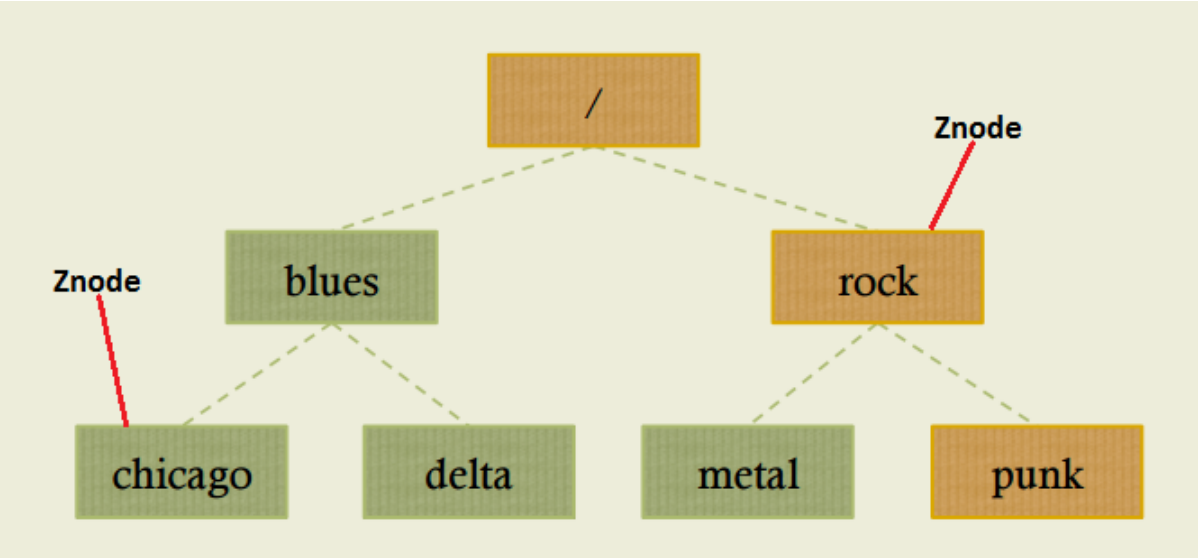

Figure 2.5: UNIX-style paths in ZooKeeper

What is ZNode?

Znodes maintain a stat structure that includes version numbers for data changes, ACL changes, and timestamps, to allow cache validations and coordinated updates.

```
ZNode = <data, version, creation flags, children>
```

Each time a znode's data changes, the version number increases. For instance, a Client also receives the version of the data, whenever it retrieves data. Though, a client must supply the version of the data of the ZNode it is changing, when a client performs an update or a delete. Because the update will fail if the version it supplies doesn't match the actual version of the data.

- **Version number** Every znode has a version number, which means every time the data associated with the znode changes, its corresponding version number would also increased. The use of version number is important when multiple zookeeper clients are trying to perform operations over the same znode.
- **Action Control List** (ACL) ACL is basically an authentication mechanism for accessing the znode. It governs all the znode read and write operations.

- **Timestamp** Timestamp represents time elapsed from znode creation and modification. It is usually represented in milliseconds. ZooKeeper identifies every change to the znodes from "Transaction ID" (zxid). Zxid is unique and maintains time for each transaction so that you can easily identify the time elapsed from one request to another request.
- **Data length** Total amount of the data stored in a znode is the data length. You can store a maximum of 1MB of data.

Once a ZooKeeper ensemble starts, it will wait for the clients to connect. Clients will connect to one of the nodes in the ZooKeeper ensemble. It may be a leader or a follower node. Once a client is connected, the node assigns a session ID to the particular client and sends an acknowledgement to the client. If the client does not get an acknowledgment, it simply tries to connect another node in the ZooKeeper ensemble. Once connected to a node, the client will send heartbeats to the node in a regular interval to make sure that the connection is not lost. [24]

ZNode Types

There are three types of Znodes as mentioned below.

Persistence Znode: All the Znodes in an ensemble assume themselves to be Persistence Znodes. These nodes tend to stay alive even after the client is disconnected.

Ephemeral Znode: These type of nodes stay alive until the client is connected to them. When the client gets disconnected, they die. These type of nodes are not allowed to have children.

Sequential Znode: It can be either a Persistence Znode or an Ephemeral Znode. When a node gets created as a Sequential Znode, then you can assign the path of the Znode by attaching a 10 digit sequence number to the original name.

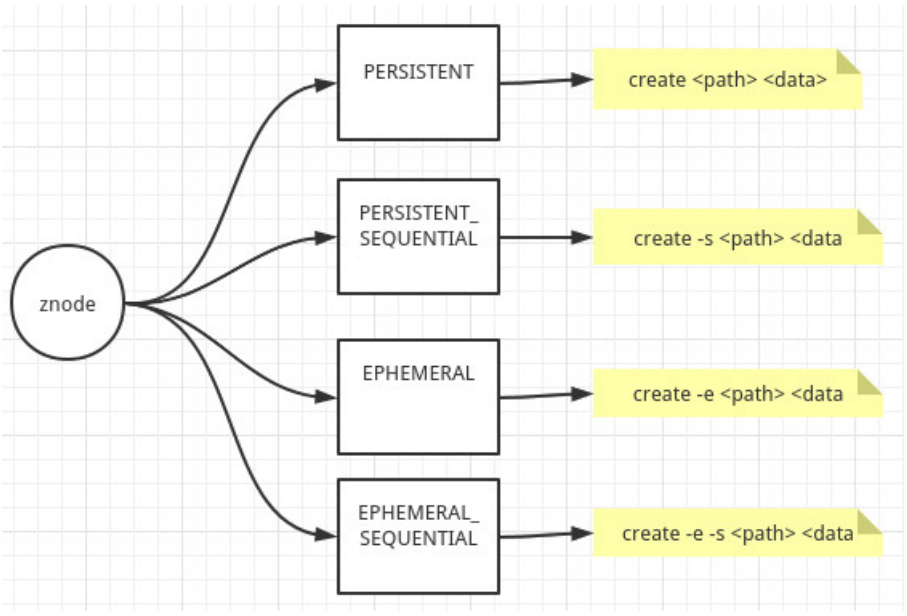

Figure 2.6: ZNode types and create command

Characteristics of Znodes

Further, there are several characteristics of Zookeeper Znodes, such as:

a. Watches
It is possible for clients to set watches on ZNodes. So, when we make changes to that ZNode, as a result, it triggers the watch and then further clear the watch. Also, ZooKeeper sends the client a notification, when a watch triggers.

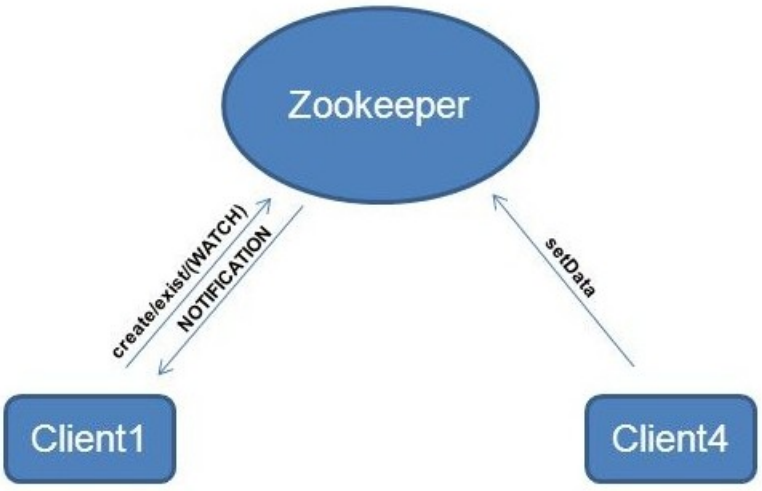

Figure 2.7: Watches in ZooKeeper

b. Data Access
At each ZNode, the data which is stored in a namespace is read and written atomically. Basically, Read process get all the data bytes which are associated with a ZNode. Whereas, the writing process replaces all the data. In addition, there is an Access Control List (ACL) of each node, so that restricts the function of all. Let's revise ZooKeeper CLI

If a client wants to read a particular znode, it sends a read request to the node with the znode path and the node returns the requested znode by getting it from its own database. For this reason, reads are fast in ZooKeeper ensemble.

If a client wants to store data in the ZooKeeper ensemble, it sends the znode path and the data to the server. The connected server will forward the request to the leader and then the leader will reissue the writing request to all the followers. If only a majority of the nodes respond successfully, then the write request will succeed and a successful return code will be sent to the client. Otherwise, the write request will fail. The strict majority of nodes is called as Quorum. [25]

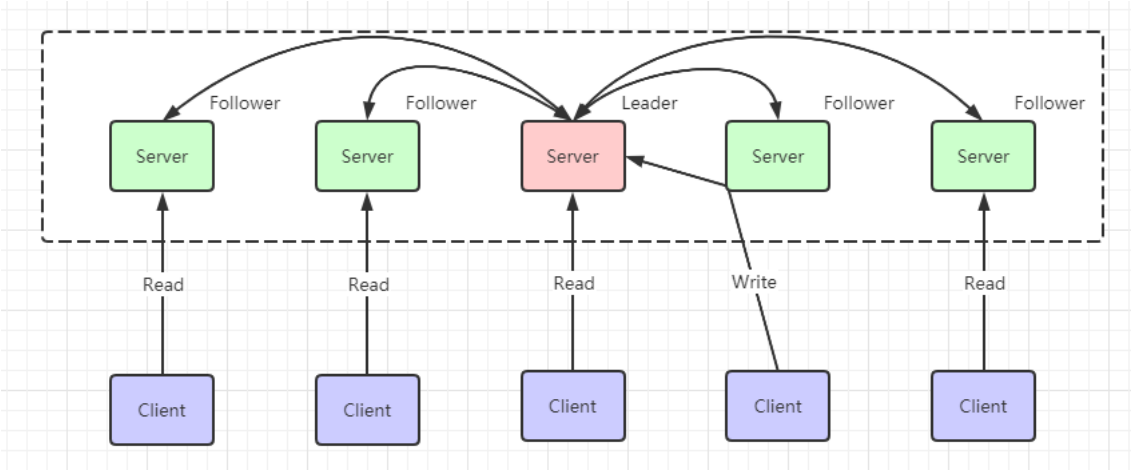

Figure 2.8: Data access in ZooKeeper

c. Ephemeral Nodes
As long as the session which creates the ZNode is active, until that time these ZNodes exists. Similarly, the ZNode is deleted, when the session ends. Thus, ephemeral ZNodes are not allowed to have children, because of this behavior only.

d. Sequence Nodes – Unique Naming
We can also request that ZooKeeper append a monotonically increasing counter to the end of the path while creating a ZNode. Well, it is a unique counter to the parent ZNode. The format of the counter is %010d — basically, it is 10 digits with 0 (zero) padding. However, to simplify sorting, the counter is formatted in this way, like ‘‘<path>0000000001’’, ‘‘<path>0000000002’’, ... [20]

Also, note that maintained by the parent node, the counter used to store the next sequence number is a signed int (4bytes), and also the counter will overflow while it is incremented beyond 2147483647. So, this was all about ZooKeeper ZNodes. Hope you like our explanation.

2.3 ZooKeeper and Kafka

Basically, Kafka — ZooKeeper stores a lot of shared information about Kafka Consumers and Kafka Brokers.

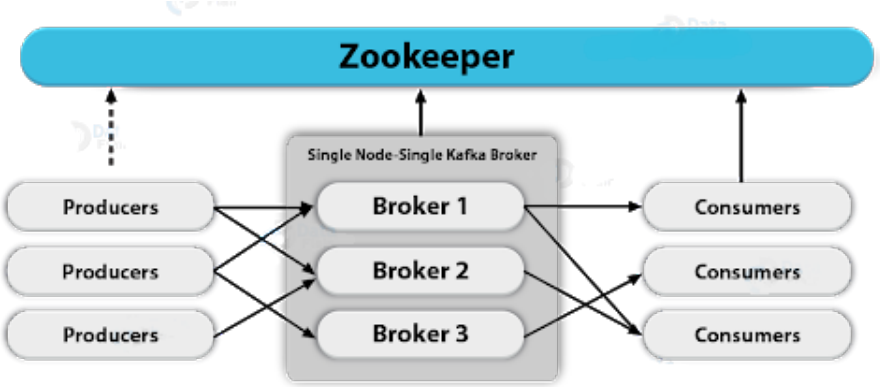

Figure 2.9: Kafka and ZooKeeper

2.3.1 ZooKeeper and Kafka Broker

The broker, topic, and partition information of Kafka are maintained in Zookeeper. In particular, the partition information, including partition and replica locations, updates fairly frequently.

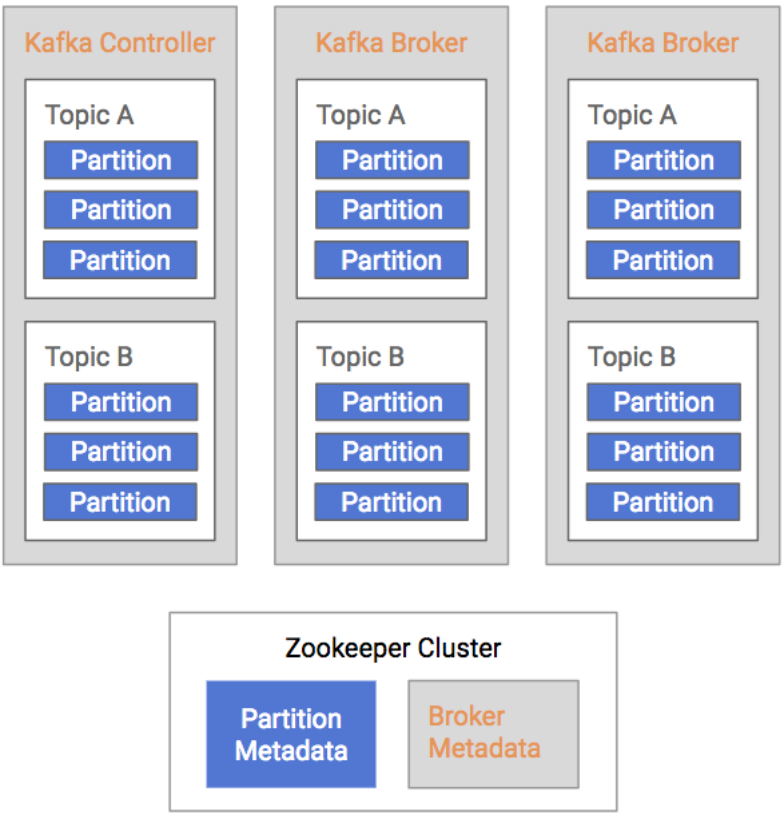

Figure 2.10: Kafka information is maintained in ZooKeeper

Zookeeper determines the state. That means, it notices, if the Kafka Broker is alive, always when it regularly sends heartbeats requests. Also, while the Broker is the constraint to handle replication, it must be able to follow replication needs.

In order to have different producing and consuming quotas, Kafka Broker allows some clients. This value is set in ZK under /config/clients path. Also, we can change it in bin/kafka-configs.sh script. [23]

However, for each topic, Zookeeper in Kafka keeps a set of in-sync replicas (ISR). The leader is one of the brokers and is responsible for maintaining the leader/follower relationship for all the partitions. When a node shuts down, it is the controller that tells other replicas to become partition leaders to replace the partition leaders on the node that is going away. Zookeeper is used to elect a controller, make sure there is only one and elect a new one it if it crashes.

Basically, Zookeeper in Kafka stores nodes and topic registries. It is possible to find there all available brokers in Kafka and, more precisely, which Kafka topics are held by each broker, under /brokers/ids and /brokers/topics zNodes, they're stored. In addition, when it's started, Kafka broker create the register automatically.

2.3.2 ZooKeeper and Kafka Consumer

ZooKeeper is the default storage engine, for consumer offsets, in Kafka's 0.9.1 release. However, all information about how many messages Kafka consumer consumes by each consumer is stored in ZooKeeper.

Consumers in Kafka also have their own registry as in the case of Kafka Brokers. However, the same rules apply to it, ie. as ephemeral zNode, it's destroyed once the consumer goes down and the registration process is made automatically by the consumer. [22]

2.3.3 ZooKeeper - Kafka Communication

Thanks to Scala classes representing Kafka, we can initiate connection from kafka to zookeeper without kafka client's consciousness. Its startup() method, initZkClient() (Kafka is seen as a ZooKeeper client) contains a call to method initializing ZooKeeper connection. There are several methods in this algorithm which we use in this Zookeeper method. Hence, as a result, the method creates a temporary connection to ZooKeeper, in this case. This session is responsible for creating zNodes corresponding to chroot if it's miAfterwarderwards, this connection closes and creates the final connection held by the server.

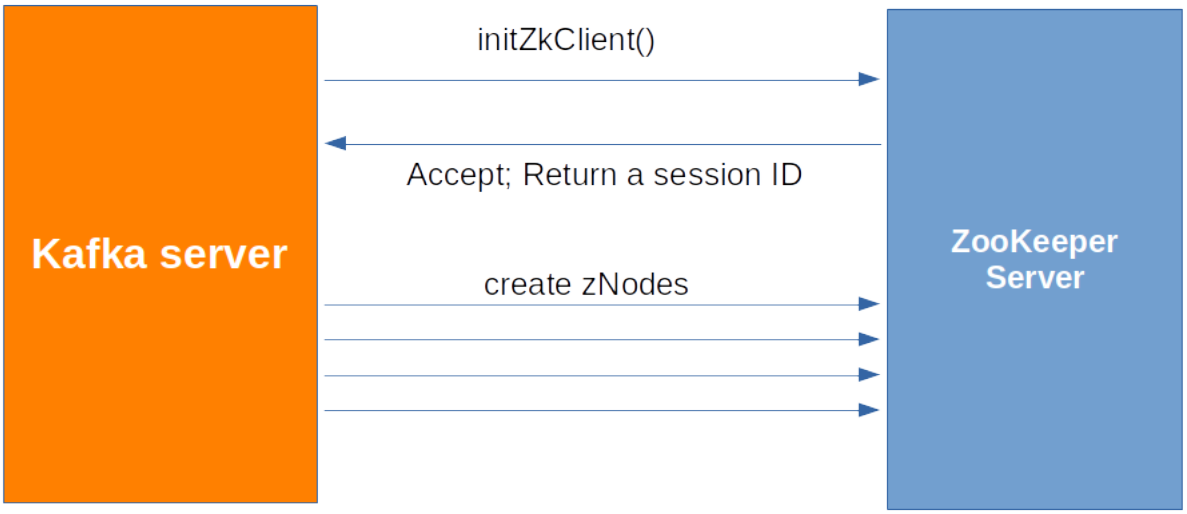

Figure 2.11: Initiate connection between Kafka and ZooKeeper

After, still inside initZkClient(), Kafka initializes all persistent zNodes, especially which server uses. We can retrieve there, among others: /consumers, /brokers/ids, /brokers/-topics, /config, /admin/delete_topics, /brokers/seqid, /isr_change_notification, /config/-topics, /config/clients. [23]

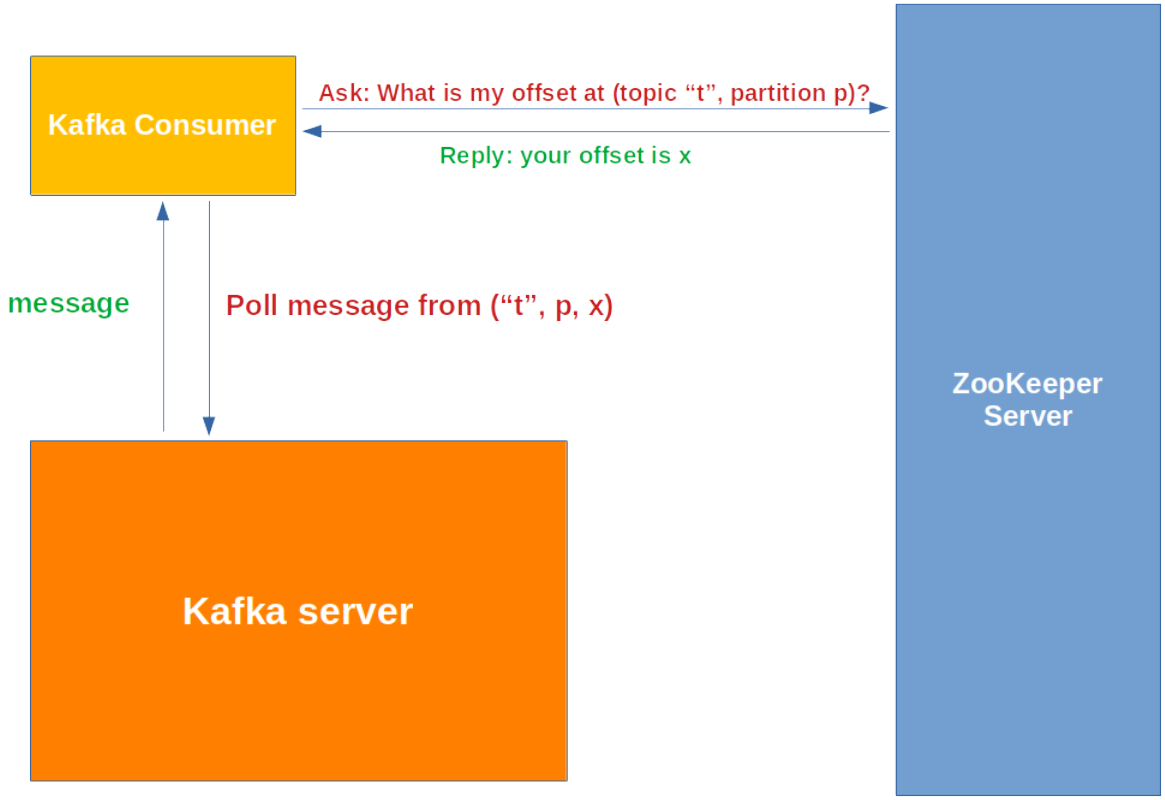

Figure 2.12: Communication between Kafka Consumer, ZooKeeper and Kafka Server

Another important role of ZooKeeper in Kafka "leader election". Nevertheless, in the study scope of this paper, we will not mention it.

In the next chapter, we will discuss about some Encrypted matching schemes as well as how to apply them into our servers to assure a new secure protocol.

Chapter 3

Encrypted matching scheme

3.1 Data storage problem

Today's markets are highly dynamic, with major change occurring randomly on frequent basis. No organization can function without data these days. With huge amounts of data being generated every second from business transactions, sales figures, promotion strategies, customer behaviors, and stakeholders, data is the fuel that drives companies. But when the data volume is huge, enterprises have to face the most pressing challenges that is storing all these huge sets of data properly. However, as these data sets grow exponentially with time, it gets extremely difficult to handle.

A potential solution for companies is to outsource data storage, thus seeking third-party experts outside of the company. Data storage outsourcing is, however, placing its trust in a third-party – relying on their expertise, their resources, and their services. Moreover, storage outsourcing shares their resources with other tenants. This sharing helps to save on costs and makes the IT scalable, but it does mean that other companies will be using some of the same servers and same devices. This creates security concerns that need to be addressed, especially if the company has strict compliance requirements. While companies may trust a Storage Service Provider's (SSP) reliability and performance, they cannot be sure 100% that an SSP is not going to use the data for other purposes, especially when the data is highly valuable.

This problem raise to a question if there is another approach to provide data confidentiality. A potential answer is cryptography. On the same idea, Changyu Dong, Giovanni Russello, and Naranker Dulay are published an article of "Shared and Searchable Encrypted Data for Untrusted Servers" [2]. The report pointed that server side encryption is not appropriate when the server is not trusted. Instead, the client must encrypt the data before sending it to the SSP and later the encrypted data can be retrieved and decrypted by the client. This could solve the problem of data leakage, but also raise a challenge of data management functionalities such as key management and data searching. Hence, the rest of this chapter will be saved to present several encrypted matching schemes to figure out the solution for the mentioned concern.

3.2 Related work

3.2.1 Privacy-Preserving Queries

Overall architecture

In 2006, Zhiqiang Yang, Sheng Zhong, and Rebecca N. Wright published an article titled "Privacy-Preserving Queries on Encrypted Data" on the 11th European Symposium On Research In ComputerSecurity (Esorics) [3], which discussed data confidentiality concern in database systems. As same as the authors in the previously mentioned paper [2], the article raised a problem on performing queries when data is encrypted. To solve the problem, they proposed an elegant scheme for performing queries on encrypted data and also provided a secure index to speed up queries by two-step mapping.

A system on Figure 3.1 is considered where data is encrypted and stored in tables. In the front-end, when the user has a query, the query is translated to one or more messages that are sent to the database. Upon receiving the message(s), the database finds the appropriate encrypted cells and returns them to the front end. Finally, the front end decrypts the received cells.

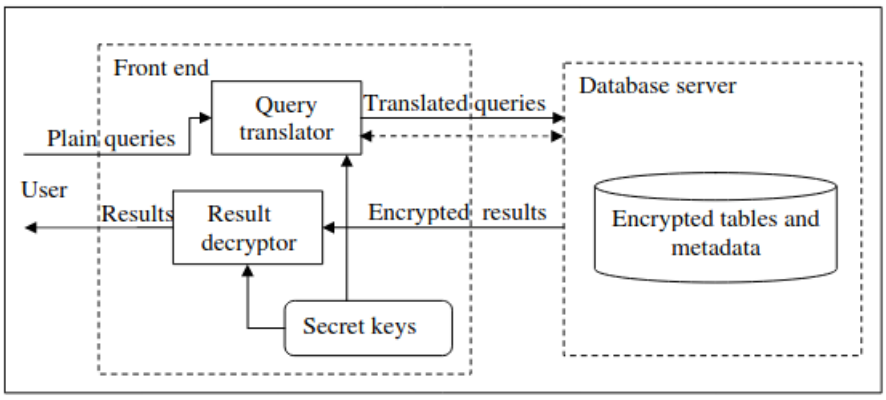

Figure 3.1: Overall architecture.

Here, the database server is not fully trusted, because it may be vulnerable to intrusion. Once an intruder breaks into the database, he can observe not only the encrypted data in the database, but can also control the whole database system for a time interval. This assumption is reasonable due to the fact that a database administrator can physically reset the database server from time to time or when intrusions are detected. Additionally, the architecture on Figure 3.1 also assumes that the communication channel between the user and the database is secure, as there exist standard protocols to secure it, as well as the user's front-end program. Besides, all data and metadata, including user logs and scheme metadata, are required to be stored encrypted.

Table and queries

A table is denoted T. Suppose that T has n rows and m columns (or attributes), $T_{i,j}$ is the cell at the intersection of the i^{th} row and the j^{th} column; here (i, j) refers to the coordinates of the cell. For the j^{th} attribute A_j, its domain is denoted D_j.

For a table T, an encrypted table T is stored in the database, and each cell of T is a bitstring of exactly k_1 bits. When encrypting a cell, the encryption algorithm appends a random string of k_2 bits to the plaintext. Hence, the input to the encryption algorithm is a k_0-bit string, where $k_0 = k_1 + k_2$. The purpose of using a random string k_2 bits is that multiple occurrences of a plaintext should lead to different cipher texts. In addition, k_0 is chosen so that it is long enough to resist brute-force key search attacks.

Privacy-Preserving Queries

According to Yang et al. [3], Privacy-Preserving Query is defined as below:

Definition 1. *(Privacy-Preserving Query) A one-round query protocol* reveals only α beyond the minimum information revelation *if for any polynomial* poly() *and all sufficiently large* k_0*, there exists a probabilistic polynomial-time algorithm* $\mathcal{S}$ *(called a* simulator*) such that for any* $t <$ poly(k_0)*, any polynomial-size circuit family* $\{\mathcal{A}_{k_0}\}$*, any polynomial* $p()$*, and any* $Q_1, \ldots, Q_t$*,*

$$|\Pr[\mathcal{A}_{k_0}(Q_1, \ldots, Q_t, q_1, \ldots, q_t, T') = 1] - \Pr[\mathcal{A}_{k_0}(Q_1, \ldots, Q_t, \mathcal{S}(\alpha, R(Q_1), \ldots, R(Q_t), T')) = 1]| < 1/p(k_0).$$

A query protocol is ideally private *if it reveals nothing beyond the minimum information revelation.*

where

- Q is the query a user intends to perform on the table T;
- $R(Q)$ is the set of coordinates of the cells satisfying the condition of query Q;
- α is a random variable referring to quantity information leaked by the protocol.

The above definition can be viewed as an adaptation of the definition of secure protocol in the semi-honest model [4].

Solution

The idea is that for each cell $T_{i,j}$, the encrypted cell $T'_{i,j}$ is divided into 2 parts:

$$T'_{i,j} = (T'_{i,j}(1), T'_{i,j}(2))$$

The first part $T'_{i,j}(1)$ is a simple encryption of $T_{i,j}$, using a block cipher $E()$, while the second part, $T'_{i,j}(2)$, is a "checksum" that, together with the first part, enables the database to check whether this cell satisfies the condition of the query or not.

$T'_{i,j}(1)$ and $T'_{i,j}(2)$ satisfy a secret equation determined by:

$$E_{f(T_{i,j})}(T'_{i,j}\langle 1\rangle) = T'_{i,j}\langle 2\rangle,$$

where f is a function. When the user has a query with condition $A_j = v$, she only needs to send $f(v)$ to the database. $f()$ is defined to be an encryption of v using the block cipher $E()$ because it should be infeasible to derive v from $f(v)$ because otherwise an intruder can learn v when observing $f(v)$.

Being noted that each encrypted cell with the same plaintext value has a different encryption, thus if an intruder breaks into the database and sees the encrypted table, he cannot tell whether two cells have the same plaintext value or not.

The security of the scheme derives from the security of the block cipher used. In cryptography, secure block ciphers are modeled as *pseudorandom permutations* [6]. Here, encryption key of the block cipher is the random seed for the pseudorandom permutation. For each value of the key,the mapping from the cleartext blocks to the ciphertext blocks is the permutation indexed by the value of the seed. Also, in this setting, the authors proved that even if the intruder has access to the whole database, he can learn nothing about the encrypted data.

3.2.2 RSA-Based Proxy Encryption

Overall architecture

The second referenced article is "Shared and Searchable Encrypted Data for Untrusted Servers" of the author group Changyu Dong, Giovanni Russello, and Naranker Dulay [2]. In this paper, authors proposed a scheme for multi-user searchable data encryption, which does not require a fully trusted server. The server can search an encrypted keyword on the encrypted data. More importantly each authorised user in the system has his own unique keys which simplifies key revocation and avoids data re-encryption. All the authorised users can insert encrypted data, decrypt the data inserted by other users and search encrypted data without knowing the other users' keys. The keys of one user can easily be revoked without affecting other users or the encrypted data at the server.

The scheme's overall architecture is presented on Figure 3.2.

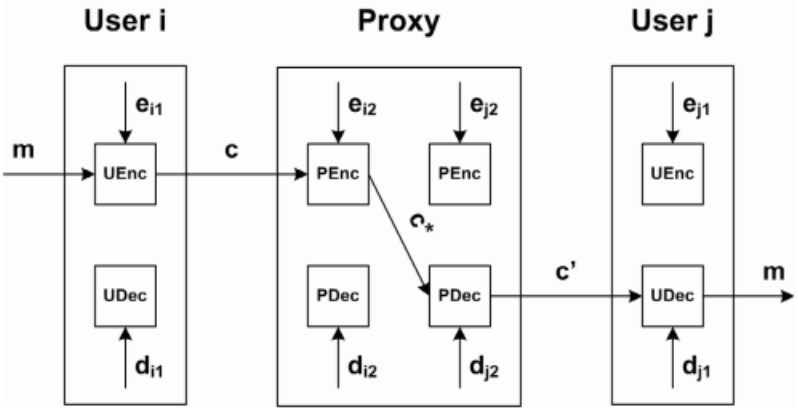

Figure 3.2: RSA-based Proxy Encryption Scheme.

The notion of proxy encryption was first introduced by Blaze et al. [5] in 1998. In RSA-based proxy encryption scheme as on Figure 3.2, a group of symbols *(IGen, UGen, UEnc, UDec, PEnc, PDec)* is used to denote the proxy encryption scheme, where:

- *IGen* is the master key generation algorithm which is identical to the key generation algorithm in the standard RSA. *IGen* needs only to be run once at the beginning of the system setup.
- *UGen* is the algorithm for generating the key pairs for the users and the proxy. For each user i, *UGen* takes the output of *IGen* and finds e_{i1}, e_{i2}, d_{i1}, and d_{i2} such that $e_{i1}e_{i2} \equiv e \mod \varphi(n)$ and $d_{i1}d_{i2} \equiv d \mod \varphi(n)$.
- *UEnc* is the algorithm for user encryption. For a message m, user i encrypts it using his encryption key $K_{uei} = e_{i1}$. The resulting ciphertext is $c = m^{e_{i1}}$.
- *PEnc* is the algorithm for proxy encryption. When the proxy receives a ciphertext c from user i, it re-encrypts it using the corresponding encryption key.
- *PDnc* is the algorithm for proxy decryption. Before sending the ciphertext to user j, the proxy decrypts it using the corresponding decryption key.
- *UDec* is the algorithm for user decryption. When a user j receives a ciphertext c' from the proxy, he decrypts it using his decryption key.

A trusted key management server (KMS) controlled by the data owner is aplied to manage the keys. To be general, a producer is called client i and a customer is called client j. Suppose that client i pushed data in a untrusted server and client j wants to search for data from the same server without security loss. As stated on the name, the proxy is built based on RSA technique, where there are private and public keys. On the step of *UGen*, the set of e_{i1} , e_{i2}, d_{i1}, and d_{i2} are found. While (e_{i1}, d_{i1}) are kept by client i, (e_{i2}, d_{i2}) are kept by the server. If the user is removed from the system at a later stage, the KMS can send a instruction to the data server to remove the key pair (e_{i2}, d_{i2}) at the server side.

Data Encryption

The encryption algorithm is shown in Figure 3.3 where each data denoted by D_x is associated with a set of searching keywords $\{W_1, W_2, ..., W_n\}_x$.

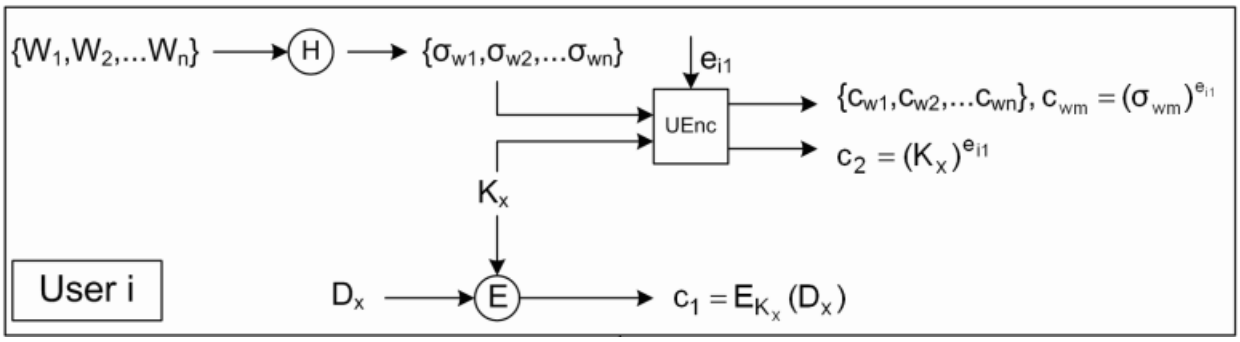

Figure 3.3: Data Encryption Scheme on the user side.

On the side of client i, the output of variables sent to the server is $\{c_1, c_2, \{c_{w1}, c_{w2}, ..., c_{wn}\}\}$, where

- c_1 is the ciphertext generated by encrypting D_x under the key K_x;
- c_2 is the result after encrypting the key K_x under the user's piece of RSA encryption key, $c_2 = (K_x)^{e_{i1}}$;
- For each search keyword W_m, the client uses a hash function H to compute σ_{wm} and computes $c_{wm} = (\sigma_{wm})^{e_{i1}}$.

After receiving the tuple, the server first computes $c_2^* = c_2^{e_{i2}}$ as on Figure 3.4.

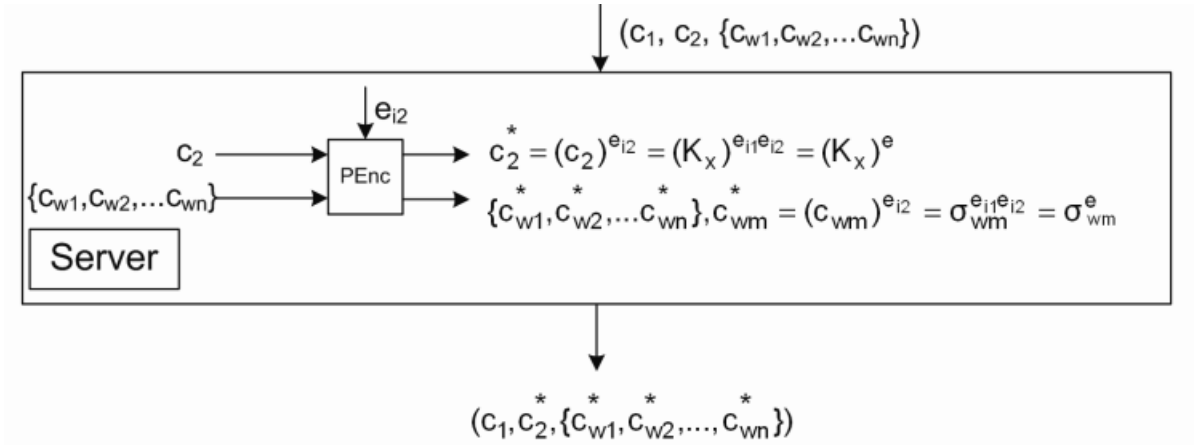

Figure 3.4: Data Encryption Scheme on the server side.

Similary, the final cipher stored on the server is a tuple $\{c_1, c_2^*, \{c_{w1}^*, c_{w2}^*, ..., c_{wn}^*\}\}$.

Keyword search

If a user j wants to retrieve all the documents on the server which contain a keyword W, j sends Q to the server, where $Q = \sigma^{e_{j1}}$ and $\sigma = H(W)$.

The server re-encrypts Q as $Q^* = Q^{e_{j2}}$, then search to see if there exists a c^*_{wm}, which is equal to Q^*. If there is matched result, the server computes $c'_2 = (c^*_2)^{d_{j2}}$ and sends back the result list of (c_1, c'_2) to client j.

Data Decryption

After receiving result set from the server, client j will decrypt it to get plaintext D_x. Simply, user j can calculate $(c'_2)^{d_{j1}} = (c^*_2)^d = (K_x)^{ed} = K_x$, and find D_x by decryting function $E()$. An unauthorised user cannot decrypt the data because the server does not have the corresponding proxy decryption key.

Other Considerations

After implementation, the author proved by experimental results that the scheme shows a good performance. The time spent on the matching operation is much more significant when the database becomes large. However, it rests the concern on a collusion attack, where a user colludes with adversary, who knows all the server side keys, they can easily recover the master keys by combining their keys. Therefore, it is needed more to investigate and develop a new index scheme for the multi-user system.

Conclusion and Future Work

In this paper, we presented our understandings on Apache Kafka and Zookeeper project. Besides, we also discuss security's concern in data storage outsourcing, as well as several researches considered as potential solutions for this problem.

All mentioned schemes can perform secured searches and updates on the encrypted data. For example, the RSA-based Proxy Encryption Scheme by Yang et al. [3] pointed out a method which does not require a shared key for multi-user access, each user in the system has a unique set of keys. However, a weakness of this scheme is that it allows statistical attacks on the queries. With the aim of researching and implementing a sufficient encrypted matching scheme to improve the security's efficiency in Zookeeper in Kafka system. Our expectation is to continue digging theoretical basis of Zookeeper's working mechanism in order to build algorithmic diagrams and apply them to the workflow of Zookeeper and Kafka servers.

For the second semester, we expect to complete the project as soon as possible, so we have established a chart corresponding to the process of our project as Figure 3.5 below:

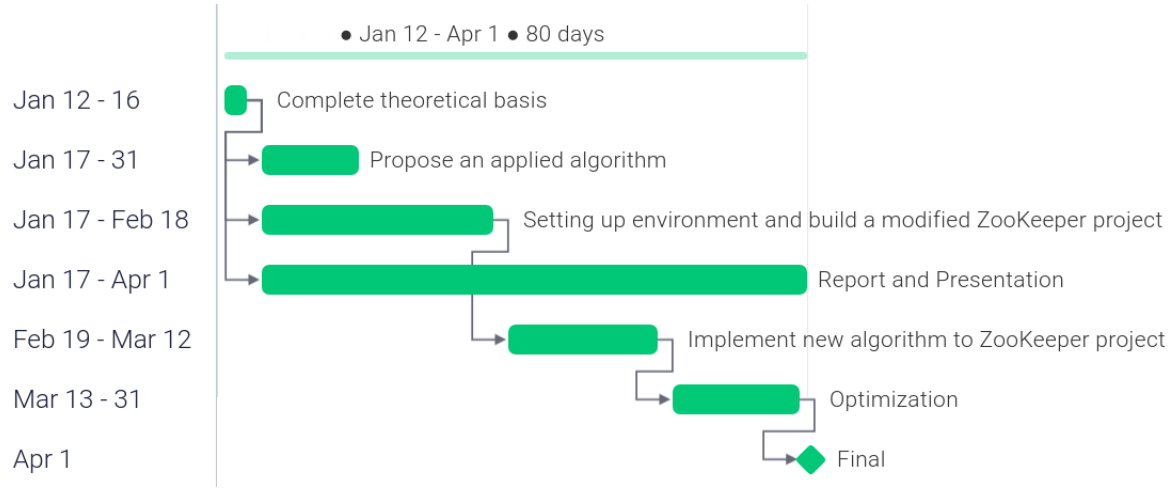

Figure 3.5: Plan chart for the second semester.

1. Completing theoretical basis
   This part includes researching more related articles and papers, along with figuring out a potential encrypted matching scheme for implementation on the next steps.
   Duration: Expected to finish in 5 days.

2. Proposing an applied matching algorithm
   This process refers to verify correctness and evaluate practical possibilities of researched algorithms.
   Duration: Expected to finish in 15 days.

3. Setting up environment and build a modified Zookeeper project
   Duration: Expected to finish in 1 days.

4. Implementing adapted algorithm to Zookeeper
   Duration: Expected to finish in 2 weeks.

5. Optimization
   Duration: Expected to finish in 18 days.

6. Making report and Presentation
   Duration: from start to delivery of the project.

Bibliography

[1] Neha Narkhede, Gwen Shapira and Todd Palino. *Kafka - The Definitive Guide*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., July 2017.

[2] Changyu Dong, Giovanni Russello, and Naranker Dulay. *Shared and Searchable Encrypted Data for Untrusted Servers*. Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2AZ, UK, 2008.

[3] Yang, Z., Zhong, S., Wright, R.N. *Privacy-Preserving Queries on Encrypted Data*. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 479–495. Springer, Heidelberg, 2006.

[4] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

[5] Blaze, M., Bleumer, G., Strauss. *Divertible Protocols and Atomic Proxy Cryptography*. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 127–144. Springer, Heidelberg, 1998.

[6] S. Goldwasser and M. Bellare. *Lecture notes on cryptography*. Summer CourseLecture Notes at MIT, 1999.

Websites:

[7] *Apache Kafka Overview*. Cloudera Runtime 7.1.1, 2019-12-18. https://docs.cloudera.com/runtime/7.1.1/kafka-overview/kafka-overview.pdf

[8] https://kafka.apache.org/uses

[9] https://data-flair.training/blogs/

[10] https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

[11] https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[12] https://data-flair.training/blogs/zookeeper-tutorial/

[13] https://zookeeper.apache.org/doc/r3.1.2/zookeeperOver.html

[14] https://hadoopabcd.wordpress.com/2015/05/02/zookeeper-distributed-coordination-service/

[15] https://techvidvan.com/tutorials/apache-zookeeper-tutorial/

[16] https://web.archive.org/web/20131209063307/http://wiki.apache.org/hadoop/ZooKeeper/PoweredBy

[17] https://engineering.fb.com/2018/07/19/data-infrastructure/location-aware-distribution-configuring-servers-at-scale/

[18] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/zookeeper-at-twitter.html

[19] https://lucene.apache.org/solr/guide/6_6/solrcloud.html

[20] https://data-flair.training/blogs/zookeeper-znodes/

[21] https://www.edureka.co/blog/zookeeper-tutorial/

[22] https://www.linkedin.com/pulse/role-apache-zookeeper-kafka-malini-shukla/

[23] https://medium.com/@rinu.gour123/role-of-apache-zookeeper-in-kafka-monitoring-configuration-c5bd1a7e4226

[24] https://www.tutorialspoint.com/zookeeper/zookeeper_workflow.htm

[25] https://www.tutorialspoint.com/zookeeper/zookeeper_quick_guide.htm