Kafka Web Gateway

Alexandre Silva
Miguel Marques

Advisor: Pedro Félix

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

July 2023

Instituto Superior de Engenharia de Lisboa

Kafka Web Gateway

47192 Alexandre Marques da Silva

47204 Miguel Jorge Bacelar Marques

Advisors: Pedro Miguel Henriques Santos Félix

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

July 2023

Abstract

This project aims to expand Apache Kafka to Web Browser clients by developing a Kafka Web Gateway application. Kafka, a distributed streaming platform, is widely used for data pipelines and streaming applications, while web clients typically rely on standard protocols such as WebSocket. The project addresses scenarios where businesses use Kafka as their back-end data communication platform and require real-time data streams from Kafka topics to be efficiently consumed and produced by web clients. The Kafka Web Gateway acts as an intermediary, facilitating communication between Kafka and web clients via WebSocket.

The application incorporates authentication and authorization mechanisms to ensure secure access to the Kafka Web Gateway.

The main functionalities of the Kafka Web Gateway include establishing connections with Kafka clusters, subscribing to specific topics for real-time data consumption, managing bidirectional data flow between Kafka and WebSocket clients, implementing authentication and authorization mechanisms, providing APIs for message acknowledgement and retries, and enabling a unique efficient message consumption by key. The project also offers a ready-to-deploy environment with load balancing capabilities, Docker files for easy deployment, and libraries for interacting with the Gateway Application.

This report provides a comprehensive understanding of the Kafka Web Gateway application. It begins with an introduction outlining the project's motivation, goals, and main functionalities. Subsequent chapters explore the fundamentals of Kafka, the problems to be solved, the architecture of the Kafka Web Gateway, server-side implementation details including authorization and authentication, web client and administration aspects, and concludes with a summary of findings, challenges, and future enhancements.

Keywords: Apache Kafka; WebSocket; Kafka Web Gateway; Scalability; Message consumption by key; Docker deployment;

Resumo

Este projeto tem como objetivo expandir o Apache Kafka para clientes browser da Web. O Kafka, uma plataforma de streaming distribuída, é amplamente utilizado em pipelines de dados e aplicações de streaming, enquanto os clientes da web geralmente dependem de protocolos padrão como o WebSocket. O projeto aborda cenários em que as empresas utilizam o Kafka como plataforma de comunicação de dados de back-end e exigem que os fluxos de dados em tempo real dos tópicos do Kafka sejam consumidos e produzidos de forma eficiente pelos clientes da web. O Kafka Web Gateway atua como intermediário, facilitando a comunicação entre o Kafka e os clientes da web através de WebSockets.

A aplicação incorpora mecanismos de autenticação e autorização para garantir o acesso seguro ao Kafka Web Gateway.

As principais funcionalidades do Kafka Web Gateway incluem estabelecer conexões com clusters do Kafka, subscrever a tópicos para consumo de dados em tempo real, gerenciar o fluxo bidirecional de dados entre o Kafka e os clientes WebSocket, implementar mecanismos de autenticação e autorização, fornecer APIs para confirmação de mensagens e reenvios, e permitir um consumo eficiente de mensagens por chave. O projeto também oferece um ambiente pronto para distribuição com capacidades de balanceamento de carga, ficheiros Docker para fácil instalação e bibliotecas para interação com a aplicação Gateway.

Este relatório fornece uma compreensão abrangente da aplicação Kafka Web Gateway. Começa com uma introdução que destaca a motivação, os objetivos e as principais funcionalidades do projeto. Os capítulos subsequentes exploram os fundamentos do Kafka, os problemas descobertos, a arquitetura do Kafka Web Gateway, detalhes de implementação no lado do servidor, incluindo autorização e autenticação, aspectos do cliente da web e da administração, e conclui com um resumo das descobertas, desafios e aprimoramentos futuros.

Palavras-chave: Apache Kafka; WebSocket; Kafka Web Gateway; Escalabilidade; Consumo de mensagens por chave; Deploy usando Docker;

Contents

List of Figures

Chapter 1

Introduction

The current chapter sets out the motivation and aims of the project, as well the main functionalities to be developed.

1.1 Motivation

This project addresses a bridge between Apache Kafka[1] and Web Browser clients. Kafka protocol[1] is not supported on Web clients[2]. So, in certain scenarios, e.g., when a business uses Kafka as its back-end data communication platform, there can be a need to bridge the gap between Kafka and Web clients, allowing real-time data streams from Kafka topics to be consumed by Web clients. This motivates the development of a Kafka Web Gateway application that acts as an intermediary, facilitating communication between Kafka and Web Browser clients, and subsequently WebSocket clients.

1.2 Goals

The primary goal of this project is to create a Kafka Web Gateway application that enables Kafka features to Web clients. By establishing a WebSocket connection between Web Clients and the Gateway, and then Kafka, the project solution aims to provide a seamless flow of data, allowing Web Clients to receive updates from Kafka topics.

Specific objectives include:

- Developing a reliable and efficient communication layer between Kafka and WebSocket clients.
- Implement authentication and authorization mechanisms to ensure secure access to the Kafka Web Gateway application.
- Designing an API for both client and server interactions.

[1] Kafka uses a binary TCP-based protocol

[2] The client side (user side) of the Web. Also called Web Browser client, it typically refers to the Web browser in the user's machine or mobile device.

- Allowing full product customization to each need, either by allowing clients to have control over what to receive and how.

1.3 Main functionalities

The Kafka Web Gateway application will offer the following main functionalities:

- Establishing a connection with Kafka clusters and subscribing to specific topics for real-time data consumption.
- Handling incoming WebSocket connections and managing the bidirectional flow of data between Kafka and WebSocket clients.
- Implementing authentication and authorization mechanisms to ensure secure communication and protect sensitive data.
- Providing APIs for client applications to interact with the Kafka Web Gateway, supporting message acknowledgement and retries.
- The Client API will bring a unique feature to Kafka consumption, the ability to efficiently consume messages by key (one of the main focus of this project).
- Ready-to-deploy environment consisting of a load balancer, docker files to easily deploy a Hub and Record Router nodes and libraries to interact with the Gateway Application.

1.4 Report structure

This report is organized into several chapters to provide a comprehensive understanding of the Kafka Web Gateway application. The structure is as follows:

Chapter 1: `Introduction` (current chapter) provides an overview of the project's motivation, goals, main functionalities, and the structure of the report.

Chapter 2: `Kafka System` explores the fundamental concepts of Kafka, its architecture, and the main APIs used for interacting with Kafka.

Chapter 3: `The Challenge` exposes the issues found in order to accomplish the project goals.

Chapter 4: `Architecture` discusses the general architecture of the Kafka Web Gateway application, including server components, data models, APIs, and the technologies employed.

Chapter 5: `Gateway Implementation` delves into the specific details of the server-side implementation, focusing on authorization and authentication mechanisms, how does the Gateway consume Kafka, reroute events and exposed APIs as well as other relevant aspects.

Finally, the report concludes with a summary of the findings, the challenges faced, and possible future enhancements.

Throughout this report, we will explore the technical aspects, design choices, and implementation details of the Kafka Web Gateway application.

Chapter 2

Kafka System

2.1 What is Kafka?

Kafka is an open-source distributed streaming platform designed as a high-throughput, fault-tolerant and scalable messaging system. Its three main capabilities consist of:

- Reading and writing messages like a message queue through the publish/subscribe pattern;
- Storing messages with fault tolerance;
- Processing streams[1].

With the aim of accessing the functionalities of Kafka, there are multiple client APIs available. The clients used in this project are the following Java Client APIs officially provided by Kafka:

- Kafka Admin API;
- Kafka Producer API;
- Kafka Consumer API;
- Kafka Streams API.

2.2 Kafka Architecture and Main Concepts

2.2.1 Records, Partitions and Topics

- **Broker** is an individual node of Kafka.
- **Cluster** is a group of brokers.

[1] Continuous flow of data/messages.

- **Records** are the unit of data within Kafka, alternatively referred to as messages, events and rows, due to being equivalent terminologies. A record is composed of a key, a value, a timestamp of when the record was produced, and additionally, custom metadata headers. A record is always related to an offset.

  A minimal representation of records can be observed as vertical tuples within a partition in figure 2.1.

- **Topics** play a role in the categorization of records in Kafka, it acts as a logical container or channel for organizing and categorizing streams of related data. Such can be metaphorically represented as a TV channel of a specific theme where records can be represented as each video frame.

- **Internal Topics** are topics that Kafka uses internally, not meant for consumption or production by applications but for internal Kafka management and operations

- **Partitions** provide horizontal scalability and parallelism by dividing a topic's records into multiple ordered and immutable sequences. Partitions also afford replication across all the Kafka brokers allowing parallelism in the consumption of the partition's records and fault-tolerance. An illustration can be found in figure 2.1.

  Although records are ordered in a partition, records are not necessarily ordered topic-wise.

- **Offset** is a field every stored record is associated with that represents their order/position in a topic partition. This field is stored separately from the record.

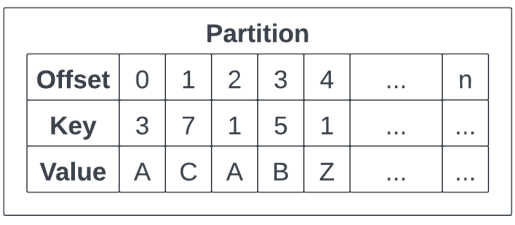

Figure 2.1: Kafka Partition and Records

- **Configurations** in Kafka follow a key-value pair structure. These configurations can be applied to brokers, topics and all clients.

2.3 Kafka Client Java APIs

2.3.1 Kafka Admin API

The Kafka Admin API provides a programmatic way to perform cluster management tasks instead of relying solely on command line tools, even though it's still one of the main tools to manage Kafka. One essential tool in this API is the AdminClient class.

Using AdminClient, developers can execute commands like creating topics, either by specifying the number of partitions and replication factor or leaving them to the broker default settings. It offers more control and flexibility compared to command line tools. For example, instead of manually running a command each time a new topic is needed, an app can be created to request such topic creation. This API supports various administrative tasks, including changing configurations, managing access control lists (ACLs), creating/deleting/listing topics and partitions, describing/listing consumer groups, and describing clusters. It provides a way to build user-facing applications that don't rely on shell scripts and allows control and monitoring of cluster operations. Overall, the Kafka Admin API, is a tool for programmatic administration of Kafka clusters, enabling automation, customization, and enhanced control over cluster management tasks.

2.3.1.1 Topic creation peculiarity

When using the Kafka Admin API, it's important to understand that most operations are asynchronous, meaning that they return a Future object representing the result of the operation. However, there can be a delay in the actual execution of certain operations, such as creating a topic in a multi-broker Kafka environment. The response from the Admin API may indicate that the request was successful, but it doesn't necessarily mean that the topic has been fully created and propagated to all brokers in the cluster. The synchronization process between brokers/nodes plays a vital role in ensuring consistency across the Kafka cluster. After a topic creation request is received by one broker, that broker is responsible for creating the topic and then informing the remaining brokers of its creation. This replication and synchronization process takes some time, and during this period, publishing messages to the newly created topic may fail since not all brokers are yet aware of its existence. To mitigate this issue, it's recommended to introduce some form of delay or wait time after receiving a successful response from the Admin API for topic creation. This allows for an appropriate amount of time for the topic metadata to propagate across the cluster and ensures that all brokers are aware of the new topic before attempting to produce or consume messages. It's important to consider these synchronization aspects and design your application logic accordingly when working with the Kafka Admin API, especially in scenarios where immediate access to newly created topics is required.

2.3.1.2 Topic Configurations

- **CLEANUP_POLICY_CONFIG** designates the retention policy used by the created topic. Either "delete", the default policy, which will delete old records when their retention time or size limit is reached. Or "compact" that replaces a record of the same key every time a record is produced to the topic.

2.3.2 Kafka Consumer API

The Kafka Consumer API is responsible for fetching messages from Kafka. Therefore the Consumer retrieves records published to the topics or specific partitions of the subscribed topics. This API employs a pull-based approach to fetch records, maintaining its offset to ensure sequential message consumption.

The mentioned offset can either be managed locally only or with the aid of Kafka's internal topic "__consumer_offsets", an internal topic Kafka uses to store all the requested consumer offsets.

2.3.2.1 Consumer Group

A **Consumer Group** is a Consumer mechanism that allows parallelism, load-balancing and fault-tolerance in consumption from a subscribed topic by allowing multiple Consumer processes to possess the same group_id. By sharing the same group_id, these Consumer processes, potentially in different machines, are considered as part of the same Consumer Group and will collaborate in consumption.

When multiple Consumers from the same Consumer Group are subscribed to the same topic, each member of the Consumer Group will consume from a different subset of the partitions in the topic, however no partition is shared between the mentioned subsets and can only be consumed by one Consumer per Consumer Group at a time. If a Consumer Group consists of more Consumers than the number of partitions in a subscribed topic, those remaining members will stay on hold until one of the other members disconnects or fails.

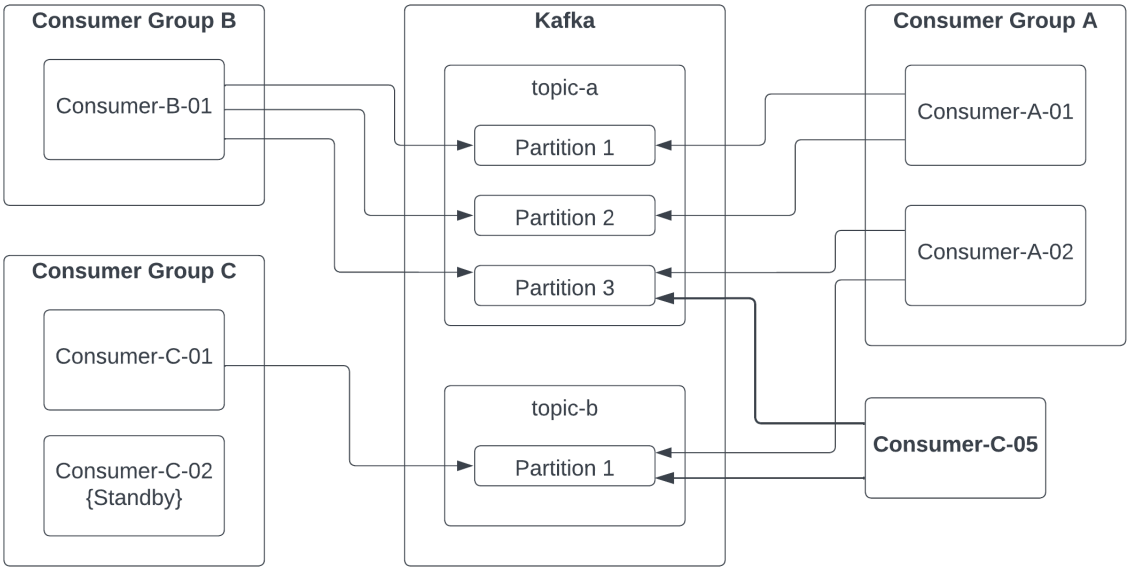

Figure 2.2: Consumers example

Figure 2.2 displays an example of one Consumer instance that doesn't belong to any Consumer Group and multiple Consumer Groups with the respective consumer instances consuming from Kafka topics. Each Consumer instance may or may not belong to the same process or machine.

Multiple cases can be observed such as the "Consumer Group C" that is only subscribed to "topic-b" which has only one partition while the Consumer Group has two consumer instances. Since only one partition is available to be assigned to the consumer instances, only one Consumer instance will consume records simultaneously, while the remaining will stay in a state of standby. Since Kafka tries to distribute the available partitions equally between the consumer instances of a Consumer Group, the minimum number of Consumer instances actively consuming in this situation is one. However the example of the "Consumer Group B" depicts a case that a Consumer Group is formed by only one instance, hence consuming all the partitions by itself. It can also be observed that "Consumer-C-05" possesses the id "C" but is not part of the "Consumer Group C". That is due to the fact that this Consumer does not use the subscribe operation, but the assign operation, mentioned with more detail in section 2.3.2.4 Operations.

2.3.2.2 Rebalance

Rebalance is a process that unfolds when subscribed partitions or members of the Consumer Group change. Kafka itself manages this process by redistributing the partitions between all the active Consumers within that Consumer Group.

2.3.2.3 Thread Safety

Since the Kafka Consumer API in not thread-safe, only one thread can be used for each consumer instance. And since the Consumer is also synchronous, with blocking operations, a thread should not be shared between Consumer instances.

2.3.2.4 Operations

Main operations a consumer instance can perform:

- **commit** manually sends the current offset to the "__consumer_offsets" topic. This operation possesses both synchronous and asynchronous variations.
- **assign** is used to manually choose which exact topic partitions will be consumed. Even if a Consumer possesses the same group_id as other Consumers, the assign operation will not allow the Consumer to join a Consumer Group.
- **subscribe** is performed when the objective is to consume from multiples topics but leave the assignment responsibility to Kafka itself. The usage of this operation will either create a Consumer Group if it doesn't exist, or join an existing Consumer Group.
- **poll** is a synchronous operation and is responsible for pull requests to Kafka by offsets or timestamps. If not enough data is available to be consumed then it will wait for a custom submitted amount of time until it returns. Other operations such as commit, assign and subscribe are only communicated to Kafka through a poll call.
- **wakeup** is an thread safe operation executed to break the poll loop.

The subscribe and assign operations are incompatible, when applied to a consumer the previous operation is overwritten.

As mentioned, most operations including the poll operation are synchronous due to most Kafka Consumer operations not being thread safe. This aspect leads to the usage of multiple Consumers from the same Consumer Group to attain parallelism while gathering data from Kafka.

2.3.2.5 Configurations

- **AUTO_OFFSET_RESET_CONFIG** defines the offset used when a Consumer starts consuming from a partition. The available values are "earliest" which means the Consumer starts consuming from the earliest offset, and "latest" which means the Consumer continues consumption from last consumed offset;
- **KEY_DESERIALIZER_CLASS_CONFIG** holds a class used for deserializing the record keys;

- **VALUE_DESERIALIZER_CLASS_CONFIG** holds a class used for deserializing the record values;

- **BOOTSTRAP_SERVERS_CONFIG** holds the host and port of all Kafka brokers.

2.3.3 Kafka Producer API

The Kafka Producer API is responsible for sending messages to Kafka. It revolves around the KafkaProducer class, which is simpler compared to the consumer counterpart. The producer is configured with a few properties and offers a send method to send records to Kafka.

The specific configurations for the producer depend on the desired use case and requirements. Important considerations include whether message loss is tolerable, whether accidental duplication of messages is acceptable, and the acceptable latency for end clients to receive messages. These factors directly impact the throughput of Kafka. For instance, with a latency tolerance of 500ms, Kafka can handle up to a million messages per second.

To produce messages, the first step is to create a ProducerRecord. This record must include the topic to which the record will be sent, as well as the key and value. Optional fields such as key, partition, timestamp, and headers can also be specified. If a partition is not explicitly provided, a partitioner is used to determine the target partition based on the key of the record.

The producer adds the record to a batch of records that will be sent to the same topic and partition. Upon successful write, it returns a RecordMetadata object containing information such as the topic, partition, and offset of the record within the partition.

Configuring the producer properly is crucial. Let's summarize some of the most relevant configuration options for this project:

Replication Factor: Topics have a replication factor, which determines how many replicas of each partition are maintained. With N replicas, Kafka can tolerate up to N-1 failures.

Acknowledgment Settings: The producer offers three acknowledgment modes:

- Ack-0 (Fire-and-forget): This mode has the lowest latency but does not wait for any acknowledgment of message arrival.

- Ack-1: This mode provides better durability[1] and guarantees that the data was written to the replica leader.

- Ack-all: This mode ensures perfect durability but has higher latency. It ensures that data is written to both the replica leader and all of its followers[2]. This project will use Ack-all due to the criticality of the data.

[1] Durability means that the message is available even after one/some copies of message failure.
[2] Brokers that introduce redundancy

Producer Idempotency: This feature determines whether the producer may write duplicate messages to the topic partition in case of a retriable error. Idempotency is active by default and should be used in this project as well.

In conclusion, the Kafka Producer API is crucial for sending messages to Kafka. By configuring the producer appropriately, considering factors like replication, acknowledgment settings, and idempotency, developers can achieve the desired balance between durability, latency, and reliability for their project.

2.3.3.1 Configurations

- **BATCH_SIZE_CONFIG** is the size, in bytes, of the batch of records to be sent. Will wait for *LINGER_MS_CONFIG* milliseconds to send records in case batch is not fully populated;
- **LINGER_MS_CONFIG** is the time, in milliseconds, the producer will wait to send records in case current *BATCH_SIZE_CONFIG* is not fully populated yet;
- **KEY_SERIALIZER_CLASS_CONFIG** holds a class used for serializing the record keys;
- **VALUE_SERIALIZER_CLASS_CONFIG** holds a class used for serializing the record values;
- **BOOTSTRAP_SERVERS_CONFIG** holds the host and port of all Kafka brokers.

2.3.4 Kafka Streams API

The purpose of Kafka Streams consists in consuming records from selected input topics, process that data, and then produce it to selected output topics. For consuming and producing records the Kafka Streams API uses the Consumer and Producer Client APIs internally.

2.3.4.1 Topology

Since the Streams API utilizes a Consumer, a group_id can be provided to force multiple streams instances to benefit from having Consumers from the same Consumer Group.

A Streams instance logical structure is represented through what Kafka calls a topology, manifesting all the stream details except the configurations. A topology can be represented as a node graph consisting of source, processing and sink nodes. An example can be observed in figure 2.3

After the creation of a stream instance the topology cannot be changed, consequently the stream needs to be shut down to establish any changes.

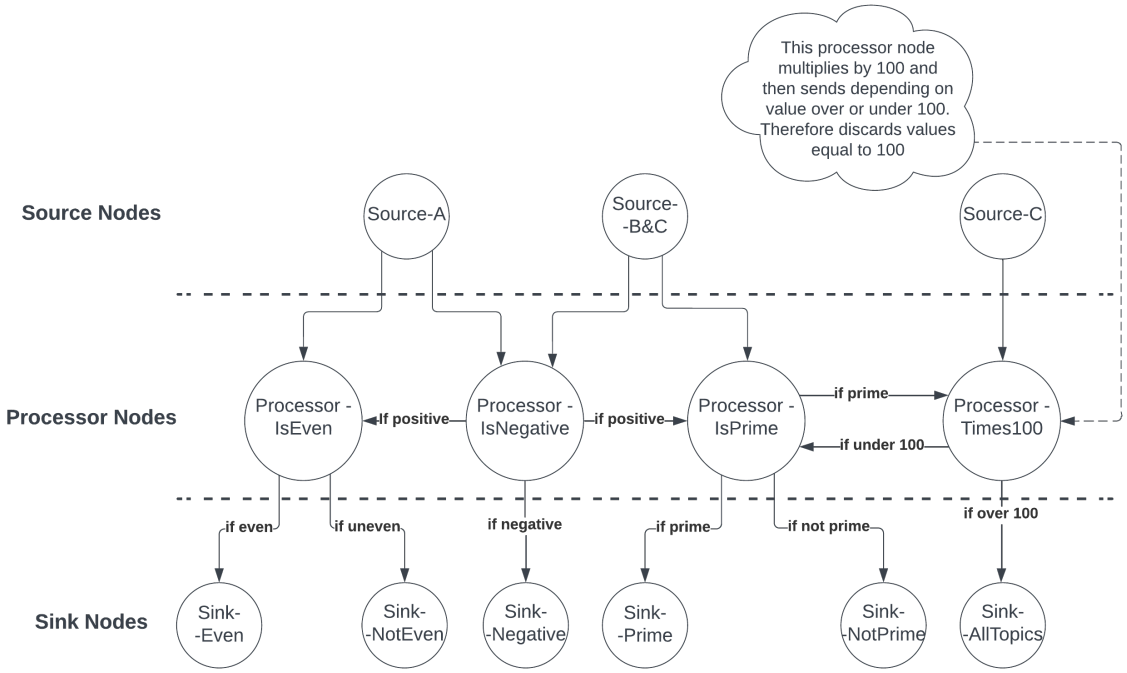

Figure 2.3: Topology example

- Source nodes represent consumption of records from specific topics, referred as input topics. The received records are then sent to selected processing and/or sink nodes;

- Sink nodes signify production of records to selected topics, referred as output topics. Records received originate from processor and/or source nodes;

- Processor nodes perform operations from received records from source and/or other processor nodes, and sends the processed data to the selected processor and/or sink nodes.

2.3.4.2 Tasks and Threads

When a Kafka Streams API Client is instanced, by default it possesses only one thread for itself. At any time a new thread can be added asynchronously. Each thread runs an established subset of tasks, a similar concept to the partitions distribution between Consumer instances in a Consumer Group. In the case of multiple Streams instances, by sharing the same stream id, the consumer group is shared too. Therefore leading to the same advantages such as parallelism and fault tolerance.

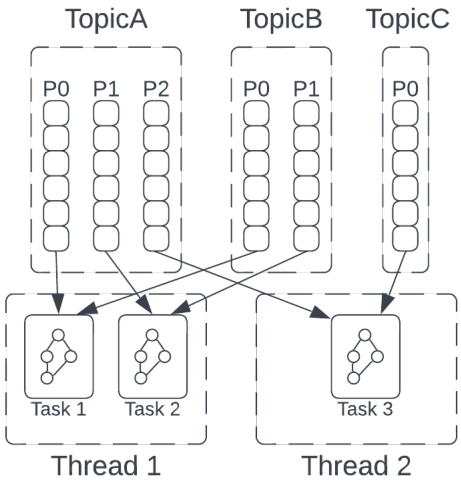

Figure 2.4: Tasks and Threads example

As observed in figure 2.4, the number of tasks created will be equal to the number of partitions the subscribed topic with the highest number of partitions has.

A task is a representation of the topology that the consumed records will have to go through, considering only the distributed subset of partitions associated.

Figure 2.4 also illustrates threads, each thread can be thought of as belonging to the same stream or different streams, for the purpose of the example.

Every time a stream instance or a thread of a stream is added or removed from a group subscription, a rebalance similar to the Consumer Group is triggered and the tasks distribution is recalculated.

2.3.4.3 Configurations

Kafka Streams Client API configurations consist of its own configurations but also Consumer and Producer normal configurations since Kafka Streams internally implements both Consumer and Producer Client APIs.

- **APPLICATION_ID_CONFIG** is an identifier for the stream instance. Also used for the internal Consumer Group id;
- **LINGER_MS_CONFIG** is a Producer configuration and should be changed to low values in case rate of messages is too slow for *BATCH_SIZE_CONFIG*, delaying processed records unnecessarily;
- **AUTO_OFFSET_RESET_CONFIG** is a Consumer configuration that holds importance every time a stream starts;

- **DEFAULT_KEY_SERDE_CLASS_CONFIG** holds a class used for serializing and deserializing the record keys;
- **DEFAULT_VALUE_SERDE_CLASS_CONFIG** holds a class used for serializing and deserializing the record values;
- **BOOTSTRAP_SERVERS_CONFIG** holds the host and port of all Kafka brokers.

2.4 Limitations

When considering an interaction between Kafka and a large scale application, two limitations become visible.

As mentioned in [2], as a rule of thumb, each broker shouldn't exceed 4000 partitions, and a cluster shouldn't exceed 200000 partitions. Due to this limitation, segmentation of large scale business logic will need to be divided by record key instead of by topics or partitions. e.g. associating each user to a topic with one partition would not work in a cluster if more than 200000 users exist, however instead of dividing users by topic, users need to be divided by the record's key.

This limitation presents an issue. A consumer that seeks a single user's records would consume all the records of a subscribed topic or partition, being that Consumer the one responsible for filtering the consumed records, resulting in discarding unwanted records.

Another limitation consists on the number of client connections that Kafka can handle at the same time. Multiple sources such as [3] recommend a maximum of 4000 client connections per broker. A client consists of any Kafka Client API instance (Consumers, Producers, Admins, and Streams). When a large scale system with hundreds of thousands of clients uses Kafka, Kafka won't be able to handle that amount of client connections.

Chapter 3

The Challenge

As mentioned in chapter 2, Kafka uses a binary protocol over TCP. A web browser cannot natively establish a TCP connection with a Kafka Cluster which leads to the need of a protocol translation intermediary infrastructure. That intermediary will then take advantage of the official Kafka Client APIs to interact with the Kafka cluster and expose a WebSocket protocol oriented API to establish bidirectional message based communication with the web browsers as depicted in figure 3.1. This procedure will allow any other platform supporting WebSockets to establish a connection with the mentioned intermediary infrastructure.

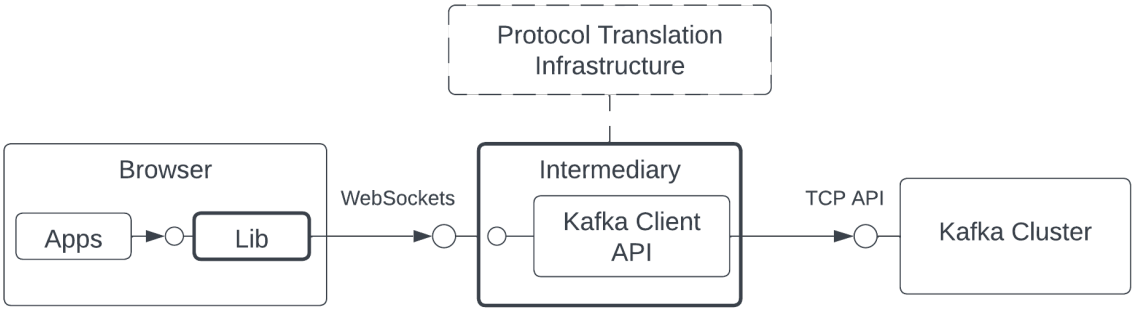

Figure 3.1: Challenge Representation Diagram

To address this limitation, the intermediary infrastructure solution provides the ability of associating a number of client browsers to a much reduced number of Kafka Clients.

With this approach, the web browser uses a library with the capability of communicating with the exposed intermediary infrastructure's WebSocket API.

Associated with the scalling limitation issue, since one of the goals of this project is to allow a high number of web clients to access Kafka then some strategy needs to me applied in order to supply each web client with only the information they seek.

For example: If a Battleship application with thousands of players and thousands of matches is using this solution then a strategy to represent a game in Kafka needs to be created since a topic cannot represent a game due to the low topic limit for this sort of scale

(4000 topics if each topic possesses a partition only).

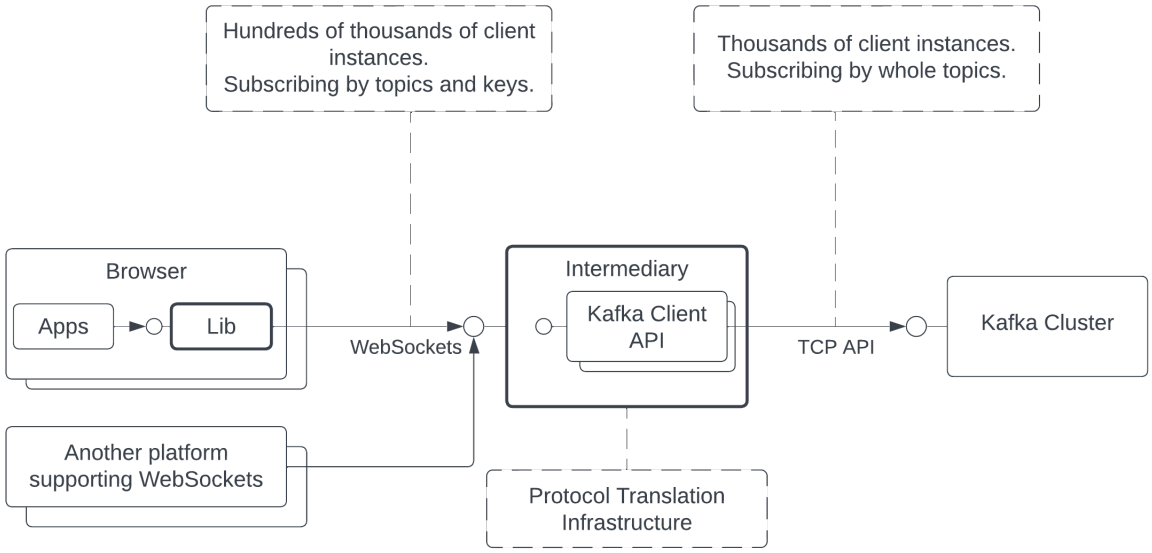

Figure 3.2: Detailed Challenge Representation Diagram

As mentioned in figure 3.2 the solution provides the option for the web clients to specify the keys from each topic that they seek to consume, while the intermediary infrastructure subscribes to the whole topics in Kafka. In the mentioned example a key could be used in each record to distinguish what information each web client will consume.

Chapter 4

Architecture

4.1 Solution Architecture

This solution's architecture revolves around the supply of Kafka based functionalities towards Web Browsers with our JavaScript library and other Web Client Applications supporting WebSockets.

A high-level illustration of the architecture of this solution can be seen in figure 4.1. The greyed elements are the project solution parts, otherwise it's external.

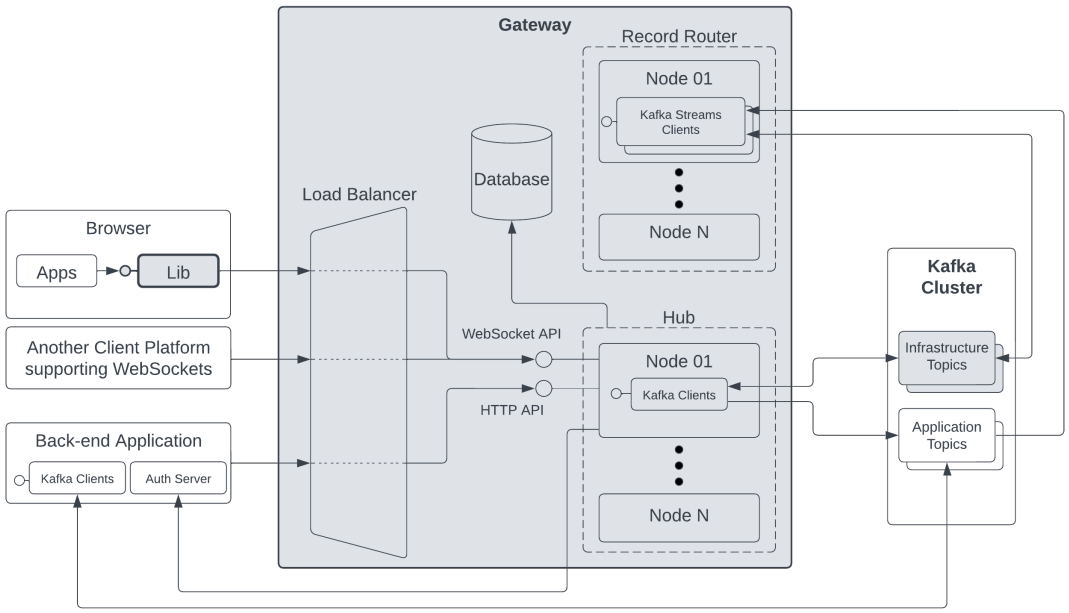

Figure 4.1: Solution's Macro Architecture Diagram

To accomplish this goal the following components were established:

- **Hub** – this component exposes an HTTP and WebSocket API, managing users and their respective requests regarding the Kafka based functionalities this project solution provides. This component can have multiple nodes running;

- **Browser JavaScript library** - provides a documented interface to establish a WebSocket connection with the Gateway's exposed WebSocket API;
- **Record Router** – A component running Kafka Streams Java Client API, responsible for filtering what records each Hub node will consume. This component can have multiple nodes running;
- **Load Balancer** - responsible for distributing the workload of WebSocket and HTTP requests among the Gateway's Hub running nodes.

And the following external components are expected:

- **Browser Application** - an application running in browser that uses our library to communicate with the Gateway;
- **Kafka Cluster** - a provided group of Kafka Server nodes that represent a Cluster;
- **Back-end Application** - a back-end application for the browser application that might communicate with Kafka and for convenience also be the authentication server;
- **Authentication Server** - a provided external authentication server to authenticate users.

4.2 WebSocket

First, let us introduce the WebSocket Protocol, they are essential in modern web applications as they enable a persistent and bidirectional connection between the client and the server. Unlike traditional HTTP, which follows a unidirectional request-response model, WebSockets facilitate real-time, full-duplex communication, allowing the server to push data to the client without the need for explicit client requests.

Before the advent of WebSockets, developers relied on techniques such as HTTP polling or long polling to achieve real-time updates. These methods involved sending periodic requests from the client to the server to check for new data. However, they incurred unnecessary overhead due to the frequent exchange of requests and responses, even when no new data was available.

WebSockets provide a more efficient solution to this problem. They establish a persistent connection between the client and the server, allowing both parties to send data whenever necessary. This bidirectional nature of WebSockets enables the server to proactively send updates to the client without the client requesting them.

4.2.1 Messages

The main message types in WebSocket communication are:

`Open:` The open message signifies the successful establishment of a WebSocket connection between the client and the server. It is sent by the server to the client after the initial handshake is completed. This message indicates that the connection is ready for data exchange.

`Message:` The message type is used to send actual data or information between the client and the server. It can contain any payload, such as text or binary data, depending on the application's requirements. This message type allows real-time communication, enabling instant updates and notifications from the server to the client and vice versa. The message may be coalesced or split by an intermediary. When a fragmented message is received, it can be either consumed partially or concatenated with subsequent messages to form the complete message.

`Close:` The close message is sent by either the client or the server to initiate the termination of the WebSocket connection. It indicates that the connection will be closed and no further communication will take place. The close message may include a status code and a reason to provide additional information about the closure.

`Ping/Pong:` The ping and pong messages are used for connection keep-alive and latency monitoring. The client sends a ping message to the server, and upon receiving it, the server responds with a pong message. These messages help ensure that the connection remains active and detect any potential network issues or delays. If a party does not receive a pong response within a certain timeframe, it may assume that the connection is no longer valid and initiate a close.

4.2.2 Authentication

Authentication in the WebSocket protocol plays a role in establishing secure and appropriate communication between clients and servers. While RFC 6455 does not specify a specific authentication mechanism during the WebSocket handshake, it states that WebSocket servers can utilize any client authentication method available to generic HTTP servers, such as cookies, HTTP authentication, or TLS authentication.

There are several options for passing authentication credentials from the web application to the WebSocket server:

Sending credentials as the first message in the WebSocket connection: This method is reliable and secure, although authentication is performed at the application layer rather than the protocol layer. It also makes monitoring authentication failures with HTTP response codes impossible.

Adding credentials to the WebSocket URI as a query parameter: While fully reliable, this approach is considered less secure as the URI may end up in logs, potentially exposing credentials. Authentication at the application layer can occur before the WebSocket handshake, enabling appropriate handling of authentication failures with HTTP 401 responses.

Setting a cookie on the domain of the WebSocket URI: Cookies are widely used and robust

mechanisms for sending credentials in HTTP applications. However, when the WebSocket server runs on a different domain from the web application, the Same-Origin Policy poses a challenge. A workaround involves creating a hidden iframe served from the WebSocket server's domain, sending the token to the iframe using postMessage, setting the cookie within the iframe, and then opening the WebSocket connection.

The chosen authentication method for the Kafka WebSocket Proxy application is URI parameter, due to the authentication being performed at the application layer, not allowing unauthorized WebSockets to be opened using the first message authentication. An enforcement of secure WebSockets, also referred as WSS, should be required in order to protect URI query parameter at transport level.

4.2.3 WebSockets over HTTP, SSE, and Long Polling

For this project, WebSockets were selected as the preferred communication protocol over HTTP, Server-Sent Events (SSE), and Long Polling. Here's why:

`Real-time, Bidirectional Communication:` WebSockets enable persistent and bidirectional connections between clients and servers. This feature is essential for real-time applications where instant data updates and seamless interaction between clients and the server are required. Unlike HTTP, which follows a unidirectional request-response model, WebSockets facilitate efficient and immediate communication without the need for explicit client requests.

`Reduced Overhead:` Compared to traditional techniques like HTTP polling or long polling, WebSockets offer a more efficient solution for real-time updates. With WebSockets, the need for frequent requests and responses to check for new data is eliminated, reducing unnecessary network overhead.

`Improved Performance:` WebSockets provide a lightweight and low-latency communication channel, enabling faster data transfer between the client and the server. This is particularly beneficial for applications that rely on real-time data streaming or interactive features.

`Simplified Development:` WebSockets simplify the development process by providing a standardized protocol and API for real-time communication. This reduces the complexity of implementing custom long-polling mechanisms or managing server-side events.

Considering these factors, WebSockets emerged as the optimal choice for this project.

4.3 Database

4.3.1 Data Model

The following section presents the graphical representation of the database in its Entity Association format without the attributes, describing the existing entities and the relations

between them.

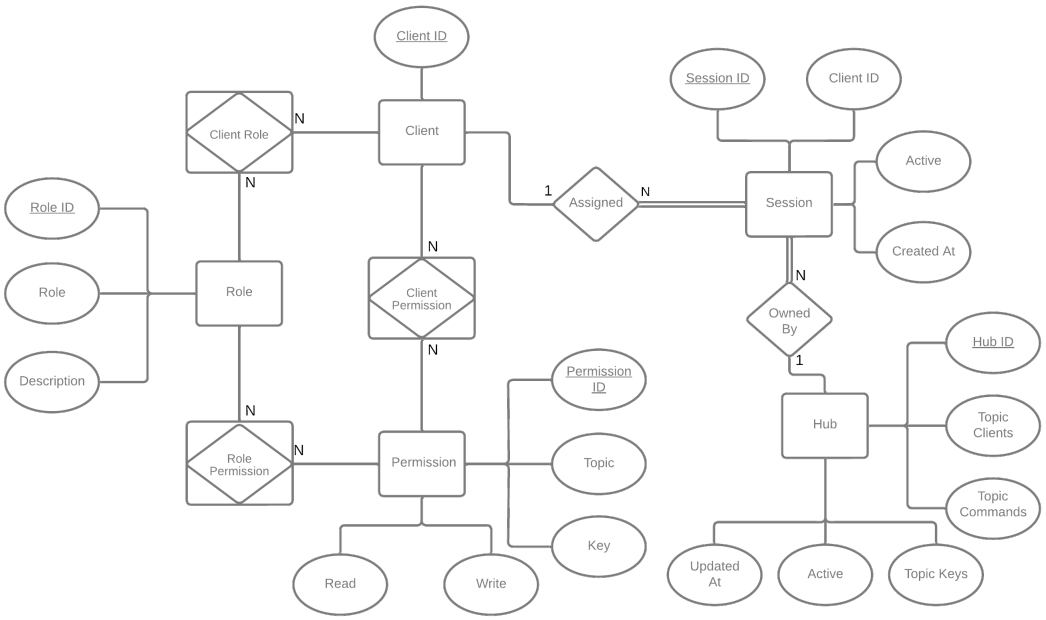

Figure 4.2: Gateway EA Model

Figure 4.4 illustrates the key entities in the system architecture. The central entity is the **Client**, which has a primary role in producing and consuming data from the Gateway. Each **Client** can be associated with a **Session**, representing either an active session or serving as a historical record of past sessions. For simplicity, the system supports only one session per **Client**.

Each **Session** is owned by a Gateway's **Hub** node, meaning that a WebSocket connection is established through a single **Hub**. The maintenance of a single connection in the Gateway is explained in detail in section 5.7.

The **Hub** entity possesses an ID and subscribes to two main topics: the clients topic and the commands topic. The clients topic pertains to messages consumed by the Record Router and ready to be distributed by the Hub node to the clients. The commands topic facilitates operations and communication between Hub nodes, such as disconnecting a client.

The **Hub** also maintains a keys topic, which represents the topics and keys the Hub is interested in. Specifically, it includes the topics and keys associated with clients.

4.3.2 Authorization

Regarding authorization, the system adopts a Role-Based Access Control (RBAC) mechanism with a modification that allows clients to have direct permissions. The authorization mechanism empowers developers to manage client permissions effectively. The key components of the authorization system are as follows:

- **Client Role**: Associates **Clients** with **Roles**.

- **Role Permission**: Establishes the relationship between **Roles** and **Permissions**.

- **Client Permission**: Represents the direct association between **Clients** and **Permissions**.

- **Role**: Defines the name and description of a role.

- **Permission**: Dictates the topic and, if applicable, the key. A null key represents permissions for all keys. Each permission can have read, write, or read/write authorizations.

These authorization components collectively manage and enforce access control policies for clients within the system. Further details on the authorization subject will be discussed later.

In summary, the system architecture comprises Clients, Sessions, and Gateways. Clients interact with the Gateway to produce and consume data. Sessions track client connections, and Gateways manage WebSocket connections and facilitate communication between clients. The system employs RBAC with modifications for client permissions, ensuring secure and controlled access to system resources.

4.4 Authentication

As part of the Kafka Web Gateway Application, an external authentication approach has been adopted to align with the project's goal of providing a service for developers and businesses. An external authentication application is utilized to authenticate web clients when communicating with the service. In this context, an HTTP GET request with a bearer token contract is chosen for simplification purposes. However, it is acknowledged that a stronger authentication method such as JWT (JSON Web Tokens) would be necessary to enhance application security and standardization.

To illustrate a possible use case for web client authentication, the following flow is proposed:

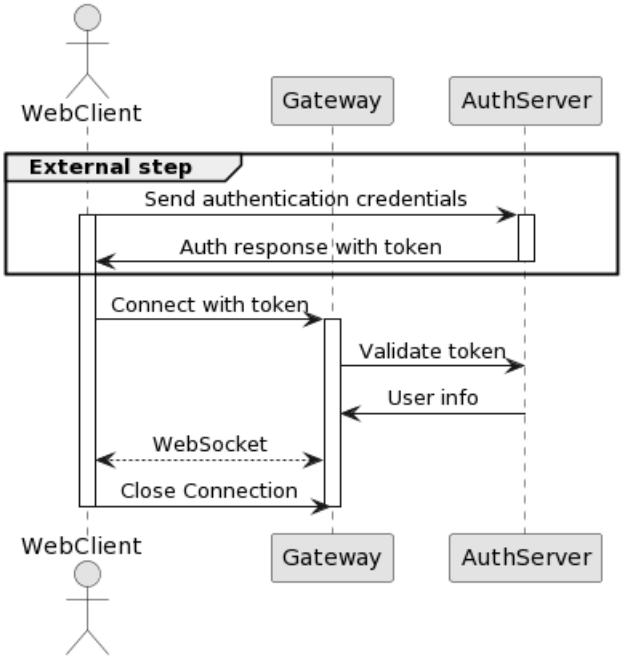

Figure 4.3: Web Client authentication flow

1. The web client initiates a step, possibly external, to obtain an authentication token required to authenticate with the Hub.
2. After receiving the token by passing authentication credentials, the web client can proceed with the authentication process.
3. The web client sends a WebSocket connect request to the Hub, including the token in the URI parameter (as previously mentioned, the token is included as a WebSocket URI parameter since the standard WebSocket RFC does not allow custom headers).
4. The Hub receives the request and intercepts the handshake/upgrade HTTP connection to a WebSocket. An intercept class within the Hub parses the URI parameter and sends an HTTP GET request to the authentication server if the parameter is present.
5. The authentication server responds with a 200 OK status and the client details, typically the client Id, in the response body to indicate successful authentication.
6. Upon successful exchange, a WebSocket connection is established between the web client and the Hub, allowing further command exchange.

For administrative tasks, any request to the API must be performed by an admin. Admin authentication is managed through the database, and an HTTP API is provided to handle the creation, permission assignment, and deletion of admins and their respective access tokens. The first admin, also known as the owner, is created with a Gateway's Hub node and is unique, with restricted modification permissions. No other owner can exist within the application.

4.5 Authorization

Regarding the authorization mechanism, a different approach is employed. An HTTP API is exposed to applications (admins) to regulate permissions. Permissions are stored in the database used by the Gateway Hubs. To exemplify a possible use case, consider a chatroom application where permissions are directly assigned to clients.

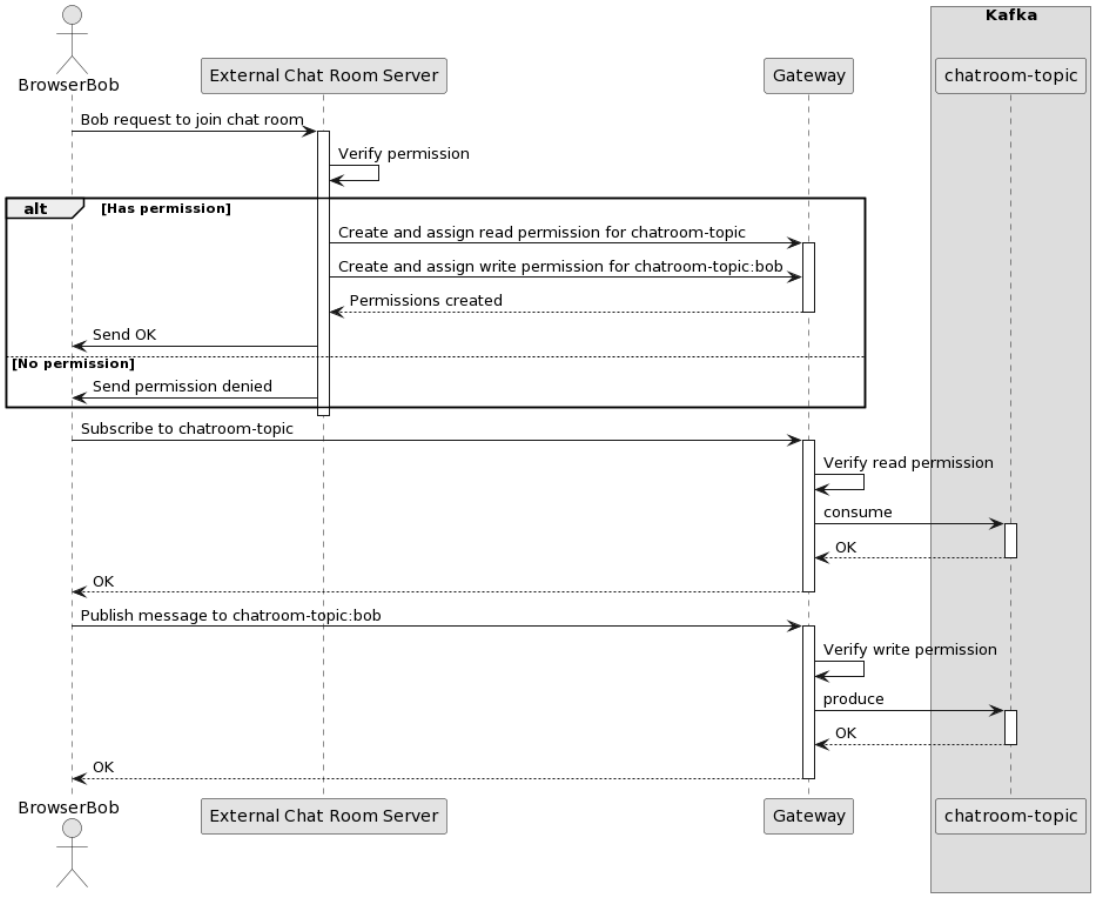

Figure 4.4: Possible Web Client authorization

1. The web client requests to an external server to join a chatroom
2. If it has permission, the external server does HTTP requests to create and assign a read permission for the "chatroom-topic" with the Gateway Hub. This external server must be an admin (have an admin token) to be able to do this requests.
3. Also requests to the Gateway Hub to assign a write permission for the chatroom-topic topic and the key bob. This allows for only the user Bob to write with this key.
4. After the external server gave permissions for the bob client to listen and write using it's key, the client can now subscribe and publish to the chatroom-topic with the Gateway.

Administrators (admins) have additional special permissions, including administrative rights to modify the authentication server, create or delete other admins and tokens, and manage permissions and roles for web clients.

4.6 Kafka Topics

4.6.1 Kafka Infrastructure Topics

Kafka infrastructure topics are created for the purpose of the Gateway's components intercommunication only. Some examples in section 5.3.1 Kafka Infrastructure Topics.

4.6.2 Kafka Application Topics

Kafka application topics are created within the context of the application using the Gateway. These topics can either be created through a Gateway user or an external server that has access to the Kafka Cluster.

4.7 Gateway Hub

The Gateway's Hub component is responsible for exposing the WebSocket and HTTP APIs and handle the clients requests. This components manages the users authentication, permissions and Kafka based functionalities through message commands.

One of the main Kafka functionalities this component provides is the consumption and production of messages/records. To work in conjunction with the Record Router component for that purpose, the Hub component only produces records to the applications topics, and consumes and produces records to the infrastructure topics.

Aside from users, which are application related users, there are infrastructure related users called admins, responsible for administrating the gateway components with some operations such as changing the authentication server for example.

4.7.1 WebSocket API

The WebSocket API is the primary interface for clients to communicate with the Kafka WebSocket Gateway.

Clients send commands as JSON payloads over the WebSocket connection. The Gateway Hub processes the commands exactly as it was sent, partial messages are not supported, and sends back appropriate responses, ack or error messages indicating their success, or client messages data to be consumed from subscriptions.

All commands will be encapsulated in these objects:

```
open class Command(val type: String)

data class ClientMessage(
    var messageId: String?,
    val command: Command
)
```

The *messageId* is currently used for two scenarios:

- To validate the success or failure of an operation, for example, when a publish command is sent with a *messageId* from the client to the server, the server will respond either an Ack or Err message representing the success or not of the operation, with the same *messageId*.

- For the server to acknowledge a record sent to the client, in this case, the client must send an Ack message if it consumed/received the record, otherwise retries can happen if such functionality is enabled.

The WebSocket API supports the following commands:

- **Subscribe**: The `subscribe` command allows clients to subscribe to one or more topics and receive messages from them. Clients can specify the topics they want to subscribe to along with optional keys for topic partitioning. The gateway validates the client's permissions before subscribing to the topics.

  ```
  data class Subscribe(val topics: List<TopicType>): Command("subscribe")
  ```

  This is the message that is sent from the Gateway to the client from subscribed topics.

  ```
  data class Message(
      val topic: String,
      val partition: Int,
      val key: String?,
      val value: String,
      val timestamp: Long,
      val offset: Long
  ) : Command("message")
  ```

- **Consume**: The `consume` command is used by clients to consume messages from subscribed topics. This command is typically used to retrieve historical messages. Not currently supported in this architecture, further analysis is needed to implement such.

  ```
  data class Consume(val maxQuantity: Int?) : Command("consume")
  ```

- **Publish**: The `publish` command allows clients to publish messages to a specific topic. Clients can provide the topic name, optional key for partitioning, and the message payload. The gateway validates the client's permissions before allowing the message to be published.

  ```
  data class Publish(
      val topic: String,
      val key: String?,
      val value: String
  ) : Command("publish")
  ```

- **Pause**: The `pause` command is used to pause the consumption of messages from subscribed topics. Clients can send this command to temporarily stop receiving messages.

```
data class Pause(val topics: List<TopicType>) : Command("pause")
```

- **Resume**: The `resume` command is used to resume the consumption of messages from paused topics. Clients can send this command to resume receiving messages after pausing.

```
data class Resume(val topics: List<TopicType>) : Command("resume")
```

These commands are used internally, they should not be sent directly by a client:

- **Ack**: The `ack` command is used by clients to acknowledge the successful processing of a message. Clients can send an acknowledgement for a received message, indicating that it has been successfully processed. The gateway uses acknowledgements to track message delivery status.

```
data class Ack : Command("ack")
```

- **Error**: The `error` command is sent by the gateway to notify clients about any errors or failures that occur during message processing or command handling.

```
data class Err(val message: String?) : Command("error")
```

The util *TopicType* object is defined as the following:

```
data class TopicType(val topic: String, val key: String?)
```

4.7.2 Browser Library

The appendix A is a TypeScript implementation of a web client library for interacting with the Kafka WebSocket Gateway. Typescript is transpiled to JavaScript to be able to run on Browsers. This library allows clients to connect to the gateway, subscribe to topics, publish messages, and receive real-time updates.

The GatewayClient class represents the main entry point of the library. Here's an explanation of its components and methods:

`Constructor:` Initializes the client with the URL of the Kafka WebSocket Gateway. Set up various properties such as the WebSocket connection, subscriptions, operation callbacks, and status flags.

`Connect:` Establish a WebSocket connection to the gateway using the provided authentication token. It sets up the necessary event listeners for handling open, message, and close events. When the connection is successfully established, the optional onOpenCallback is invoked.

`Disconnect:` Closes the WebSocket connection and performs necessary clean-up such as clearing subscriptions and logging the disconnection.

`onOpen:` Registers a callback function to be executed when the WebSocket connection is opened successfully.

`onClose:` Registers a callback function to be executed when the WebSocket connection is closed.

`Subscribe:` Subscribes to a topic with an optional key and registers a message callback function. Send a subscribe command to the server and returns a promise that resolves when the subscription is acknowledged or rejects if an error occurs.

`Publish:` Publishes a message to a topic with an optional key. Send a publish command to the server and returns a promise that resolves when the publication is acknowledged or rejects if an error occurs.

4.7.3 HTTP API

In addition to the WebSocket API, the Kafka WebSocket Gateway also exposes an HTTP API for admins to interact with the Gateway. The HTTP API provides a RESTful interface for managing administrative tasks and performing operations that do not require real-time streaming. The HTTP API includes the following endpoints:

4.7.3.1 Admin

GET /api/admin/adminId: Retrieves the admin with the specified adminId.
POST /api/admin: Creates a new admin.
PUT /api/admin/adminId: Updates the admin with the specified adminId.
DELETE /api/admin/adminId: Deletes the admin with the specified adminId.

Example of a request and response body related to the creation of an Admin:

```
data class Admin(
    var adminId: Int?,
    val name: String,
    val description: String?,
    val owner: Boolean,
    val administrative: Boolean,
    val permission: Boolean
)
```

In the response, the same object will be returned with the *adminId* fullfiled if it was successful.

4.7.3.2 AdminToken

POST /api/admin/adminId/tokens: Creates a token for the admin with the specified adminId.

DELETE /api/admin/adminId/tokens/tokenValidation: Deletes the token with the specified tokenValidation for the admin with the specified adminId.

4.7.3.3 ClientPermission

POST /api/client-permission: Creates a new client permission.

DELETE /api/client-permission/clientId/permissionId: Deletes the client permission with the specified clientId and permissionId.

4.7.3.4 ClientRole

POST /api/client-role: Creates a new client role.

DELETE /api/client-role/clientId/roleId: Deletes the client role with the specified clientId and roleId.

4.7.3.5 Permission

POST /api/permission: Creates a new permission.

DELETE /api/permission/permissionId: Deletes the permission with the specified permissionId.

4.7.3.6 Role

POST /api/role: Creates a new role.

DELETE /api/role/roleId: Deletes the role with the specified roleId.

4.7.3.7 RolePermission

POST /api/role-permission: Creates a new role permission.

DELETE /api/role-permission/roleId/permissionId: Deletes the role permission with the specified roleId and permissionId.

4.7.3.8 Setting

Configurations of the Gateway, for example, the authentication server URL.

POST /api/setting: Creates a new setting.

PUT /api/setting/name: Updates the setting with the specified name.

DELETE /api/setting/name: Deletes the setting with the specified name.

For a full API documentation, check here.

4.8 Gateway Record Router

This component's purpose is to ensure that the Gateway's Hub component receives all the records it is subscribed to through its users, with an accuracy of not only the topics but the keys too, which is a functionality Kafka does not possess. Otherwise, each of the Hub's nodes would be consuming more records than needed and would discard many of the received records while filtering and routing the records to the clients.

To perform this objective, the Record Router component communicates with the Hub Component through Kafka infrastructure topics to be informed of the current subscriptions and to deliver the records each Hub node seeks through subscriptions. And with that knowledge, consumes records from all the application topics that all the Hub nodes are subscribed to. Guaranteeing each message is processed only once.

More detailed information about this component in section 5.4

Chapter 5

Gateway Implementation

This chapter describes with more detail what each of the solution's components does and the flow of some use-cases. The delivered implementation is not a business ready software to be distributed, it's to prove the solution is viable.

5.1 Technologies

The Kafka Web Gateway application is made up of Hub servers developed in Kotlin using the Spring framework, with Apache Tomcat serving as the servlet engine. It utilizes a PostgreSQL database, with JDBC for communication, to store various information such as administration details, permissions, and gateway information. Additionally, the Record Router component is built on Spring and employs Kafka Streams.

To facilitate remote management of the gateway, an HTTP API is exposed by all Gateway's Hub nodes, allowing users to modify the Gateway State through remote methods. For interaction with the gateway, a JavaScript or TypeScript web client library is provided, enabling users to leverage the WebSocket protocol.

Collectively, these components integrate to form the Kafka Web Gateway Application, enabling communication between Kafka and web clients.

5.2 Gateway Hub properties and lifecycle

In this section, we explore the configuration of a Gateway Hub node and it's lifecycle.

5.2.1 Configuration

General server properties:

- `server.port`: This property specifies the port number on which the embedded server should listen. When running multiple deployments of an application, it's important to ensure that each deployment uses a different port to avoid conflicts.

- `spring.kafka.bootstrap-servers:` This property is used to configure the bootstrap servers for connecting to a Kafka cluster. Kafka is a distributed platform, and the bootstrap servers are the initial set of servers to which the client connects to discover the full cluster. You need to provide the hostname and port number of at least one Kafka broker.

- `spring.datasource.url:` This property specifies the URL or connection string for the database that the application will connect to.

- `spring.datasource.username:` This property sets the username used to authenticate the application's connection to the database.

- `spring.datasource.password:` This property specifies the password associated with the username used for database authentication.

As previously mentioned, there is a unique admin, called owner. This owner that cannot be deleted or created through the HTTP API and is only created through the instantiation of a Gateway Hub node.

There are currently supported the following Gateway properties:

- `gateway.config.auth.server:` The authentication URL to authenticate Web clients. It can also be changed using the HTTP API with the 'auth-server' setting name.

- `gateway.config.owner.username:` The owner (admin) username. Creates in the database if an owner does not already exist.

- `gateway.config.owner.description:` The owner description.

- `gateway.config.owner.token:` The owner token to be used for HTTP API authentication. Only the owner can alter its tokens.

As previously mentioned, there is a unique admin, called owner. This owner that cannot be deleted or created through the HTTP API and is only created through the instantiation of a Gateway Hub.

5.2.2 Lifecycle

On the initialization of a Gateway Hub node, a few requirements need to execute, respectively:

- Related to the properties of this node, the creation of the owner if one doesn't exist already and the authentication server URL.

- A successful connection to a Kafka cluster must happen in order to, create topics associated with this Hub node, respectively, the gateway-id-clients, gateway-id-keys

and gateway-id-commands. Also, to register consumers for these topics and the system-topic. Topic's responsibility will be explained in the next section.

- The insertion of this new Hub node on the database with its information for other Hub nodes to be able to know which topics they can use to communicate with.

On the shutdown, the deactivation of the Hub node in the database is performed.

5.3 Kafka Topics

5.3.1 Kafka Infrastructure Topics

As mentioned in 4.6.1, Kafka infrastructure topics are created for the purpose of the Gateway's components intercommunication only. The infrastructure topics used are:

- **system-topic** - holds information regarding the startup, shutdown and other operations that might interest multiple components;
- **gateway-id-clients** - a topic populated by the Gateway's Record Router component for the Hub nodes to consume. Each Hub node will possess it's own topic, replacing the "id" with the Hub's id;
- **gateway-id-keys** - a topic used for the Gateway's Hub nodes to communicate their subscriptions to the Record Router component. Each Hub node will possess it's own topic, replacing the "id" with the Hub's id;
- **gateway-id-commands** - a topic used for the Gateway's Hub nodes to communicate with each other, populating the topic of the node they seek to communicate with. Therefore, every Hub node will possess it's own topic, replacing the "id" with the Hub's id;

5.3.2 Kafka Application Topics

As mentioned in 4.6.2, Kafka application topics are created within the context of the application using the Gateway. These topics can either be created through a Gateway user or an external server that has access to the Kafka Cluster.

5.4 Gateway's Record Router

This component uses Kafka Streams Java Client API to consume all the records from the topics the Gateway's Hub nodes clients are subscribed to, which are then forwarded to the respective Hub node based on record keys requested by each client of said Hub node.

This component is scalable through the creation of more processes running this service, meaning that since this component is implementing Kafka Streams, advantages like parallelism, load-balancing and fault-tolerance, inherited from the Consumer Groups, would be manifested between this component's nodes too.

The cause for introducing this component to the project arose from a challenge met in the Gateway's Hub component implementation when connecting to Kafka to adopt its functionalities, illustrated in figure 5.1. Each user subscription is based on topics or specific record keys within each topic, but a Kafka Consumer can only consume records based on offsets or time, which means that if a topic receives a high amount of records, but the gateway only needs records with a certain key, then the gateway's consumer ends up consuming unwanted records that will then be discarded. This would mean the filtering of records by key would happen in each Hub node, replicating the filtering of possibly the same records, which is not ideal.

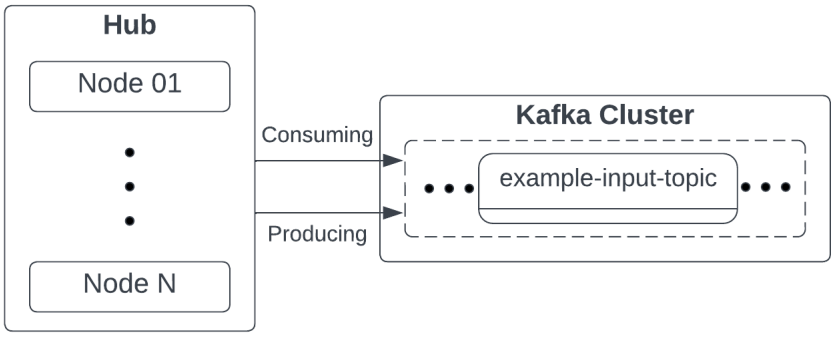

Figure 5.1: Direct usage of Kafka functionalities through Gateway's Hub

To solve this issue, a new component was added, implementing Kafka Streams.

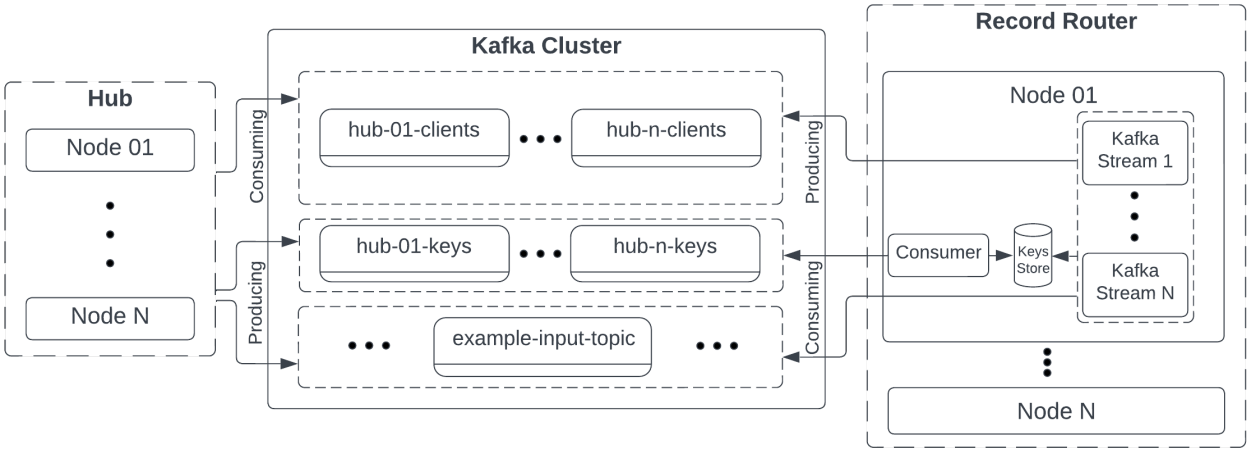

Figure 5.2: Record Router Component Solution Architecture

As observed in figure 5.2 this solution's flow would start with each Gateway's Hub node producing records containing the topics and respective keys as needed by their users. These records would be produced to specific topics, a different topic for each Hub node, where each record key would contain the desired topic and the record value would hold the keys to consume from that topic. That topic would be configured to have a compact cleanup-policy, retaining only the latest value of each key. That way, the topic would hold only the currently needed topics and respective keys. Those topics names would follow the template "gateway-id-keys" (e.g. "gateway-01-keys").

Each Hub node also has an associated topic which will be the only topic they will consume from, meaning it contains all the records from all the request topics and keys.

Record Router would then take advantage of a Consumer to consume from all the "gateway-id-keys" topics and save it locally. Using that data, this component has the awareness of what topics and keys each Hub node requires. This component's main functionality consists in consuming from the topics that all the Hub nodes requested and produce the desired records to the Hub nodes topics through Kafka Streams API instances.

5.4.1 Stream Logic and Topology

As mentioned in the Kafka Streams API section, each stream instance possesses a topology representing its logic.

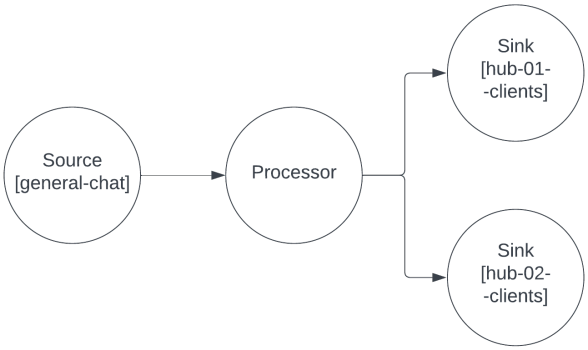

Figure 5.3: Kafka Stream Topology Use Case example

In figure 5.5 is displayed an example of a Kafka Streams Topology created in this project for a Chatroom use-case. The example depicts all the currently running Gateway's Hub nodes which may or may not have subscriptions towards a topic named "general-chat" that represents a single chat in an application. The existence of this stream and corresponding topology mean that at least one of the Hub nodes clients possesses a subscription over the referred topic.

As mentioned, this topology displays three layers, the source node that represents the node responsible for consuming from topics which in this case is consuming from one topic only as that was the solution established for this project. Then can be observed the processor node, which is responsible for routing each of the received records from the source node to the respective sink nodes through consultation of the current subscriptions, and if needed, edit the received records. And finally the sink nodes that represent all the currently running Hub nodes that depending on the processor routing outcome, may or may not receive records consumed from the source node.

In this project's solution all the processor nodes unconditionally edit metadata, more precisely a header is added with the name of "origin-topic" and the corresponding submitted value being the topic from which the record was just consumed. This operation is performed due to the act of the gateway consuming the records from the "gateway-id-clients" topic, the records topic is considered to be the current one, in this case the "gateway-id-clients". Therefore, for the gateway to determine the topic each record is originated from, in order to forward the records to the respective clients, the gateway reads the metadata header "origin-topic" to perceive the topic that record was originally produced to.

5.5 Gateway's Hub

The Hub is the bridge between the Web and this project's solution. With the possibility of being composed of multiple nodes and with the responsibilities of handling client's authenti-

cation, permissions, subscriptions and Kafka based functionalities provided by the Gateway.

In following operation examples it is assumed a WebSocket connection is already ongoing.

5.5.1 Subscriptions

In order for a subscription requested by a client to be applied, the respective read permission needs to be present in the gateway's database. The mentioned permissions are submitted by special users denominated as 'admin' that communicate with the Gateway through the exposed HTTP API.

After a client's subscription request is validated through the presence of the respective permission, the Gateway proceeds to save that subscription in the database and in memory. The subscriptions in memory will then be used for routing of the received records to each client.

Then, the gateway will produce records to the "gateway-id-keys" topic for the Record Router component to use for routing the desired records to the current Hub node, representing the currently subscribed topics and respective keys related to that topic from all the currently subscribed clients in that Hub node.

5.5.2 Record Consumption

If a client has active subscriptions and an open connection with the Hub node then the client will receive the desired records. The gateway will always be consuming record from a specific topic with a name format of "gateway-id-client", a topic reserved for each node. This topic will be populated will all the records that the Record Router component sends for the determined Hub node.

After the Hub node consumes each record, the record will then be routed to the clients that seek those records depending on their origin topic and key. To perform this routing operation the subscriptions saved in memory will be used to determine which clients require each consumed record.

5.5.3 Record Production

Similar to the subscription process, a client to accomplish the production of a record to a topic, the corresponding writing permission needs to be present in the gateway's database. If the record production request is permitted, the gateway will directly produce that record to the desired topic.

5.6 Subscription and Consumption Use-Case

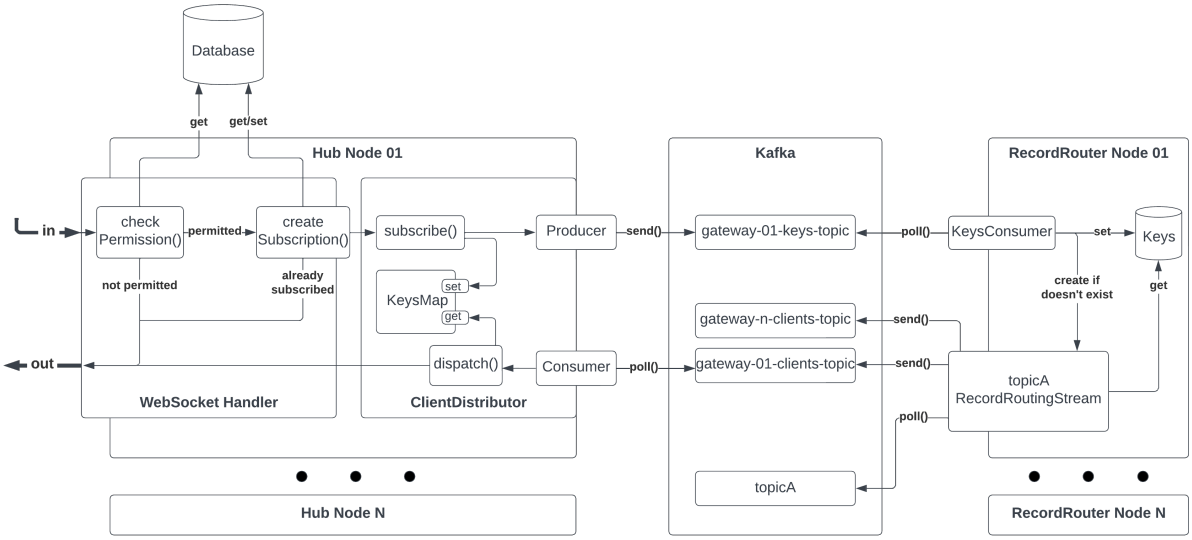

Figure 5.4: Gateway Subscribe Use-Case Example

The figure 5.4 depicts a Gateway subscribe use-case where a client requested a subscription.

The request reaches the WebSocketHandler class, responsible for receiving and handling WebSocket messages, and since the command received is "subscribe" then the Gateway starts by checking if the client possesses the permission to subscribe to the chosen topic and key. For the purpose of explaining, it will be assumed that the Hub node id is "01" and the chosen subscription is to topic "topicA" and key "1".

If the permission was denied the gateway will then respond with failure, else the gateway will proceed with the subscription by searching the subscriptions in the database and in case it's already present the gateway will respond with failure, else the subscription will be inserted in the database and forward the subscription to the ClientDistributor class, responsible for setting up the in-memory and Kafka subscriptions. In memory the solution found consists of a map that possesses a Pair of Topic and Key for the map key and the value being a list containing the sessions of all the clients subscribing to that topic and key, e.g. the new Pair would be {"topicA", "1"}. This way, when the gateway receives a record, it forwards the record to the corresponding clients through a search in this map.

After saving the values to this map, the subscription will also be sent by the Producer to a topic called "gateway-01-keys-topic" that will hold all the currently active topics and keys the Hub node seeks.

The Record Router nodes will be consuming from the "gateway-01-keys-topic", distributing the generated tasks for stream processing parallelism. Each of the nodes will check if the stream to process records from the new subscription already exists, in each node exists

a stream to consume from a subscribed topic, if not then it's created and the topic and keys are saved in memory. The created stream will consume from the subscribed topic, add the custom metadata header "origin-topic" to every record that will be forwarded to a Hub node, and forwards said records to the subscribing nodes based on topic and key saved in memory to the respective "gateway-id-clients-topic" of each Hub node. In this case, every record with the key "01" is forwarded to the "gateway-01-clients-topic".

The Hub node has a Consumer listening to the topic "gateway-01-clients-topic" which holds all the records the Record Router component determined that the Hub node seeks based on the subscriptions present in the "gateway-01-keys-topic". After each record is consumed, that record's key and "origin-topic" metadata header will be used to search the in-memory subscriptions map to forward each record to the respective clients.

5.7 Synchronization between Gateway Hubs

In this section, we explain how we can communicate between Gateway Hubs. When we have multiple WebSocket servers, usually it's needed ways to communicate between them, for example, to notify other servers that the client X disconnected, or notify a server shutting down. In this case, we explore the scenario when the Gateway only allows one active session per client. If a client disconnected without terminating the WebSocket connection, the connection is still pending for the ping pong timeout. But now, the client reconnected with the Gateway but to another Gateway Hub node. So, a method for the new Gateway Hub node to inform the initial Gateway Hub node to terminate it's WebSocket connection with the client.

To handle this use case, we utilized Kafka as the underlying engine for communication between the Gateway Hub nodes. Synchronization between WebSocket servers is a common requirement, and since we were already using Kafka, it seemed fitting to leverage its capabilities.

First, let's recall some concepts. The *hub-X-commands* topic in Kafka represents operations to be executed within the Gateway Hub node it is associated to, where each operation is represented by a key-value pair. The key represents the type of operation, such as the deletion of a WebSocket session, while the value contains the corresponding details of the operation.

Figure 5.5 illustrates the Gateway reconnect use case, showcasing the interaction between the various components involved, in this case the client X did not close the connection with Gateway Hub 2 and is attempting to connection with Gateway Hub 1.

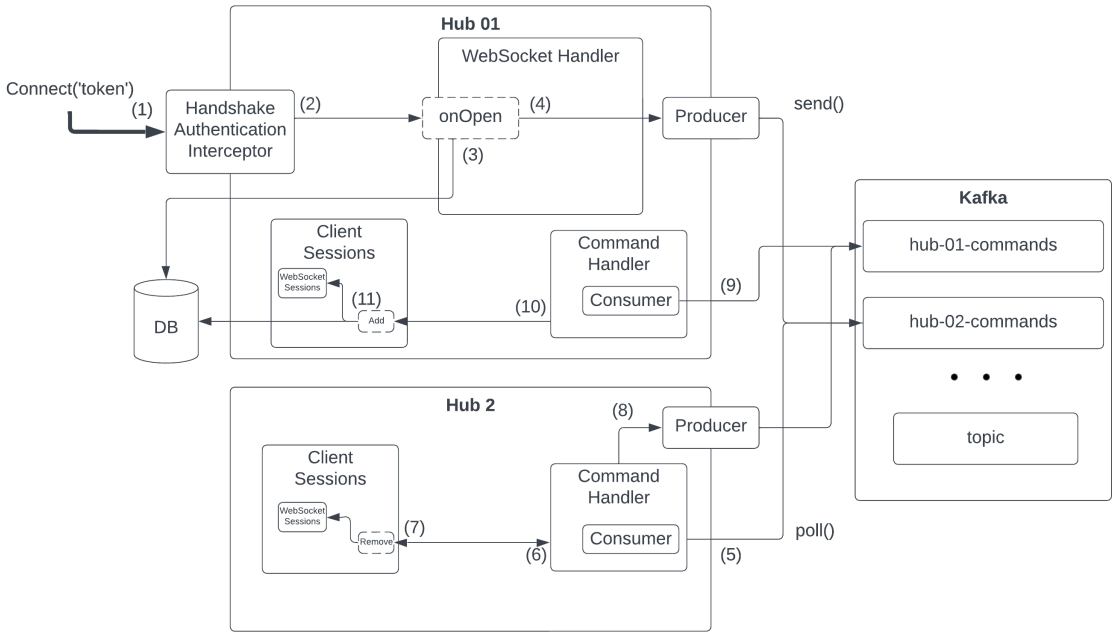

Figure 5.5: Gateway reconnect use case

1. The web client executes a connect command providing a token to authenticate with.
2. The *Interceptor* intercepts the WebSocket handshake/upgrade and validates the token against the external authentication server.
3. The Hub 1 consults the database to check if there's any active sessions for the client.
4. Found an active session for the client belonging to the Hub 2, so it sends a message through the singleton Kafka producer to the hub-02-commands topic requesting to remove the session.
5. The hub-02-commands consumer on Hub 2 receives the message requesting the deletion of the client session.
6. The *Client Sessions*, responsible for storing client WebSocket sessions, receives the request and closes the client WebSocket session from its storage.
7. After removal, Hub 2 produces a success response to the hub-01-commands topic indicating the success of the operation
8. The hub-01-commands consumer on Hub 1 receives the message indication the success of the session removal.
9. Informs the *Client Sessions* that it's ready to accept the new WebSocket session.
10. Adds the WebSocket session to the storage and registers in the database.

As previously referred, this is a possible solution to communicate between Gateway Hubs.

5.8 Browser library implementation

The GatewayClient class represents the main entry point of the library. Here's an explanation of its internal methods.

`HandleMessage:` Handles incoming messages received from the gateway. It checks the message type and performs the appropriate actions, such as executing the message callback for subscriptions, sending acknowledgments, or handling errors.

`Send:` Sends a JSON payload to the server over the WebSocket connection.

`SendAck:` Sends an acknowledgment for a received message to the server.

`getSubscriptionKey:` Generates a unique key to identify subscriptions according to the topic and key.

The `IGatewayMessage` interface represents the structure of a message received from the gateway. It includes properties such as message ID, command type, topic, partition, key, value, timestamp, and offset.

The `ISubscription` interface represents the structure of a subscription. It includes properties such as the topic, key, message callback, last message ID, and acknowledgment status.

To use this library, you need to import the GatewayClient class and create an instance by providing the URL of the Kafka WebSocket Gateway. Then, you can call the connect method with an authentication token to establish a connection. Afterward, you can use the subscribe method to subscribe to topics, the publish method to publish messages, and register callbacks using onOpen and onClose methods if needed.

This library is built using the browser WebSocket API[4], for other systems supporting WebSockets that want to use this library, adaptations are needed to support such.

Chapter 6

Conclusion and Future Work

This chapter describes challenges faced during the execution of this project, what was accomplished and how, and future work improvements.

6.1 Conclusion

This project was brought to us as only a problem concerning the connection between a Web Browser and Kafka. With no knowledge of the Kafka System we performed an in-depth research of Kafka's ecosystem and behaviour.

After acquiring and understanding of Kafka's System we devised the solution's architecture and a proof of concept implementation.

Since browsers cannot establish a connection with Kafka, therefore unable to use Kafka's functionalities. This project's solution, the Kafka Web Gateway, an intermediary, resolves that problem and also overcomes Kafka's limitations of small-scale maximum client connections and inability to filter sought messages by a message key, useful to allow better granularity in stored data.

To establish the connection between Web Browsers and the Gateway we developed two documented interfaces implementing WebSockets to allow a bidirectional flow of messages. One interface in the form of a JavaScript library meant to run in the Web Browser Clients, and another situated in the Gateway, in a component with the possibility of multiple nodes exposed to the Web, named Hub. Also possessing an HTTP API for administrative operations.

Kafka's Consumer and Producer Client APIs were used for communication with Kafka for both application and infrastructure context purposes. However, to filter the messages by key, another component with with the capacity for multiple nodes was developed, named Record Router. This component implements Kafka Streams Client API to forward sought messages by topic and key, guaranteeing each message is processed only once.

To demonstrate the feasibility of the project's solution, a proof of concept in the form of a Kotlin project was implemented with all the solution's architecture components, deployable

via docker, along with a browser Chatroom application. This application uses the browser WebSocket interface library to establish a connection to the Gateway, and a Server application as that application's back-end and authentication server.

6.2 Future Work

The solution's architecture was designed to sustain all the Kafka Client's API operations, requiring only extra infrastructure topics to communicate between components if needed.

However, the proof of concept implementation does not currently support all the functionalities we desired.

Currently, the Kafka functionalities the Gateway implementation supports are:

- **subscribe -** used for clients to subscribe to topics and optionally keys too. Right now also starts consuming records automatically;
- **publish -** user for clients to send messages to Kafka topics.

And the Kafka functionalities that are not implemented are:

- **consume -** used to start consuming the subscribed topics;
- **pause -** used to pause specific subscriptions;
- **resume -** used to resume specific paused subscriptions.

More work needs to be done regarding authentication methods and overall Gateway security and customization regarding certificates and business needs. An improvement should use JWT tokens[1] To add more functionalities, no major changes need to be implemented in the Hub component, but the Record Router might need some effort.

Aside from missing functionalities, the Record Router component currently possesses an inappropriate strategy to save the subscriptions, since those are saved in-memory and duplicated between the nodes. To fix this incident, a solution such as using the Kafka Streams API local storage functionality might be the most appropriate. Latency between production and consumption of records by the Hub component would need to be tested in order to define which strategy is better, since one of Kafka's priorities is low latency between the production and consumption of a record.

[1]JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

Bibliography

[1] Apache Software Foundation. Apache kafka, Jan 2011.

[2] Jun Rao. Apache kafka supports 200k partitions per cluster, 2018.

[3] Huawei Cloud. Kafka specifications, 2023.

[4] Mozilla Corporation. The websocket api (websockets) - mdn, 2004.

Appendix A

Browser Library

```
export interface IGatewayMessage {
    messageId: string,
    command: {
        type: string,
        topic: string,
        partition: string,
        key: string | undefined,
        value: string,
        timestamp: string,
        offset: number
    }
}

export interface ISubscription {
    topic: string;
    key: string | undefined;
    messageCallback: Function | undefined;
    lastMessageId: string | undefined;
    ackSent: boolean;
}

export default class GatewayClient {
    readonly #url: string;
    #socket: WebSocket | undefined;
    #fullTopicsSubscriptions: Map<string, ISubscription>;
    #keysSubscriptions: Map<string, ISubscription>;
    #operationsCallback: Map<string, { expiresAt: number, operationCallback: Function | undefined }>;
    #onOpenCallback: Function | undefined;
    #onCloseCallback: Function | undefined;
    #active: boolean;

    constructor(url: string) {
        this.#url = url;
        this.#socket = undefined;
        this.#fullTopicsSubscriptions = new Map();
        this.#keysSubscriptions = new Map();
        this.#operationsCallback = new Map();
        this.#onOpenCallback = undefined;
        this.#onCloseCallback = undefined;
        this.#active = false;
    }

    connect(token: string) {
        if (this.#socket?.readyState !== WebSocket.CLOSED) {
            this.disconnect(true)
        }

        this.#socket = new WebSocket(`${this.#url}?token=${token}`);

        this.#socket.onopen = () => {
            console.log('WebSocket connection established.');
            if (typeof this.#onOpenCallback === 'function') {
                this.#active = true
                this.#onOpenCallback();
            }
        };

        this.#socket.onmessage = (event) => {
            this.#handleMessage(JSON.parse(event.data));
        };

        this.#socket.onclose = () => {
            console.log('WebSocket connection closed.');
            if (typeof this.#onCloseCallback === 'function') {
                this.#active = false;
                this.#onCloseCallback();
            }
        };

        return this;
```

```
}

disconnect(clearSubscriptions: boolean) {
    if (this.#socket) {
        this.#socket.close();
        this.#socket = undefined;
        if (clearSubscriptions) {
            this.#keysSubscriptions.clear();
            this.#fullTopicsSubscriptions.clear()
        }
        console.log('WebSocket connection disconnected.');
    }
}

reconnect(token: string) {
    if (this.#socket?.readyState !== WebSocket.CLOSED) {
        this.disconnect(false)
    }

    this.connect(token);

    const fullTopics = Array.from(this.#fullTopicsSubscriptions.values());
    const keysSubscriptions = Array.from(this.#keysSubscriptions.values());

    for (const subscription of fullTopics) {
        this.subscribe(subscription.topic, subscription.key, subscription.messageCallback);
    }

    for (const subscription of keysSubscriptions) {
        this.subscribe(subscription.topic, subscription.key, subscription.messageCallback);
    }
}

onOpen(callback: Function) {
    this.#onOpenCallback = callback;
}

onClose(callback: Function) {
    this.#onCloseCallback = callback;
}

subscribe(topic: string, key: string | undefined, messageCallback: Function | undefined): Promise<void> {
    const messageId = crypto.randomUUID()

    const promise = new Promise<void>((resolve, reject) => {
        const callback = (message: any) => {
            if (message.command.type === 'ack') {
                resolve();
            } else if (message.command.type === 'error') {
                reject(new Error(message.command.message));
            }
        };

        this.#operationsCallback.set(messageId, {
            expiresAt: Date.now() + 60_000,
            operationCallback: callback,
        });
    });

    const subscriptionKey = this.#getSubscriptionKey(topic, key);
    if (key == null) {
        if (!this.#fullTopicsSubscriptions.has(topic)) {
            this.#fullTopicsSubscriptions.set(topic, {
                topic,
                key: undefined,
                messageCallback: messageCallback,
                lastMessageId: undefined,
                ackSent: false,
            });
        }
    } else if (!this.#keysSubscriptions.has(subscriptionKey)) {
        this.#keysSubscriptions.set(subscriptionKey, {
            topic,
            key: undefined,
            messageCallback: messageCallback,
            lastMessageId: undefined,
            ackSent: false,
        });
    }

    // Send the subscribe command to the server
    const payload = {
        messageId,
        command: {
            type: 'subscribe',
            topics: [{
                topic: topic,
                key: key
            }]
        }
    };
    this.#send(payload);

    return promise;
}
```

```
    publish(topic: string, key: string | undefined, message: string): Promise<void> {
        const messageId = crypto.randomUUID()

        const promise = new Promise<void>((resolve, reject) => {
            const callback = (message: any) => {
                if (message.command.type === 'ack') {
                    resolve();
                } else if (message.command.type === 'error') {
                    reject(new Error(message.command.message));
                }
            };

            this.#operationsCallback.set(messageId, {
                expiresAt: Date.now() + 60_000,
                operationCallback: callback,
            });
        });

        const payload = {
            messageId,
            command: {
                type: 'publish',
                topic: topic,
                key: key,
                value: message
            }
        };
        this.#send(payload);
        return promise;
    }

    #handleMessage(message: IGatewayMessage) {

        if (message.command.type === 'ack' || message.command.type === 'error') {
            console.info(`Ack || error received for ${message.messageId}`);

            let operationsCallback = this.#operationsCallback.get(message.messageId);

            if (operationsCallback) {
                if (operationsCallback.operationCallback) {
                    operationsCallback.operationCallback(message)
                }
                this.#operationsCallback.delete(message.messageId)
            }

        } else if (message.command.type === 'message') {

            const subscriptionKey = this.#getSubscriptionKey(message.command.topic, message.command.key);
            let subscription;

            if (this.#fullTopicsSubscriptions.has(message.command.topic)) {
                subscription = this.#fullTopicsSubscriptions.get(message.command.topic)
            } else {
                subscription = this.#keysSubscriptions.get(subscriptionKey);
            }

            // if it has subscription, execute callback
            if (subscription && message.messageId !== subscription.lastMessageId) {
                if (subscription.messageCallback) {
                    subscription.messageCallback(message)
                }
                subscription.lastMessageId = message.messageId;
            }

            this.#sendAck(message.messageId);
        } else {
            console.error("Unknown message received, check gateway for root cause.")
        }
    }

    #send(payload: any) {
        if (this.#socket && this.#socket.readyState === WebSocket.OPEN) {
            const message = JSON.stringify(payload);
            this.#socket.send(message);
        }
    }

    #sendAck(messageId: string) {
        const payload = {
            messageId,
            command: {
                type: 'ack'
            },
        };
        this.#send(payload);
    }

    #getSubscriptionKey(topic: string, key: string | undefined) {
        return `${topic}-${key}`;
    }
}
```