Hochschule für angewandte Wissenschaften
München

Fakultät für Informatik und Mathematik

Bachelor's Thesis

Computer Science

High performance scalable message processing with distributed in-memory cache of object state

---

Author:	Robin Jägers
Matriculation number:	
Submission Date:	09. Juli 2023
Supervisor:	Prof. Dr. Johannes Ebke
Company:	SCHWARM Technologies Germany GmbH Rindermarkt 15 D-80331 München

I confirm that this bachlor's thesis is my own work and I have documented all sources and material used.

München, 09. Juli 2023 Robin Jägers

Abstract

Within this bachelor thesis, the possibility and efficiency of using a log-based message broker as an object state store is examined. The main goal is to show the high performance a distributed in-memory local state store can achieve and why a log based message broker is a useful complement to create a reliable and scalable object store. The project of this work is designed as state store for current device data of an IoT system based on the microservice architecture.

As log-based message broker Kafka is used in this thesis with its additional Streams DSL, which simultaneously perform the functions of a data consumer, producer and a processor. Kafkas' Streams DSL is the perfect fit for this project, because it already provides an automatic state store handling.

Firstly this work describes the implementation how to replace an object database with a distributed local state store. Secondly it shows the providing of the state to other services over a changelog topic. And thirdly the project illustrates how the changelog can be materialized into a view, which makes the data accessible to the outside via a REST API.

In the performance analysis of this paper, the comparison between the local state store implementation and a predecessor implementation, with *MongoDB* as object store, is made. Also, the differences in performance and resource utilization between the two types of key-value stores, that Kafka Streams provides, are investigated. The two types are the only in-memory and the persistent key-value store using *RocksDB*.

The thesis successfully shows that the implementation of such a distributed local state store can be efficiently and easily integrated into a microservice architecture. Furthermore the service is capable to process much more messages per second than the *MongoDB* implementation. And in addition the resource utilization is much lower in such a system that does not use an external database. Moreover, it is shown that the latency, which is measured by the time the service needs to process one message, does not depend on the number of messages per second. This cannot be achieved with the implementation in *MongoDB*.

Contents

1 Introduction

1.1 Motivation

In modern software architecture, it is becoming more and more relevant to process and store a large amount of data correctly and efficiently. Especially in Internet of Things (IoT) systems with many devices, where messages from a single device tend not to contain a large amount of data but are received in a small time interval, it is important to ensure the correctness and efficiency of processing, storing and querying the device state (A. B. Alaasam et al. 2020). The general approach is to store this state inside external datastores, but distributed data systems with local stores are receiving more and more attention, because of the fast process times (Kleppmann 2017). Therefore it is becoming increasingly important to provide an efficient algorithm that guarantees correct processing.

1.2 Planned Concept

In this work, the implementation of a distributed and scalable in-memory state storage is planned. This storage is used to store the current state of a physical device as Digital Twin (DT) at any time. Due to the fact, that the state store is in-memory and distributed, the application should efficiently persist the data on an external resource. Furthermore it should also provide the state to the outside with an Application Programming Interface (API), having access to all objects, even if the state store is distributed over many service instances.

1.3 Expected Results

The result implementation should be easy to integrate into an IoT system based on a microservice architecture. And it should fulfill the relevant requirements for an object state storage, e.g. good scalability and persistency of data. Comparing the performance of the proposed implementation of this work to the previour implementation relying on an external, NoSQL database, significant improvements should be apparent.

1.4 Report Outline

The first chapter covers the state of the art of the technologies used within the project. Followingly, the conceptual and architectural design of the target implementation is described along with the included microservices that compose the scalable in-memory state storage. In the third chapter the detailed implementation of the services is shown. After that the performance test results between different technologies of state storages and also between the old and the new implementation are illustrated and explained and

summarized in the next chapter. Lastly, the potential for improvements is assessed and outlined.

2 State of the Art

In the beginning of this chapter technologies which are useful for the implementation of a distributed object state storage are explained. After that the different types of message brokers are explained, pointing out their importance in distributed data systems. The second part of the message brokers shows that log-based message brokers with additional features are already designed to store data in a distributed way. Finally the special framework named Kafka is explained with its special features, which simplifies the distributed storage of an object state.

2.1 Distributed Data Systems

When it comes to modern software systems, the most important concerns are: Scalability and reliability. This means that the systems should work correctly even in the face of adversity and they should allow the amount of data to grow (Kleppmann 2017, P. 6).

In order to fulfill these requirements, distributed data systems are used, which by design are equipped with good scalability and reliability by adding instances and replicate them across multiple machines. The implementation of distributed systems without a state is trivial, but performing stateful operations on a distributed setup can lead to a lot of complexity. Therefore, until recently, the general approach was to store the state on an extern central datastore, but distributed data systems with internal states will become the default in the future (Kleppmann 2017, P. 18).

2.2 Microservice Architecture

A common architecture for distributed data systems is the microservice architecture. The microservice architecture can be described as a design pattern that divides an application into a number of small independent services, where each service is responsible for a certain functionality (A. B. A. Alaasam et al. 2019). By design the microservice architecture is a stateless service and preferable to a stateful one. A stateful microservice has the ability to retain state information that was generated previously. So this service can produce output, that also depends on the state generated in previous interactions. In contrast the stateless service can only work with the current data being sent within the interaction. But stateless microservices are ideal, because they are easily scalable and reliable through redundancy and in terms of a failure, the service does not have to deal with state recovery strategies (Furda et al. 2018).

2.3 Digital Twin - Object State

A DT can be seen as a virtual representation of an existing object product in the real space. The digital twin supports monitoring, controlling and state prediction of the real machine based on its object state data (A. B. A. Alaasam et al. 2019).

The concept of DT also includes the middleware as connection between the real space and the virtual space.

2.4 Message Brokers

Streaming middleware is needed for data exchange between loosely-coupled microservices (A. B. A. Alaasam et al. 2019). Message brokers can be the main part of that middleware, whereby a message broker runs as a server and services can connect to it as clients. Clients are divided into producers, writing messages to the broker, and consumers, receiving messages by reading them from the broker (Kleppmann 2017, P. 443).

One improvement that comes with message brokers is the asynchronous processing. This means that producers do not have to wait on a response from consumers to continue processing. Additional message brokers support peak shaving, that means that a broker buffers data, so that consumers and producers can consume or send data according to their own processing capacities (Fu et al. 2021).

2.4.1 Queue-Based Message Brokers

Traditionally message brokers, defined in standards like JMS (Hapner et al. 2002) or AMQP (OASIS 2012), are responsible for routing the messages to different queues, based on the message routing keys (Milosavljević et al. 2021). Thus producers do not have to send messages to specific queues directly. Instead they send it to exchanges inside the broker and the exchanges route them to the queues (cf. figure 2.1). Consumers then get the messages from the queues they assigned to (Vineet and Xia 2017).

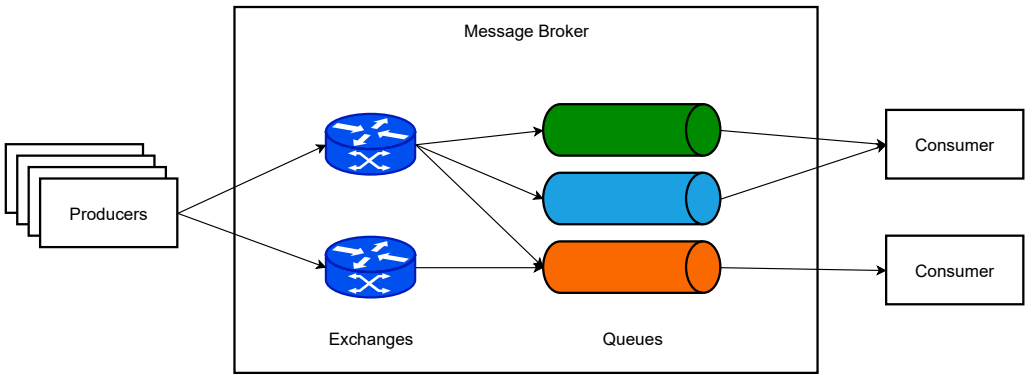

Figure 2.1: *Queue-Based Message Broker*

Messages that are completely delivered to their consumers will be deleted by the broker immediately after, even when they are durably stored on disk (Kleppmann 2017, P.

446). Because of that, each consumer which is added to the system can receive only new messages and cannot recover prior ones, since they have already been consumed and thus deleted from the queue. So in case of a consumer crash you cannot expect the same result after restarting the consumer. The broker guarantees ordering within a queue (T and K 2019), thus allowing applications to create an order of related events by routing these event types from multiple topics all together into one queue (Vanlightly 2018). Topics can be seen as categories of messages types, comparable to a folder in a filesystem.

Examples for this kind of messaging middleware systems are: RabbitMQ, ActiveMQ, AWS SQS, Google Cloud Pub/Sub (Kleppmann 2017, P. 444).

2.4.2 Log-Based Message Brokers

Because of the fact that traditional message brokers delete the message after being consumed, log-based message brokers are intended to be a hybrid, combining the durable storage approach of databases and the low latency notification system of message brokers. A producer applies data messages to a log, which can be seen as an append-only sequence of records on disk, while a consumer receives the data by reading the log sequentially from beginning to end and waiting for a notification, that new messages are appended to the log, when it has reached the end of the log (Kleppmann 2017, P. 447).

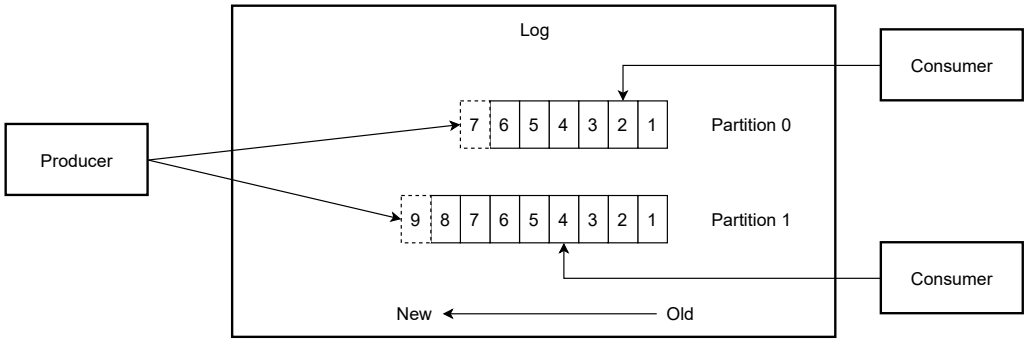

Figure 2.2: *Log of a Log-Based Message Broker*

Logs are naturally persistent and a shared resource, which means that they can be divided into many partitions (seen in fig. 2.2), or also known as shards, for balancing load and reaching a higher throughput (Vanlightly 2018). Throughput means the amount of events processed within a defined time period. Different partitions can also be hosted on different brokers and thus also on different machines to reach higher throughput than a single disk can offer (Kleppmann 2017, P. 447). Within a partition messages get an increasing sequence number assigned by the broker, so that the messages are totally ordered within the partition (Kleppmann 2017, P. 447).

For better reliability replications of a partition can be stored on different brokers, where one replica is the leader and all others are follower replicas. The leader is accessed by the consumers and followers stay synchronized in case of a leader crash (Fu et al. 2021).

Examples for log-based messaging middleware systems are: Apache Kafka, Apache Pulsar, AWS Kinesis, Azure Event Hubs, Twitter's DistributedLog (Vanlightly 2018; Kleppmann 2017).

Comparison to Queue-Based Brokers

The main difference to the queue-based broker is, that a log-based broker trivially supports fan-out messaging. Fan-out means that the messages can be consumed by many consumers in parallel and that the messages are not directly deleted after a consumer read them (Kleppmann 2017, P. 448). The queue-based approach has to duplicate and publish the message to many queues to achieve this kind of messaging.

The downside of the log-based approach is that messages in one partition can only be processed on the same consumer instance, thus delaying all messages of one partition, if the consumer is working slowly. In contrast a queue can be consumed by many consumers and messages can be processed on a message-by-message basis (Kleppmann 2017, P. 449). On the other hand, with the partition to one consumer allocation, the log-based broker guarantees that all messages of one partition are ordered whereas using a queue-based broker with many consumers, we cannot assume that the message processing is ordered. We realize that scaling comes at the cost of ordering within a queue-based system (Vanlightly 2018).

2.4.3 Performance Comparison

When we speak of the performance of a message broker, two key metrics are important. The throughput as mentioned before and the latency, which measures the time a message takes to be transmitted between endpoints (Fu et al. 2021).

In these authors' tests (Fu et al. 2021; T and K 2019; Dobbelaere and Esmaili 2017) we can see that Kafka outperforms most of the message brokers like ActiveMQ, RabbitMQ, RocketMQ and Pulsar in terms of throughput. The test (T and K 2019) shows that NATS can have a comparable throughput but scalability is not well supported by NATS. The only test that shows that Pulsar performs better than Kafka was executed by (Intorruk and Numnonda 2019) but only with two different message sizes.

Regarding latency RabbitMQ, ActiveMQ and Pulsar have a better performance than Kafka, which is shown in (Fu et al. 2021; Intorruk and Numnonda 2019), especially when the consumer is too slow and the messages have to be read from disk instead of the OS Cache (Dobbelaere and Esmaili 2017). RabbitMQ is very performant when it comes to lightweight communication, as we can see in (Gracioli et al. 2018), which was performed on embedded systems. But on a server environment with good configurations and bigger message sizes Kafka is also capable of providing low-latency, shown by (T and K 2019; Dobbelaere and Esmaili 2017).

When we speak of data buffering Kafka is highly stable and has lower average Central Processing Unit (CPU) usage than RabbitMQ (Milosavljević et al. 2021).

As a result it can be stated that log-based brokers are more performant in high throughput while queue-based brokers are capable of a lower latency.

2.4.4 Message Broker vs Datastore

Traditional queue-based message brokers are not suitable for long-term data storage, because they assume that their queues are short and in the case they have to buffer messages, each individual message takes longer to process (Kleppmann 2017, P. 444).

In contrast a log-based broker definitely has similarities with a datastore. In fact a database uses commit, transaction or change logs, to which all database events are applied to allow auditing, replication and recovery. The snapshot representation of the database can always be derived from the immutable change log, but the contrary however is not true (Rooney et al. 2019).

Databases support indexes and various ways of querying for data where the result is typically a point-in-time snapshot of the data, while message brokers do not support arbitrary queries but they notify all clients when data change (Kleppmann 2017, P. 444).

2.5 Log-Based Message Broker as Database

As seen before a database and a log-based message broker have many features in common. Logs are already a part of the internal structure of databases for replication and recovery (Fang et al. 2011). In this section it is shown under wich circumstances a log-based broker has the ability to replace the internal part of a database and can also be accessed from outside with the help of materialized views.

2.5.1 Cleanup Policy

The cleanup policy of a broker is one important factor for storing persistent data in the log. Since the broker can only keep a limited amount of history data for each log, log compaction is a method to keep at least the last entry of every key inside. The principle of the compaction is, periodically looking for log records with the same key and keeping only the record with the newest timestamp. Having the possibility to delete records, every key with a null value, which is also known as a tombstone, is removed during compaction (Kleppmann 2017, P. 456).

This compaction reduces disk space for each log, because the space depends only on the number of keys and not on the number of messages, different to the logs with time-based expiry of messages (Kleppmann 2017, P. 456). Change logs of relational databases resemble these logs with this compaction policy (Rooney et al. 2019).

2.5.2 Persistency

As mentioned above, replications of log partitions can be stored on different message brokers (Fu et al. 2021). With this feature and the possibility that the different brokers can run on different machines, data can be persistently stored in a log-based broker, as long as the log is using the log compaction policy.

2.6 Stream Processing

Log-based message brokers are often associated with stream processing, because the most log-based message brokers have the ability and APIs which can perform stream processing. Earlier we assumed that consumers always pull data from the log in a certain time interval and then receive a batch of records, which leads to the problem that changes are only reflected in the stored data or output records after a certain time has elapsed. The idea behind stream processing is that the fixed time slices are abandoned and every event is processed at the time it happens (Kleppmann 2017, P. 439).

Event streams have a few attributes, such as being infinite and ever growing. They are also ordered, immutable, and replayable (Narkhede et al. 2017, P. 248-249).

On the one hand stream processing has the advantages compared to batch processing, that events are processed immediately. On the other hand state processing becomes more critical in the stream-processing world rather than in the batch-processing world (A. B. Alaasam et al. 2020).

2.6.1 State

When we talk about state in stream processing applications, there are two main approaches in storing.

The first and trivial approach is to store the data on external datastores, often a NoSQL system like Cassandra or MongoDB. The advantage is that there is usually no capacity problem, because they store the data on the disk. And the fact that the external store can be accessed by all instances of the application is very useful to change many objects in one instance at the same time. The main disadvantage is that the datastore causes additional latency when processing an event (Narkhede et al. 2017, P. 253).

The second approach is to store the data on an local or also named internal store, which is mostly managed with an internal embedded, in-memory database running within the application. The most obvious advantage is the extremely fast processing of internal storage. But on the other side, this approach brings with it many complications a stream processing application must address. First of all the local state is now limited to the memory size available to the application instance (Narkhede et al. 2017, P. 253-258). The other problem is the persistency, which seems to be the main problem of internal application states, because in terms of a crash, the instance state is gone. So it is important to keep the state in a separate resource, for example, in an external database, external file system, caching service or sending the state to a dedicated log with log compaction, similar to a database changelog (Kleppmann 2017; A. B. A. Alaasam et al. 2019). In terms of partition rebalancing it is also important that the internal state store can also be rebalanced, so that every instance has the correct state for the partitions the instance is processing (Narkhede et al. 2017, P. 258).

2.6.2 Stream-Table Duality

Streams contain a history of changes, while tables represent a current state of the world. That means they are two sides of the same coin, because they both hold the current state

of the data, but tables are not capable of holding the history of the data. With that fact, a stream can always be converted into a table, but for converting a table into a stream, we additionally need to capture all changes that modify that table. Converting a stream into a table is often referred to as materializing the stream into a view (Narkhede et al. 2017, P. 253-254).

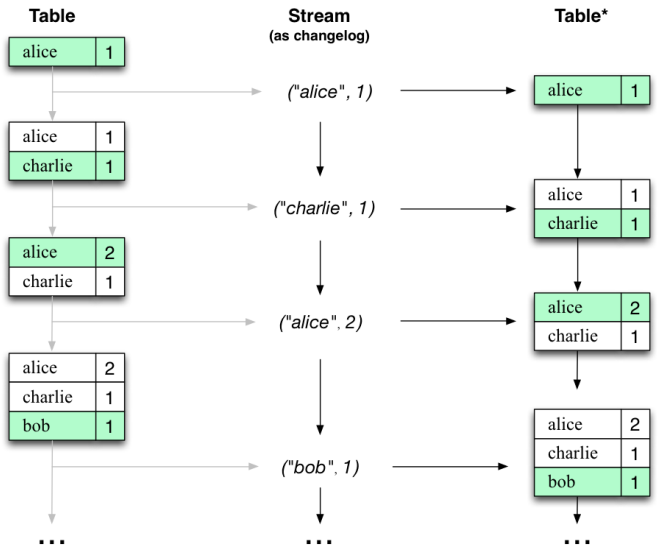

Figure 2.3: *Stream-Table Duality - From (Kafka 2023)*

2.6.3 Materialized Views

When reading all events of a stream from the beginning to the end, and changing a state storage as we go, we have a table representing the state at a specific time (Narkhede et al. 2017, P. 254). With log compaction we do not have to read all messages to get the current state. Using this materializing of a stream, we gain a lot of flexibility by allowing several different read-views, for example one key-value view and one view that is applied to a search index, so that the simple key queries can be performed by the first view and the complex search queries by the other view. Debates about normalization of database data become irrelevant with this approach of translating data from write-optimized event logs into read-optimized materialized views (Kleppmann 2017, P. 454-462).

2.7 Apache Kafka

The main idea of Kafka was to implement a messaging system for log processing that combines the benefits of traditional log aggregators and messaging systems. It was created at LinkedIn as a distributed and scalable messaging system enabling applications to consume log events in real time (Kreps et al. 2011). Kafka is often described as "distributed commit log" or more recently as "distributed streaming platform" (Narkhede et al. 2017, P. 4).

In Kafka messages are categorized into topics and these topics are broken down into partitions, where a partition can be viewed as a single log to which messages are written to in an append-only fashion (Narkhede et al. 2017, P. 5). Consumers are divided into consumer groups, with each group consuming all partitions of one topic. Consumers can be different applications or also many instances of the same application. With these consumer groups Kafka supports load balancing so that only one consumer in a group receives events from a partition (Kleppmann 2017, P. 448). With default settings Kafka supports the at-least-once delivery semantic, but most of the time a message is delivered exactly once to each consumer group (Rooney et al. 2019; Kreps et al. 2011).

To achieve a low-level latency, Kafka supports a so-called zero-copy strategy. This technology does not cache messages in process, instead the messages are only cached in the underlaying file system page cache (Kreps et al. 2011). In the best scenario, where consumers are not far behind in reading messages, we have a very high cache hit ratio and then reads are nearly free in terms of disk I/O (Dobbelaere and Esmaili 2017).

Brokers are completely stateless at Kafka, because the consumer itself maintains the information of the current offset - i.e. how many messages are already consumed. But in addition to Kafka brokers must always run Zookeeper, which also keeps track of the offsets consumed in each partition. Further Zookeeper detects the addition and the removal of brokers or consumers and, in this case, Zookeeper triggers the rebalancing process of each consumer as described later (Kreps et al. 2011).

2.7.1 Why use Kafka?

As seen before, Kafka as an message broker performs very good with high throughput and can also provide low-latency. With its flexible scalability it can easily handle any amount of data (Narkhede et al. 2017, P. 10). So when it comes to an application that needs high throughput, where each message is to be processed fast and where message ordering is important, Kafka is a good choice, because it is well maintained and with the additional features on top it can be used as well for the state storage use case (Kleppmann 2017; Milosavljević et al. 2021, P. 449).

2.7.2 Partitioning

Partitions are the way how Kafka provides redundancy and scalability. The keys of a message determine to which partition the record is appended to, but the mapping is consistent only as long as the number of partitions does not change. When consumers are differently utilized or when a consumer fails or a new one launches, partitions are moved from one consumer to another, which is called rebalancing. Rebalances are important to provide high availability and scalability. In the normal course of events, however, they are fairly undesirable, also because in a stateful stream processing the consumer loses its current state during rebalancing and has to restore the state for the new partitions (Narkhede et al. 2017).

2.7.3 Cleanup Policy

Kafka has a log compaction cleanup policy that is important for storing persistent data inside the topics. As mentioned before log compaction deletes all outdated data of one key and only keeps the most recent record. A log with the compaction policy is divided into two segments, the dirty and the clean segment. The dirty segment is the group of all messages that are appended to the log after the last compaction and the clean segment is already compacted. The log compaction of Kafka does not take place permanently. By default it verifies all 15 seconds if the amount of dirty segments is too high and when the dirty to clean ratio is over 50 percent the compaction will start (Zelenin and Kropp 2022, P.109-111). With a tombstone (cf. section 2.5.1) for one key the entry will be deleted during compaction, while with a null key inside the dirty segment the compaction will fail (Narkhede et al. 2017, P. 110).

With the compaction policy Kafka supports the use cases to store the current state inside the topic, because when recovering from a crash, only the latest state of a key is important (Narkhede et al. 2017, P. 110).

2.7.4 Persistency

Kafka relies on persistent data structures and the file system page cache (Dobbelaere and Esmaili 2017). With the replication of the partitions on different brokers, Kafka guarantees availability and durability when single nodes fail. Kafka divides the replicas into one leader replica, which handles all produce and consume requests to guarantee consistency and ordering, and many follower replica, whose only task is to stay synchronized (Narkhede et al. 2017, P. 97).

2.7.5 Kafka Streams DSL

Apache Kafka already provides two streams APIs that are easy to use - a low level Processor API and a high-level Streams Domain Specific Language (DSL) (Narkhede et al. 2017, P. 265).

The Streams DSL is a feature for transforming stream data from a Kafka source topic to a destination topic by using map, transform or processor functions (A. B. Alaasam et al. 2020). It also introduces a synchronization between local state stores within the application and a Kafka changelog topic, which allows to restore the local store from the topic in cases of application failure or partition rebalancing (A. B. A. Alaasam et al. 2019). The Streams DSL uses consumers and producers as base and provides higher-level functionalities on top (Narkhede et al. 2017, P. 6).

Kafka Streams adds the threading model to the parallelization model with the partitions. In the threading model the number of threads within one partition can be defined to parallelize the stream processing. Amongst these threads there is no shared state, so each thread has its own state store like an additional instance. A thread also acts like a consumer in a consumer group, so that every partition can only be accessed by one thread, which means that the number of partitions has to be bigger than or equal the number of all threads over all instances of the streaming application (Confluent 2023).

2.7.6 Materialized Views

With the Streams DSL Kafka also offers a lot of ways to create materialized views of topics. For example there is the *KTable*, which always stores the latest value of a key into a local key value store (Zelenin and Kropp 2022, P.138). The *GlobalKTable* does the same but the table will be populated with data from all partitions of the topic. The disadvantage, however, is that the storage of each local store is increased and the table is mostly designed for small and relatively static data (Seymour 2021).

The difference is that all applications which use the *KTable* store only one partition (shown in fig. 2.4), while all applications which implement the *GlobalKTable* store all partitions each (shown in fig. 2.5).

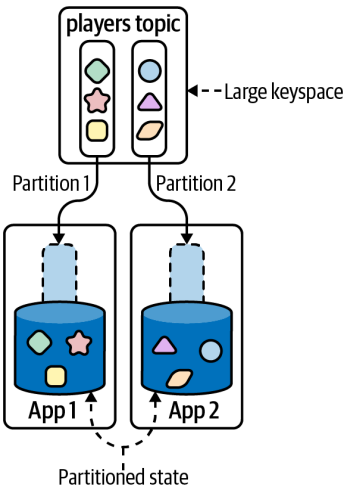

Figure 2.4: *Materialized Views - KTable - From (Seymour 2021)*

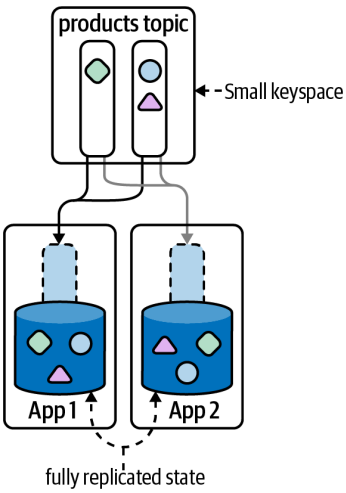

Figure 2.5: *Materialized Views - GlobalKTable - From (Seymour 2021)*

2.7.7 Kafka as State Cache

Kafka Streams has the possibility to create a database system from the scratch. As central element the changelog topic is used and then the platform can be extended with materialized views like tables and search indexes in a distributed way (Zelenin and Kropp 2022, P.156).

The local in-memory state store provided by Kafka Streams can be used inside the stream processing application, which also writes all changes into a compacted changelog topic, that then can be used with materialized views in other services. Kafka also brings a really good data persistency, because Kafka can always recover the state store from the changelog topic. Moreover the Streams API offers the possibility to use *RocksDB*[1] as addition to the in-memory state store, because it also persists the data to disk for quick recovery (Narkhede et al. 2017, P. 258).

[1] RocksDB: `https://rocksdb.org/`

3 Concept

This chapter is about the concept of the main project. The main target of this project is to build an application which stores the current object state of a device as a DT and provide the current state data for other applications, like frontend services. It should also identify the actual state changes and make them available, so that other applications can work with them to create historical timeseries for example.

This application should handle a high throughput with a low-level latency very efficiently in terms of utilized compute capacity.

In the first section the architecture of the project is explained. After that the concepts used in the test are shown, following by the environment in which the project should be deployed.

3.1 Architecture

The application is based on a microservice architecture, because it handles only a single task of a whole IoT system. The task is to store the current state and provide the state and the changes that are made to other services.

Central part of the application is firstly consuming new object changes, secondly storing the state inside local in-memory state stores and thirdly providing this store to other services through a changelog topic. Local state stores are used, because as shown in 2.6.1, external datastores cause additional latency which results in the service being able to handle a lower throughput than with a local state store. This application has similarities to a Change Data Capture (CDC) architecture, which is used by the most databases to capture and distribute a stream of database updates, so that the updates are available on other applications or datastores (Kleppmann 2017, P. 454).

With Kafka Streams DSL there is the possibility to create such a CDC architecture, because we can use the in-memory stores the Streams API provides, thus automatically writes all updates to a compacted changelog topic. Kafka is also often used by other databases as a changelog, because as already shown before in section 2.7, the log is persistent to disk and replicated for fault tolerance (Kreps 2017). So in the main part there is a local state store, that can handle the state changes as fast as possible. Further there is the external changelog topic stored in Kafka, which persists and replicates the data and also provides the state to potentially many materialized views. One of these views provides a Representational State Transfer (REST) API with methods to get all or one specific object from the state store. The changelog topic together with the state should only be changed by one service, handling the state store, while all other services can only consume the topic to own read-only state stores.

Kafka is used in this project because, together with the Streams DSL, it combines the messaging between the services and the handling of the local state storage with the automatic persistence and distribution over the changelog topic. Kafka also handles a lot of throughput with a low-level latency as shown in 2.4.3 and the accumulation of

data does not make it slower (Kreps 2017). Changing the local store inside the Kafka Streams processing application is only an overhead and not recommended. The persistent state store, which is implemented with *RocksDB* is also very performant, because it is an embedded, very fast key-value store, which flushes the data asynchronously to disk (Golder 2022). And using the in-memory state store is even faster, since it is maintained by a hash map structure held inside the Java Virtual Machine (JVM)'s heap (Kafka 2023). In the main part the objects are only queried by the key, so it is not relevant to use a more queryable database. The only part where a more queryable datastore could make sense, is in an additional frontend service, to allow more complex queries, but this is not part of this project and only discussed in chapter 6.

The application is divided into three services also shown in figure 3.1:

1. Split Message Service: The main entry point of the device data.
2. Object Changes Handler: The central part of the application, which stores the state inside the local storage and provides it to other services.
3. Frontend Service: The REST API entry point for querying the state store from outside.

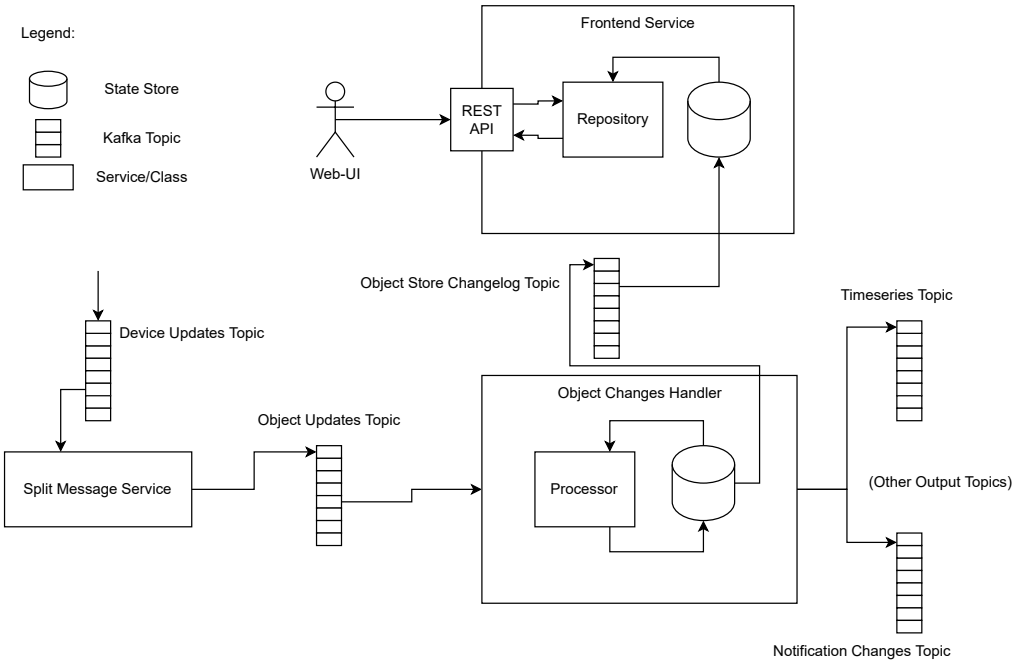

Figure 3.1: *Project Overview Architecture*

3.1.1 Split Message Service

This service consumes all data that comes from the devices and splits the data into one message per object. In the whole IoT system one device can send many changes of different object identifier (ID)s. With that fact, the split into one message per object is necessary for correct handling and processing of the object state inside the next service, because the data

has to be partitioned with the object ID as partition key. This guarantees an ordering of the same object IDs and an assignment to the processing object changes handler instance that holds the correct object in the local state store.

3.1.2 Object Changes Handler

The Object Changes Handler service is the main and intern part of the project architecture. It uses Kafka Streams to simultaneously perform the functions of a data consumer, producer and processor (A. B. A. Alaasam et al. 2019). As ingoing stream the service consumes the topic of the split service, which is partitioned by the object ID. For processing the stream it is necessary to write a stateful Kafka Streams Processor, which handles the data and authorization validations just as the change synchronization of the object data inside the local state store (shown in fig. 3.2). The authorization is a service inside the IoT project, which stores and handles the permissions of each user for the different resources, such as an object for example. For this reason the object changes handler has to check, if the user, which in this case is the device, has the permission to change the object.

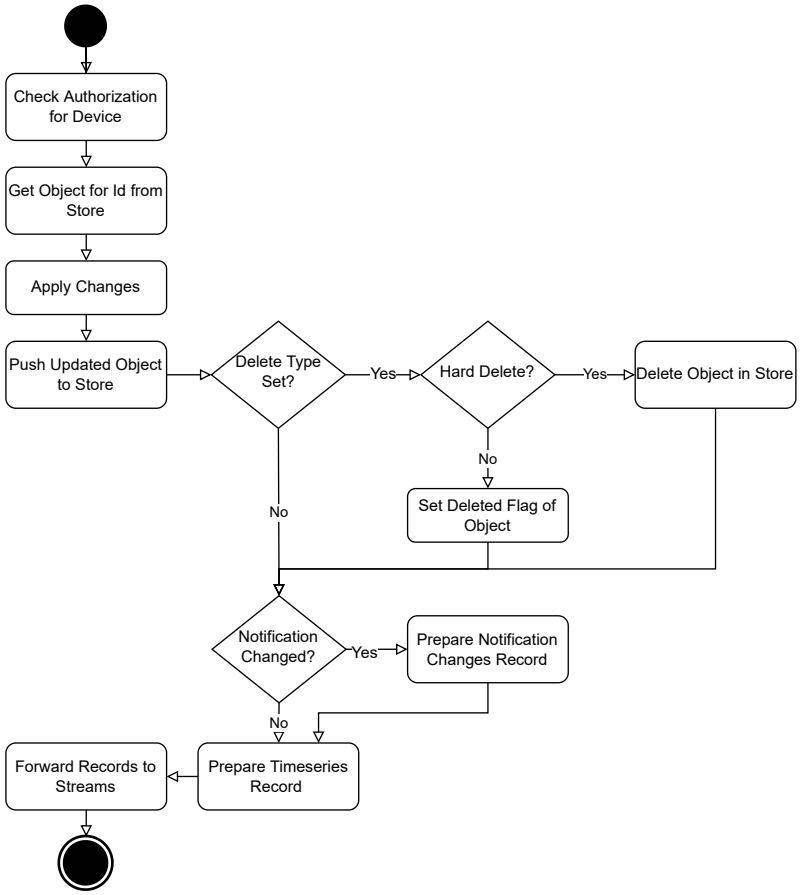

Figure 3.2: *Object Changes Handler - Processor - Activity Diagram*

The service should also include the same core functionalities as the predecessor implementation. This comprises the possibility of deleting objects either completely from disk or of attributing a tag to them, which means that the object has a delete tag with the message timestamp.

As a local state store, the decision is between an in-memory or a persistent (*RocksDB*, cf. 3.1) state store implementation. The primary benefits of the persistent store are that the state can exceed the size of available memory. Additionally, in terms of a failure, the state store can be recovered from disk faster than the in-memory store, which only restores from the changelog topic. On the other side the in-memory store is operationally less complex and faster than the persistent store, especially when the data has to be queried from the disk (Seymour 2021). The project shows the implementation of both stores and also the comparison in the test results.

Lastly the service applies all state changes to outgoing topics which can be consumed by other services, like the timeseries or the notification service of the IoT project. The timeseries service requires all updates made to the actual data, so that they can be stored as historical data series. The notification service gets the notifications that are set by the device or the object changes handler itself during validations.

3.1.3 Frontend Service

For the REST requests from other services like a web based user interface (Web-UI) there is the frontend service. This service only has to consume the changelog topic and store it into a local state store as materialized view, so that it can be easily queried from the internal REST API. It should handle the query of all objects and a specific object called by the ID.

For materialization the *KTable* of Kafka Streams can be used. The *GlobalKTable* would not be performant enough in this case, because it is made for few and static data, as it is discussed in section 2.7.6. The disadvantage is that, if there are many instances of this service, one instance does not contain all partitions of the changelog topic and thus does not contain the entire state. That means for the REST API, that sometimes requests end up at the wrong instance, which does not hold the object of the ID. For this reason Kafka supports metadata calls to find out the right instance for a specific key. With this information the service should perform a so called remote query to access the correct instance (Schmid 2018). For a query which should return all objects there is also a metadata call, which returns all active instances holding partitions of the state store (Confluent 2023).

In the following diagram a request for an specific object by ID is shown.

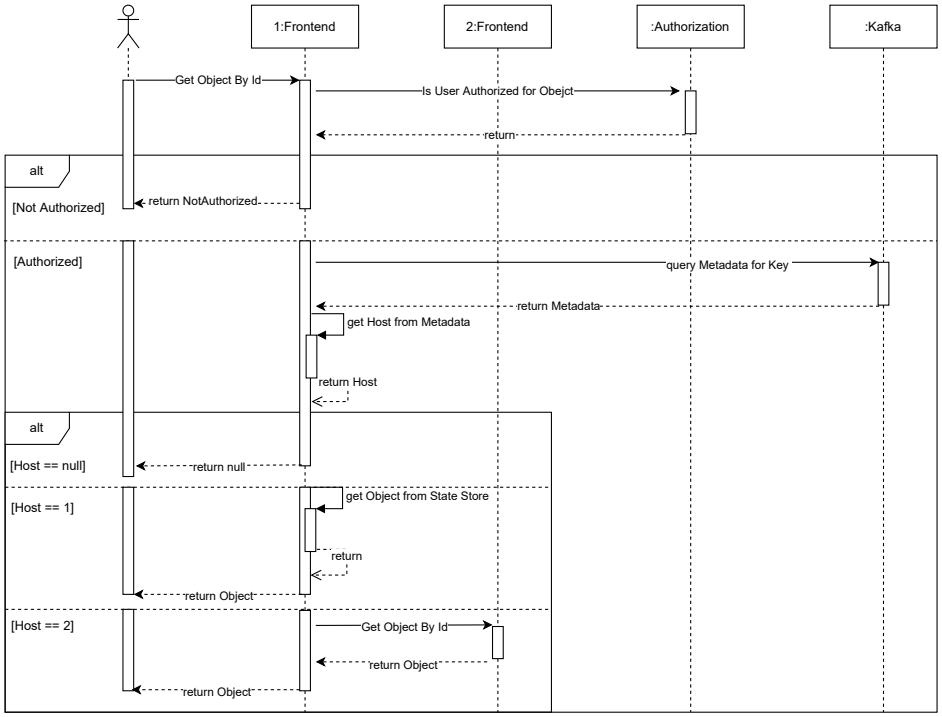

Figure 3.3: *Frontend Handler - Get ID Request - Diagram*

Additionally to the diagram the second instance also executes all the steps in the same way as the first one does, because it can be used exactly the same REST API method that is used from outside. But for the "get all" query an extra method for internal communication is needed, because otherwise the frontend service instances would run into a loop. This is due to the fact that all active frontend instances are queried in the "get all" method.

3.2 Testing

For testing the services it is necessary to track the metrics of the services and compare them with the predecessor implementation, which uses a NoSQL Database for storing the state. With the collected data of the metrics, the old system can be compared with the new services and it can be evaluated if the new implementation has a better performance. The different state store types are also to be tested and compared to investigate the real difference in latency in this environment.

3.2.1 Simulation Service

For creating reproducible data that can be used in the test environment, it is necessary to create a simulation service, which simulates test data of many devices in a certain interval. Each device should contain a list of object IDs for which the data should always be changed. The outgoing topic of this service is the topic, which the split service or the

old Model service will consume in the following. This service is only necessary for the test environment and should not impact the productive environment.

3.2.2 Predecessor

The old existing application of the IoT project, which processed and stored the DT data, is so called the Model service. As object datastore the implementation uses *MongoDB*[1]. *MongoDB* is a very powerful and automatic scalable database for object storage, where the records stored inside are similar format to JavaScript Object Notation (JSON) objects. The database also has a rich query language to support all types of read and write operations (Chauhan 2019). This service consumes the Kafka topic directly from the devices and handles all objects inside. Therefore it is not necessary to split the data into one message per object. The REST API also requests the database directly.

For comparability of the new and the old system, the old service will be also tested with the same metrics.

3.2.3 Metrics

The essential comparison between the new and the predecessor implementation is the performance, that can be measured by the throughput and the latency of the service (Fu et al. 2021). For this performance metrics we can use a counter and a timer and build the arithmetic mean over a timespan. Also important for the service performance is the CPU usage and the memory consumption.

For testing the frontend performance the time should be tracked in which different REST API calls are processed. The main calls for the test are the "get all objects" and the "get one specific object by the ID". The arithmetic mean is not the best metric to use here because it does not indicate the delay experienced by the user. So for user responses it is better to use percentiles. The 50% percentile is also known as the median, but for response times it can be useful to compare also the high percentiles, known as tail latencies, because they directly affect the user experience of the service (Kleppmann 2017, P. 14-15).

3.3 Environment

The microservice architecture used for this project fits a container-virtualized infrastructure with technologies used such as *Docker*[2] and *Kubernetes*[3] (Pahl and Jamshidi 2016). Such container-based virtualization is much more lightweight and resource efficient than one based on a Virtual Machine (VM). Because the containers are sharing the same operation system as isolated processes on the core-level of the system. By deploying every service as a container, the infrastructure can easily perform vertical and horizontal scaling, according to the resource needs and the throughput the services have to process (Al-Dhuraibi et al. 2018).

[1] MongoDB: `https://www.mongodb.com/`
[2] Docker: `https://www.docker.com/`
[3] Kubernetes: `https://kubernetes.io/de/`

4 Results

In this chapter the implementation of the application services is shown. Further the the previous model and the testing service implementations are described. After that the setup of the test environment is explained and at the end there is a comparison and analysis of the test results.

4.1 Implementation

As described in the concept, the application is divided into three microservices. All services are implemented in *Java*[1] as *Dropwizard*[2] *ServiceApplications*. For the data connection between the services Kafka topics are used as explained before. The consumer service checks if the topic exists and creates it if not.

4.1.1 Split Message Service

The Split Service consumes the device data with Kafka Streams, transforms and then applies it to the topic of the Object Changes Handler. To manage the Kafka Streams in Dropwizard we have to create the streams inside a class implementing the *Managed* interface of Dropwizard. After that it can be added into the Dropwizard lifecycle inside the *run* function of the *ServiceApplication*.

```
1 @Override
2 public void run(ServiceConfiguration configuration,
3     Environment environment, ObjectMapper mapper)
4     throws Exception {
5     ...
6     var transformer = new MessageTransformer(...);
7     environment.lifecycle().manage(transformer);
8 }
```

Listing 4.1: *Dropwizard Lifecycle*

In the overridden *start* function of the *MessageTransformer* class, which implements the *Managed* interface, the Kafka topology can be defined with the *StreamsBuilder*. First the input stream is defined, then the transformation and at last the output topic, where the stream should be applied to. For the use of own class definitions in the topics, an own serializer, called *Serdes* in Kafka, is added. For the transformation of the one device event into many object events, the Streams API has the function *flatMap* which can map one stream event to many with the help of a lambda function (Kafka 2023). Inside the lambda we only divide the input device key-value pair into a list of object key-value pairs, where the key is always the object ID.

[1] Java: https://www.java.com/
[2] Dropwizard: https://www.dropwizard.io/en/stable/

```
@Override
public void start() throws Exception {
    var inputSerde = Serdes.serdeFrom(
        inputTypeSerializer, inputTypeDeserializer);

    ...

    StreamBuilder streamBuilder = new StreamsBuilder();
    KStream stream = streamBuilder.stream(inputTopicName,
        Consumed.with(Serdes.String(), inputSerde));
    KStream transformed = stream.flatMap(this::transform);
    transformed.to(this.outputTopicName,
        Produced.with(Serdes.String(), outputSerde));
    ...
}
```

Listing 4.2: *Split Service - Kafka Topology*

Before starting the streams we have to adjust the properties. For this purpose we create a *Properties* object and then add it to the Kafka Streams constructor. For the Split Service we only need the standard configuration. So we have to set the server config to the Kafka host and port of the environment. Additionally the application ID which should be unique within the Kafka cluster. Also, all Split Service instances should have the same ID so that the cluster can identify all service instances as a consumer group. With the fact that the application ID is used as consumer group, the partitions are divided between all instances of the Split Service, as explained in section 2.7. The last property is the client ID, which should be unique for every instance (Confluent 2023).

```
final Properties props = new Properties();
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, host);
props.put(StreamsConfig.APPLICATION_ID_CONFIG, appId);
props.put(StreamsConfig.CLIENT_ID_CONFIG, clientId);

this.streams = new KafkaStreams(streamBuilder.build(),
    props);

// Exception handling

this.streams.start();
```

Listing 4.3: *Split Service - Kafka Streams Configuration*

With this implementation we now have created an output stream written into a topic that can be consumed by the Object Changes Handler. The events of the topic are partitioned with the object ID as partition key, so that there is the guarantee of an in-order processing of one object ID (Log Ordering, cf. 2.4.2). This also means that events of an object ID are always consumed by the same Object Changes Handler instance, which is important for the state handling inside the service.

4.1.2 Object Changes Handler

The Object Changes Handler consumes the object data, processes and validates it, stores it in an in-memory local store and then sends the updated data to outgoing topics. As seen above in the service to manage the streams in *Dropwizard* we use a class which implements the *Managed* interface.

In this service we mainly use the same Kafka Streams topology as seen before in listing 4.2. The only difference is that the service uses the *process* function of Kafka Streams, which gets a processor supplier as a parameter.

For the configuration of streams this time we add two properties to the default ones shown in listing 4.3. One is the replication factor configuration setting the replication factor for the changelog topic created by the state store. We should put it at least on 3, because as explained in section 2.7.7 this is the part the state store data is persistent especially when using the in-memory store only. The *STATE_DIR_CONFIG* configuration sets the directory where the state data is stored on the disk, but it only comes into play when we use the persistent state store. In a production system this has to be set to a directory that is persistent even if the instance fails (Confluent 2023).

```
1 props.put(StreamsConfig.STATE_DIR_CONFIG, stateStoreDir);
2 props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 3);
```

Listing 4.4: *Object Changes Handler - Kafka Streams Configuration*

This time a class that implements the *Processor* interface of Kafka Streams handles all the input data. In the *Processor* interface Kafka already provides the possibility to access state stores through the *ProcessorContext*. The stores have to be created in the topology itself or in the processor supplier. For creating the stores inside the supplier there is a function called *stores*, which must be overridden. The function has to return then a set of state store builder (Kafka 2023).

```
1  public class ObjectChangesProcessorSupplier
2      implements ProcessorSupplier<IK, IV, OK, OV> {
3
4      private final String storeName;
5
6      @Override
7      public Processor<IK, IV, OK, OV> get() {
8          return new ObjectChangesProcessor(this.storeName);
9      }
10
11     @Override
12     public Set<StoreBuilder<?>> stores() {
13         var modelObjectSerde = Serdes.serdeFrom(
14             storeTypeSerializer, storeTypeDeserializer);
15
16         var storeSupplier = Stores
17             // For in memory only state store
18             .inMemoryKeyValueStore(this.storeName);
19             // For persistent state store
20             .persistentKeyValueStore(this.storeName);
21         StoreBuilder<KeyValueStore<UUID, ModelObject>>
```

```
22                 storeBuilder = Stores.keyValueStoreBuilder(
23                     storeSupplier, Serdes.UUID(),
24                     modelObjectSerde);
25             return Collections.singleton(storeBuilder);
26         }
27     }
```

Listing 4.5: *Object Changes Handler - Processor Supplier*

The stores can then be accessed in the *init* function of the *Processor* via the store name.

```
1  @Override
2  public void init(ProcessorContext<OK, OV> context) {
3      this.keyValueStore =
4          context.getStateStore(this.storeName);
5      this.context = context;
6      ...
7  }
```

Listing 4.6: *Object Changes Handler - Processor - Init Function*

With this initialization, the key-value store can be easily accessed in the actual *process* function of the processor. We can now get the data of the object by a simple *get* call, append then the new object changes and push the updated object back with the *put* function. At the end of the *process* function the updated object can be appended as a record to the outgoing stream for the other services by forwarding it into the context. To permanently delete objects, there is the possibility to set a delete type in the incoming object data (cf. fig. 3.2). The Object Changes Handler can then delete it permanently with the *delete* function of the key-value store.

```
1  @Override
2  public void process(Record<IK, IV> inputRecord) {
3      var objectId = inputRecord.key();
4      ModelObject oldObject =
5          this.keyValueStore.get(objectId)
6      ...
7      this.keyValueStore.put(objectId, updatedObject);
8      ...
9      ModelObject deletedObject =
10         this.keyValueStore.delete(objectId)
11     ...
12     this.context.forward(new Record<OK, OV>(objectId,
13         objectChanges, timestamp));
14 }
```

Listing 4.7: *Object Changes Handler - Processor - Process Function*

Kafka is already creating a compact changelog topic of the intern state store and with every *put* call the new object is also applied to the topic. By deleting the object inside the key-value store, there is a record applied to the topic with a tombstone for the ID so that Kafka deletes this object in the next compaction period, as explained before in section 2.7.3.

As shown above the changelog topic is the only opportunity to persist the data when we us the in-memory store as local store. However, the changelog topic is also used for the persistent store. Thus we can access the current object state in every other service through this topic. The implementation of this is shown right away in the Frontend Service.

4.1.3 Frontend Service

The Frontend Service consumes the changelog topic as *KTable*, which at each time stores the latest value of a key inside a materialized local store. It also provides a REST API for other services, like a Web-UI, to get, create, update or delete model objects.

The *KTable* stores only data of the partitions assigned to the service instance, which makes API requests more complicated, because we have to implement remote queries. A *GlobalKTable* does not have this problem, but we cannot use it here, because it is designed for small and static data and then the service is also not scalable. The difference between these two tables is described before in section 2.7.6.

For the initialization of Kafka Streams we need a different topology than for the other services (shown in listing 4.2). To the *StreamsBuilder* we only add a table, for which we define the type of the materialization of the local store. For the materialization we need to add a store supplier which is created similar to the state store in the object changes handler, shown in listing 4.5.

```
1 ...
2 Materialized<UUID, ModelObject,
3     KeyValueStore<Bytes, byte[]>> materialized =
4     Materialized.as(storeSupplier);
5 materialized = materialized.withKeySerde(Serdes.UUID())
6     .withValueSerde(modelObjectSerde);
7
8 streamBuilder.table(this.cacheTopicName, materialized);
```

Listing 4.8: *Frontend Service - Kafka Topology*

The configuration of Kafka Streams is quite similar to the other services. The default properties, shown in listing 4.3, and also the state directory configuration has to be set, if we want to use the persistent state store, shown in listing 4.4. Additional to these properties there is a performance optimization configuration for the *KTable*, which has the effect that the source topic of the table is also used as the changelog topic (Confluent 2023). This has the positive effect that Kafka does not create a new changelog topic for the table materialization with exact the same data inside. Because of this we save disk space and CPU usage of Kafka. When we want to set the optimization we also have to overload the *StreamsBuilder* build method, as shown in the listing 4.9 right after (Kafka 2023).

As second additional property we have to set the application server configuration, which is necessary for the remote queries we want to implement. This should be set on the host and port of the container of each Frontend Service instance (Confluent 2023).

```
1  ...
2  props.put(StreamsConfig.APPLICATION_SERVER_CONFIG,
3                  String.format("%s:%s", nodeHost,
4                  nodePort));
5  props.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG,
6      StreamsConfig.OPTIMIZE);
7
8  this.streams = new KafkaStreams(streamBuilder
9      .build(props), props);
```

Listing 4.9: *Frontend Service - Kafka Streams Configuration*

After starting the stream the state store materialization of the *KTable* can be accessed through the *store* function of Streams. To do this, we have to define the store query parameters, which can be set with the store name and the type. The store name was set in the store supplier shown in listing 4.5. For the type we use in the frontend the read only key-value store, because we only need the objects for specific IDs and are not interested in objects in a specific timespan.

```
1  ...
2  this.streams.start();
3
4  StoreQueryParameters<ReadOnlyKeyValueStore<UUID,
5      ModelObject>> storeQueryParameters =
6      StoreQueryParameters.fromNameAndType(objectsStoreName,
7          QueryableStoreTypes.keyValueStore());
8  var store = this.streams.store(storeQueryParameters);
9
10 var modelRepository = new ModelRepositoryImpl(
11     objectsStoreName, this.streams, this.store);
12
13 this.environment.jersey().register(new ModelRestApi(
14     modelRepository));
```

Listing 4.10: *Frontend Service - KTable Store Access*

As shown in the listing 4.10, the key-value store can then be passed to the repository, which then handles all internal processing and communication with the state store. *Dropwizard* uses *Jersey*[3] as REST client. We have to register an API class, in which the endpoints are defined, to the environment, so that *Dropwizard* handles the REST calls.

In the API file we can define the endpoints by assigning *javax* annotations to the methods, like the HTTP type with *GET*, *POST*, *DELETE* and the path of the request with *Path*.

```
1  @Produces(MediaType.APPLICATION_JSON)
2  @Consumes(MediaType.APPLICATION_JSON)
3  public class ModelRestApi {
4
5      ...
6
7      @GET
```

[3]Jersey: https://eclipse-ee4j.github.io/jersey/

```
8       @Path("{id}")
9       public Optional<ModelObject> getId(@Context
10          HttpHeaders headers, @Auth ...,
11          @PathParam("id") UUID id) {
12          return this.modelRepository.getById(id, ...);
13      }
14
15      @POST
16      @Path("instance")
17      public Map<UUID, ModelObject> getInstanceObjects(
18          @Auth ..., @Valid InstanceRequest instanceRequest)
19          {
20          return this.modelRepository
21              .getObjectsFromInstance(instanceRequest, ...);
22      }
23
24      @GET
25      public Map<UUID, ModelObject> getAll(...) {
26          return this.modelRepository.getAll(...);
27      }
28
29      @POST
30      @Path("create")
31      public UUID create(@Auth ..., @Valid @NotEmpty
32          Map<UUID, Map<String, Object>> objectChanges) {
33          var transactionId = Utils.buildTransactionId(...);
34
35          this.modelRepository.create(objectChanges, ...);
36          return transactionId;
37      }
38
39      ...
40  }
```

Listing 4.11: *Frontend Service - Rest API*

In the repository we can then access the local state store with a *get* method, as we saw before in listing 4.7, or with an *all* method, which returns all objects of this local store. The main problem is that we can only access the objects with those keys, where the partition of the key is handled by the instance. This means when a frontend instance gets accessed with a request of a key not stored in this instance, we have to forward this request to the right instance. For that Kafka offers metadata calls that we can use to find out the right instance for the key and also all active frontend nodes (Confluent 2023). From the metadata we get the host and the port of the respective instance, which we defined before in the streams configuration, shown in listing 4.9. When we search for a specific object the first instance forwards the request to the right instance if possible, described in the concept diagram 3.3. When we want all objects of the other instances we have to use a different request, than the *getAll* method, for the internal communication. A request which returns all objects only of the own instance is used, because we do not want the instance to also search and request others (Schmid 2018). We already defined this

endpoint in listing 4.11 with the method *getInstanceObjects*. Listing 4.12 illustrates the implementation of the *getById* method in the repository.

```
public Optional<ModelObject> getById(UUID id, ...) {
    ...

    var metadata = this.streams.queryMetadataForKey(
        this.objectsStoreName, id,
        Serdes.UUID().serializer());

    if (metadata == null) {
        return Optional.empty();
    }

    var remoteHost = metadata.activeHost();

    if (this.hostInfo.equalsHostInfo(remoteHost.get())) {
        return Optional.of(this.store.get(id));
    }

    var target = String.format("http://%s:%s",
        remoteHost.get().host(), remoteHost.get().port());
    var response = this.client.target(target)
        .path(id.toString())
        .request(MediaType.APPLICATION_JSON_TYPE)
        .headers(...).get();

    if (response.getStatus() > 204) {
        // Error handling if host is unavailable
    }

    return response.readEntity(new
        GenericType<Optional<ModelObject>>() {});
}
```

Listing 4.12: *Frontend Service - Repository - Get Object By ID*

In this implementation shown in listing 4.13 we receive all object IDs, which the user is permitted to read, from the authorization service. With this fact we do not use the *all* method of the key-value store, realizing that most of the time we do not want all objects of the store. But the authorized object IDs are also forwarded to the other instances via the *getInstanceObjects* method. This is because sending a REST API request for each ID is slower than querying the local store for each ID in each instance.

```
public Map<UUID, ModelObject> getObjectsFromInstance(
    InstanceRequest instanceRequest, ...) {
    ...
    var modelObjectIds = instanceRequest.getObjectIds();
    var allEntries = new HashMap<UUID, ModelObject>();

    for (var id : modelObjectIds) {
        var modelObject = this.store.get(id);

        if (modelObject == null) {
```

```
11                  continue;
12              }
13
14              // Check other preconditions defined in request
15              if (...) {
16                  allEntries.put(id, modelObject);
17              }
18          }
19          return allEntries;
20      }
21
22      public Map<UUID, ModelObject> getAll(...) {
23          ...
24          var allEntries = new HashMap<UUID, ModelObject>();
25          var instanceRequest = new InstanceRequest(
26              authorisedObjectIds, ...);
27
28          for (var metadata : this.streams
29              .streamsMetadataForStore(this.objectsStoreName)) {
30              if (this.hostInfo.equalsHostInfo(metadata
31                  .hostInfo())) {
32                  allEntries.putAll(this.getObjectsFromInstance(
33                      instanceRequest, ...));
34              } else {
35                  var target = String.format("http://%s:%s",
36                      metadata.host(), metadata.port());
37                  var response = this.client.target(target)
38                      .path("instance")
39                      .request(MediaType.APPLICATION_JSON_TYPE)
40                      .headers(...)
41                      .post(Entity.json(instanceRequest));
42
43                  if (response.getStatus() > 204) {
44                      // Error handling unavailable instance
45                  }
46
47                  var remoteValues = response.readEntity(new
48                      GenericType<Map<UUID, ModelObject>>() {});
49                  allEntries.putAll(remoteValues);
50              }
51          }
52          return allEntries;
53      }
```

Listing 4.13: *Frontend Service - Repository - Get All*

The concept of the *KTable* is a read-only state storage, so we cannot create or update changes, and we also want to change the state only in the Object Changes Handler as defined before in section 3.1. For this reason all change requests like the *create* method push the changes into the Kafka topic which is consumed by the Split Service.

4.2 Testing

4.2.1 Simulation Service

The Simulation Service is implemented as a simple service starting a couple of jobs that are executed in a certain interval. Every job gets a number of object keys and then creates many objects with random number values for a simulated device. Since the main focus is on the performance of the number of objects handled by the model services, all the objects have only two values. One is a random number that should be updated in the object and the second one is an increment value, which should be incremented in the model service each time. These two types are used, because they are the most used in the predecessor service.

For job handling we can use *Quartz*[4], which is a very well featured job scheduling library for Java. We want to execute many jobs in parallel. So we set the thread pool number to the number of jobs in the scheduler configuration.

```
var props = new Properties();
props.put("org.quartz.scheduler.instanceName", schedName);
props.put("org.quartz.threadPool.class",
    "org.quartz.simpl.SimpleThreadPool");
props.put("org.quartz.threadPool.threadCount",
    numDevices.toString());

var factory = new StdSchedulerFactory(props);
this.scheduler = factory.getScheduler();
```

Listing 4.14: *Simulation Service - Scheduler Configuration*

We have to create a job that creates every time a new value for each object and then sends the data to the Kafka topic, consumed either by the Split Service or the consumer of the predecessor implementation. For the device ID we use a random ID, because we want to divide the data evenly into the Kafka partitions and the device ID is not relevant for this test case.

```
for (var objectId : jobData.getObjectIds()) {
    var tV = new ValueChange(Math.random(),
        ValueChangeType.Set);
    var tInc = new ValueChange(1,
        ValueChangeType.Increment);

    objectChanges.put(objectId, Map.of("TV", tV, ...));
}

var modelChanges = new ModelChanges(..., objectChanges);
var changeRequest = new AuthenticatedMessage<>(
    transactionId, jobData.getUserId(), modelChanges);
jobData.getChangesProducer().accept(UUID.randomUUID()
    .toString(), changeRequest);
```

Listing 4.15: *Simulation Service - Data Job*

[4]Quartz: `http://www.quartz-scheduler.org/`

In the end we only have to add these jobs and a trigger to the scheduler. For every device we create the defined object IDs and pass them to the job via the job data, so that the job sends the changes made to the same objects each time.

```
for (var i = 0; i < numDevices; i++) {
    ...
    var job = JobBuilder.newJob(SimulateDataJob.class)
        .withIdentity("Job" + i)
        .setJobData(jobDataMap).build();
    var trigger = TriggerBuilder
        .newTrigger().withIdentity("T" + i).startNow()
        .withSchedule(SimpleScheduleBuilder
            .simpleSchedule().withIntervalInSeconds(sec)
            .repeatForever())
        .forJob(job).build();

    this.scheduler.scheduleJob(job, trigger);
}
```

Listing 4.16: *Simulation Service - Schedule Job*

Now, in the *Dropwizard* configuration, we can define the interval time in seconds, the number of devices and objects within a device each time the service is started.

4.2.2 Metrics

For collecting metrics we use *Prometheus*[5], which is an open source monitoring system. It provides metrics like counter, timer or a histogram and also other metrics can be published to it. We measure latency and throughput of the services with the *Histogram* metric, by starting the timer at the beginning of the process and stop it at the end, as shown in listing 4.17 inside the Objects Changes Handler.

```
public static final Histogram processHistogram =
    Histogram.build().buckets(0.00005, 0.0001, ...)
    .name("object_changes_process").register();

public void process(Record<IK, IV> inputRecord) {
    var processTimer = ObjectChangesProcessor.
        processHistogram.startTimer();
    ...
    processTimer.observeDuration();
    this.context.forward(...);
}
```

Listing 4.17: *Metrics - Histogram Implementation*

For the REST API response metrics we can use the metrics *Dropwizard* already provides. *Jersey* has different annotations that we can use to instrument methods, so that the response time or rate is written into metrics (Dropwizard 2023). We intend to track the response time, which we can achieve with the *@Timed* annotation on class basis, so that all methods

[5]Prometheus: https://prometheus.io/

are tracked. As output we get a histogram with the response time divided into different percentiles, with the 50% percentile representing the median.

4.2.3 Testing Environment

For testing all the services and Kafka as well as *MongoDB* for the predecessor implementation, the resources and dependencies have been deployed on *Azure*[6]. The utility services like the authorization service, the *Prometheus* or the Simulation Service are running on a VM named *Standard_DS2_v2 specification*[7] with 2 vCPUs and 7 GiB of memory. Kafka is running with three brokers on three VMs named *Standard_D8as_v4 specification*[8], with 8 vCPUs and 32 GiB of memory. In the tests with the new model, all services run on the *Standard_D8as_v4 specification* VM. For better comparability the predecessor implementation and the *MongoDB* replica set are run on the same machine in the comparative tests. The replica set consists of three nodes, where one node is the primary and handles all client requests and appends the updated data to the secondary nodes.

The azure infrastructure is created via *Terraform*[9], which allows you to easily build or change infrastructure components like compute instances, storage or networking. *Kubernetes* resources like the services are managed with *Helm*[10] in *Helmfiles*[11].

With this environment we can easily deploy, destroy or change the services and the configurations.

4.3 Performance Tests

4.3.1 Configuration

In all tests the same number of 50 devices are used, because it is also interesting to compare the latency change of the Split Service and that depends on the number of objects inside one device message. The device number does not affect the latency of the old predecessor implementation, because there the time is measured, which every object needs to apply its changes and not the time every device message needs. It also does not affect the latency of the new Object Changes Handler, because this service gets one message per object. So the number of objects by device are only adjusted.

In addition, the same interval time of 1 second is used for the test data in all tests and each measurement covers a period of 10 minutes.

For the frontend tests 100 messages per second are simulated. A REST client sends a request every 5 seconds in a timespan of 5 minutes per method. In order to test the remote queries, the *getId* call was executed twice for every configuration. Once the query is sent to the correct instance, where the ID is stored, and once the query is sent to the wrong instance, which then must forward it to the instance that stores the ID.

[6] Azure: `https://azure.microsoft.com/de-de`
[7] Standard_DS2_v2 specification: `https://azureprice.net/vm/Standard_DS2_v2`
[8] Standard_D8as_v4 specification: `https://azureprice.net/vm/Standard_D8as_v4`
[9] Terraform: `https://developer.hashicorp.com/terraform`
[10] Helm: `https://helm.sh/`
[11] Helmfiles: `https://helmfile.readthedocs.io/en/latest/`

The partitioning and replication of Kafka is always the same. 12 partitions are always used, which allow to divide by 2 and 3 for the tests of the new services with up to 3 instances and which also allow to divide by 2, 6 and 12 for the old predecessor implementation. This has the reason that all instances should have the same number of partitions at any time for the best performance of the respective service. More partitions are not equal to a higher throughput and would also bring problems with it, because every client buffers per partition and so the memory consumption increases with the partitions (Zelenin and Kropp 2022, P. 60).

The Kafka brokers and also the *MongoDB* cluster was not adjusted during the tests. Kafka is always running with three brokers and clusters on three different VMs. And *MongoDB* is also running with one replica set consisting of three nodes. Both configurations were already described in section 4.2.3. Every time the CPU and memory usage of these two tools are shown, the summary of all clusters, brokers or nodes are displayed. The new model services are also summarized in the resource utilization, because a comparison between the old predecessor implementation and the whole new system is made. And the predecessor implementation contains all three functionalities inside one service, while the new system is divided into these three services. The three services are also performing three different jobs, so that the comparison between these services is not relevant.

All CPU resources are measured and displayed in *cpu* units, whereby 1 CPU unit is equivalent to 1 virtual core. With this unit, the test results are mostly independent of the computing power of the VM used. This is also the unit for resource requests used in container creation in *Kubernetes* (Kubernetes 2023).

4.3.2 State Store Type Comparison

In the first comparison the following tests are executed for the two types of the local state store.

		Instances		
State Store	Simulated Msg/s	Split	Object Changes	Frontend
In-Memory	1000; 2000; 5000; 10000	1	1	1
In-Memory	1000; 2000; 5000; 10000	1	2	1
In-Memory	1000; 2000; 5000; 10000	1	3	1
Persistent	1000; 2000; 5000; 10000	1	1	1
Persistent	1000; 2000; 5000; 10000	1	2	1
Persistent	1000; 2000; 5000; 10000	1	3	1

Table 4.1: *State Store Type Comparison - Test Configuration*

In this comparison we want to look at the performance difference between the two state store types Kafka provides as key-value stores. The first one is the in-memory store which only holds the data in the memory of the instance. The second one is the persistent state store, which holds the data also in Random Access Memory (RAM), but also asynchronously persists it to disk, as discussed in section 2.7.7. For all the tests only

the Object Changes Handler was increased in the number of instances, because the other services are not relevant for the comparison.

Chart 4.1 illustrates the latency comparison of the two types of state stores as well as the comparison of a different number of Object Changes Handler instances.

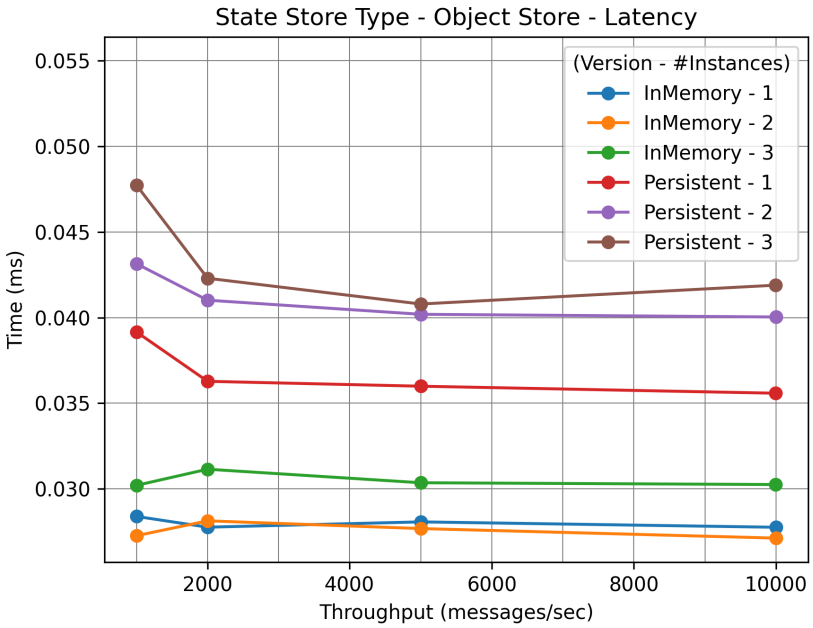

Figure 4.1: *State Store Type - Object Store - Latency*

As we can see, the latency of the persistent state store is higher, but it does not have a great impact. We also see that the number of instances affects latency, but only to a small extent. Even three instances with the in-memory store have lower latency than one with the persistent state store. It is also shown that the amount of throughput does not affect the latency. In summary, the selection of the state store has the greatest impact on latency in this test.

In the next charts we want to compare the CPU and memory usage of the Kafka brokers and the model services.

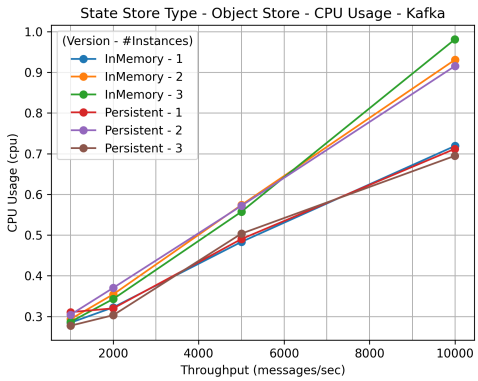

Figure 4.2: *State Store Type - Object Store - CPU Usage - Kafka*

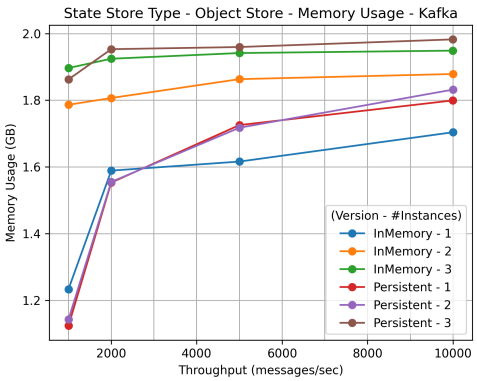

Figure 4.3: *State Store Type - Object Store - Memory Usage - Kafka*

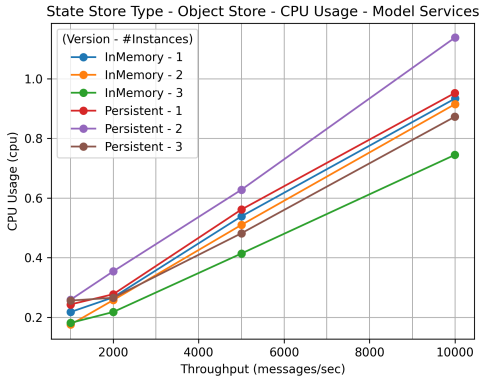

Figure 4.4: *State Store Type - Object Store - CPU Usage - Model Services*

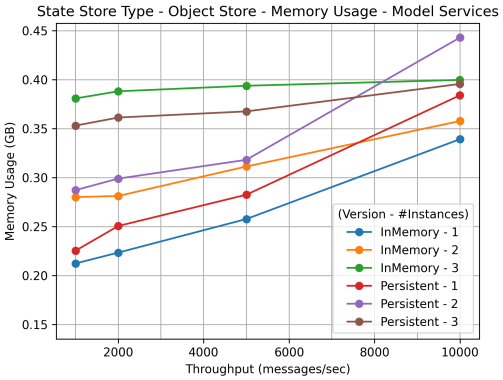

Figure 4.5: *State Store Type - Object Store - Memory Usage - Model Services*

We can see that there is no clear difference between the two state store types when it comes to CPU and memory usage of Kafka or the model services. The only difference is seen between the number of instances. The memory consumption of Kafka and the model services increases with the number of instances. And the CPU usage of the model services decreases with the number of instances as shown in chart 4.4. We also see that the CPU utilization is linearly dependent on the amount of throughput, while the memory consumption is not so much increasing due to the number of messages.

4.3.3 In-Memory State Store Performance

Now we want to see what one instance of the Object Changes Handler can achieve with the in-memory state store implementation. For this purpose, the following tests are performed, using the test results of one instance with the in-memory state store from above (cf. table 4.1).

State Store	Simulated Msg/s	Instances		
		Split	Object Changes	Frontend
In-Memory	100; 200; 300; 500; 1000	1	1	1
In-Memory	1000; 2000; 5000; 10000	1	1	1
In-Memory	20K; 25K; 30K; 35K; 40K	1	1	1

Table 4.2: *In-Memory State Store Performance - Test Configuration*

The other two services are also running with one instance each, like in the tests before. For this purpose the next chart 4.6 shows the difference between the simulated and the actual processed number of objects. And besides that the latency required by the Object Changes Handler to process the amount of objects is shown in chart 4.7.

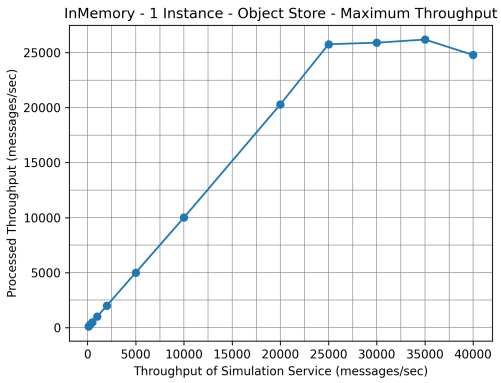

Figure 4.6: *InMemory State Store - Object Store - Maximum Throughput*

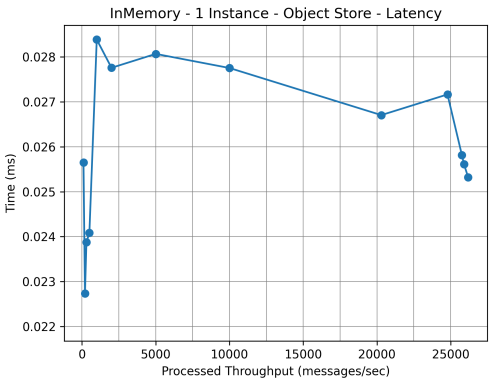

Figure 4.7: *InMemory State Store - Object Store - Latency*

As you can see the maximum number of objects that can be processed by this implementation is round about 26 thousand objects per second. And the time required for each object to be processed did not change significantly with the number of objects. Only the small numbers under 1000 objects per second have a better latency.

The following charts display the CPU and memory usage of Kafka and the model services for that scenario.

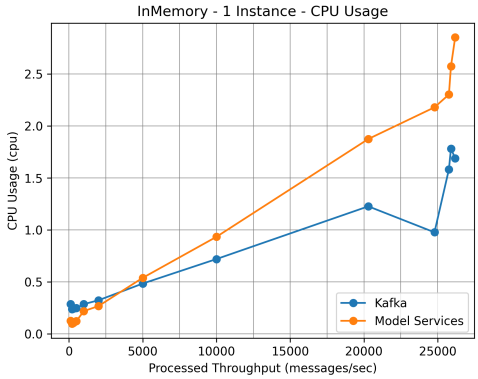

Figure 4.8: *InMemory State Store - CPU Usage*

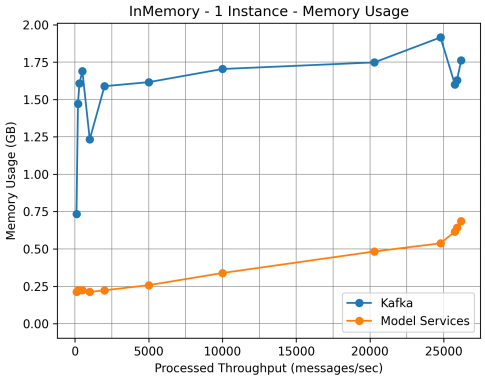

Figure 4.9: *InMemory State Store - Memory Usage*

The CPU usage of the model services and also of Kafka increase linearly with the number of messages. Only after 25 thousand messages the load of the CPU and memory of the services are rising significantly to handle this amount of incoming data. Most of the time the memory consumption of Kafka and the model services was increase linearly on a small base.

4.3.4 Split Message Service

To determine the relationship between the number of objects per device and the latency of the Split Service, the following tests were run with a different number of objects and instances:

		Instances		
Number Devices	Simulated Msg/s	Split	Object Changes	Frontend
50	1000; 2000; 5000; 10000	1	1	1
50	1000; 2000; 5000; 10000	2	1	1
50	1000; 2000; 5000; 10000	3	1	1

Table 4.3: *Split Message Service - Test Configuration*

With these tests the correlation between the number of objects per device and the latency is shown.

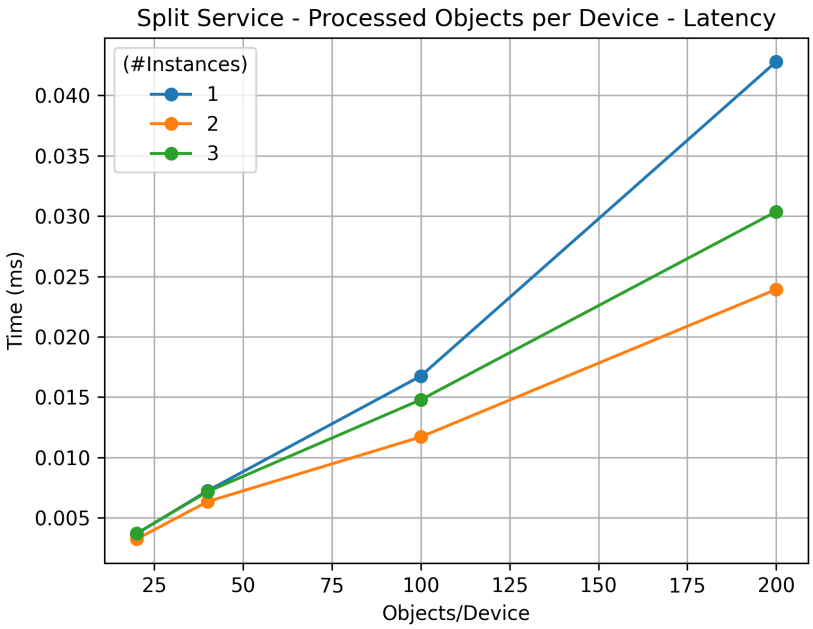

Figure 4.10: *Split Service - Latency*

Obviously the latency of the split service is not dependent on the number of instances, which means that the number of devices that are processed by an instance does not affect the latency. The latency increases only with the number of objects per device.

4.3.5 New - Old Comparison

At last we want to see the difference between the new solution and the old predecessor model implementation. In this scenario 100 up to 1000 messages are used, because the high latency of the predecessor implementation only allows lower throughput for each instance as we can see in chart 4.11. In comparison to the old model, the in-memory state store implementation is used here, because as seen above in chart 4.1 the two state store types do not have a significant difference in their latency time. In addition only one instance of the in-memory solution is used due to the reason, that the one instance of the Object Changes Handler can already perform 25 thousand messages per second. It did not have to be scaled for these numbers used in this scenario in a production environment.

Apply Changes Performance

The tests performed for the comparison are listed in this table 4.4, while the results of the in-memory state store performance test are taken from the test above (cf. table 4.2).

Version	Simulated Msg/s	Instances	
		Old Model	New Services
New - InMemory	100; 200; 300; 500; 1000	0	1
Old - MongoDB	100; 200	1	0
Old - MongoDB	100; 200	2	0
Old - MongoDB	100; 200; 300; 500	6	0
Old - MongoDB	100; 200; 300; 500; 1000	12	0

Table 4.4: *New/Old Comparison - Apply Changes Performance - Test Configuration*

For the processing of the changes per model object the time each object needs to get authorized, validated, compared and stored is tracked and the average time per throughput is shown in chart 4.11. Chart 4.12 is displayed with a logarithmic scale for the latency to illustrate the difference between the old and the new service, because the difference is too big to show it in the linear scale.

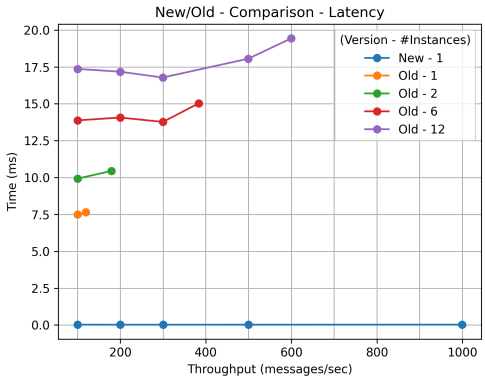

Figure 4.11: *New/Old - Comparison - Latency*

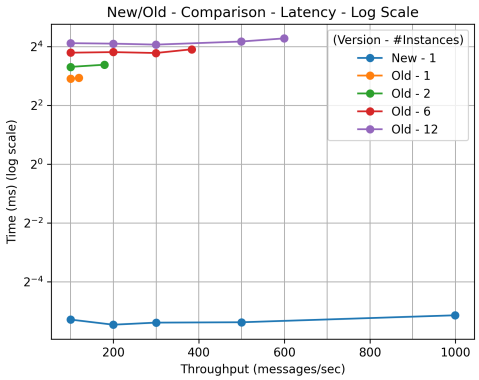

Figure 4.12: *New/Old - Comparison - Latency (Log Scale)*

We can realize that the latency of the old system is up to 400 times higher than the new system. And the latency of the old service increases up to 30% for each instance added to the environment. Chart 4.12 shows that the processing time of the new model service does not really increase with the number of messages, what has already been established in the latency comparison between the state store types (cf. 4.1) and also in the in-memory test with one instance (cf. 4.7). The latency of the predecessor implementation in 4.11 shows, that there is a slight correlation between the throughput and the latency.

The CPU and memory usage of the model services and Kafka is shown in the following charts.

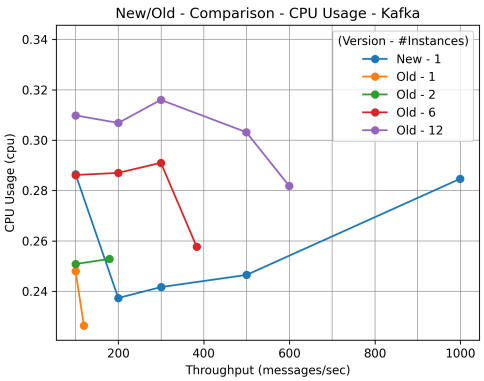

Figure 4.13: *New/Old - Comparison - CPU Usage - Kafka*

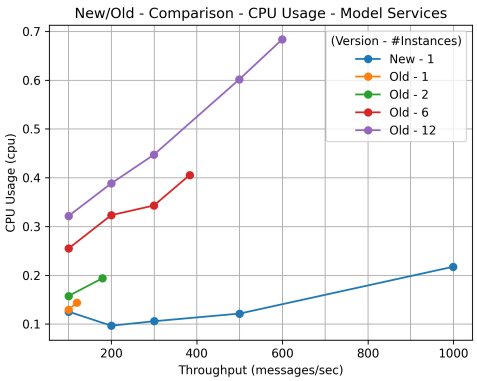

Figure 4.14: *New/Old - Comparison - CPU Usage - Model Services*

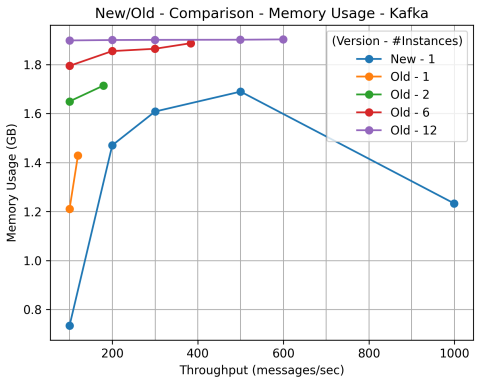

Figure 4.15: *New/Old - Comparison - Memory Usage - Kafka*

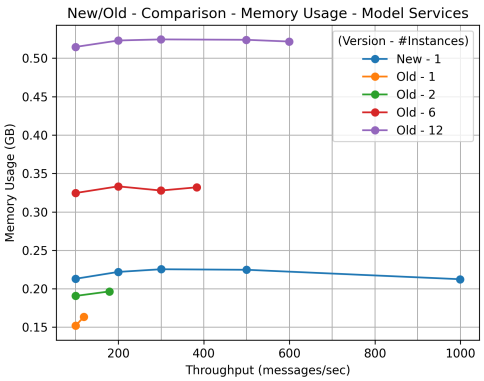

Figure 4.16: *New/Old - Comparison - Memory Usage - Model Services*

We can see that the CPU usage of the model services depends on the number of instances and the throughput, while the throughput effects the CPU more on the old system than on the new services. The memory usage of the services is mostly depending on the number of instances, but the new service has a higher consumption than the old service with more or equal than two instances. The CPU and memory usage of Kafka is also only dependent on number of instances, but we cannot see a clear difference between the new and the old system.

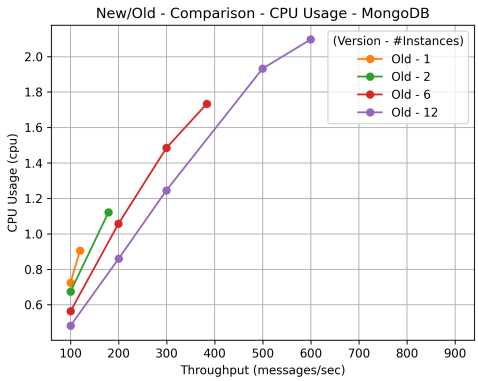

Figure 4.17: *New/Old - Comparison - CPU Usage - MongoDB*

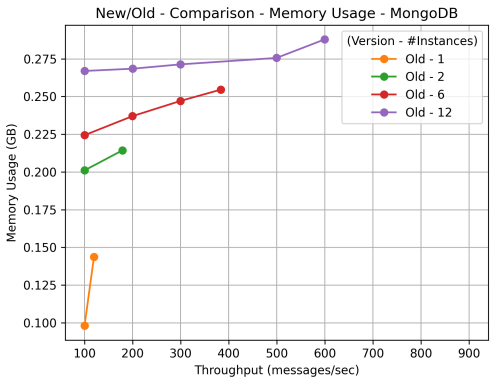

Figure 4.18: *New/Old - Comparison - Memory Usage - MongoDB*

The above charts shown the CPU and memory consumption of all *MongoDB* nodes additionally used in the tests with the predecessor implementation. The first chart 4.17 clearly shows the relationship between the increase in CPU utilization of *MongoDB* and the amount of throughput. On the contrary, it also shows that the CPU usage decreases with the number of instances. Memory consumption of *MongoDB* increases the most with the addition of model instances (cf. chart 4.18).

Frontend Request Performance

Finally, we also present the difference between the implementations when it comes to frontend REST API requests. For that all implementations are tested with 100 messages per second and with a different number of frontend instances as we can see in the following table.

		Instances			
Version	Simulated Msg/s	Old Model	Split	O. Changes	Frontend
Old - MongoDB	100	1	0	0	0
Old - MongoDB	100	2	0	0	0
Old - MongoDB	100	3	0	0	0
New - InMemory	100	0	1	1	1
New - InMemory	100	0	1	1	2
New - InMemory	100	0	1	1	3
New - Persistent	100	0	1	1	1
New - Persistent	100	0	1	1	2
New - Persistent	100	0	1	1	3

Table 4.5: *New/Old Comparison - Frontend Request Performance - Test Configuration*

Chart 4.19 illustrates the time the REST API needs to respond the "get all objects" request between the old model service and the two state store types of the new system for different numbers of frontend instances.

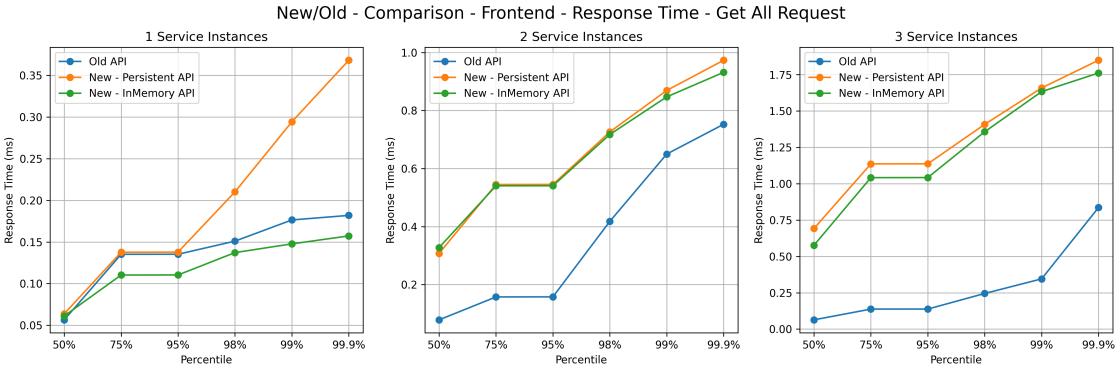

Figure 4.19: *Frontend - Get All*

We can see that the response time of the "get all" call with one instance is mostly the same in all three implementations. When it comes to more than one instance the implementation with *MongoDB* is significantly faster. The response time of the new service also increases with the number of Frontend Service instances, resulting from the current implementation, where all instances are queried one after the other (shown in listing 4.13). This can be improved with asynchronous requests, which will be discussed later in chapter 6. The chart also shows that the two state store types did have quite similar response times, only with one instance the in-memory state store is the fastest of the three implementations.

The next two charts show the response time of the "get object by ID" request, which searches for one specific object ID and returns the object. The first chart 4.20 presents the average response time of all requests. In the second chart 4.21 we can see the difference between the response times sending the request to the instance which stores the ID and the instance which has to forward the request to another instance, because it does not store the partition of the ID in the local state store (Remote Query, cf. section 3.1.3).

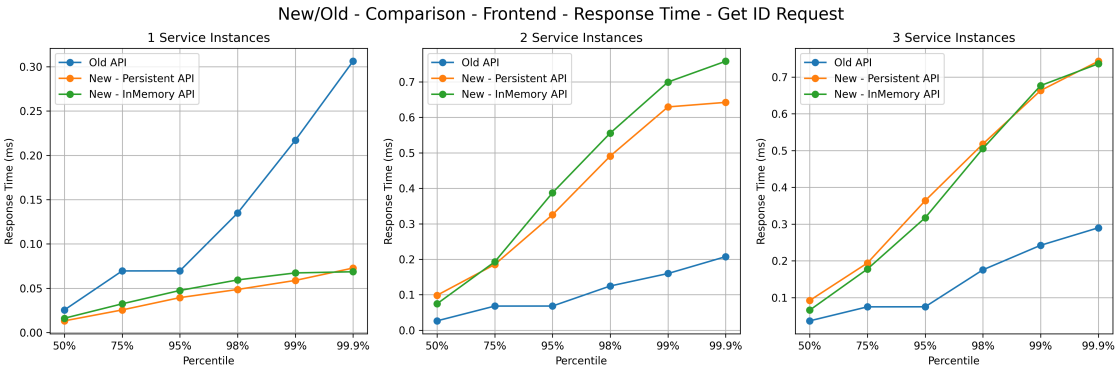

Figure 4.20: *Frontend - Get Object by ID*

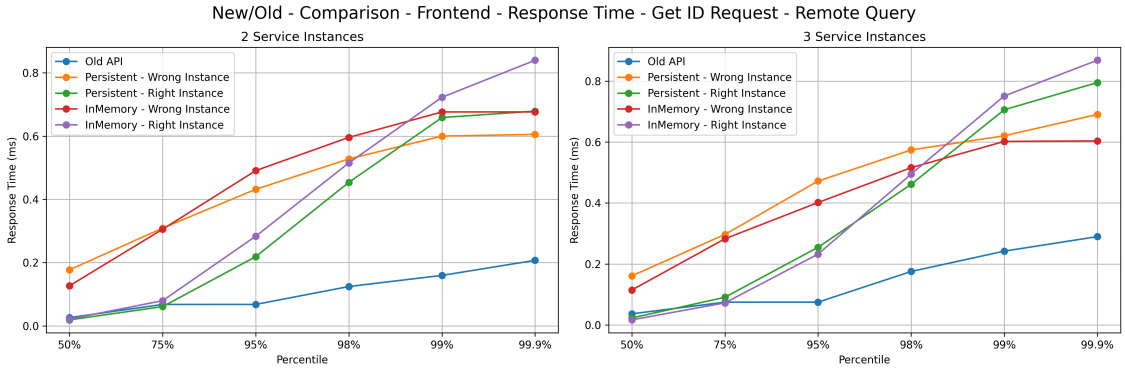

Figure 4.21: *Frontend - Get Object by ID - Remote Query - Wrong/Right Instance Comparison*

We can see that response time of the "get object by ID" request is with both state store implementations faster than the old system, when the frontend service is running with one instance. But when it comes to many instances the old system performs much better, especially when we call the wrong instance. As demonstrated in the above tests the two state store types are quite similar in response time. In chart 4.21 we see that the remote query effects the response time on the smaller percentiles, but when we come to the high percentiles, also named tail latencies, the response times are converging. This can be explained by the metadata call, which is executed in both scenarios as described before in section 3.1.3. But we can also see in that chart that the response time for the percentiles up to 75% are not worse than the old system when the query accesses the right instance.

5 Summary

The implementation of this thesis confirms that an in-memory object state store using a log-based message broker such as Kafka is possible. Kafka Streams supported us with the provided functionalities, making it relatively easy to replace a database for an object state store as shown in the implementation section 4.1. This implementation of a distributed state storage is easily scalable and also persistent, since the changelog topic is stored on disk. The topic also can be replicated with distributed replications across many brokers on different servers resulting in a very good persistence of the data.

In the implementation we also saw that we can provide the state store data via the changelog topic to various services, which then make the object data available in different materialized views, like the key-value read-only store shown in section 4.1.3.

In the test results we realized that the performance of this implementation is much better than the predecessor implementation of this service using a NoSQL database as object storage like *MongoDB*. Latency is up to 400 times lower than the old service (cf. chart 4.11) and the CPU usage is also up to 5 times lower with the new implementation. The memory consumption of the services is higher in the new system. But the sum of all CPU and memory usage is already higher with only one instance of the old system if we also add the resource utilization required by *MongoDB*, which is shown in these two charts: 4.17, 4.18. The new services do not require any additional resources, because they only use Kafka and the resources utilized by the service itself. The big difference between the latencies also comes from the fact that we need many instances of the old model service to process the same throughput that a single new service can handle. The other reason is that the latency of the new system does not increase with the amount of throughput as seen in chart 4.7, while the latency of the predecessor implementation increases slightly (shown in chart 4.11). When comparing the latencies of the new system with the different types of state stores, chart 4.1 illustrates that the new system is much more parallelizable and thus better scalable when we need a higher throughput. The throughput with one instance of the new system is also much higher and we can process up to 25 thousand messages per second, as shown in chart 4.6.

As far as response times to REST API requests are concerned, we recognized that the new Frontend Service can achieve just as fast processing times as the predecessor implementation, if only one frontend instance is running. But when it comes to many instances we have a significantly worse response time especially when we look at the tail latencies. This bad performance comes from remote queries, this means when we have to call many instances the API request from one instance to another is very time consuming. As the tests are only performed with 100 messages per second, the response times of *MongoDB* are optimized. We are not talking about real production times, since for the *MongoDB* cluster it is easy to hold the 100 objects in the RAM. In a production environment, however, objects that are older and no longer stored in RAM have to be queried from the disk. And when *MongoDB* is additional a sharded resource and not only used with one replica set the *mongos* instance has to route the queries to the shards, which

also adds a amount of time to the request latency (MongoDB 2023). Chart 4.21 shows the difference between a *getID* request to the instance which stores the ID on the one hand and an instance, which has to forward the request to an other instance on the other hand. We noted that the most requests are much slower when the Frontend Service instance has to forward the request to an other instance. At the same time we also saw that the tail latencies are mostly the same, which may be due to the fact that the metadata call takes its time. This poor performance of the REST API with many instance can be improved, for which suggestions are made in chapter 6.

In the tests, it is also showed that the only difference between the two types of state stores is that the latency of the in-memory store is lower than the one of the persistent store, while CPU and memory usage are nearly the same. This results from the fact, that the persistent state store also keeps all objects in memory, when the memory has enough free space, and only additionally persists the data asynchronously to disk. This means that the latency increases if the memory is too small, but in this scenario the in-memory state store would not work correctly. Because of the fact, that we do not expect such large objects, that the size of the memory is the relevant factor, and with the reason that the only other advantage of the persistent state store is, that the data can be restored faster, the in-memory state store is the right type for this scenario. With the in-memory state store we can achieve higher throughput and the service can also be deployed as stateless *Kubernetes* resource, which simplifies the deployment of the service.

6 Future Work

In the REST API response time tests shown in section 4.3.5, it was fairly clear that the performance of the requests for remote queries across many instances can be improved. The implementation of the *getAll* request can be optimized by making the API calls to other instances asynchronously. Another performance issue are the REST calls themselves, these text protocols use all technologies for the web, from JSON as representation format down to HTTP and the entire networking stack (Goetsch 2017). Replacing REST with binary protocols, as it is also done at *MongoDB* with the so named *Wire protocol* (MongoDB 2023), will reduce the response time of remote queries. To prevent the converging of the tail latencies in the *getId* remote query comparison (shown in chart 4.21), it could be examined whether the metadata call is a deciding point. For that the local state store can first be queried for the ID and if it does not return an object, the metadata call will be executed.

To perform more complex queries on the actual state, a new service can be implemented, which also consumes the changelog topic but then runs an indexing algorithm over the data. So there would be another materialized view of the data, which would allow queries like filtering or searching for a specific value within an object.

The other solution for more complex queries is to investigate other external databases, which can consume the changelog topic. For this solution, the database must be able to handle the number of data updates. Otherwise, with an upstream *KTable*, the database would always consume only the latest values, no matter how fast it can process them.

The impact of the threading model explained in section 2.7.5 can also be examined, by increasing the number of threads per instance in contrast to adding instances of the model services. Increasing the number of threads should have the same effect as adding new instances, because chart 4.1 already shows that the number of instances does not have a big influence on the latency, unlike the old system shown in chart 4.11.

List of Figures

List of Listings

List of Tables

Bibliography

Alaasam, A. B. A., G. Radchenko, and A. Tchernykh (2019). "Stateful Stream Processing for Digital Twins: Microservice-Based Kafka Stream DSL". In: *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*. DOI: 10.1109/SIBIRCON48586.2019.8958367.

Alaasam, A. B., G. Radchenko, A. Tchernykh, and J. González Compeán (2020). "Analytic study of containerizing stateful stream processing as microservice to support digital twins in fog computing". In: *Programming and Computer Software*.

Chauhan, A. (2019). "A review on various aspects of MongoDb databases". In: *Int. J. Eng. Res. Sci. Technol* 5.

Confluent (2023). *Confluent Kafka Streams Documentation*. URL: https://docs.confluent.io/platform/current/streams.

Al-Dhuraibi, Y., F. Paraiso, N. Djarallah, and P. Merle (2018). "Elasticity in Cloud Computing: State of the Art and Research Challenges". In: *IEEE Transactions on Services Computing* 2. DOI: 10.1109/TSC.2017.2711009.

Dobbelaere, P. and K. S. Esmaili (2017). "Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*. DEBS '17. Barcelona, Spain: Association for Computing Machinery. ISBN: 9781450350655. DOI: 10.1145/3093742.3093908. URL: https://doi.org/10.1145/3093742.3093908.

Dropwizard (2023). *Dropwiard Metrics Documentation*. URL: https://metrics.dropwizard.io/4.2.0/manual.

Fang, R., H.-I. Hsiao, B. He, C. Mohan, and Y. Wang (2011). "High performance database logging using storage class memory". In: *2011 IEEE 27th International Conference on Data Engineering*. DOI: 10.1109/ICDE.2011.5767918.

Fu, G., Y. Zhang, and G. Yu (2021). "A Fair Comparison of Message Queuing Systems". In: *IEEE Access*. DOI: 10.1109/ACCESS.2020.3046503.

Furda, A., C. Fidge, O. Zimmermann, W. Kelly, and A. Barros (2018). "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". In: *IEEE Software* 3. DOI: 10.1109/MS.2017.440134612.

Goetsch, K. (Nov. 2017). *Are Binary Frameworks an Alternative to REST?* URL: https://www.linkedin.com/pulse/binary-frameworks-alternative-rest-kelly-goetsch.

Golder, R. (July 2022). *Kafka Streams: State Store*. URL: https://medium.com/lydtech-consulting/kafka-streams-state-store-30110bf4f24.

Gracioli, G., M. Dunne, and S. Fischmeister (Apr. 2018). "A Comparison of Data Streaming Frameworks for Anomaly Detection in Embedded Systems". In: *1st International Workshop on Security and Privacy for the Internet-of-Things (IoTSec)*.

Hapner, M., R. Burridge, R. Sharma, J. Fialli, and K. Stout (2002). "Java message service". In: *Sun Microsystems Inc., Santa Clara, CA*.

Intorruk, S. and T. Numnonda (2019). "A Comparative Study on Performance and Resource Utilization of Real-time Distributed Messaging Systems for Big Data". In: *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. DOI: 10.1109/SNPD.2019.8935759.

Kafka, A. (2023). *Apache Kafka Version 3.4 Documentation*. URL: https://kafka.apache.org/34/documentation/streams/developer-guide.

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. Ed. by A. Spencer and M. Beaugureau. O'Reilly Media.

Kreps, J. (Sept. 2017). *It's Okay To Store Data In Kafka*. URL: https://www.confluent.io/blog/okay-store-data-apache-kafka/.

Kreps, J., N. Narkhede, J. Rao, et al. (2011). "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. 2011. Athens, Greece.

Kubernetes (2023). *Kubernetes Documentation*. URL: https://kubernetes.io/docs/.

Milosavljević, M., M. Matić, N. Jović, and M. Antić (2021). "Comparison of Message Queue Technologies for Highly Available Microservices in IoT". In: *IcETRAN*.

MongoDB (2023). *MongoDB Documentation*. URL: https://www.mongodb.com/docs/manual/.

Narkhede, N., G. Shapira, and T. Palino (2017). *Kafka: The Definitive Guide*. Ed. by S. Cutt. O'Reilly Media.

OASIS (2012). *Oasis advanced message queuing protocol (amqp) version 1.0*. https://www.oasis-open.org/standard/amqp/.

Pahl, C. and P. Jamshidi (2016). "Microservices: A Systematic Mapping Study." In: *CLOSER (1)*.

Rooney, S., P. Urbanetz, C. Giblin, D. Bauer, F. Froese, et al. (2019). "Kafka: the Database Inverted, but Not Garbled or Compromised". In: *2019 IEEE International Conference on Big Data (Big Data)*. DOI: 10.1109/BigData47090.2019.9005583.

Schmid, R. (Nov. 2018). *Queryable Kafka Topics with Kafka Streams*. URL: https://medium.com/bakdata/queryable-kafka-topics-with-kafka-streams-8d2cca9de33f.

Seymour, M. (2021). *Mastering Kafka Streams and ksqlDB*. Ed. by A. Spencer and M. Beaugureau. O'Reilly Media.

T, S. and S. N. K (2019). "A study on Modern Messaging Systems- Kafka, RabbitMQ and NATS Streaming". In: *CoRR*. arXiv: 1912.03715.

Vanlightly, J. (May 2018). *Event-Driven Architectures - The Queue vs The Log*. URL: https://jack-vanlightly.com/blog/2018/5/20/event-driven-architectures-the-queue-vs-the-log.

Vineet, J. and L. Xia (2017). *A Survey of Distributed Message Broker Queues*. arXiv: 1704.00411 [cs.DC].

Zelenin, A. and A. Kropp (2022). *Apache Kafka: von den Grundlagen bis zum Produktiveinsatz*. Ed. by S. Gottmann. Carl Hanser Verlag München.

Glossary

API Application Programming Interface.

CDC Change Data Capture.
CPU Central Processing Unit.

DSL Domain Specific Language.
DT Digital Twin.

ID identifier.
IoT Internet of Things.

JSON JavaScript Object Notation.
JVM Java Virtual Machine.

RAM Random Access Memory.
REST Representational State Transfer.

VM Virtual Machine.

Web-UI web based user interface.