UNIVERSITY OF TARTU
Institute of Technology
Computer Engineering Curriculum

Jonathan Karu

Towards a GraphQL Proxy for Apache Kafka

Bachelor's Thesis (12 ECTS)

Supervisor:
Riccardo Tommasini, PhD

Tartu 2019

Towards a GraphQL Proxy for Apache Kafka

Abstract:
Apache Kafka is a open-source stream-processing framework which is quickly becoming an industry standard for event-driven applications. Kafka is written in Scala and, thus, general purpose JVM (java virtual machine) programming languages can be used to write native applications that interact with a Kafka cluster using the Kafka protocol. However, the need to generalize the interaction to other languages, e.g. python, has pushed Kafka developers to abstract the Kafka protocol using the REST API. The REST proxy is a component of the Kafka suite that enables the use of a RESTful (representational state transfer) API (application programming interface) to communicate with a Kafka cluster, and thus allowing to control Kafka using HTTP.

GraphQL is a data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data [1]. Recently GraphQL has been gaining traction. GraphQL enables developers to define which data will be returned by a request, moreover GraphQL allows to interact with this data type through a full-fledged query language.

In this thesis we propose the GraphQL proxy, which enables communication with a Kafka cluster using GraphQL.

Võtmesõnad:
Kafka, GraphQL, web, streams, Avro, REST

CERCS: P175 Informatics, systems theory;

Liikudes Apache Kafka GraphQL Proxy poole

Lühikokkuvõte:
Apache Kafka on avatud lähtekoodiga voogtöötluse raamistik, millest on saamas tööstusstandard südmustepõhiste rakenduste arendamisel. Kafka on kirjutatud Scalas ning seega saab kasutada üldotstarbelisi JVM (java virtuaalmasin) programmeerimiskeeli, et kirjutada rakendusi, mis töötavad Kafka protokolliga. Sellegipoolest, on vajadus Kafka laiendamiseks teistesse keeltesse, näiteks pyhon, pannud Kafka arendajad välja töötama Kafka jaoks REST (representational state transfer) API (application programming interface). REST proxy on Kafka komponent, mis lubab kasutada RESTful APIt, et suhelda Kafka klastriga ning seega lubab kasutada Kafkat üle HTTP.

Hiljuti on populaarsust kogunud veebiliides GraphQL. GraphQL on avatud lähtekoodiga andmepäringu keel, mis lubab arendajatel rangelt defineerida päringule vastu saadetavad andmed. Kuna Avro skeemid on väga sarnased GraphQLi skeemidele, siis võib GraphQL olla parem viist Kafka klastriga suhtlemiseks kui REST.

Keywords:
Kafka, GraphQL, veeb, vood, striimimine, Avro, REST
CERCS: P175 Informaatika, süsteemiteooria;

Contents

List of Figures

List of Tables

Abbreviations, constants, definitions

API - Application Programming Interface
REST - Representational State Transfer
ACL - Access Control List
JSON - Javascript Object Notation
RESTful - an application that implements REST architecture constraints
HTTP - HyperText Transfer Protocol
JVM - Java Virtual Machine

Unsolved issues

1 Introduction

Apache Kafka is an open-source stream-processing framework which enables the use of fault-tolerant, scalable and durable messaging logs. Kafka proves useful when a system needs to decouple, which means that services work asynchronously. Confluent, the company who maintains Kafka, supports different native clients in several language to communicate with Kafka. However many organizations demand more freedom in choosing a language to implement Kafka platform [6]. With this in mind, Confluent created the Kafka REST Proxy, which eases the use of Kafka adopting web based technologies. REST is an architectural style to ease the communication between web services. REST has been a de facto standard through many years since its introduction in 2000 by Roy Fielding [7], however in 2015, Facebook released GraphQL, a spec which enables developers to design their API in a graph-like structure and define a type hierarchy.

The idea to develop GraphQL, came from the fact that GraphQL is capable of returning exactly the data, which is requested and nothing more, in contrast to RESTful interfaces which may require multiple requests and return excess data and it also enables to model the API in a graph-like structure, which is more natural than what REST provides. The goal of this thesis is to study the idea of using GraphQL instead of REST for designing a proxy to Kafka protocol that allows to interface with the Kafka cluster.

This thesis is divided in to four sections: a) background, which describes REST APIs, GraphQL and Kafka, b) reasoning to use graphql with kafka, c) implementation and d) conclusion. In the GraphQL introduction the author explains the main concepts and attributes of GraphQL, such as the syntax, fields, types, variables. In the Kafka introduction the author explains Apache Kafka and its main concepts, such as topics, brokers, consumers, producers and the schema registry, The reasoning section brings forth the main reasons to use GraphQL instead of REST when communicating with a Kafka cluster. The implementation section focuses on the GraphQL requests and resolver functions which communcate with the Kafka cluster in the implementation of the project.

1.1 GraphQL vs REST

This section will briefly go over the main differences along with pros and cons of GraphQL and REST.

Both REST and GraphQL have entry points to the data, in GraphQL they are query, mutation and subscription types and in REST they are endpoints. Both interfaces also have a way to differentiate a request that reads data or writes data. GraphQL enables clients to request for related data about a resource in a single request, however in REST, this requires multiple requests. When querying for data in GraphQL as opposed to REST, the client can specify which data it would like to receive, whereas in REST, the client gets what the server has prepared, this is referred to as overfetching.

Overfetching with REST

When using RESTful interfaces to communcate with a server, requests usually return more data than necessary, this is called overfetching. This leads to higher bandwith requirements, because irrelevant data for the client is usually returned as well. GraphQL alleviates this problem, by allowing the client to specify which fields to return on requests. This means that the server filters out the fields on the data which is returned. In REST, however, the client would have to get the full request object and then filter out the necessary fields, a workaround in REST architectures would be to define multiple endpoints, each defining a subset of fields which the client might request, however this often makes things too difficult, this would also mean making fields resources, which is against the goal of REST.

If the mean temperature of each room were to be calculated, a request for all the temperatures would be made. If this was done in REST, the object would return irrelevant information, which is the timestamp for the taken measurement. Figure 1.1 shows the server response for the REST query. Figure 1.2 shows the server response for the GraphQL query which only queries the necessary information.

```
Request endpoint: /api/v1/temperatures
Response JSON object:
{
    "data": {
        "temperatures": [
            {
                "room": 412,
                "value": 21,
                "timestamp": "1589112659430"
            },
            {
                "room": 412,
                "value": 22,
                "timestamp": "1589112659435"
            },
            {
                "room": 410,
                "value": 21,
                "timestamp": "1589112659432"
            }
        ]
    }
}
```

Figure 1.1. Example GET request and response for temperatures

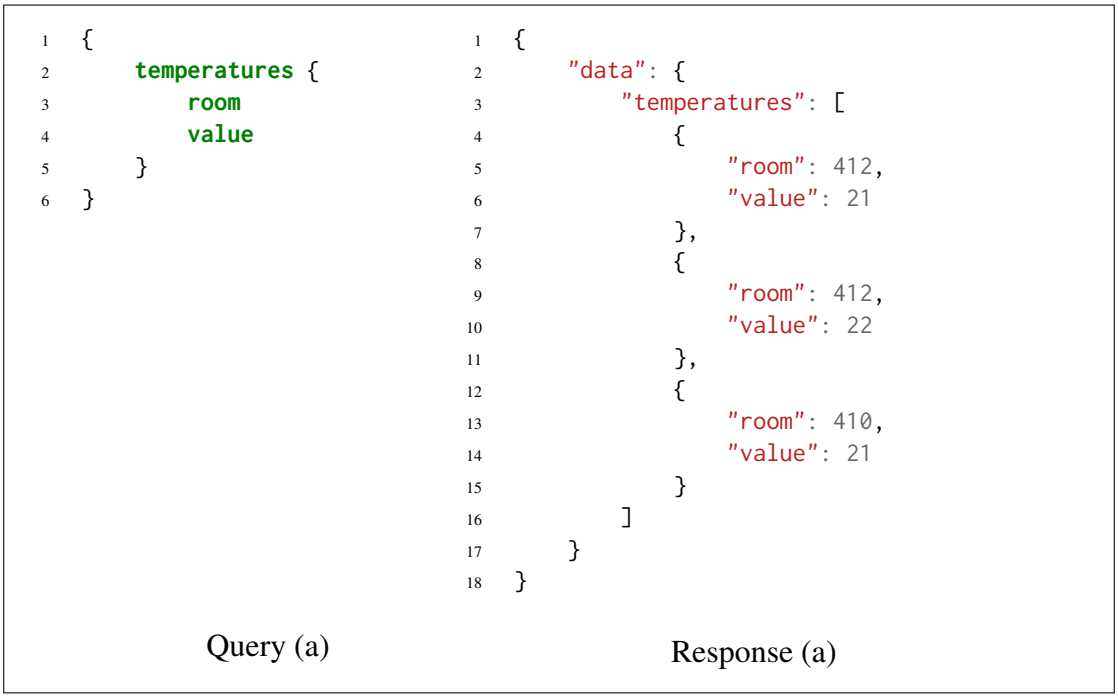


Figure 1.2. Example GraphQL query and response for temperatures

Overfetching is an important issue when dealing with data intensive applications, because unnecessary data makes the system more demanding in terms of resources. When making continuous requests, the amount of unnecessary data should be kept to a minimum, to keep the latency as low as possible.

2 Background

This section explains the main ideas and technologies related to GraphQL, Kafka and the proxy implementation. To understand Kafka, the publish/subscribe model must be explained first and to understand the pros and cons of GraphQL a definition and comparison with REST will be explained.

2.1 Web APIs

A **web API** is a construct to allow developers to create complex functionality more easily, it is a way to abstract more complex code away and provide a more easier syntax on the web. A web API can be used by clients to create, modify, fetch or delete resources provided by the server. Web APIs are usually developed with ease of use in mind and have different implementations, in this chapter we will introduce two web APIs: REST and GraphQL.

2.1.1 REST

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems [7]. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements [7].

REST principles
REST has 6 guiding constraints which must be satisfied if an interface needs to be referred as RESTful [8]:

1. **Client-server** - By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

2. **Stateless** - Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

3. **Cacheable** - Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

4. **Uniform interface** - By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple

architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

5. **Layered system** - The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.

6. **Code on demand (optional)** - REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

The key concept in any REST application is the resource. A resource is an object (e.g. image, html document) which can have methods to create, change, delete or return the object. A resource has an resource identifier attributed to it, which contains information about the resource. Currently the most popular way to transport resources is using the JSON (javascript object notation) data model.

To access and manipulate resources, resource methods are required. The most common methods to work with resources are HTTP methods (GET, PUT, POST, DELETE). An example GET method can be seen on figure 1.1.

Usually REST APIs are defined as a list of endpoints, where an endpoint refers to some underlying resource. Table 2.1 contains some example endpoints along with methods and actions, when a method is called on an endpoint.

Method	Endpoint	Action
GET	/temperatures/412	Return temperatures for room 412
POST	/temperatures/412	Add a new temperature for room 412
DELETE	/temperatures/412	Delete a temperature from room 412

Table 2.1. Example REST endpoints, methods and actions

2.1.2 GraphQL

GraphQL is a query language for APIs and a runtime for fulfilling those queries with existing data [1]. A GraphQL service is created by defining the schema and the corresponding backend resolvers. A GraphQL schema defines the possible set of requests and types that the current application is capable of using. An example GraphQL schema can be see on figure 2.1. GraphQL allows developers model the data with objects and relations in mind (as graphs). Figure 2.6b shows a graph representation of the schema on figure 2.6a.

```
type Query {
    temperature: Temperature
}
type Temperature {
    room: Int
    value: Int
    timestamp: String
}
```

Figure 2.1. Example GraphQL schema

In a GraphQL service, the set of types defined is called the service “schema” [9]. A schema is composed of types and root operation types (query, mutation and subscription), which determines the place in the type system where the operations begin.

In the schema on figure 2.2, which has the capabilities of getting a hotel room temperature as well as sending a room temperature to the server, the type RoomTemperature has a field "location" with type "Room", which resolves to fields: "number" of type Int and "occupied" of type Boolean.

```
type Query{
    roomTemperature: RoomTemperature
}
type Mutation{
    addRoomTemperature(roomNumber: Int, value: Int): RoomTemperature
}
type RoomTemperature{
    room: Room
    value: Int
    timestamp: String
}
type Room{
    number: Int
    occupied: Boolean
}
```

Figure 2.2. GraphQL schema with Query and Mutation types for RoomTemperature model

On figure 2.3, a query with a return type of Temperature is defined, which contains data about a temperature measurement. The query will return a Temperature object, which

has fields for the location of the measurement, the value in degrees and the timestamp of the measurement. For this example query a java backend is used, which can be seen on figure 2.3. Using this resolver, the query on figure 2.4a can be sent to the server.

```java
public DataFetcher getTemperature(){
    return datafetchingenvironment -> {
        return ImmutableMap.of(
            "room", 412,
            "value", 21,
            "timestamp", "1589112659430"
        );
    }
}
```

Figure 2.3. Example Java resolver for query Temperature

```
{
    temperature {
        room
        value
        timestamp
    }
}
```

GraphQL query (a)

```
{
    "data": {
        "temperature": {
            "room": 412,
            "value": 21,
            "timestamp": "1589112659430"
        }
    }
}
```

Server response (b)

Figure 2.4. Temperature query and response

GraphQL currently supports the following **scalar types**:

Scalar type	Description
Int	signed 32 bit integer
Float	signed double-precision floating-point value
String	UTF-8 character sequence
Boolean	true or false
ID	unique identifier, serialized as a string

Table 2.2. GraphQL scalar types

GraphQL currently has three types of **operations** available [10]:

- Query - a read only fetch.
- Mutation - a write followed by a fetch.
- Subscription - a long-lived request that fetches data in response to source events.

A *query* can be compared with a HTTP GET request, which only returns some data from server, usually without any side effects. A *mutation* operation can be compared with a HTTP POST request, which posts some data to the server and then returns some information about the posted data (usually the data itself), mutations usually have side effects. A *subscription* is a continuous flow of data from the server, it can be seen as a live query (connection stays open until cancelled).

A GraphQL operation can select a set of information it needs from the server and will receive exactly that information [11]. A selection set is similar to selections in SQL. An example of querying only the timestamp field from a temperature object can be seen on figure 2.5.

```
{
    temperature {
        timestamp
    }
}
```

Query (a)

```
{
    "data": {
        "temperature": {
            "timestamp": "1589112659430"
        }
    }
}
```

Response (b)

Figure 2.5. Query for only the timestamp of a temperature

A selection set is mostly composed of **fields**, which describe discrete pieces of information available to request within a selection set [12]. All GraphQL operations must eventually resolve to scalar types which can also be represented as leaf nodes in a graph [12]. Figure 2.7 shows a query for a roomtemperature object, in which all queried fields resolve to scalar values.

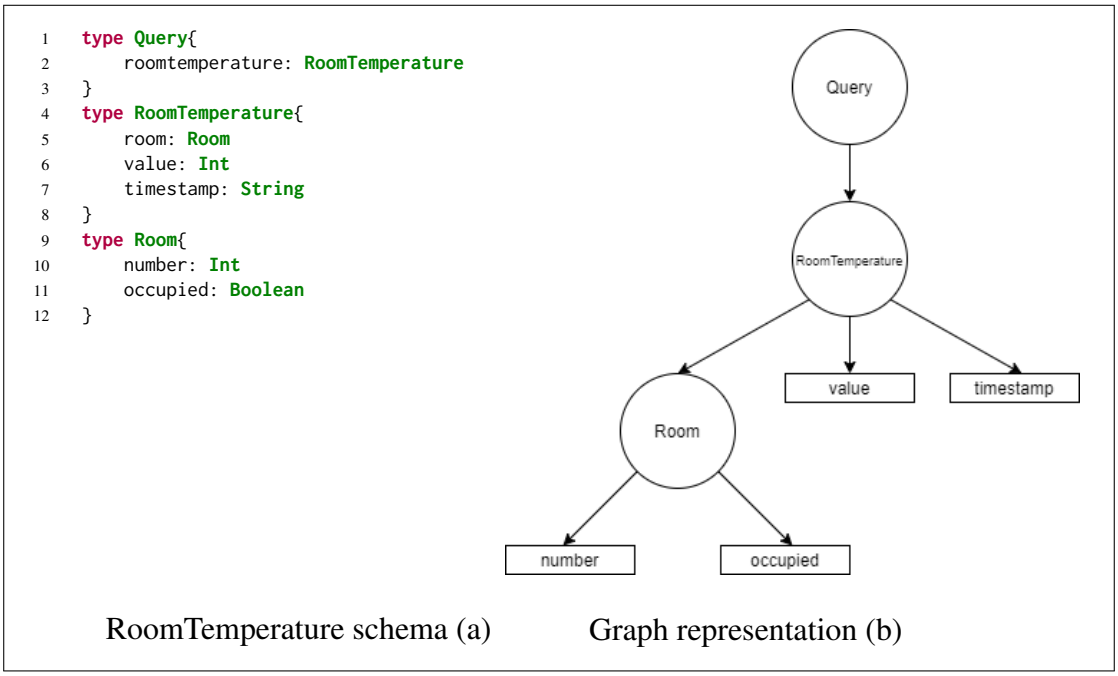


Figure 2.6. Graph representation and schema for RoomTemperature model

```
{
    roomtemperature {
        room {
            number
            occupied
        }
        value
    }
}
```

Query (b)

```
{
    "data": {
        "roomTemperature": {
            "room": {
                "number": 412,
                "occupied": false
            },
            "value": 21
        }
    }
}
```

Response (b)

Figure 2.7. Query for RoomTemperature

Fields are **functions**, which return values and accept **arguments** that may alter their behaviour [13]. These arguments often map directly to function arguments within a GraphQL backend implementation. An example query that supports arguments can be seen on figure 2.8. Here the query temperatureFromRoom accepts an integer, which specifies what room the client is interested in querying.

```
{
    temperatureFromRoom(room: 412) {
        value
        timestamp
    }
}
```

Figure 2.8. Query which accepts room number as a parameter

2.2 Streaming data

In today's world streaming data is more relevant than ever, due to an increased amount of microservices, streaming services and growth in general computational power and the amount of IoT devices. Figure 2.9 shows an infographic, which brings forward the fact that we are truly living in the age of information. Since the amounts of data that modern systems have to handle is growing at an increasing speed, the need for new technologies is also on the rise. To handle large amounts of data, asynchronicity can not

be looked over, the following chapter introduces the underlying principles to build these asynchronous systems.

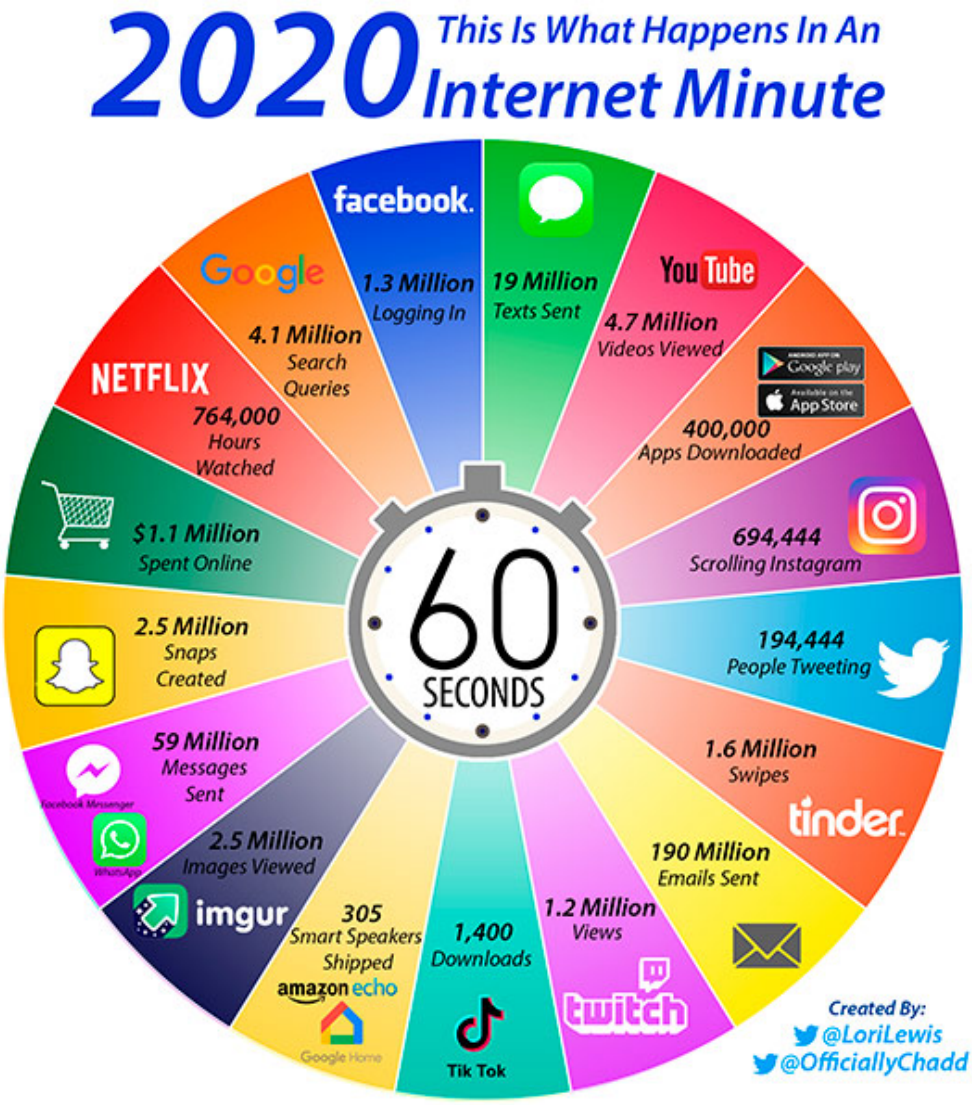


Figure 2.9. The Internet Minute [2]

Data which is continuously generated by various sources is called Streaming data. Streaming data is processed using Stream Processing engines. An example of streaming data would be temperature measurements. For example a hotel has 500 rooms, which is periodically taking temperature measurements from sensors and then sending the data to a central server. This data is represented as an unbounded stream of events, which has no predefined volume and an event is for example a temperature measurement.

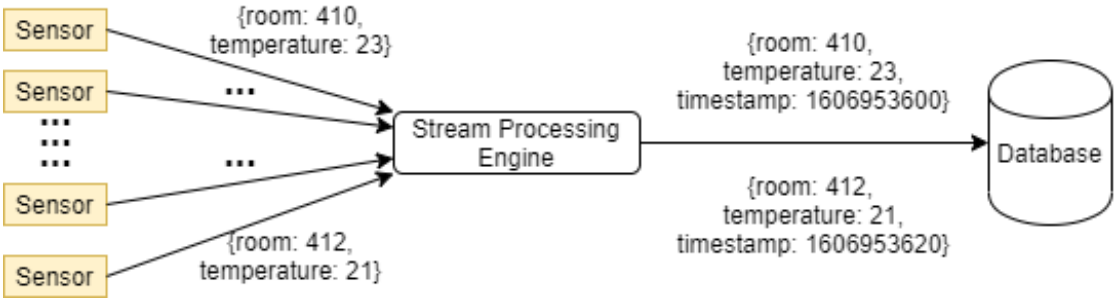


Figure 2.10. Example temperature measurement data streaming pipeline

2.3 Pub/Sub

Due to increased amount of data which is being processed by devices, synchronous services are not a feasible choice for large-scale application any more.

Pub/Sub also known as publish/subscribe is an asynchronous messaging service that decouples services that produce events from services that process events [14]. Subscribers have the ability to express their interest in an event and are subsequently notified of any event, generated by a publisher, which matches their interests [3]. The strength of this messaging service is that publishers and subscribers are fully decoupled, which means that publishers and subscribers decide when they work. Producers publish information on a software bus and consumers subscribe to the information they want to be notified about on that bus [3]. This information is usually referred to as an event. Apache Kafka is one of the most popular platforms that uses the pub/sub model. Kafka is also one of the main parts of this thesis. Say we have four clients sending temperature measurements to a server. If the server is handling clients one by one, at most a client would have to wait for three clients to finish communicating with the server before it can send data. This would result in making the client unable to collect any more data before sending the existing data to the server. Increase this to a hundred clients and soon the clients have 1% uptime and the if amount of servers were to increase, the clients would have to be notified. Instead, a publish/subscribe service could be used for increased uptime for the clients and easier scalability of the servers.

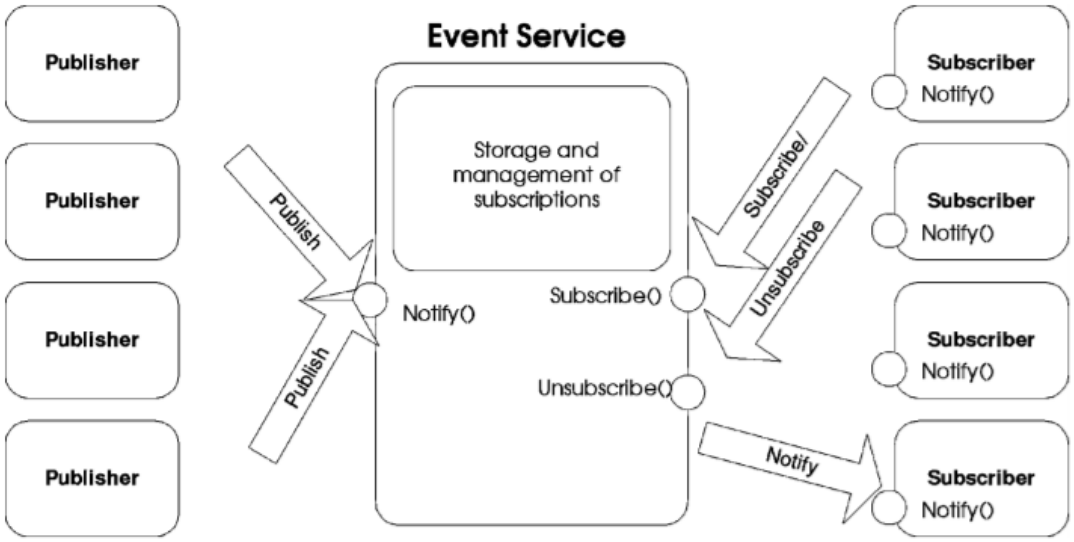


Figure 2.11. A simple object-based publish/subscribe system. [3]

2.4 Kafka

Apache Kafka is an open-source distributed streaming platform which uses the pub/sub model. Kafka is run as a cluster of a single or multiple servers which are referred to as brokers. Streams of records that are received are stored in an append only logs by brokers. A record consists of a key, a value and a timestamp. Clients, also known as producers and consumers, and brokers communicate through a high-performance TCP protocol.

2.4.1 Topics, brokers and records

A topic in Kafka is a category that organizes records logically. Multiple clients can publish and subscribe to the same topic. The Kafka cluster maintains a partitioned log for each topic. A Kafka server handling the topics within it is called a broker. A Kafka cluster has one or more brokers, the producers and consumers don't directly communicate to each other but instead they use the broker to exchange messages. Kafka brokers are stateless so a Zookeeper instance maintains the state. A partition is an immutable, ordered sequence of records, where new messages are appended. Each new record, which is appended to a log, is assigned a sequential id called the offset, which identifies each record within the partition [4]. Kafka retains all records for a retention period which can be configured, after the retention period has completed for a message, it will be deleted [4].

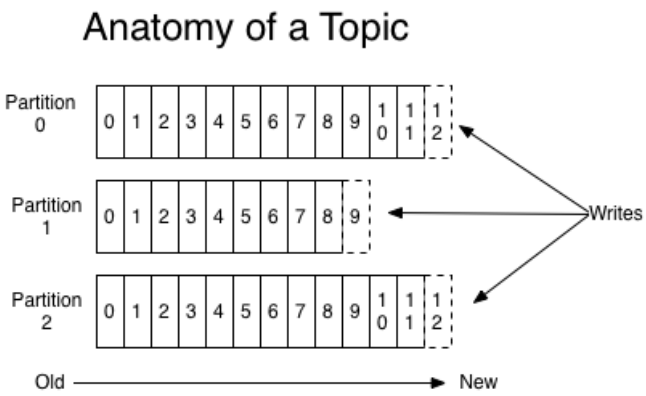


Figure 2.12. Topic's partitions [4]

Data size doesn't have any effect on Kafka's performance. This comes from the fact that consumers only retain the offset of the consumer on the topic. The offset is controlled by the consumer and can be changed as needed, which means that records can be consumed in any order. For example, the offset can be reset to the start of the topic, to start consuming from the beginning of the topic or to the end to consume newest records. The number of consumers can be an arbitrary amount which can be controlled at runtime.

Partitioning allows to scale the Kafka cluster to store data that would not fit on a single server and to allow parallel consumption of messages. The amount of partitions can be arbitrary. Partitions also act as an unit of parallelism. The partitions of a topic are distributed over the Kafka cluster with brokers. Each partition can be replicated over an arbitrary amount of servers and each partitions has a leader broker and zero or more follower brokers. The leader broker handles all read and write requests for the partition and the followers replicate the leader's actions [4]. When a leader broker fails a new leader will be elected from the followers. Each broker acts as a leader for some partitions and a follower for others, which means the load within the cluster is well balanced. Message ordering is guaranteed within a single partition but not across multiple partitions. If total ordering is required, a single partition must be used. A topic with three partitions can be seen on figure 2.12.

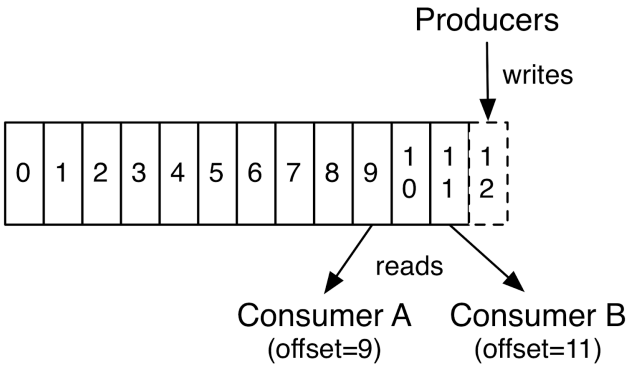


Figure 2.13. Producers and consumers [4]

Records that are produced to a topic consist of a message key, value and timestamp. A Kafka topic stores records as byte arrays, which means records have to be serialized at production and deserialized at consumption. When a producer or consumer instance is created, a serialization/deserialization type for both the key and the value must be provided.

Kafka has the next guarantees [4]:

- Messages sent to a topic will be appended in the order they are sent.
- A consumer consumes records in the order they are appended to the topic.
- A topic with replication factor of N can tolerate N-1 server failures before any records are lost.

2.4.2 Producer

In Kafka, the entity that writes messages to a topic is called a producer. A producer partitioner maps each message to a topic partition, and the producer sends a produce request to the leader of that partition and the partitioners shipped with Kafka guarantee that all messages with the same non-empty key will be sent to the to the same partition, however a partition can also be specified when producing a record. If a key is provided, the partitioner will hash the key with murmur2 [15] algorithm and divide it by the number of partitions. An example of a producer can be seen on figure 2.13.

Records written to the partition leader are not immediately readable by consumers. When all in-sync replicas have acknowledged the write, then the message is considered committed, which makes it available for reading.

2.4.3 Consumer

A consumer is the entity which reads messages from a topic. After the consumer receives its assignment from the coordinator, it must determine the initial position for each assigned partition. Typically consumption starts at either the earliest or latest offset. As a consumer reads messages from a partition, it must commit the offsets corresponding to the messages it has read, so that when the consumer shuts down, its partitions will be re-assigned to another member, which will begin consumption from the last committed offset of each partition. An example of consumers can be seen on figure 2.13.

A consumer group is a set of consumers which cooperate to consume data from some topics, the partitions of all the topics are divided among the consumers in a consumer group. When new consumers arrive and old consumers leave the group, partitions are re-assigned. This is known as rebalancing the group.

2.4.4 Zookeeper

Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services [16]. The data within Zookeeper is divided across multiple nodes, if a node fails, Zookeeper can perform instant failover migration. If a node is shutting down, the controller tells all the replicas to act as partition leaders to fulfill the duties of the partition leaders on the node that is about to fail [16]. When a node shuts down a new controller is elected. Zookeeper holds configuration data which contains the list of existing topics, partitions for each topic, location of replicas, list of configuration overrides for each topic and which node is the preferred logic, etc. [16]. Finally, Zookeeper maintains a list of all the brokers.

2.4.5 Schema registry

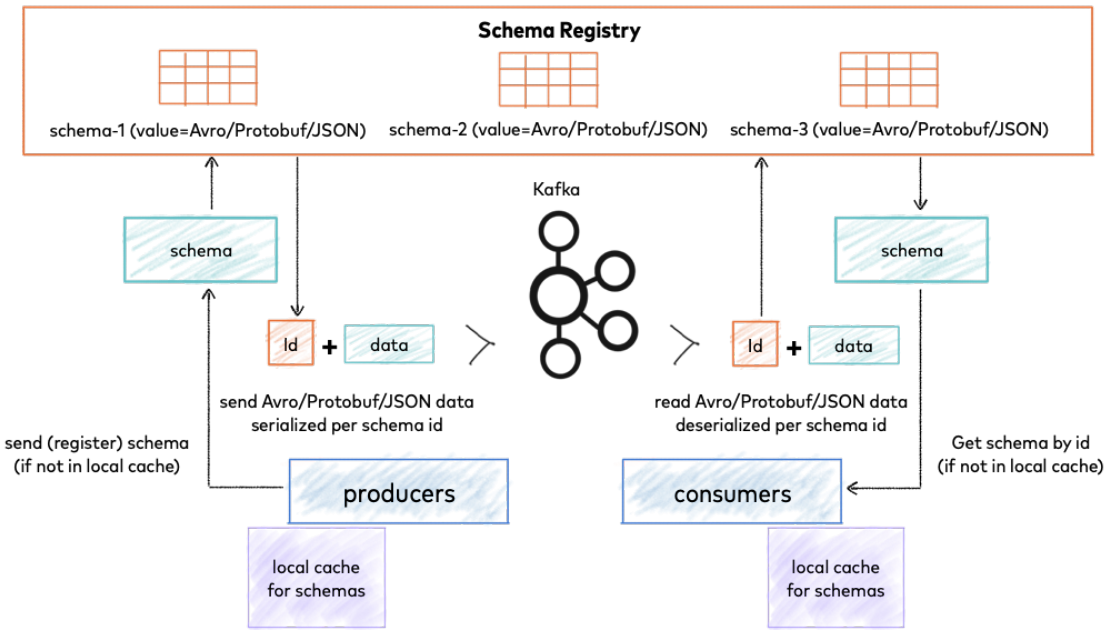


Figure 2.14. Storing and retreiving schemas

Confluent Schema Registry provides a serving layer for metadata. It provides a RESTful interface to store and retrieve Avro, JSON and Protobuf schemas. The Schema Registry stores a versioned history of all schemas based on a specified subject name strategy. It lives outside of and separately from Kafka brokers, consumers and producers can talk to the Schema Registry to send and retreive schemas that describe the data models for the messages. When a schema is registered, the Schema Registry returns a globally unique ID, which identifies the registered schema.

Avro

Apache Avro is a data serialization system, which provides rich data structures, compact and fast data formats along with simple integration with dynamic languages [17]. Apache Kafka uses Avro as one of the main serialization types because it has the following features [18]:

1. It has a direct mapping to and from JSON.
2. It has a very compact format.
3. It is very fast.

4. It has great bindings for a wide variety of programming languages.
5. It has a rich extensible schema language defined in pure JSON.
6. It has the best notion of compatibility for evolving your data over time.

Avro relies on schemas, when Avro data is read, the schema used when writing it is always present. This permits fast serialization as well as making the data together with its schema fully self-describing. An example Avro schema can be seen on figure 2.15.

The schema specifies how the records will be serialized and deserialized, which means the schema must be present at both operations. Usually the client application can get the schemas from the Schema Registry, which is introduced in chapter 2.4.5.

```
{
    "type": "record",
    "name": "Temperature",
    "namespace": "com.KafkaGraphQL",
    "fields": [
        {
        "name": "room"
        "type": "int"
        },
        {
        "name": "value",
        "type": "int"
        },
        {
        "name": "timestamp",
        "type": "string"
        }
    ]
}
```

Figure 2.15. Avro schema for Temperature model

2.4.6 Streams API

Currently, Kafka Streams is the most widely adopted API from Kafka. A stream represents an unbounded, continuously updated data set. Kafka Streams defines a processor topology, which can be represented as a graph, each stream processor has a source processor and a sink processor. Data is streamed from a source processor to a sink processor, which enables complex joins and transformations on the data being processed. The Streams API is not used in this thesis.

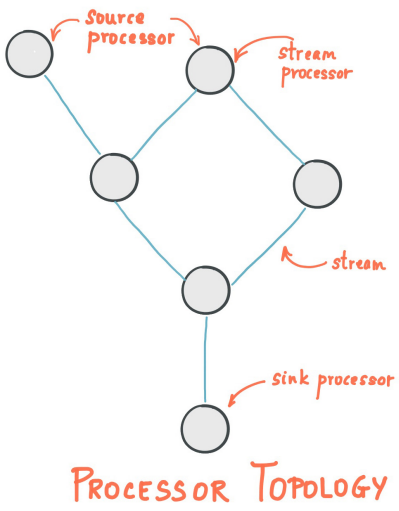


Figure 2.16. Kafka Streams processor topology

2.4.7 Confluent REST Proxy

The Confluent REST Proxy is an HTTP-based proxy for communicating with a Kafka cluster [6]. It supports producing and consuming messages along with providing the metadata about the cluster. The proxy supports binary, JSON and Avro formats for record encoding and integrates with the schema registry. The proxy was made to meet the growing demands of many organizations that want to use Kafka, but also want more freedom to select languages beyond those for which stable native clients exist today [6].

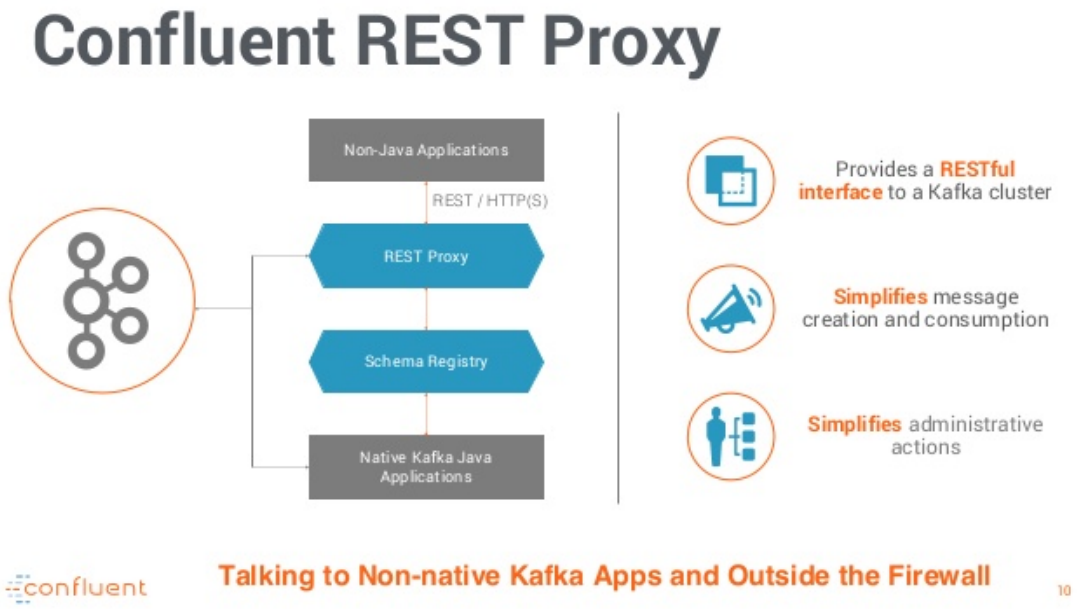


Figure 2.17. Confluent REST Proxy [5]

The Confluent REST Proxy is a HTTP wrapper of Java libraries [6]. It uses the existing libraries provided with the Apache Kafka project, which also includes modules to access the cluster's metadata. The proxy requests use embedded data - serialized key and value data that Kafka deals with [6]. To consume messages from a topic, the proxy requires that a consumer instance is created and subscribed to a topic by the client. These consumers are stateful and tied to a particular proxy instance. Some of the key operations that the proxy supports are shown on table 2.3.

Name	Description
GET /topics	Get a list of Kafka topics.
GET /topics/(string:topic_name)	Get metadata about a specific topic.
POST /topics/(string:topic_name)	Produce messages to a topic
GET /topics/(string:topic_name)/partitions	Get a list of partitions for the topic.
GET /topics/(string:topic_name)/partitions/ (int:partition_id)	Get metadata about a single partition in the topic.
POST /consumers/(string:group_name)	Create a new consumer instance in the consumer group.
POST /consumers/(string:group_name)/instances/ (string:instance)/subscription	Subscribe to the given list of topics or a topic pattern to get dynamically assigned partitions. If a prior subscription exists, it would be replaced by the latest subscription.
GET /consumers/(string:group_name)/instances/ (string:instance)/subscription	Get the current subscribed list of topics.
GET /consumers/(string:group_name)/instances/ (string:instance)/records	Fetch data for the topics of the consumer

Table 2.3. Some of Confluent REST Proxy methods

3 Reasoning to use GraphQL with Kafka

This chapter explains the reasoning behind why GraphQL could be used to communicate with the Kafka cluster instead of a RESTful interface.

3.1 Streams of records

Since Kafka is a stream-processing framework, a proper Kafka interface for communicating with a Kafka cluster should implement streams to some extent. The Confluent REST

Proxy only supports getting records as a request and not in the form of a stream, which is because RESTful interfaces are unable to define a request which might return infinite data. Kafka streams could be implemented in REST using a workaround involving webhooks and in GraphQL with GraphQL Subscriptions.

In REST architecture the client has to send requests each time they need new information, this means that for getting the latest pieces of information, the client has to periodically send the same request to the server, which means that the client might make requests more often than is required. One solution to this problem is using webhooks, which means that the client will specify an URL, to which the server will send events as they happen.

GraphQL has the subscription operation type, which supports streaming data from the server. GraphQL subscriptions are usually implemented by the client and server using WebSockets to communicate. WebSockets is a technology that makes it possible to open a two-way interactive communication session between the user's browser and a server [19]. The client sends a subscription query to the server and specifies what data it is interested in, a socket is created and when the server has new data for the client, it is pushed to the client via the socket.

4 Implementation

In this section the implementation details for the GraphQL interface for Kafka will be described. The implementation is based on the Confluent REST proxy java backend that was redesigned to comply to graphql principles. Porting the REST proxy to GraphQL enabled us to reuse some existing server functions used to interface Kafka with REST, however the GraphQL logic and resolvers still had to be written accordingly to the existing functions.

Preexisting codebase used for this implementation and not written by us can be found on: https://github.com/confluentinc/kafka-rest and https://github.com/graphql-java/graphql-java-subscription-example.

The GraphQL proxy implementation and Streaming Avro data via GraphQL was implemented using the following:

- Java 8.
- Apache Kafka 2.5.0 - java library for Kafka.
- Confluent REST Proxy v5.4.0 - reusing some functions from Confluent.
- graphql-java 6.0 - java library for GraphQL.
- RxJava 2.1.5 - a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

- Spring boot 2.2.6 - a framework to develop web applications.
- Jetty web server - servlet engine and http server.

4.1 GraphQL Proxy

In total our implementation of the GraphQL proxy supports 11 operations, which are shown on table 4.1.

The Confluent REST Proxy provided functions to communicate with the Kafka cluster on the server side. These functions were available through the DefaultKafkaRestContext.java class, which provides wrappers for Kafka AdminClient [20]. The AdminClient class is the administrative client for Kafka, which supports managing and inspecting topics, brokers, configurations and ACLs (access control list).

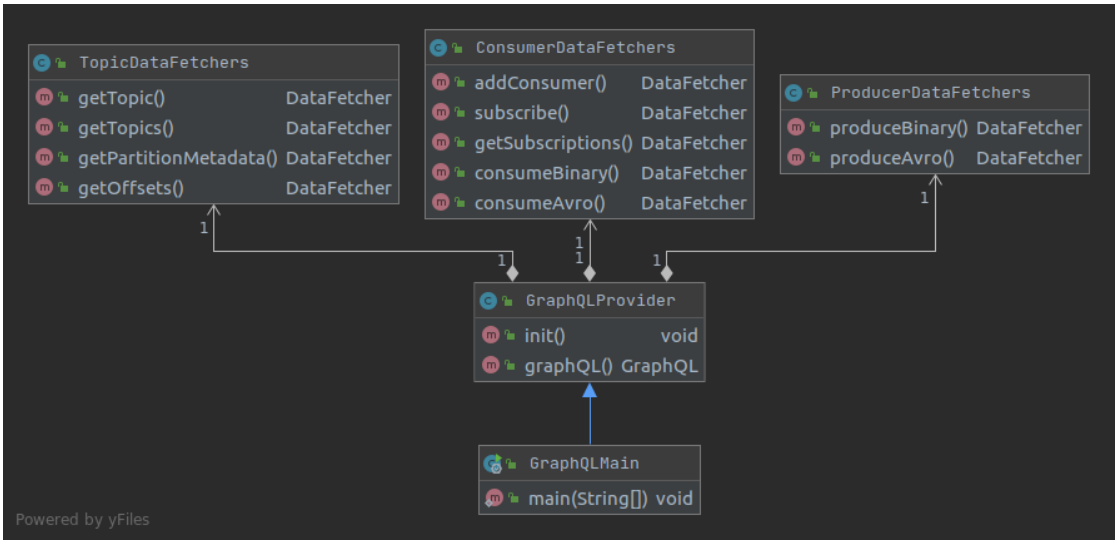


Figure 4.1. UML diagram for the GraphQL Proxy

Type	Name	Descriptions
Query	topics	Returns a list of existing topics
Query	topic	Returns metadata about a single topic
Query	partitionMetadata	Returns metadata about a partition
Query	getOffsets	Returns offsets of a partition
Query	consumeBinary	Returns a list of binary records from a consumer
Query	consumeAvro	Returns a list of Avro records from a consumer
Query	getSubscriptions	Returns a list of subscribed topics for a consumer
Mutation	ProduceBinary	Produces binary records to a topic
Mutation	ProduceAvro	Produces Avro records to a topic
Mutation	addConsumer	Creates a consumer instance
Mutation	subscribe	Subscribes a consumer to a list of topics
Subscription	temps	Starts a stream of temperatures from the server

Table 4.1. Table of supported operations in our implementation of GraphQL proxy

4.1.1 Data model

To communicate with the Kafka cluster, GraphQL types for topics, producers, consumers and records had to be designed. The models for these entities can be seen on figure 4.2, the figure is not exhaustive of all the types written for this thesis.

```
type Query {
    topics: [Topic] # Returns an array of topics
    consumeBinary(group: String, instance: String): [ConsumedRecord]
    consumeAvro(group: String, instance: String): [ConsumedRecord]
}
type Mutation {
    produceBinary(topic: String, records: [BinaryRecord]): [Offset]
    produceAvro(topic: String, records: [AvroRecord]): [SchemaInfo]
}
type Topic {
    name: String
    configs: [Property]
    partitions: [Partition]
}
type Property {
    key: String
    value: String
}
type Partition {
    id: Int
    leader: Int
}
type Offset{
    partition: Int
    offset: Int
}
type SchemaInfo {
    keyId: Int
    valueId: Int
}
type ConsumedRecord { # Return type for a record query
    topic: String
    key: String
    value: String
    partition: String
    offset: String
}
input BinaryRecord { # Input object to create a new record
    key: String
    value: String
    partition: Int
}
input AvroRecord {
    key_schema: String
    key_schema_id: Int
    value_schema: String
    value_schema_id: Int
    key: String
    value: String
}
input Consumer { # Input object to create a new consumer
    name: String
    format: String
    auto_offset_reset: String
    auto_commit_enable: String
    fetch_min_bytes: Int
    request_timeout_ms: Int
}
```

Figure 4.2. Modelling Kafka with GraphQL types

4.1.2 Querying Metadata

To query metadata, the methods which AdminClient offers and were used in this thesis are the following:

- describeTopics(Collection<String> topicNames) - get metadata about a collection of topics.
- listTopics() - lists the available topics in the cluster.

Using the listTopics() method, a GraphQL resolver was written, which asks the AdminClient wrapper for a list of existing topics, which are then returned to the client, this list enables to get information about partitions and offsets as well.

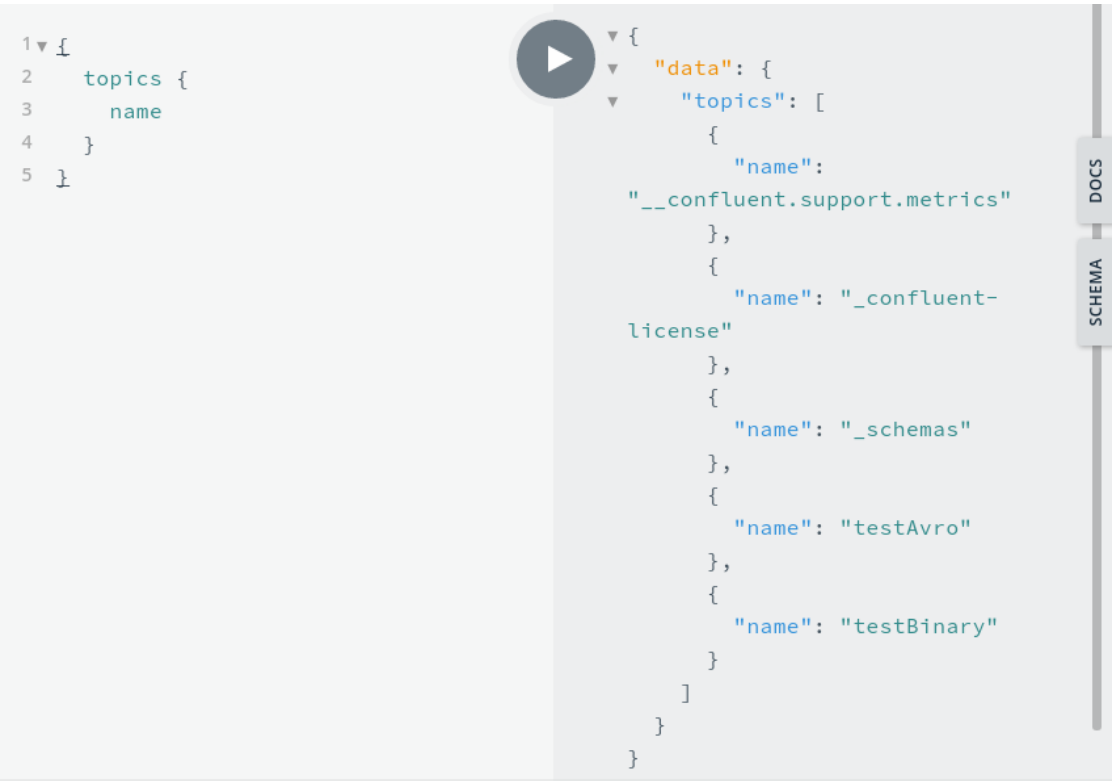


Figure 4.3. Query for existing topics

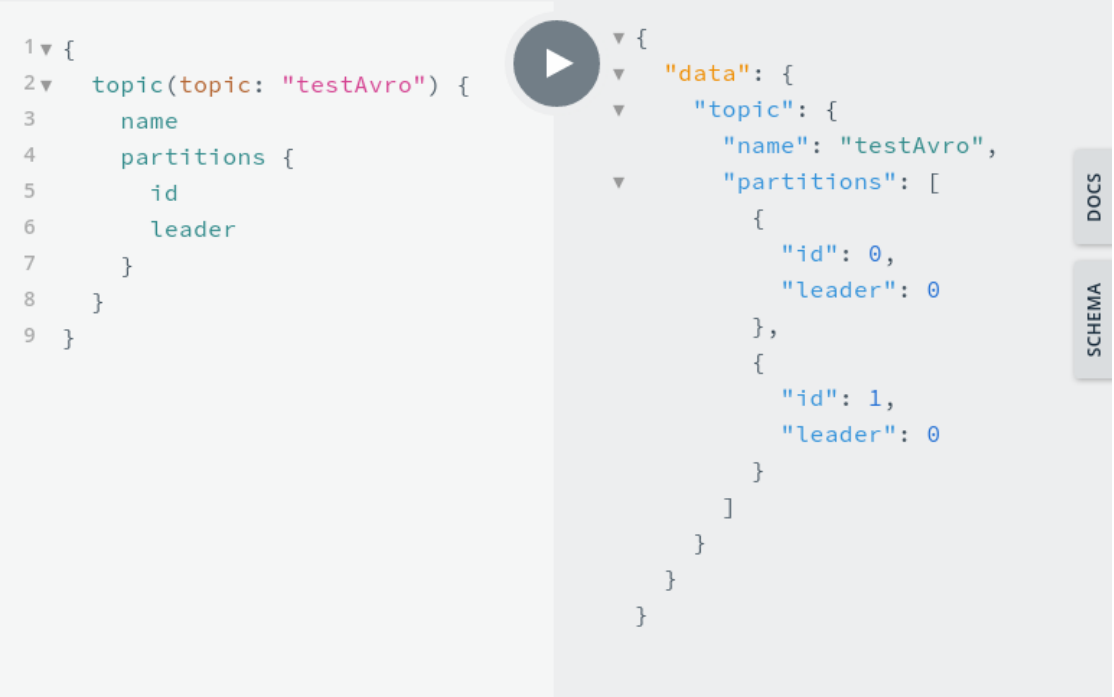


Figure 4.4. Querying metadata about a single topic

4.1.3 Producing to a topic

For record production, the Confluent REST Proxy uses a shared pool of producers, who will send the provided records to a topic. The client specifies records via the REST interface which will then be sent to a producer with the desired serialization type. The serialization type is specified in the request and the ProducerPool object stores a map with keys specifying the serialization type and values with corresponding producer objects.

To produce data to a topic, our implementation supports two GraphQL mutations:

- produceBinary(topic: String, records: [Record]), produces the provided records to the topic.
- produceAvro(topic: String, records: [AvroRecord]), produces the provided Avro records to the topic.

Binary producing In our implementation, the client can specify the key, value and partition of the record to be produced as well as a topic. The keys and values provided, must be strings, which will then be converted into byte arrays in the class ProducerDataFetchers' (figure 4.1) resolver method produceBinary and afterwards sent to the specified topic with a NoSchemaProducer instance, which will just send the records to the partition as they are provided.

Avro producing In our implementation, Avro producing works by specifying the schemas or schema ids for both the record's key and value when posting records. If the schema is not provided, the id must identify the schema and the schema must match with key and the value of the record. The resolver method in class ProducerDataFetchers' called produceAvro then forwards data to a AvroRestProducer instance, which encodes the key and value and sends them to the topic specified, the resolver then responds the client with the schema id for both the key and the value (if the schema didn't exist in the registry before, a new schema is registered).

Figure 4.5. Producing Avro records to a topic

4.1.4 Consuming from a topic

To consume records from a topic, our implementation has three steps:

1. Create a consumer instance in a consumer group and specify record type
2. Subscribe the created consumer to a topic
3. Consume records using the consumer instance

To create a consumer, the client must specify a set of parameters, describing the consumer instance, these parameters are:

- Group - consumer group where the consumer instance will be created.

- Name - name for the created consumer.
- format - record format - either "binary" or "avro".
- auto_offset_reset - sets the auto.offset.reset setting for the consumer.
- auto_commit_enable - sets the auto.commit.enable setting for the consumer.
- fetch_min_bytes - sets fetch.min.bytes setting for the consumer.
- consumer_request_timeout_ms - sets the consumer.request.timeout.ms setting for the consumer, this controls the maximum time to wait for records if the maximum request size has not been reached.

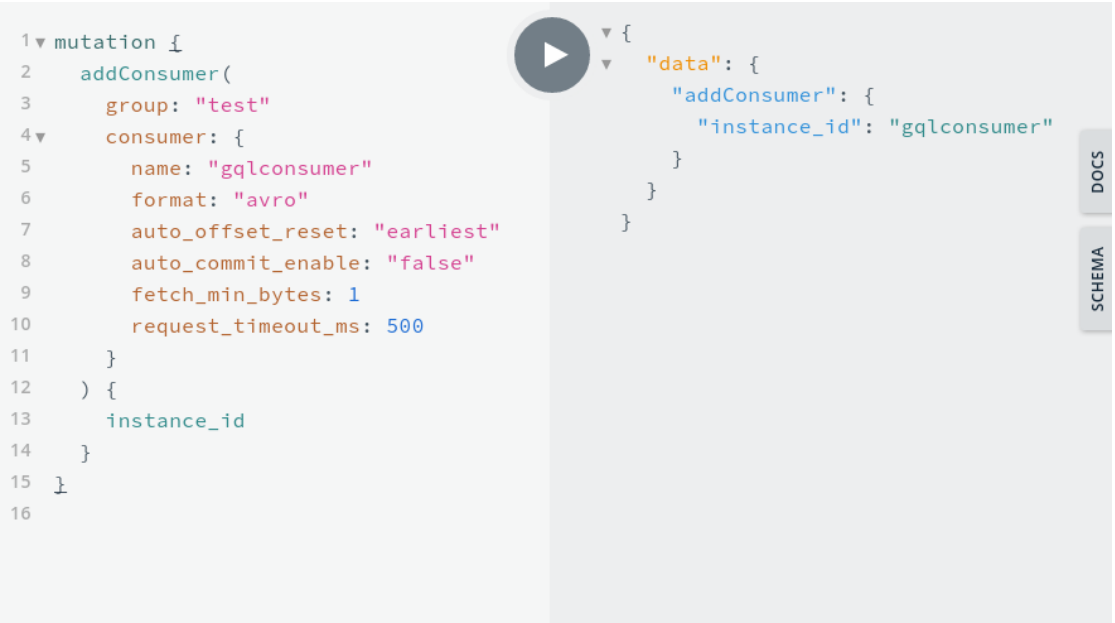


Figure 4.6. Creating an Avro consumer

Both the group and the name must be specified to consume data from a topic. After sending the request the server creates a new consumer instance, which is now ready to be used. To use the consumer instance for consumption, it must subscribe to a set of topics, which can be done with a subscribe mutation. The subscribe mutation takes parameters of group, consumer name and an array of topic names, to which the consumer instance will subscribe to. This mutation doesn't return anything.

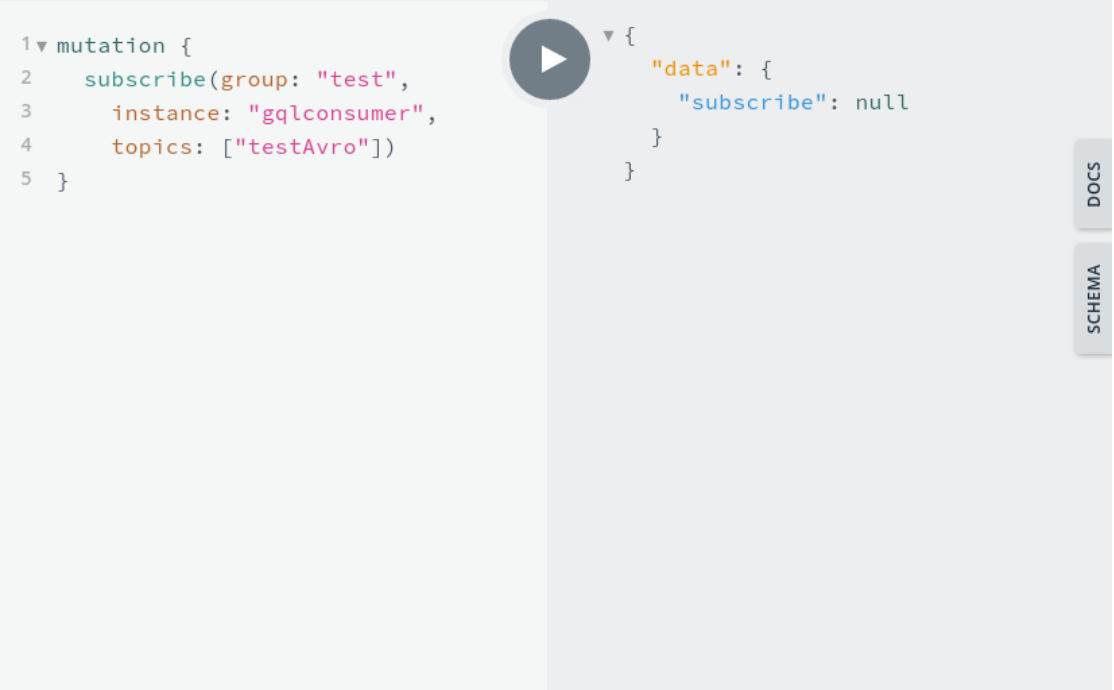


Figure 4.7. Subscribing a consumer instance to a topic

Once the consumer has subscribed to a set of topics, a query can be made to the server, which will return keys and values from the subscribed topics, when the query is made, the offsets are saved for the consumer and for every subsequent query, records will be returned starting from that offset.

Figure 4.8. Consuming Avro records

REST Proxy vs GraphQL Proxy

Table 4.2 compares some of the total times of the Confluent REST Proxy and our implementation. The times were measured using the ubuntu time utility, which measures the runtime of a command, we used curl as the command in both proxies. According to the benchmarks, GraphQL performed better overall. This means that in terms of execution and response time, graphql is a better choice for such a proxy.

Operation	REST Proxy	GraphQL Proxy
Get a list of topics	0.493s	0.056s
Get metadata about a topic	0.079s	0.044s
Produce a binary message	0.183s	0.083s
Produce an Avro message	0.279s	0.117s
Consume 5 Avro messages	1.023s	0.500s

Table 4.2. Comparison of REST proxy vs GraphQL proxy

4.2 Streams with GraphQL subscriptions

To stream records from a topic over GraphQL, GraphQL subscriptions and RxJava was used by us. When the server recieves a subscription request from the client, the server

subscribes to a publisher of events and starts sending data over the websocket to the client.

4.2.1 Streaming temperatures

In the following, a temperature example is used for subscriptions and streaming. The subscription is of type Temperature similar to the one shown on figure 2.1. A publisher instance is created, which is consuming the desired Kafka topic in 100 ms intervals for new records and upon consuming new records, sending the records over a WebSocket to the client. For the demo a producer instance is running that is generating random temperature measurements at random intervals up to 2 seconds apart.

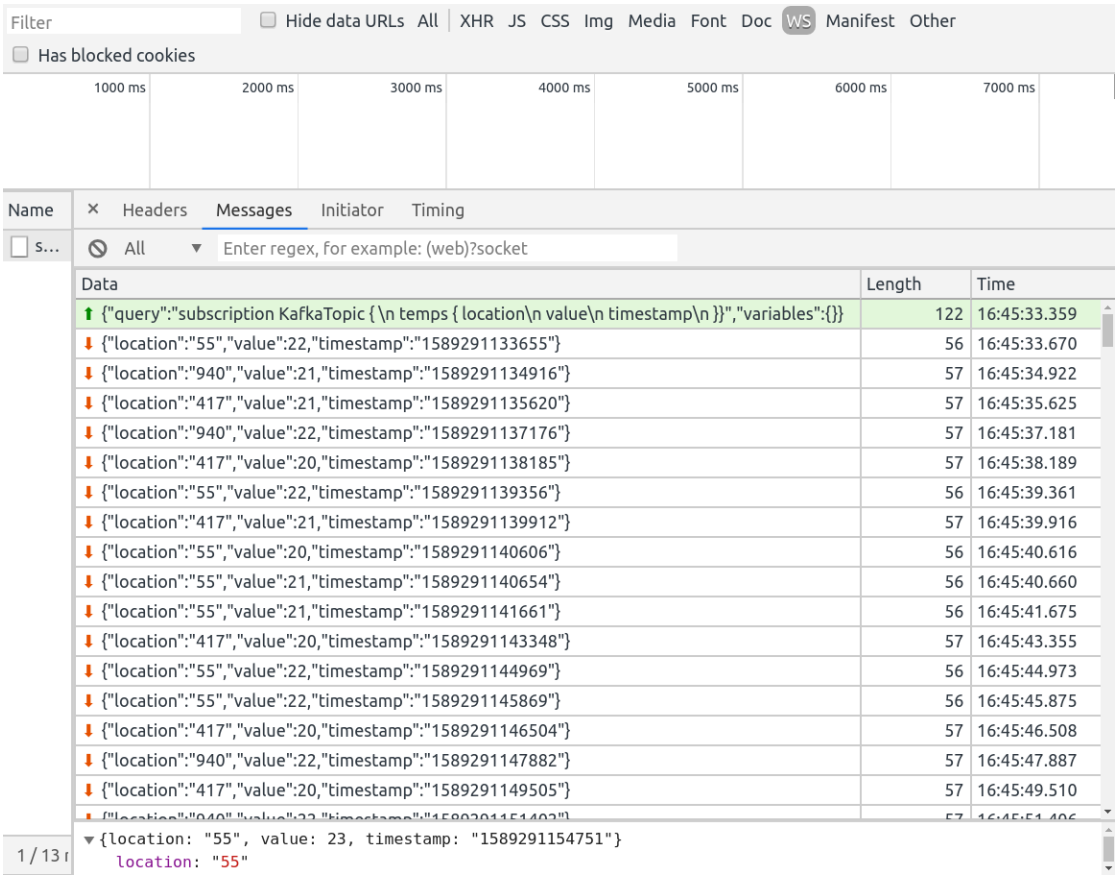


Figure 4.9. Consuming temperature measurements via GraphQL subscriptions, shown using Google Chrome's inspect element interface

5 Conclusion

During the writing of this thesis a GraphQL proxy was written, which provides a web API to communicate with the Kafka cluster. This enables the use of Kafka without it's native clients (Java, Go etc.), this makes the Kafka cluster much more accessible. During the thesis 11 GraphQL operations were developed, which enable to query for metadata, produce and consume records.

5.1 Future work

This thesis did not interface GraphQL with Kafka to the full extent. The current streaming implementation is not able to stream records of generic types and since Avro schemas can in theory be parsed into GraphQL schemas, then it makes sense that such a parser would prove useful in the future as well.

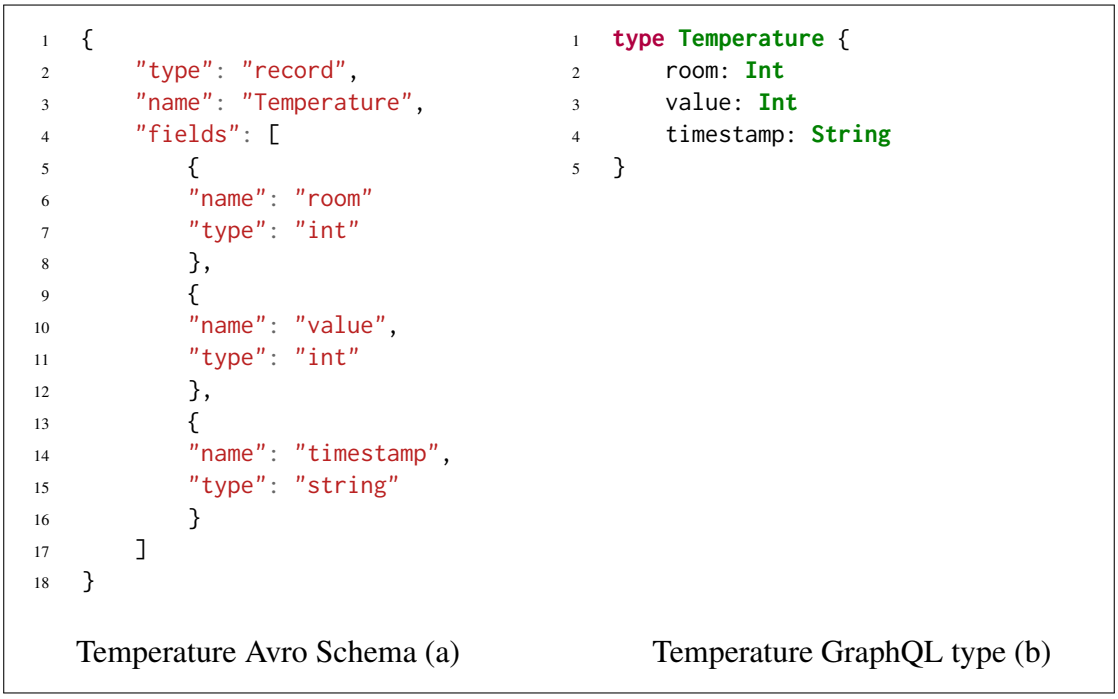
5.1.1 Kafka Streams

In the future our work could be extended to work with Kafka streams API using GraphQL subscriptions. This would mean streaming records similarly to streaming temperatures, but enabling the server to send arbitrary types of Avro records. First the used temperature subscription could be generalized to a type AvroRecord, which has fields for the key, value and timestamp. Then those fields could be generified to return either binary keys and values or support Avro records using JSON string as in the GraphQL proxy proposed in this implementation.

5.1.2 GraphQL to Avro parser

Avro and GraphQL schemas share an inherent similarity, which can be seen on figure 5.1. In the future, a parser for these schemas to translate one to the other could be implemented to avoid explicit conversion of the schemas. To do these translations some equivalence between types must be implemented and fields which resolve to custom types, could be translated recursively.

GraphQL schemas are similar to Avro schemas because both define types and fields on those types. In both GraphQL and Avro, fields can be scalar types or complex types, which have fields as well.



```
{
    "type": "record",
    "name": "Temperature",
    "fields": [
        {
        "name": "room"
        "type": "int"
        },
        {
        "name": "value",
        "type": "int"
        },
        {
        "name": "timestamp",
        "type": "string"
        }
    ]
}
```

Temperature Avro Schema (a)

```
type Temperature {
    room: Int
    value: Int
    timestamp: String
}
```

Temperature GraphQL type (b)

Figure 5.1. Avro and GraphQL equivalents for Temperature model

References

[1] "Graphql | a query language for you api." https://graphql.org/.

[2] "Infographic: What happens in an internet minute 2020." https://www.allaccess.com/merge/archive/31294/infographic-what-happens-in-an-internet-minute.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," 2003.

[4] "Apache kafka." https://kafka.apache.org/intro.

[5] "Confluent rest proxy and schema registry (concepts, architecture, features)." https://www.slideshare.net/KaiWaehner/confluent-rest-proxy-and-schema-registry-concepts-architecture-features/10.

[6] "A comprehensive rest proxy for kafka." https://www.confluent.io/blog/a-comprehensive-rest-proxy-for-kafka/.

[7] R. T. Fielding, "Architectural styles and the design of network-based software architectures," 2000.

[8] "Rest api tutorial." https://restfulapi.net/.

[9] "Graphql specification - schema." http://spec.graphql.org/June2018/#sec-Schema.

[10] "Graphql specification - operations." http://spec.graphql.org/June2018/#sec-Language.Operations.

[11] "Graphql specification - selection sets." http://spec.graphql.org/June2018/#sec-Selection-Sets.

[12] "Graphql specification - fields." http://spec.graphql.org/June2018/#sec-Language.Fields.

[13] "Graphql specification - arguments." http://spec.graphql.org/June2018/#sec-Language.Arguments.

[14] "What is pub/sub? | cloud pub/sub documentation | google cloud." https://cloud.google.com/pubsub/docs/overview.

[15] "Murmurhash." https://en.wikipedia.org/wiki/MurmurHash.

[16] “What is zookeeper and why is it needed for apache kafka?.” https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka_what_is_zookeeper.html.

[17] “Apache avro 1.9.2 documentation.” https://avro.apache.org/docs/1.9.2/.

[18] “Why avro for kafka data?.” https://www.confluent.io/blog/avro-kafka-data/.

[19] “The websocket api (websockets).” https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[20] “Adminclient (kafka 2.3.0 api).” https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/admin/AdminClient.html.

Appendix

Acknowledgements

I would like to thank Riccardo for the opportunity to work with him, during which i learned a lot of valuable things.
In memory of professor Sherif Aly Ahmed Sakr.

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Jonathan Karu**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Towards a GraphQL Proxy for Apache Kafka,

   supervised by Riccardo Tommasini, PhD.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Jonathan Karu
20.05.2020