Multi-Tenant Apache Kafka for Hops

Kafka Topic-Based Multi-Tenancy and ACL-Based Authorization for Hops

MISGANU DESSALEGN MURUTS
Multi-Tenant Apache Kafka for Hops

Kafka Topic-Based Multi-Tenancy and ACL-Based Authorization for Hops

Misganu Dessalegn Muruts

Master of Science Thesis
Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

15 November 2016

Examiner: Dr. Jim Dowling
Supervisor: Gautier Berthou

TRITA Number: TRITA-ICT-EX-2016:120
Abstract

Apache Kafka is a distributed, high throughput and fault-tolerant publish/subscribe messaging system in the Hadoop ecosystem. It is used as a distributed data streaming and processing platform. Kafka topics are the units of message feeds in the Kafka cluster. Kafka producer publishes messages into these topics and a Kafka consumer subscribes to topics to pull those messages. With the increased usage of Kafka in the data infrastructure of many companies, there are many Kafka clients that publish and consume messages to/from the Kafka topics. In fact, these client operations can be malicious. To mitigate this risk, clients must authenticate themselves and their operation must be authorized before they can access to a given topic. Nowadays, Kafka ships with a pluggable Authorizer interface to implement access control list (ACL) based authorization for client operation. Kafka users can implement the interface differently to satisfy their security requirements. SimpleACLAuthorizer is the out-of-box implementation of the interface and uses a Zookeeper for ACLs storage.

HopsWorks, based on Hops - a next generation Hadoop distribution, provides support for project-based multi-tenancy, where projects are fully isolated at the level of the Hadoop Filesystem and YARN. In this project, we added Kafka topic-based multi-tenancy in Hops projects. Kafka topic is created from inside Hops project and persisted both at the Zookeeper and the NDBCluster. Persisting a topic into a database enabled us for topic sharing across projects. ACLs are added to Kafka topics and are persisted only into the database. Client access to Kafka topics is authorized based on these ACLs. ACLs are added, updated, listed and/or removed from the HopsWorks WebUI. HopsACLAuthorizer, a Hops implementation of the Authorizer interface, authorizes Kafka client operations using the ACLs in the database. The Apache Avro schema registry for topics enabled the producer and consumer to better integrate by transferring a pre-established message format. The result of this project is the first Hadoop distribution that supports Kafka multi-tenancy.

Keywords: Hadoop, Kafka, Hops, HopsWorks, Multi-Tenancy, Kafka Topics, Schema Registry, Messaging Systems, ACL Authorization
Acknowledgements

I would like to express my profound gratitude to Dr. Jim Dowling, for his continuous support during my thesis period. His open-minded approach to new ideas, insightful and tactful feedback to technical disagreement were the invaluable assets to the success of the project. Moreover, learning from him was the thrilling experience that I would not pass without singling it out.

My advisor, Gauthier Berthou, was also of much help to get the goals of the thesis achieved. His door was always open when I needed to talk to him. I am also indebted to recognize the substantial assistance from Ermias, Theofilos and the remaining Hops-team members. Moreover, I enjoyed their friendship and, of course, the Friday fika.

My extended appreciation goes to the Swedish Institute (SI) which sponsored my two years master study in Sweden. My contract with SI did not terminate with my master completion. I believe the support has laid a foundation for my future career.

Last but not least, my heartfelt gratitude goes to my family for standing by me to share my dream. The unfailing support they offer and the enduring confidence they bestowed on me keeps constantly energizing me.
## Contents

1 **Introduction** 1  
1.1 Background 1  
1.2 Problem 3  
1.3 Purpose 3  
1.3.1 Benefits, Ethics and Sustainability 4  
1.4 Delimitation 4  
1.5 Outline 4  

2 **Background** 7  
2.1 What is Messaging 7  
2.2 Kafka Architecture 8  
2.3 Kafka Main Characteristics 11  
2.4 Kafka and other Messaging Services 12  
2.5 Apache Kafka Use Cases 12  
2.6 About SICS Hops Project 13  
2.7 Multi-Tenant Architecture 14  
2.7.1 Multi-Tenancy in Cloud Computing 14  
2.7.2 Multi-Tenancy in Database 14  

3 **Related Works** 17  
3.1 Kafka Security 17  
3.2 ACL Based Authorization 17  
3.2.1 Kafka SimpleAclAuthorizer 18  
3.2.2 DefaultPrincipalBuilder 19  
3.2.3 Other Implementations 19  
3.3 Schema Registry 20  
3.4 Hops Project based Multi-tenancy 21  

4 **Methodology** 23  
4.1 Goal 23  
4.2 Solution 23
5 Analysis
   5.1 Kafka Topic ............................................. 25
      5.1.1 Existing Apache Kafka Topic Operations ............. 25
      5.1.2 Hops New Topic Operations .......................... 27
   5.2 Topic Availability Invariant ............................... 27
      5.2.1 Topic Availability States ............................ 28
      5.2.2 Why Always to State Four ......................... 28
   5.3 Why Topic Sharing ..................................... 29
   5.4 Hops Schema Registry .................................... 29
      5.4.1 Kafka Clients Access to Avro Schema ............... 30
      5.4.2 Avro Schema Compatibility and Evolution ............ 31
   5.5 Synchronizing Zookeeper and Database for Kafka Topics .... 31
      5.5.1 Topic Synchronization Failure Scenarios .......... 32
      5.5.2 Possible Synchronization Approaches ............... 33
   5.6 Hops ACL Definition .................................... 35
      5.6.1 The Role Concept ................................... 37
      5.6.2 Fine-grained ACLs and Role-base ACLs .............. 37
   5.7 HopsAclAuthorizer Class .................................. 39
      5.7.1 HopsPrincipalBuilder Class ........................ 40
      5.7.2 When is a topic operation authorized .............. 40

6 Test Configurations ................................... 43
   6.1 Kafka Broker Configurations ............................. 43
   6.2 Zookeeper Configuration .................................. 44
   6.3 Configuring Kafka Clients ............................... 44
   6.4 Local Resources ....................................... 45
   6.5 Kafka Util Library ..................................... 45
   6.6 Spark job ............................................... 46

7 Conclusions, Recommendations and Future Works ............. 47
   7.1 Conclusions and Recommendations ...................... 47
   7.2 Future Works ......................................... 48

Bibliography ................................................. 49
List of Figures

2.1 Forms of messaging .......................................................... 9
2.2 Kafka Architecture [1] ..................................................... 9
2.3 Kafka consumer group abstraction [2] .............................. 11
2.4 Shared database, shared schema [3] ................................. 15
# List of Tables

2.1 Degree of multi-tenancy in cloud computing .......................... 14
5.1 Sample Hops ACL definitions ............................................. 38
5.2 Hops ACL definition levels ................................................ 39
# List of Listings

<table>
<thead>
<tr>
<th>3.1</th>
<th>Schema example</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Sample Kafka broker configuration</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Acronyms and Abbreviations

This document requires readers to be familiar with terms and concepts of distributed systems, cloud computing multi-tenancy architecture and data access authorizations. For clarity, we summarize some of these terms and concepts in the above context.

**ACL**  
Accesses Control List

**CA**  
Certificate Authority

**Hops**  
Hadoop Open Platform-as-a-Service

**IaaS**  
Infrastructure as a Service

**IDL**  
Interactive Data Language

**JSON**  
JavaScript Object Notation

**KIP**  
Kafka Improvement Proposals

**OSDM**  
Open Source Development Model

**PaaS**  
Platform as a Service

**REST**  
Representational State Transfer

**SaaS**  
Software as a Service

**SASL**  
Simple Authentication and Security Layer

**SLA**  
Service Level Agreement

**SSL**  
Secure Socket Layer

**TLS**  
Transport Layer Security
Chapter 1

Introduction

Nowadays, the amount of data either flowing on the Internet or enterprises have on their storage is large and complex. Such data is often generated and/or collected from sensors and mobile devices, social media user activities, traditional enterprises, e-commerce customer activities and others. This collected data can be structured, semi-structured or totally unstructured making it challenging to manage, analyze and generate insights from it.

Big data refers to a huge volume of data that cannot be analyzed and stored using the traditional methods and tools of data processing. The challenges in handling Big Data are defined by three of the Big Data Systems properties, often termed as 3’Vs [4, 5]. It usually has one or more of them - high volume, high velocity and/or high variety.

- **Volume** - refers to the size of data. Data volume can be growing rapidly and endlessly despite the limited capacity of the data processing resources.

- **Variety** - refers to the various data source types and the various data formats. Data may come from business transactions, social media user activities and IoT sensors in different formats (e.g. text documents, audio, video, financial transactions and others).

- **Velocity** - refers to how quickly data is generated, delivered, stored and retrieved. Data can be in batch form, can arrive periodically or in real-time.

Other rarely mentioned 2’Vs are also added as characteristics of Big Data.

- **Veracity** - refers to the uncertainty or inaccuracy of the data as the data may come from unknown and ever-changing sources. Data filtering, structuring will be needed to keep data consistency [4, 5].
Value - refers to the quality of data stored and its commercial importance [6]. Storing data is a waste of storage resources if that data cannot be analyzed to generate meaningful insights and used for better business operations.

Big Data, in terms of its properties, is explained as a volume of data with high acquisition velocity that has different data representations. Storing data is a waste of storage resources unless its opportunities are harnessed and used to empower organizations, by providing faster and better decision making and better customer experiences, through otherwise inaccessible data patterns and insights. Such volume of data inundates businesses on a day-to-day basis, making it difficult to analyze using traditional relational approaches [7]. Continuous researches by industries and academia to maximize opportunities that can be gained from big data is giving way for the emerging data driven-economy.

Echoing the motto of ‘data is the future oil’, big companies have been investing on Big Data Analytic and making use of the data they have. Due to their limited storing and processing capacities for big data, traditional relational databases are becoming like things of the past. Even the highly centralized enterprise data centers are incapable of processing a huge volume of data. Big data is distributed data. Big Data Analytic is the modern approach to examine large data and uncover hidden patterns in it giving insights. In other words, it is the application of advanced analysis techniques against a large data stored in a distributed environment.

Apache Hadoop has become the de facto Big Data Analytic platform. It is an open-source framework for distributed storage and distributed processing of a large amount of data using a cluster of commodity hardware. It does not need a custom designed computing hardware. It rather uses the low performance and easily affordable computing hardware. It has many distributions - Cloudera, Hortonworks and MapR being the most widely used distributions.

The distributed nature of data in Big Data systems comes with its own challenges and opportunities. Balancing the storing and processing loads among dispersed commodity computers is one of the benefits that enabled to analyze Big Data. On the other hand, communication among the distributed processing components and coordinating them are among the challenges that should be addressed. These components can communicate by messaging - a mechanism that enables computer processes to send/receive signals or data among themselves. There are different messaging platforms used in different systems. But Apache Kafka is usually used with Hadoop and it is the focus of the thesis work.
1.1 Background

In recent years, Apache Kafka has become the de facto high throughput messaging system within the Hadoop ecosystem. Popular architectures for handling data-in-motion, such as Dataflow and Lambda architectures, are built using Kafka. Kafka support for Hadoop is still at the early beta stage despite Kafka’s importance to Hadoop. Prior to Kafka version 0.9, it had a few consideration to data security. The Kafka community introduced a security feature only after this version.

At KTH/SICS, we are developing the Hadoop Open Platform-as-a-Service (Hops), a next generation architecture for Hadoop that improves scalability and security. Hops does not yet support Kafka. In this project, we will integrate a multi-tenant Kafka service into HopsWorks.

1.2 Problem

Apache Kafka is one of the important components of a company’s data infrastructure. As a messaging service, it is used to store some of the critical data in a cluster of servers. Limiting access to the data, only to company internal employees will not guarantee that the data is not misused. Data should be secured from both internal and external users’ threats. Even internal users have different duties and data requirements. So, the permission to access this data should be based on the data that the user wants for legitimate reasons.

The challenge is how to secure this crucial data from possible internal and external threats that may otherwise compromise company data?

1.3 Purpose

This project aims at building a multi-tenant Kafka cluster for HopsWorks. This Kafka cluster will support ACL-based user authorization. By ACL based user authorization, it means that after authenticating Kafka client connections, their requests to access secured resources will be authorized using access control lists defined for those resources. The main purposes of this project includes the following parts:

- **Multi-tenant Kafka for HopsWorks**: HopsWorks has the concept of projects. A user should be either a project owner or project member to perform any job on HopsWorks. For jobs that require access to Kafka topics, a user will be able to create and access topics within the project. Topic owner will also be able to share the topic across projects if needed.
• **Hops schema registry**: a Kafka topic will have associated Apache Avro schema for data integrity. Schema registry will be implemented to store into and retrieve from a schema.

• **ACL-based authorization**: a topic will have associated user access control lists (ACLs) defining which user performs what on it. Hops ACL-based authorizer implementation will authorize user requests based on the topic ACLs persisted into a database.

### 1.3.1 Benefits, Ethics and Sustainability

Data security and privacy is essential for the very existence companies that own classified data. Every company needs a secure way of data handling either during communication or storage. Often times, data in a network is at higher risk as it is exposed to a wide range of users compared to data in local storage. Securing such data is crucial for the successful functioning of enterprises because exposed data can reach to the extent of endangering their existence.

Integrating Kafka into HopsWorks can introduce security holes unless appropriately configured. For instance, by sharing a topic across projects and allowing topic operations to project members, the topic resident data is open to all users which might not be desired in an environment where there is a classified data. Access to such data should be authorized so that intruders are blocked from causing security and privacy problems. Even though Kafka is scalable and high throughput messaging system, which may discourage the concept of topic sharing across projects, topic sharing is still essential for the efficient use of resources.

### 1.4 Delimitation

This work depends on many other previous works. It mainly uses the Apache Kafka project and the Hops. It does not change the Kafka project though. It only changes the implementation of the pluggable authorizer interface provided by the project. Basing on the Hops feature that it provides project as a service, and the functionality that SSL keys and certificates can be generated both for Hops project members and Kafka brokers on demand, this thesis work will add a multi-tenant Kafka to HopsWorks.

### 1.5 Outline

As introduced above, this paper discusses on how to integrate a multi-tenant Kafka into HopsWorks. Chapter 2 goes through a deeper discussion of the Apache
1.5. Outline

Kafka and its main concepts relevant to the multi-tenancy feature followed by chapter 3 that briefs the related previous works attempted to provide secure Kafka cluster, authorization of client requests to the cluster and schema registries. Chapter 4 discusses the methodology used, the goals and the proposed solution of this project. The solution is analyzed thoroughly in chapter 5 that covers the detailed thesis works. Chapter 6 goes through the test configurations and test files developed to test the multi-tenant Kafka. Finally, chapter 7 concludes the paper by providing conclusions, recommendations and future works.
Chapter 2

Background

Kafka is a distributed fault-tolerant, high throughput, partitioned, replicated publish/subscribe messaging system, originally developed at LinkedIn and now open sourced under the Apache License. This chapter explores Apache Kafka with detail discussions to its architecture, design principles and characteristics. It also discusses multi-tenancy architecture both in cloud computing and database domains.

2.1 What is Messaging

Messaging is a way to allow computer processes to communicate with one another by sending messages along a communication medium. These messages can contain information about event notifications or service requests. If both the sender and the receiver processes have almost similar speeds and/or processes exchange messages pairwise, a direct connection between the communicating parties can provide the messaging medium. This is a point to point messaging, as illustrated in figure 2.1a below.

However, when either the sender process has higher speed than the receiver process, and/or the sender process wants to send the message to multiple receivers and/or the receiver process wants to receive messages from multiple senders, a messaging service that implements standard messaging concepts such as batching, broadcasting, ordering is needed. An implementation of such concepts meddles the communication between the end parties. The hypothetical messaging system depicted in figure 2.1b, is only for the illustration of the later scenario. Otherwise, it is an opaque messaging system that does not implement any messaging concepts or semantics for that matter.

These two modes of messaging are briefed here to spring into Apache Kafka which implements the later mode. The thorough discussion of these modes and
their comparisons set aside, one notable difference between them is that the first mode supports only synchronous communications while the later can support both synchronous and asynchronous communications. Additionally, a messaging system can be seen as a level of indirection entity in between to decouple message senders from recipients. One advantage of decoupling message recipients from messages senders is that they do not need to know about each other.

Messaging services can be implemented using either publish/subscribe based or queue based model [8, 9]. In a queue based implementation, senders push messages into queues and receivers read those messages. Messages can either be pulled by receivers or pushed by the queue depending on the implementation of the queue. In some implementations, only a single sender can push messages to a queue and a single receiver can read messages from the queue. In this approach, the queue is used as a message buffer to balance the sender and receiver processes speeds. This is simply a point-to-point link with a buffering capability to support asynchronous messaging. If multiple receivers are reading from the queue, however, each receiver reads a portion of the messages on the queue as each message has to be read by only one consumer. To achieve this, the queue should implement a mechanism that identifies which messages should be sent to which receivers and how, either in a round-robin fashion or following another algorithm.

In a publish/subscribe (pub/sub in short) based messaging system implementation, there are sets of topic channels to which senders push messages and from which a message is broadcast to all receivers subscribed to receive messages from the topic. The queue based implementation is a degenerate case of pub/sub where there is only one topic channel, one sender and one receiver.

Kafka implemented as a pub/sub system abstracts the queue-based approach also as discussed in section 2.2 below.

### 2.2 Kafka Architecture

Apache Kafka is an advanced messaging system where its operational architecture looks like figure 2.2. It consists of the following main components: producer, consumer, broker, Zookeeper. It supports a plug-able authorizer module.

- **Message**: is the transferable unit of data in Kafka. It is an array of bytes.

- **Broker**: Kafka is a cluster of servers each called a broker. Each broker needs a separate configuration. Kafka broker configuration is specified in the server.properties file, with unique configuration values for the properties: broker.id, port and log.dir.
2.2. Kafka Architecture

(a) Point to point messaging

(b) Pub/sub messaging service

Figure 2.1: Forms of messaging

Figure 2.2: Kafka Architecture [1]

- **Topic**: an abstraction which a message is pushed into and consumed from. A topic is partitioned and replicated, and for each topic partition there is a leader broker and other followers if required.

- **Producer**: a Kafka client process that publishes messages into topics of its choice. It has the ability to choose which message to publish into which topic partitions. A topic partition will be discussed later.

- **Consumer**: a Kafka client process that consumes messages from topics. Each consumer instance labels itself with a consumer group identification.

- **Apache Zookeeper**: a high performance distributed coordination service with a hierarchical namespace. The main Zookeeper roles in Apache Kafka are:
1. Coordination: managing cluster membership and electing a leader, watching broker failure and replacing it when it happens, providing node watchers others.

2. Metadata storage: it stores the topic information such as topic replication and partition information, resource ACLs, and maintains consumer offsets. A description how Zookeeper stores the metadata is discussed as Kafka data structure in Zookeeper [10].

3. It also provides quotas for clients as of Apache Kafka version 0.9.

For each topic, Kafka cluster maintains a configurable number of partitions. A partition is an ordered sequence of messages and continually appended to. Each message is identified by an offset and retained for a configurable number of days.

Kafka provides a single consumer abstraction called consumer-group that generalizes both the queue based and pub/sub implementations, see figure 2.3. Consumers label themselves within a consumer group, and each message is read by a single consumer in the consumer group [8]. This abstraction allows a group of consumers to divide up the work of consuming and processing Kafka messages. Each consumer group is subscribed to every topic partition, which effectively broadcasts every message to all consumer groups. But within a consumer group, only a single instance of consumer processes the delivered message. If each consumer group consists of only a single consumer, Kafka becomes a publish/subscribe system where every message is broadcast to a single instance consumer group and processed by it. If all consumer instances belong to a single consumer group, Kafka degenerates to a queue system where each message is read by a single consumer instance and processed by a single consumer instance within the group. The Consumer group is the main concept behind the horizontal scalability feature of Kafka. It allows a pool of processes to divide up the work of consuming and processing records [2], it will be discussed in section 2.3 below.

The number of topic partitions determines the number of consumer instances in a consumer group that can process messages in parallel. For a topic that has N number of partitions, it can support up to a maximum of N consumers. If there are fewer partitions than the number of consumers in a consumer group, the extra consumers will be kept idle. If there are more partitions than consumers, to the contrary, a single consumer instance may consume from more than one partitions. Generally, more number of partitions leads to high throughput. However, availability and latency requirements might be compromised as more partitions introduce more open file handles in brokers, end-to-end latency, and more memory requirements in the clients [11]. Each partition is replicated into a configurable number of replica servers. One of the replica servers acts as a leader
for that partition and the remaining servers follow the leader. In cases the leader fails, one of the followers is promoted to be a new leader of the partition.

Figure 2.3: Kafka consumer group abstraction [2]

2.3 Kafka Main Characteristics

Kafka, often rethought as a distributed transaction log, was developed to provide messaging services. A few of the basic design principles are listed below.

- **Scalability** - Kafka scales horizontally by adding more commodity nodes on the fly. So, the cluster size can grow/shrink dynamically without cluster downtime. Deploying more commodity nodes than a single high-performance server increases the number of I/O operations that can handle a huge volume of data. For instance, N times partitioned topic can optimally support N consumers whether the deployment is a single node or a cluster of nodes. However, due to the limited number of I/O operations on a single server, the consuming processes will be fast if the topic partitions are distributed in a cluster of nodes. Scaling provides a high throughput to both producers and consumers by distributing the load to the whole cluster.

- **Durability** - Unlike other messaging systems, Kafka persists messages onto a disk with a strong durability guarantee. The advantage of persisting messages onto disk is twofold. It ensures that data on a disk is not lost when a broker gets down. It also aggregates the data for batch processing.
• **Fault Tolerance** - Kafka can survive N-1 node failures in a cluster of N nodes if the message configuration is properly configured. Each topic partition is replicated into a configurable number of replicas and the leader for that partition keeps a list of in-sync-replicas. This list follows the leader so that each node in the list reflects the leader states. In case the leader broker is down, one of the in-sync-replicas will be elected as new leader of the partition and Kafka keeps serving. This fail-over process is transparent to the application.

### 2.4 Kafka and other Messaging Services

Unlike other messaging systems like RabbitMQ and Apache ActiveMQ, Kafka provides the following services [12, 13, 14].

- It scales out horizontally very easily by adding more commodity servers. The main approach to scaling out in Kafka is by adding more consumers to a consumer group.
- It provides high throughput for both publishing and consuming.
- It persists messages onto disks. This feature enables Kafka to process both batch and streaming data.
- It automatically balances consumers during failure. Each consumer in a consumer group processes different messages. If one of them fails, the consumer re-balance algorithm should run so that a new consumer will be assigned to each partition whose messages were processed by the failed consumer.

### 2.5 Apache Kafka Use Cases

Apache Kafka is now a reliable and mature project deployed in the data pipelines of many industry leaders such as LinkedIn, Spotify, Yahoo, Netflix, Paypal and others [15]. Some of the use cases [8, 16, 17] where these companies use Apache Kafka for are listed below.

- **Messaging** - it is used as a messaging broker, decoupling consumers from producers in large scale message processing applications. It can also be used to replicate database. The original database will act as Kafka producer and the destination database will act as Kafka consumer. In addition, DBAs can use Kafka to decouple apps from databases.
• **Website activity tracking** - it can be used to track user website activities (page view, search) or user social media activities (shares, posts, comments, twits etc), user preferences on music streaming applications (favorite song, playlist etc), customer preferences on e-commerce services (item preferences, customer demographic groups etc) and other industry specific activities.

• **Time-based Message Storage** - it can be used to store email, chat, and SMS notifications.

• **Dataset sharing** - Kafka can also be used to share datasets across projects in Hops cluster.

### 2.6 About SICS Hops Project

Hops is a next generation distribution of Apache Hadoop, developed at KTH/SICS. There are many other Apache Hadoop distributions, with three of the top distributions provided by, Cloudera, Hortonworks and MapR. Each of these distributions has improved some key features of the original Hadoop. Hops, like the other distributions, introduces its own innovative features, with the main focus on scalability, high availability and customizable metadata architecture for Hadoop.

Hops supports Hadoop-as-a-Service, project-based multi-tenancy, secure sharing of datasets across projects and other services. Its WebUI is called HopsWorks. HopsWorks has both user and administrator views for Hops. It is used to manage projects, search and visualize datasets, run jobs and interactively analyze data. It is built on the concept of projects. Under normal system environments, a Hops user creates an isolated project at the HopsWorks WebUI and this user is responsible for managing all the privileged actions on the project. These actions include adding/removing users to/from the project members list, sharing datasets with other projects and other services. It also provides email-based user authentication and optional two-factor authentication.

In HopsWorks, project members can be added/removed or changed their project roles dynamically. There are two types of roles, *DATA OWNER* and *DATA SCIENTIST*, each with its own clear actions.

A first release Hops documentation, Hops 2.4, is available on the Hops website, [www.hops.io](http://www.hops.io).
2.7 Multi-Tenant Architecture

Multi-tenancy is an architecture where a shared environment serves multiple customers cost-effectively and securely. In such systems, a tenant is a business client served by the shared environment.

Multi-tenancy can be provided at the infrastructure level or application level with varying degrees of sets of shared resources. The main hassle in such systems is tenant data isolation. It can be realized using either virtualization or ACL-based access. Another architecture called the single-tenant is an architecture where, unlike in multi-tenancy, each customer gets dedicated and isolated sets of resources to fulfill its needs [18]. A good comparison of single-tenant and multi-tenant architectures is provided in [19].

2.7.1 Multi-Tenancy in Cloud Computing

Resource pooling, considered as one of the five cloud computing essential characteristics [20], is achieved using multi-tenancy. Multi-tenancy in the cloud can be provisioned at the following three cloud computing service models.

- **IaaS**: multiple tenants share infrastructures like servers and storage devices. In IaaS multi-tenancy is achieved using virtual machines.
- **PaaS**: multiple tenants may share the same operating system.
- **SaaS**: multiple tenants share a single application instance and database storage. Tenant data isolation is achieved using tenantId.

How much of the SaaS, PaaS or IaaS is shared among multiple tenants can also be used to describe the degree of multi-tenancy in cloud computing. The table 2.1 shows the different degrees of multi-tenancy.

2.7.2 Multi-Tenancy in Database

Multi-tenancy realization in database architecture depends on the type of application the database is attached to. Hence, multi-tenancy can be enforced at various
levels depending on the technical and business considerations of both the database providers and customers. The transition from the most isolated data architecture to the most shared data architecture is a continuum of different possible architectures. The most common architectures [21, 22] are briefed below.

- **Separate databases**: each tenant has a separate database. This approach is a horizontally scaled single-tenant architecture where each database instance belongs to a single-tenant.

- **Shared database, separate schema**: multiple tenants use the same database, but each tenant has separate tables grouped by a tenant specific schema.

- **Shared database, shared schema**: multiple tenants’ data is stored in the same database tables. In this approach, each record of the tables is uniquely identified by tenantId and customer data isolation is attained by this id. Figure 2.4 is a sample database table to illustrate how data of multiple tenants co-exist in shared database and shared schema architecture.

![Table Example](image)

**Figure 2.4: Shared database, shared schema [3]**

As stated previously, the choice of which architecture level to deploy depends on the customer SLAs (like data isolation and security, efficiency, speed and other requirements) and the business, architectural and operational models of the database provider [23].
Chapter 3

Related Works

This chapter summarizes the related works that have been done in the areas of ACL-based authorization in Apache Kafka and multi-tenancy implementations under different abstractions like databases, projects and others.

3.1 Kafka Security

Apache Kafka has become one of the crucial components of a company’s data infrastructure. As opposed to its widespread usage, Kafka cluster had no built-in security features in all versions prior to 0.9.0.0, making the whole Kafka cluster accessible for any user [24]. But after the release of version 0.9.0.0, Kafka supports [8] authentication of the following communications using either TLS/SSL or SASL - Kafka clients with brokers, inter-broker interactions, and Kafka brokers with Zookeeper. It also supports the authorization of client operations by the Kafka brokers. Kafka brokers and clients should be configured with SSL keys and certificates to authenticate their communications.

3.2 ACL Based Authorization

Kafka does not examine the message contents to check if they are not tampered, delayed or malicious from the origin. However to authenticate which it is communicating to and to authorize what this entity can perform on it, Kafka 0.9.0.0 has introduced an expandable authorization interface, that can be used to implement the core authorization requirements. This interface can also be used to implement advanced and enterprise-level security features. During Kafka service start up, Kafka server reads the value of authorizer.class property from the server.properties file and instantiates an instance of this class. Then, every
client requests to the Kafka cluster will be routed via this implementation and each operation will be authorized.

The interface is documented nicely and is still open for updates under the KIP-11 [25]. It basically relies on Linux-like Access Control Lists (ACLs). The general format for these ACLs is: "Principal P is [Allowed/Denied] Operation O From Host H On Resource R" [8]. Principal P is the client (user) accessing the cluster. The Resource R is either the Kafka topic or the Kafka cluster or consumer-group which the consumer client belongs to. The possible operations are READ, WRITE, CREATE, DELETE, ALTER, CLUSTER_ACTION and DESCRIBE. As can be seen, some operations are resource type specific. For instance, CLUSTER_ACTION applies to Kafka cluster resource type whereas WRITE operation applies to a topic resource type. Apache Kafka also has a PrincipalBuilder interface which is implemented to return the customized transport layer’s peer principal which the Kafka broker is communicating with. As explained under the Kafka documentation [8], the default SSL user name will be of the form:

\[CN=writeuser, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown\]

where the 'Unknown' values are the parameter values entered during the SSL certificate creation. An implementation of this interface extracts the principal from this SSL user and this principal is authorized for the operations it is requesting based on the resource ACL definitions. The customized implementation should be specified in the broker server.properties as:

\[principal.builder.class=CustomizedPrincipalBuilderInterface\]

This section discusses the different implementations of the above two interfaces and their possible drawbacks.

### 3.2.1 Kafka SimpleAclAuthorizer

This is the default and out-of-the-box implementation of the Authorizer interface. Unless a customized implementation is specified, this implementation is used by Kafka. It keeps the general format of the Authorizer ACLs and stores the resource ACL definitions in the Zookeeper under /kafka-acls/<resource-type>/<resource-name> using a JSON format.

In this implementation, a principal can be either user or Kafka producer/consumer. Under this implementation, the following concepts are worth mentioning [8, 25].

- **Default behavior:** if a resource R has no corresponding ACLs, then no one is authorized to use the resource R except super users. Super users is a list of users that are allowed to access all resources in the cluster without authorization. Any user who wants to access every resource in the cluster
without authorization should be added to this list in the server.properties file. Super users are listed in the server.properties as

\[\text{super.users=\text{User:Bob;User:Alice}}\]

where Bob and Alice are the SSL user names. The default behavior can be reversed by setting `allow.everyone.if.no.acl.found` to true in the server.properties file. This configuration allows all requests to Resource R if it does not have an ACL definition.

- **ACL cache**: it keeps a cache of ACLs with a TTL of 1 hour. The cache is initially loaded during authorizer configuration and updated every 5 seconds. As tracked in JIRA Kafka/KAFKA-3328, the possibility of ACL definitions loss during frequent add/remove ACL operations on a cluster where there are multiple instances of the Authorizer has been resolved by making the authorizer read/write state from Zookeeper and only leveraging the cache for get/authorize requests.

- **Unfinished issues**: JIRA issue Kafka/KAFKA-3507 has opened the task of defining a standard exception for the Authorizer interface. JIRA Kafka/KAFKA-3266, also tracks the additional ACL operations list ACLs, alter ACL. These two JIRA issues are still unresolved.

- **Priority rules**: deny operation is given higher priority over allow operation permission type. For instance, if there is one ACL definition that allows all operations for all users and another ACL definition that denies a specific user for all operations, then this user will be denied for any operation.

- **Implicit ACLs**: some ACL definitions have implicit ACL definitions. For instance, a principal that has READ or WRITE permission also has the DESCRIBE permission unless there is an explicit ACL that denies DESCRIBE operation.

### 3.2.2 DefaultPrincipalBuilder

This is the Apache Kafka default implementation of the PrincipalBuilder interface that returns the transport layer’s peer principal. This principal is the Kafka client that should be authorized for the operation is requests.

### 3.2.3 Other Implementations

There are multiple messaging hubs developed on top of Apache Kafka. The IBM’s Bluemix cloud platform, Apache Sentry, RBAC, SentryKafkaAuthorizer are some
of those implementations.
Apache Sentry provides a role-based authorization control (RBAC) for Hadoop. RBAC is a powerful mechanism to provide secure access to authorized users. In this approach new data is added/removed, new users are added/removed, users are assigned/unassigned roles, roles are assigned access control list for resources.

There is also the idea of SentryKafkaAuthorizer [26] to use Apache Sentry for role based authorization in Kafka, but it is not yet implemented.

3.3 Schema Registry

As far as Kafka is concerned, the types of messages in transit or in memory need not have a specific format and/or meaning. However, for better integration of the communicating client processes, it is recommended that an additional structure and semantics be attached to a message.

The simplest way to achieve this is by using the language specific built-in serializations, like Java serialization, Ruby’s marshal and Python’s pickle. The second way to impose a structure to Kafka messages is by adding schema to the contents of the messages. A schema is composed of primitive types (e.g int, string) and complex types (e.g. array, union). A schema definition should at least contain the type, name and possibly schema properties.

There are different options to provide schema to messages starting from the simple and language-agnostic formats like JSON and XML, to the more advanced and cross-language serialization formats like Apache Avro, Google Protocol buffer, Apache Thrift and others. One interesting feature of the advanced formats is, they support schema evolution so that processes with new schema versions can decode messages encoded with old schema versions.

Each of these messaging formats has its own use cases depending on the efficiency, ease of use, support in different programming languages. In principle, all formats can be used by the same application at the same time though the best practice is to stick to a single format. Many Kafka developers, favor the open source project Apache Avro for it: is fast and very compact, has a direct mapping to/from JSON and other features [27]. Apache Avro is represented using either a JSON schema or an interactive data language (IDL) [28]. It uses schema evolution rules to enforce backward compatibility when a schema change is inevitable [29, 30, 31].
3.4. HOPS PROJECT BASED MULTI-TENANCY

Listing 3.1: Schema example

```json
# name = name of the schema
# type = type of schema
# fields = array of courses
{
    "name": "student_name",
    "type": "record",
    "fields": [
        { "name": "course_name", "type": "string" },
        { "name": "course_code", "type": "int" },
        { "name": "credit", "type": "int" }
    ]
}
```

Listing 3.1 is a simple schema example that represents a student record with courses he/she is registered for.

The Confluent platform (www.confluent.io) has a schema registry server that provides REST endpoints for registering and retrieving Avro schemas [32]. Confluent schema registry runs as a service and allows many schema CRUD (create, read, update and delete) operations. It also enforces the following schema compatibility rules when a new schema is registered. By default, it provides backward compatibility, but it can be configured/ altered to provide either of them.

- **Backward compatibility** (default): a new schema is able to decode messages encoded with all previous schemas.
- **Forward compatibility**: all previous schemas can decode data encoded with the new schema.
- **Full compatibility**: a schema is both backward and forward compatible.
- **No compatibility**: the new schema should be a valid Avro schema.

Kafka ecosystem at LinkedIn also has a separate and replicated schema registry server[33]. Each producer writes Avro messages, registers the Avro schema in the schema registry and embeds a schema-ID in each encoded message. The consumer, in turn, fetches the schema corresponding to the ID from the schema registry and decodes the message accordingly.

### 3.4 Hops Project based Multi-tenancy

As listed in section 2.6, Hops provides project-based multi-tenancy service. This service is achieved using the implementation of dynamic role-based access
control. In role-based access control, users are assigned to groups that are in turn assigned group capabilities. This is more efficient by managing user permissions at a group level than the user based access control that allows granular control by assigning fine-grained access controls to individuals.

In a Hops project, a user can have either of the two Hops project role types, *DATA OWNER* and *DATA SCIENTIST*. A user’s project role determines what actions a user can perform on the project datasets, project member list, project member role assignments [34]. For instance, as a project Data owner, a user can add/remove project members, assign roles to project members, create/delete project datasets and others. As a project Data Scientist, in turn, a user can run batch jobs, upload to a restricted dataset and others.
Chapter 4

Methodology

This chapter goes through the goals this thesis work aims at, the solution to the already identified problem and concludes by discussing the methodology used to implement the solution.

4.1 Goal

Apache Kafka developed as a messaging service for real-time data feeds, it did not originally support security and authorization features. It was deployed on trusted environments mainly focusing on the functionality, i.e. allowing all user requests without a fear of data security and privacy infringements. This assumption overlooked the threats to the environment both from the internal users and external data thieves. Internal users can legitimately compromise company data.

Our goal in this project is to solve this security gap. Upon successful completion of this project, enterprises will have the ability to fulfill the security requirements of their critical data stored in/passing through Kafka. Kafka servers will authorize all Kafka client requests using user ACLs defined for Kafka topics and persisted into a database. By adding user ACLs to topics, enterprises can secure their data from potential fraudulent users and legitimate internal users. These ACL definitions range from the most fine-grained permission model that applies for a specific user to the role-based permission model that applies to a group of users identified by user roles in a project.

4.2 Solution

This section provides the solution to implement a secure multi-tenant Kafka on HopsWorks.
Going through the features of the HopsWorks, one benefit is that it provides project as a service. When a user wants to use Hops, he/she should create a project from HopsWorks. It is from within a project that a user can run whatever big data applications that Hops supports. Similarly, if those user applications need Kafka as a service, they should run from within a project. Hence, a user can create Kafka topics from within a project.

These topics will be persisted into an NDB database and created on the Zookeeper. The user can share these topics across projects if needed. Since a project may have many members with different roles, the topic owner gives ACLs to each member of the projects based on the principle of least privilege. The topic owner can also define role-based ACL definitions. These ACLs are stored in the database. This approach builds a multi-tenant Kafka for HopsWorks.

The next phase is authenticating each user requests to the Kafka cluster. SSL keys and certificates are created for each Kafka broker and project members. These certificates should have same certificate authorities (CA). When the Kafka client applications access the cluster for a service, they will be configured with the user SSL keys and certificates. By using these security configurations, the Kafka broker authenticates the clients’ connection sessions. When there is a client request operation, the Kafka cluster extracts the user name of the client SSL key. A new implementation of the Authorizer interface, then, queries the database if there are ACL definitions to authorize the user request. External users will have no ACLs and are effectively unauthorized for any request. For internal users, each request is authorized/unauthorized using the ACL definitions. This approach solves the challenge discussed in section 1.2.

4.3 Methodology

After doing literature reviews on related implementations, the thesis work mainly focuses on developing the software that implements the solution. The source code is open source (OS).

The solution implements many functionalists, including, developing Kafka multi-tenancy, schema registry, and ACL-based authorization. The development process leverages the Open Source Development Model (OSDM). Under OSDM software development model [35, 36], the development process has a mobile structure where the arrival of a new feature or removal of an old feature does not affect the whole process. This model is also suitable for group collaborations and discussions. So, all tasks in the thesis works are treated as features. Each task is developed and tested separately in the order of multi-tenancy, schema registry and authorizer implementations. It is integrated into the main release when the feature is completed.
Chapter 5

Analysis

This chapter explains most of the thesis works, discussing the approaches followed to implement the topic-based multi-tenancy in HopsWorks, schema registry and implementation of the Authorizer interface.

5.1 Kafka Topic

Apache Kafka topic is the abstraction to which a message is continually written to and consumed from. The Kafka producer pushes messages into topic either synchronously or asynchronously depending on the durability and speed requirements of the applications. Kafka Consumers keep polling the Kafka topics consuming each message published. Kafka topic is one of the three resource types existing in Kafka. The other two resource types are Kafka cluster and consumer-group. In this project, the topic is the only Kafka resource type Kafka multi-tenancy is achieved for.

As Kafka relies on zookeeper in which topics have a global name space, each topic name must be unique. To ensure this property and the topic invariant definition discussed in section 5.2, we first persist the topic in Zookeeper and persist it to the database only once Zookeeper has accepted the project.

Kafka has a set of allowable operations on a topic. Due to the project abstraction introduced by HopsWorks and the associated project level operations, new topic operations are added to the existing Kafka topic operations. The list of all topic operations with a detailed discussion is provided below.

5.1.1 Existing Apache Kafka Topic Operations

The following topic operations are supported by the Apache Kafka. They are also preserved in this work even though the implementation of each operation is
modified to accommodate the Hops features.

- **Create topic**: creates a topic in the Kafka cluster with a default replication and partitioning factors. If the user specifies a replication factor, HopsWorks checks that this replication factor is not greater than the number of brokers currently running in the cluster. The partitioning factor is the unit of parallelism in Kafka, hence, a large number of partitions leads to high throughput [11]. On the other hand, having too many partitions in total or on a single broker is a hint to consider for service availability and latency requirements. It determines the number of consumers within a consumer-group that can access the topic concurrently. Having a large number of partitioning factor supports many consumers that can consume messages from the topic concurrently, hence speeding up the message consumption process. This will have performance problem if there are a few brokers as all the consumers will connect to them and they will be bottlenecks.

- **Topic details**: retrieves the topic partitioning, replication, partition leader and other information from Zookeeper. Topic details are saved only on Zookeeper, not in a database. This is because the topic description assignment is dynamic. The list of replicas, in sync replicas and the partition leader, vary continuously to mirror the cluster topology and the topic definition. Instead of keeping the topic details in a database and updating it every time it changes in the Zookeeper, it is preferable to store it only in the Zookeeper. Detailed topic information is retrieved from the Zookeeper using the Kafka SimpleConsumer java API.

- **Add Hops ACL for user**: the topic owner adds Hops ACL definitions to it. These ACLs can be for members of the owner project or for members of a destination project the topic is shared with. All requests on topics are authorized using these ACLs. For a topic without ACLs, the authorization depends on the Authorizer implementation of the default behavior. The default Hops ACL is configured by the property ‘allow.everyone.if.no.acl.found’ in the server.properties file and its behavior is implemented by the Authorizer implementation. Subsection 5.6 covers Hops ACL definition and how it differs from the Apache Kafka ACL definition.

- **Delete topic**: this operation deletes the topic from Zookeeper using the Kafka admin API, deleteTopic, first and next removes it from the database. It also removes the topic share and Hops ACL information on a cascade.
5.2. Topic Availability Invariant

5.1.2 Hops New Topic Operations

The following topic operations are introduced in this project.

• **Share topic**: this is a new concept in the Hops. Kafka does not have the concept of projects, hence no topic sharing across projects. In HopsWorks, the topic owner can share it with other projects of the same or different user. The owner project can then add the Hops ACLs for the destination project. Sharing a topic, without adding Hops ACLs to members of the destination project, is not enough for project members to access Kafka topic. In such scenario, only requests to topics that are open to all users will be authorized.

• **Unshare a topic**: the topic owner project can unshare the topic from the destination project which the topic is shared to at any time. Similarly, the destination project can unshare the topic after using it or immediately after receiving the share notification which effectively means declining to share a topic. The topic owner then removes all ACLs for members of the destination project.

• **Add Topic schema**: a schema is used to add integrity to the communicating entities. A topic has a schema. This schema is used by the Kafka producer to encode the messages it publishes and to decode the messages by the Kafka consumer when consuming them. This will be discussed thoroughly in section 5.4.

5.2 Topic Availability Invariant

Under any circumstance, the desirable state is when a Kafka topic exists both in the database and Zookeeper. If this state is not achieved, the system should try to keep the topic availability invariant explained below.

**If a topic exists in the Hops database, then it must also exist in the Zookeeper.**

A project user can see only topics in the database and assumes these topics are also available in the Zookeeper. For this assumption to be valid, the database should always mirror the Zookeeper. Note that the contrapositive of this topic availability invariant should also hold true. **If a topic does not exist in the Zookeeper, it does not exist in the database.**

To enforce the topic availability invariant, the topic create/delete operations should be done in the following order.
5.2.1 Topic Availability States

For any Kafka topic, at any moment in time, it exists in either of the following topic availability states.

- **State 1**: a topic exists both in Zookeeper and database
- **State 2**: a topic exists only in Zookeeper
- **State 3**: a topic exists only in database
- **State 4**: a topic does not exist in both

States 1 and 4 are the desirable states of topic availability states. If the topic is in either state 2 or state 3, however, a synchronizer should take the current state to state 4 to enforce the topic invariant explained above.

5.2.2 Why Always to State Four

Why when a Kafka topic availability state is either in state 2 or state 3 needs to go to state 4 not to state 1?

If the topic availability state is state 2, there is no way for Kafka clients to access the topic Avro schema from a database. Kafka clients’ REST calls to HopsWorks will fail. Thus, unable to publish/consume messages to/from the Zookeeper topic. Kafka clients are not authorized to create a topic and persist it to the database. Kafka topics can only be created from the HopsWors WebUI. So Kafka topic availability state cannot go from state 2 to state 1.

Similarly, if topic availability state is state 3, it cannot go to state 1 by the following reasoning. ’auto.create.topics.enable’ is disabled to disallow the
automatic topic creation in Zookeeper when the topic the clients want to interact with is not available in the Zookeeper. This move was necessary because a Kafka topic needs an Avro schema, hence, every topic creation should be done from the HopsWroks WebUI. This state is observed when a Kafka client fails to interact with the topic. In such scenario, the user should delete the topic from the database, effectively going to state 4, and either create the topic again or change the topic for the Kafka clients.

5.3 Why Topic Sharing

Why is a topic shared across projects? The main reason for topic sharing is when there is a need of communication across projects. These projects can include the owner project or can be all destination projects. During communication, either of them will run a job that publishes messages into a topic and others will have jobs that consume messages from the same topic. Which users of which projects write to a topic and which users of which projects read from a topic depends on the topic Hops ACL definitions.

The side benefit of this feature is, it avoids the unnecessary creation of a large number of topics. When a topic is not in use by the members of the owner project, members of other projects can use it. If a user from a destination project has also a specific topic name requirements, but that topic name is used by another project, the user cannot create the topic but can use it if shared. If the topic is not shared, the user can prefix the topic name with the project name.

Topic sharing is per project and Hops ACL definition is project member specific. So, though a topic is shared to a project, not all members of the project will have the same privileges on the topic unless role based ACLs are defined. It all depends on the Hops ACL definitions.

This feature extends the project based multi-tenancy feature of HopsWorks. HopsWorks supports sharing public datasets across projects.

One drawback of topic sharing is, unethical users can misuse topic data during inter-project communication. Data of previous user still residing on a shared topic can be read by unintended users who have a read access on this topic. It is up to the data owner to correctly configure the topic and decide who can perform what on the topic.

5.4 Hops Schema Registry

By using Avro schema, Kafka producer will encode its messages using the Avro schema before publishing it to Kafka topics. Kafka consumer will decode
the messages after consuming it from the Kafka topics. Effectively, Kafka messages are Avro messages. So, each Kafka topic is created with its own Avro schema. Hops schema registry implementation persists the Avro schemas into a database. The database CRUD operations on the Avro schemas have the following meanings:

1. **Create**: a new schema is created and persisted into a database. While creating a schema, if a schema of the same name exists in a database, its version number is incremented and persisted. If a schema of the same name does not exist, schema version is initialized to minimum version number (in this case 1) and is persisted. Only a validated and backward compatible Avro schema is registered.

2. **Read**: view the selected version of the schema.

3. **Update**: the highest version for this schema is incremented and assigned to the new version and persists a new/modified schema. Backward schema compatibility is enforced during schema update and if the new schema is not compatible with the old schemas, the update is rejected.

4. **Delete**: delete the selected version of the schema unless it is used by a topic.

### 5.4.1 Kafka Clients Access to Avro Schema

The Kafka clients need access to the schema that will be used for message serialization before sending publish/consume message request to the cluster. HopsWorks provides REST endpoints for clients to access the Avro schema. Allowing REST endpoints to external users has associated security risks. So, the Kafka clients should be authorized to access these endpoints.

When a user logs into HopsWorks, a new session id is given by the Glassfish server. All user operations are authenticated using this session id. If this user wants to run external clients that access the HopsWorks REST endpoints, those clients should get a way to access the user session id so that all requests from those clients will be authenticated. The solution we implemented is, to add the session id as an environment variable so that the clients can access it. After accessing it, the session id will be embedded in the header of the request to the HopsWorks REST endpoints. Since the server allows all operations for this session id, the new request will be authenticated. This way, both the producer and consumer clients will be able to retrieve the topic specific schema (schema version). After retrieving the schema, the producer encodes its messages using the schema and publishes them to the Kafka topic that uses this schema. The Kafka consumer,
5.5. Synchronizing Zookeeper and Database for Kafka Topics

in turn, decodes the consumed Avro messages and decodes them to get the row messages. It uses the Twitter object serialization and deserialization, Bijection, to convert objects back and forth. Unlike the Avro’s API, it has more friendly APIs [37]. Introducing a schema cache in the clients can reduce the number of requests to the server.

One word of caution here is, the automatic topic creation should be turned off in the broker configuration so that all Kafka topics will be created with an Avro schema. This avoids the possibility of topic creation when a Kafka producer wants to publish a message to a non-existent topic.

5.4.2 Avro Schema Compatibility and Evolution

Our implementation supports schema registering and upgrading. In each operation, the schema is validated and checked for backward compatibility before registering. Schema Evolution is applied when the schema to be used for data deserialization is different from the schema used to serialize the data. Similarly, it is applied when a schema is updated after data has been written to a topic using the older schema. The new version should be backward compatible so that already published messages with the old schema version can be read and decoded with the new schema version. The schema evolution is applied only during deserialization, so only Kafka consumer can apply the schema evolution.

Apache Avro supports schema evolution by applying the schema evolution rules. These rules can be applied either when a field is removed from or added to the old schema version [38].

5.5 Synchronizing Zookeeper and Database for Kafka Topics

Storing data at two or more different locations comes with its own challenges, including but not limited to, synchronization problem and handling conflicts. The synchronization challenges these locations may face is different depending on the type of data they store. For instance, if they log event time stamps which take place in the underlying system, obviously event ordering is the big challenge. In a sequence of events, in which order do the events take place and in what order do the different locations observe the events? This is a typical distributed systems synchronization challenge.

Similarly, since Hops Kafka topics are stored on the Zookeeper and the NDB database, they should be synchronized. The synchronization challenge here is, however, not event ordering. It is rather what the snapshots of the two locations
look like at any moment in time. At any given instance, especially when a Kafka client wants to use Kafka services, all the topics on Zookeeper should be available on the database and the same holds true the other way round. What happens if this is not the case? This section goes through the desirable topic existence states, possible failure scenarios, the options that can be used to solve the conflicts and finally discusses the implemented solution.

5.5.1 Topic Synchronization Failure Scenarios

For Kafka clients to use a Kafka topic, they should access both the database and the Zookeeper to publish/consume messages to/from the topic. They send REST calls to HopsWorks which in turn accesses the database to retrieve the Apache Avro schema of the specified topic. The Kafka clients are either Spark or Flink jobs that use Kafka as a messaging system. This is the only case where Kafka client jobs access the database for Kafka services. These clients access the Zookeeper topic to publish/consume messages to/from it. If the topic does not exist in either of the two locations, the only option is to abort.

The following topic synchronization failure scenarios are likely to happen in the system. They are ordered from the most the most likely to the less likely failure scenario.

- **Project owner removes the project itself.** This operation also implicitly removes all the project topics from the database on a cascade. However, it does not remove these topics from the Zookeeper. Since, the HopsWorks WebUI shows the list of topics in the database, the user may assume that the topics are also removed from the Zookeeper. A problem occurs when a user wants to create a topic with the same name that was deleted by the project deletion operation. Since Zookeeper has global naming space for topic names, the topic creation will fail.

  This operation is also hazardous as it removes many topics at once. Some of the project topics might have been shared across other projects and used by those project members, underlining the extra care that should be taken before performing this action. Otherwise, this operation could render user data in the Kafka topics inaccessible. Topics that were shared to this project, if any, would also be unshared. But, this will not create synchronization problem.

- **A Kafka topic has a configurable life span in the Zookeeper** with a default value of 168 hours. After that time, unless topic specific retention time is specified, the topic and its data in all its partition replicas will be deleted. But the topic still exists in a database. This is state 3 of the topic
availability states. If a user creates a topic from the HopsWorks WebUI when the system is in this state, the topic will be created in the Zookeeper, but it fails to persist it to the database as it already exists. This should be avoided as the new topic Avro schema can be different from the old schema and the topic can also belong to an unintended project. In the database, the topic belongs to the previous project, not the project in which the current topic is created. The solution is to remove the topic from the database using the HopsWorks WebUI. If Kafka clients try to interact with the non-existent topic, the clients should fail.

- **Project owner creates/removes a topic from the HopsWorks WebUI.**
  This is a normal user action under normal system operation. Topic synchronization is important as both topic creation and topic deletion are performed sequentially. A topic is created in the Zookeeper followed by persisting it in the database. The deletion process is the other way around. A synchronization problem occurs if the system fails soon after removing the topic from the database during the topic remove action or if the system fails soon after the topic is created on Zookeeper during the topic create action. While the failure in the topic removing process takes the topic availability state to state 3, the failure during the topic creation process takes the topic availability state to state 2.

### 5.5.2 Possible Synchronization Approaches

All Kafka client operations first check that the topic really exists both in Zookeeper and in a database. If this is not the case, clients should fail safely. They cannot create the topic where it is missing and resume their operations for the following two reasons. In the first place, Kafka topics need the Avro schema, but the clients do not have it. Thereof, they cannot persist a topic to the database. What if the Kafka clients keep a default Avro schema that will be used to create a topic in the database in case it is missing? A topic is tied to a project. Nevertheless, the Kafka clients do not know whether they are accessing the topic from the owner project or from the destination project. Consequently, if the Kafka clients persist a topic into the database, a topic that previously belonged to a given project can now belong to another project, which should not happen. Hence, keeping a default schema cannot solve the issue.

Similarly, ‘auto.create.topics.enable’ is disabled not to allow clients to create a missing topic in the Zookeeper. Originally, it was disabled so that no topic will be created in the Zookeeper which does not have an Avro schema. But, it has also ultimately prevented the creation of a topic on the Zookeeper when the same topic with Avro schema exists in the database.
Most importantly, Kafka clients can only request resources from Hops database via HopsWorks REST calls. They cannot persist any data to the database. As noted above, this approach cannot solve the topic synchronization problem. To solve this problem, the following alternatives are discussed:

- **Keep a cache for topics in the Zookeeper in the Authorizer implementation** - this is used by the default implementation of the Authorizer interface. A cache of topics in the Zookeeper is stored and the cache is updated every configurable amount of time. The cache mirrors the Zookeeper and the database mirrors the cache. The possible drawback of this approach is, the cache might not fully mirror the Zookeeper before it is obsolete. Even if the cache may mirror the Zookeeper, automatic topic creation in the database is disabled. This happens when the cache contains topics that do not exist in the database. Always topic creation needs an Avro schema. So the default implementation will no more work, hence we rejected this solution.

- **Zookeeper watcher client** - a Zookeeper watcher is set up for each topic and when a topic is deleted from the Zookeeper the watcher is invoked. The watcher, in turn, removes the topic from the database. The problem with this approach is the Zookeeper will have as many watchers as the number of topics it has. Too many watchers can cause session re-establishments to fail. So, it is also rejected.

- **JEE Timer Service** - this timer service is configured to run every configurable amount of time. It is also persistent i.e. if the server restarts, the timer will resume its executions. This is the best solution among the proposed solutions and is explained as follows.

When the scheduled timeout expires, the timer gets all topics from the Zookeeper and goes through each topic in the database to remove all topics from the Zookeeper that are not available in the database and the other way round. The good part with this approach is, it does not try to create a topic. It only removes a topic. All the rejected options try to create a topic in the database in some scenarios.

How often should the topics be synchronized? Ideally, the timer should receive a synchronize notification soon after it finishes processing the business logic of the notification handler. If the execution time of the business logic is smaller than the configured interval, the handler will remain idle for the duration of the difference. If the execution time of the business logic is larger than the configured interval, however, there will be queued notifications. This introduces synchronization delays. The
execution time depends on the number of topics the system has, and the comparison takes the lion’s share of the execution time as it goes through all topics with time complexity of $O(N^2)$, where $N$ is the number of Kafka topics in the system. Low timer interval favors more synchronization, but may introduce a queue of unprocessed timer notifications for there could be too many topics in the system. Long timer interval makes the system less synchronized, makes the notification handler idle for the duration of the difference and avoids the possibility of accumulating unprocessed timer notifications. Though not the optimum solution, longer timer interval is preferred to shorter timer interval, as the Zookeeper topics and database topics are rarely out of phase.

Configuration issues of this Solution

Upon the reception of delete topic operation, Kafka server marks the topic with ’marked for deletion’. This marking has no effect unless delete.topic.enable is set to true in the server.properties. If this property is not set, the marked topic appears in the list of topics in Zookeeper and a delete topic operation can be requested again for the same topic, effectively adding extra load to the timer. Also creating a new topic with the same name throws a TopicExistsException though the previous topic is already marked for deletion. Therefore, setting this property is crucial for the Kafka Admin APIs to work as intended.

The second and the serious drawback is, it requests too many IO connections to the Zookeeper. The Zookeeper may throw a connectionless exception in such cases. To avoid this error, set maxClientCnxns in the conf/zoo.cnf with a reasonable number.

5.6 Hops ACL Definition

Hops has its own ACL format that extends the Kafka ACL to fit its own requirements. Defining access control list is describing ’who can perform what on a resource from where as what role?”. So, the general format of Hops ACL is, ”Principal P from within Project Pr is [Allowed/Denied] Operation O From Host H on Resource R for the Role Ro”. Hops ACL can also be defined as ”All members in Project Pr are [Allowed/Denied] Operation O From Host H on Resource R for the Role Ro” to give group access lists. Where:

- **Principal**: this is the user who requests the Kafka cluster for a given operation on the specified resource name. A user request originates from inside a project. So, the user name is the HDFS User identified by
projectName_username (projectName underscore underscore useName ). This format gives fine-grained allow/deny ACL definition to a specific principal. If the user name is projectName_* , where wildcard (*) is used instead of a specific project member, then the ACL definition applies to all members of the project. This is a role-based ACL definition.

- **Resource:** this is the topic name. Currently, it does not include the Kafka cluster and 'consumer-group' resource types.

- **Permission Type:** the Kafka permission types-namely ALLOW and DENY.

- **Operation:** is one of the following operation types: READ, WRITE, DESCRIBE, *. The wildcard '*' represents all the three operations. However, it does not insert three rows into the database. For ACL definition with ALLOW permission type, both READ and WRITE operations imply DESCRIBE. In any other scenarios, every operation is singleton operation.

- **Host:** the source node where the request originates from.

- **Role:** the HopsWorks role types: DATA OWNER, DATA SCIENTIST, *

The two additions to the Kafka SimpalAclAuthorizer ACL definition are, the project and role concepts. It keeps the priority rule and implicit ACL definitions of the SimpalAclAuthorizer ACL definition. The topic owner can perform the following actions on Hops ACLs.

- **Add Hops ACL:** topic owner adds Hops ACLs for the topic inside the topic owner project.

- **Delete Hops ACL:** the topic owner can remove the topic ACLs.

- **List/Alter ACLs:** the list of topic ACLs are displayed as advanced features of the topic. There is also the option to alter the ACL definition if required.

Hops ACL definition is stored in a database and there is no interaction with the Zookeeper both when adding/removing ACLs and when authorizing user requests. Unlike the SimpleAclAuthorizer that stores ACLs in JSON format, Hops implementation is a plain text. The NDB cluster serves millions of requests per second and this has deterred the need of introducing ACL caches in the implementation of HopsAclAuthorizer.
5.6. Hops ACL Definition

5.6.1 The Role Concept

HopsWorks has project abstraction from which a user with the required credentials can perform any operation. A role defines what specific operations a user of a project can do. The project creator is the project owner, with DATA OWNER role, and is allowed to perform all operations like creating a project, adding members into it, creating jobs within it, accessing public dataset, deleting the project itself. All other members are with limited capabilities like accessing public dataset, running jobs from within the project when they have the authorization.

Specifying the role type in the Hops ACL definition describes what roles the user can perform. The wildcard ‘*’ specifies an ACL definition for both roles.

5.6.2 Fine-grained ACLs and Role-base ACLs

One of the good features with this implementation is, it supports the notion of a continuum ACL definitions. Table 5.1 is a sample Hops ACL definition table. The database topic_acl table, where topic ACLs are stored, has additional columns used for identifying a specific ACL and simplifying the authorization process. The sample table is the content of table topic_acl as seen from the HopsWorks WebUI.

Some of the interpretation of the ACL entries in the table below are listed as follows. Note that table 5.1 does not specify which topics belong to which projects and which topics are shared across projects.

1. According to the ACL in entry six in the table, any operation is allowed on topic ‘Twitts’. In spite of this, entry two is a deny ACL. It is worth of mentioning the priority rule here. user2@kth.se is denied because of the priority rule but others are allowed.

Similarly, according to the seventh entry in the table, no operation is authorized on topic ‘Shares’, hence operationally deemed as not important. Though the first entry allows admin@kth.se the read operation on this topic, this user cannot read from it because ‘Deny’ ACL has higher priority to ‘Allow’ ACL.

One possible non-operational scenario where this can be used is, a user may use the topic ‘Shares’ as a placeholder for future use. Since a topic name is unique in Kafka, a user can not create a given topic if the topic name is already taken by another project. So, topic ‘Shares’ can be rethought as it is kept for future use. It will not be misused because of its current ACL definition and it can be used in the future by only changing the ACL definitions. But, the number of topics a user can create per project is limited.
### Table 5.1: Sample Hops ACL definitions

<table>
<thead>
<tr>
<th>Topic</th>
<th>Member</th>
<th>Permission</th>
<th>Operation</th>
<th>Host</th>
<th>Shares</th>
<th>Topic</th>
<th>Member</th>
<th>Permission</th>
<th>Operation</th>
<th>Host</th>
<th>Shares</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temp.</td>
<td>Data Scientist</td>
<td>Deny</td>
<td>Read</td>
<td>bbc1.sics.se</td>
<td>*</td>
<td>MyKafka</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Rain</td>
<td>Data Scientist</td>
<td>Deny</td>
<td>Read</td>
<td>bbc1.sics.se</td>
<td>*</td>
<td>MyKafka</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Twitts</td>
<td>Data Owner</td>
<td>Deny</td>
<td>Write</td>
<td>localhost</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Weather</td>
<td>Data Owner</td>
<td>Deny</td>
<td>Read</td>
<td>bbc1.sics.se</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* Denotes wildcards that match all topics and roles.
2. Entry three in table 5.1 allows admin@kth.se a read operation on topic 'Temp.' This entry implicitly allows the same user 'Describe' operation with all other entry values the same.

3. Entry two is an example of the most fine-grained Hops ACL definition. It can not be broken into any smaller definitions. Entry six and seven, in contrast, are the most coarse definitions. They can be decomposed into many more less coarse or fine-grained definitions.

This definition gives the notion of continuum ACL definitions, ranging from the most fine-grained to the most coarse definitions. With four (member, operation, host and role) of the fields allowing wildcard values, there are sixteen (16) levels of ACL definitions. Table 5.2, lists these levels. Note that 0 represents a specific value while 1 represents a wildcard value.

4. Role-based Hops ACLs are defined for all data owner members or data scientist members of a project. Entry five in table 5.1 defines an example of a role-based access control. This ACL denies any write operation on the topic 'Rain' from the host bbc1.sics.se for all data scientist members of the project Weather. In table 5.2, role-based authorization is defined by those entries where the role field is set to a wildcard value. In the database, as many rows as the number of project members are inserted during a single role-based ACL insertion. So, entries 5, 6, and 7 are broken down into role-based ACLs in the database.

### 5.7 HopsAclAuthorizer Class

This is the Hops implementation of the Authorizer interface. This implementation does not have any interaction with the Zookeeper as no ACLs are stored on it. All
actions on topic ACLs are performed from the HopsWorks frontend. So, there is no add/remove ACLs operation that routes via this implementation.

This implementation is leveraged during authorization. An instance of the implementation is created during initial Kafka broker configuration. The instance accesses the database to retrieve the stored ACLs when there is an authorize request by a Kafka client. It either allows or denies the operation for the principal build by the HopsPrincipalBuilder based on the ACL definitions in the database.

The property 'allow.everyone.if.no.acl.found' in the Kafka broker configuration is per resource type and resource name. If this property is set in the configuration file, then that value will be applied to all resource types, i.e., CLUSTER, TOPIC and CONSUMER_GROUP. However, there are ACLs only for TOPIC resource type. To solve this problem, the HopsKafkaAclAuthorizer uses this property only for TOPIC resource type and allows all authorize requests to all other resource types by default.

Unlike the SimpleAclAuthorizer that keeps a cache of ACL, HopsAclAuthorizer does not have a cache. HopsWorks does not even need to keep a cache of Hops ACLs. We leveraged the high-speed NDBCluster, so ACLs can be accessed from database in a fraction of time.

5.7.1 HopsPrincipalBuilder Class

This is the Hops implementation of the PrincipalBuilder that returns an instance of KafkaPrincipal. It extracts the user name from the SSL username in the form of projectName_username and creates an instance of KafkaPrincipal. KafkaPrincipal is an implementation of the transport Layer Principal interface. Unlike the default PrincipalBuilder of the SimpleAclAuthorizer, this implementation checks the type of the SSL user.

5.7.2 When is a topic operation authorized

There is no difference with the default SimpleAclAuthorizer as to when a topic operation is authorized. The precedence rule, super user concept, and implicit ACL definitions of the SimpleAclAuthorizer are preserved in this implementation too. The difference is, however, while an additional ACL is created and stored for the implicit ACL definitions in the SimpleACLAuthorizer, no additional ACL definition is created in the HopsAclAuthorizer. The implicit Hops ACLs are checked when Kafka client requests are authorized. The user operation is authorized if at least one of the following is true:

- if a user is a super user.
• if ‘allow.everyone.if.no.acl.found’ is set to true and there are no ACLs for
  the topic and the user is a member of the topic owner project.

• if there are specified or implied allow ACLs in the database for the given
  operation.
Chapter 6

Test Configurations

This chapter discusses the test configurations and test files developed to test the project works.

6.1 Kafka Broker Configurations

Each Kafka broker needs SSL key and certificate to authenticate Kafka clients. The SSL key is generated using a REST call to Hops SSL generator and the public key is signed by an intermediate CA. The trust store and key store locations are added in the configuration file of the brokers.

The broker configuration should set the 'authorizer.class.name' to HopsACLAuthorizer and 'principal.builder.class' to HopsPrincipalBuilder.

The HopsACLAuthorizer needs the URL, type, and the login credentials of the database to retrieve the Hops ACLs during Kafka client request. These values should also be passed in the configuration and the HopsACLAuthorizer gets them during initial Kafka broker configuration.

Below is a portion of Kafka broker configuration which includes most of the necessary properties added/changed in this implementation.

```
<table>
<thead>
<tr>
<th>Listing 6.1: Sample Kafka broker configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>listeners=PLAINTEXT://10.0.2.15:9092, SSL://10.0.2.15:9091</td>
</tr>
<tr>
<td>security.inter.broker.protocol=PLAINTEXT</td>
</tr>
<tr>
<td>ssl.client.auth=requested</td>
</tr>
<tr>
<td>ssl.keystore.location=/var/lib/kagent/keystores/node_server_keystore.jks</td>
</tr>
<tr>
<td>ssl.keystore.password=hidden</td>
</tr>
<tr>
<td>ssl.truststore.location=/var/lib/kagent/keystores/node_server_truststore.jks</td>
</tr>
<tr>
<td>ssl.truststore.password=hidden</td>
</tr>
<tr>
<td>allow.everyone.if.no.acl.found=false</td>
</tr>
<tr>
<td>delete.topic.enable=true</td>
</tr>
</tbody>
</table>
```
Kafka cluster should run at least with two ports. One of them should be a plaintext port that will be used by the SimpleConsumer API to request topic partitioning details from within the HopsWorks. SimpleConsumer API cannot be configured with SSL key and certificates. The second port should be SSL port that is used to authenticate Kafka clients’ communications with the Kafka brokers.

6.2 Zookeeper Configuration

There is no additions to Zookeeper configurations. The only case where Zookeeper configuration is given due attention is when the topic synchronization timer in section 5.5.2 creates too many I/O connections to the Zookeeper and closes them repeatedly. Zookeeper throws an exception when there are many open connections to it. To avoid the possibility of throwing an exception by the Zookeeper, maxClientCnxns=0, should be added to zoo.cnf so that it allows limitless client connections to it. It can be set to any number depending on the maximum number of client connections needed.

6.3 Configuring Kafka Clients

Kafka clients should also be configured with SSL keys and certificates so that they will be authenticated by the Kafka brokers and their requests will be authorized by the HopsACLAuthorizer. Hops Kafka clients are user jobs that run from inside Hops project. These jobs are usually either Spark or Flink jobs that require Kafka as a service.

These jobs should be configured with the user SSL keys and certificate to be able to request services. Each user member of a project has its own SSL key and certificate. They are generated during project creation for the project owner, and member addition for other members. Each user has unique SSL and certificate per project it owns or is a member of. Kafka clients will authenticate to the cluster with their own principal. A principal is a user who is a member of the project inside which the job is running and is identified by the project name and the
username. So, SSL keys and certificates are generated with the principal name of projectname_username and then stored in the database.

When Kafka clients want to communicate with the cluster, how can they access the user key and certificate from the database? Giving Kafka clients access to the database is risky. The safest way to retrieve the values is by loading them as the application’s local resources.

The Kafka clients use the Kafka util library in part 6.5 to create instances of Kafka producer and consumer.

## 6.4 Local Resources

YARN applications perform their work by running containers which most of the time have dependencies on files. Applications might need those files either at startup or during the rest of their life cycles. For remote files, instead of accessing those files each time they are needed by the applications, YARN provides the concept of resource localization that loads/copies remote files to the local file system [39].

In our implementation, when a user runs a job from a project, the SSL keys and certificates of the user are retrieved from the database and loaded as the job’s local resources to the local file system. The Kafka Util library can then load these values and configure the Kafka producer and consumer.

## 6.5 Kafka Util Library

This library contains the Kafka producer and consumer clients, and HopsKafkaUtil class that instantiates a singleton instance of each of these clients. Other systems which intend Kafka as their messaging system can interact with the Kafka cluster using this utility class. Generally, this library performs the following actions:

- Accesses the user SSL keys and certificates from the local resource. It then configures the Kafka producer and consumer with the truststore and keystore locations.
- Create instances of the Kafka producer and consumer. These clients are configured with the necessary parameters including, but not limited to, broker address, topic name, serializer/deserializer classes and others.
- Retrieve the Avro schema used by the topic from the database. The projectId of the project inside which the topic is created is used as a path parameter in the REST call to retrieve the schema. The user session id is also needed
to authenticate the schema request operation. It is added as a parameter in the request header. Both of them are passed as environment variables. The topic name is passed as an argument in the schema path.

6.6 Spark job

This is a simple Spark application developed for testing the implementation with same application producing and consuming messages to and from the Kafka cluster. It is a customization of the Spark PI application. What type of messages the Spark application wants to publish and what to do with the consumed messages are up to the developer of that application. The goal here is to see this application producing and consuming messages. It creates the Spark context and immediately accesses the Kafka util library to create an instance of:

- the Kafka producer that publishes Avro messages and logs the sent messages into the HopsWorks log files,
- the Kafka consumer that consumes those Avro messages, decodes them to get the raw messages and similarly logs them into the HopsWorks log files.

The Spark Job starts the Kafka consumer followed by the producer so that the consumer will consume all the messages published.
Chapter 7

Conclusions, Recommendations and Future Works

This chapter goes through summarizing the main features implemented in this project. It finally concludes by suggesting some possible future works to be investigated on.

7.1 Conclusions and Recommendations

We have implemented a multi-tenant Kafka environment that supports ACL-based authorization. Kafka multitenancy enabled user data to reside in the same cluster. Access to each data is configured by the data owner. So, for an appropriately configured environment, one user’s data is completely isolated from another user’s data. Such data isolation at a Kafka topic level is achieved by defining access control lists to the topic.

The data isolation does not restrict one user access per Kafka topic. Since Kafka topics can be shared across projects, many project members may access the topic. Defining topic access rights is recommended to be based on the principle of least privilege so that users are given the lowest level of rights that enable them to do their jobs. Moreover, project users have different roles and the ACLs can be defined either for individual users or for user roles. This approach helps for better communication across projects.

With this implementation, enterprises can deploy Kafka in their data architecture without the fear of security breaches. When data is in Kafka, the data owner can limit user access to that data by defining access control lists. Allow ACLs are defined only for those users who are trusted to access the data. By default, data is not accessible to any user unless specific allow ACLs are defined for that data. By doing so, unwanted users are prohibited from accessing the data.
Chapter 7. Conclusions, Recommendations and Future Works

Started with the goal of building a secure and multi-tenant Apache Kafka for HopsWorks, the above summary explains the thesis goals are met.

7.2 Future Works

The following ideas are identified as future works.

1. As discussed in section 5.3, topic sharing is a new feature for HopsWorks. Currently, if a topic is already taken by another project, the only option is to prefix the topic name with the project name. A possible workaround is to provide a request to the topic owner for sharing.

2. Currently, when the owner shares the topic with another project, it does not send a notification to the destination project notifying the project owner to either accept or decline the invitation. The sharing process is performed automatically.

3. In this implementation, Hops ACLs are added only for topics. It does not add Hops ACLs for Clusters and Consumer_Group resource types. Currently, those resource types will use the default Hops ACL.
Bibliography


