Chair of Information Technology

Master's Thesis

A DSMS approach to support surveillance data based services in U-space

Daniel Pfisterer, BSc

February 2024

MONTANUNIVERSITÄT LEOBEN
www.unileoben.ac.at

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich diese Arbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, den Einsatz von generativen Methoden und Modellen der künstlichen Intelligenz vollständig und wahrheitsgetreu ausgewiesen habe, und mich auch sonst keiner unerlaubten Hilfsmittel bedient habe.

Ich erkläre, dass ich den Satzungsteil „Gute wissenschaftliche Praxis“ der Montanuniversität Leoben gelesen, verstanden und befolgt habe.

Weiters erkläre ich, dass die elektronische und gedruckte Version der eingereichten wissenschaftlichen Abschlussarbeit formal und inhaltlich identisch sind.

Datum 12.02.2024

[illegible]

Unterschrift Verfasser/in
Daniel Pfisterer

Danksagung

Diese Arbeit möchte ich meiner großartigen Familie widmen, die mich immer tatkräftig unterstützt hat. Es war nicht immer einfach im Studium, doch es hilft enorm, wenn man stets jemanden hinter sich hat, der einem den nötigen Rückhalt bietet, sei es finanziell oder emotional. Auch wenn man es manchmal für selbstverständlich nimmt, das ist es keinesfalls. Liebe Familie, ich weiß eure Unterstützung sehr zu schätzen und bin euch unheimlich dankbar für die Möglichkeiten, die ihr mir gegeben habt. Ohne euch hätte ich das nie geschafft.

Mein Dank gilt auch meinem Betreuer Herrn Dr. Ronald Ortner für all die Unterstützung während meiner Arbeit. Insbesondere weiß ich die Flexibilität zu schätzen, die es mir erst ermöglich hat meine Arbeit in Spanien zu schreiben. Lieber Herr Ortner, vielen herzlichen Dank für die umfangreiche Betreuung meiner Arbeit, die trotz der geografischen Distanz reibungslos funktioniert hat.

An dieser Stelle möchte ich mich auch noch bei Herrn Dr. Juan José Ramos Gonzalez bedanken, der mir das Thema meiner Arbeit angeboten und es mir ermöglicht hat fünf großartige Monate in Spanien zu verbringen. Lieber Juanjo, vielen Dank für deine Unterstützung und dein Vertrauen in mich, dieses Thema zu bearbeiten.

Abstract

The market for *UAS (Unmanned Aerial System)* holds a lot of potential for growth in the near future, both in industrial and consumer applications. With an increasing rate of adoption, standardized regulations and technical developments are crucial to enable safe UAS operations. One initiative that blends both is the definition of a *U-space* that provides digital services for a safe access to airspace. In order to provide these services in real-time, with 1Hz as reference value, the implementation of a prototype of a *DSMS (Data Stream Management System)* is proposed.
The developed DSMS is specified based on a combination of regulatory and problem-specific requirements. For defining the technical specifications the system is divided into three parts (ingestion, processing, and storing). According to the specifications a DSMS should be fault-tolerant and scalable while providing tools for stateful computations and SQL-compatibility. These specifications serve as basis for selecting the frameworks for the implementation of the prototype. For the data ingestion *Apache Kafka* is utilized, the stream processing is done with *Kafka Streams*, and all data is stored in a *PostgreSQL* database. Based on these frameworks the architecture of the whole system is designed. As Kafka is responsible for most of the data handling, designing the data flow within, from and to Kafka proved to be crucial for a successful implementation. To simplify testing and deployment of the prototype, all frameworks are implemented as containerized applications using *Docker*. Yet, even as containerized applications, applying these frameworks is not trivial. A smooth data exchange between the different components, is only possible with consistent schema definitions. Coordinating the partitioning logic from Kafka with the stream processors for scalability requires careful adjustment of all parameters. Overcoming these challenges demands a deep understanding of the whole system with all its components and the interactions between them.
The finished DSMS prototype implements a selection of U-space services which are tested with a customized simulator. Although results have to be viewed with care, as all tests are conducted in a controlled environment, the results demonstrate the feasibility of using a DSMS to provide U-space services.

Kurzfassung

Das Wachstumspotential für den Markt für UAS (Unmanned Aerial System) ist groß, sowohl für industrielle als auch für Verbraucheranwendungen. Auf Grund der steigenden Adaptionsrate braucht es standardisierte Regulierungen und technische Entwicklungen im Bereich der Sicherheit. Mit der Definition eines U-Space zur Bereitstellung digitaler Services für einen sicheren Zugang zum Luftraum werden beide Anforderungen abgedeckt. Um diese Dienste in Echtzeit bereitstellen zu können (mit 1Hz als Richtwert) wird die Entwicklung eines *DSMS-Prototypen (Data Stream Management System)* vorgeschlagen. Eine Kombination aus regulatorischen und problemspezifischen Anforderungen stellt die Basis für die Spezifikation des entwickelten DSMS dar. Zur Definition der technischen Spezifikationen ist das System in drei Teile unterteilt: Datenverwaltung, Datenverarbeitung, und Datenspeicher. Dabei gelten Fault-tolerance und Skalierbarkeit als besonders wichtig für ein DSMS. Weiters muss eine Datenverarbeitung mit State möglich und SQL-Kompatibilität vorhanden sein. Diese Spezifikationen dienen als Grundlage für die Wahl der Frameworks, die zur Implementierung des Prototypen verwendet werden. Zur Datenverwaltung wird *Apache Kafka* verwendet; *Kafka Streams* erledigt die Datenverarbeitung, bevor alle Daten in einer *PostgreSQL* Datenbank gespeichert werden. Die Architektur des Systems basiert auf diesen Frameworks, wobei besonders die Rolle von Kafka hervorzuheben ist. Da Kafka für die Datenverwaltung verantwortlich ist, ist die Gestaltung des Datenflusses innerhalb, von, und zu Kafka ein entscheidender Erfolgsfaktor. Zur Vereinfachung des Entwicklungsprozesses sind alle Frameworks mit *Docker* als Container-Anwendungen implementiert. Doch selbst dann ist die Implementierung dieser Frameworks nicht einfach. Ein reibungsloser Datenaustausch zwischen allen Komponenten ist nur mit konsistent definierten Schemata möglich. Um Kafka Streams skalierbar zu machen, muss es mit der Partitionierungslogik von Kafka durch Parameterkonfigurationen abgestimmt sein. Die Implementierung verlangt ein tiefgreifendes Verständnis des ganzen Systems mit allen Komponenten und deren Interaktionen.
Der fertige DSMS-Prototyp implementiert eine Auswahl an U-Space Services, welche mit einem Simulator getestet werden. Aufgrund der kontrollierten Testumgebung sind die Ergebnisse mit Vorsicht zu betrachten. Dennoch zeigen die Resultate, dass ein DSMS verwendet werden kann, um U-Space Services bereitzustellen.

Contents

List of Figures

List of Tables

Acronyms

- *ACID* - Atomicity, Consistency, Isolation, and Durability
- *API* - Application Programming Interface
- *BASE* - Basically Available, Soft State, and Eventual Consistency
- *CAP* - Consistency, Availability, and Partition-resilience
- *CDC* - Change Data Capture
- *DAG* - Directed Acyclic Graph
- *DSL* - Domain Specific Language
- *DSMS* - Data Stream Management System
- *EU* - European Union
- *FP* - Flight Plan
- *GUI* - Graphical User Interface
- *HDD* - Hard Disk Drive
- *IDE* - Integrated Development Environment
- *JDBC* - Java Database Connectivity
- *JMX* - Java Management Extension
- *RAM* - Random Access Memory
- *RF* - Replication Factor
- *SSD* - Solid State Drive
- *UAM* - Urban Air Mobility
- *UAS* - Unmanned Aerial System
- *UAV* - Unmanned Aerial Vehicle

Terminology

- *Producer* - Any device or system that can create a data stream is referred to as producer (e.g. a sensor on a UAS).
- *Consumer* - Any system or sub-system that consumes a data stream to produce some output is referred to as consumer (e.g. a DSMS or one of its sub-systems).
- *Operation* - Any number of tasks carried out by a single UAS between landings is referred to as one operation.
- *U-space (services)* - Digital procedures supporting safe, efficient and secure access to airspace so UAS can safely execute their operations are referred to as U-space services or U-space.
- *Message* - Any data sent from a producer to a consumer is referred to as message.
- *Event* - The content of a message that describes something unique and immutable that happened at some point in time is referred to as event.
- *Data stream* - A continuous, real time series of messages that is order based on a timestamp and cannot be saved to storage entirely is referred to as data stream. Unless otherwise stated, the terms *message stream* and *event stream* are used synonymously with data stream.
- *Batch data* - A massive collection of messages that is available in its entirety right from the beginning is referred to as batch data.
- *DSMS* - A system that can handle data streams and deliver results in real time is referred to as DSMS. Unless otherwise stated, the term *system* is used synonymously with DSMS.
- *Log* - An append-only sequence of messages ordered by a unique, sequential identification number is referred to as log.
- *Prototype* - A DSMS in development that supports selected U-space services (e.g. legal recordings) is referred to as prototype.
- *Ground-based control system* - The superordinate system of the DSMS that coordinates all UAS activities in U-space is referred to as ground-based control system.

1. Introduction

In this chapter, an argument for the relevance of the topic is given before the problem definition and the subsequent research question are presented. Finally, a short overview of the thesis' structure is provided.

1.1. Relevance

UASs (Unmanned Aerial System) or UAVs (Unmanned Aerial Vehicle), commonly referred to as drones, are mostly known for their consumer applications (e.g. taking a picture from above). However, they are also applied in various industries. The top three industries are energy, construction and agriculture, where UAS are utilized to do mappings and inspections[1]. This leads to a global UAS market with an estimated worth of $30.6 billion in 2022; yet there is potential for growth. Until 2030 a steady compound annual growth rate of approximately 7.8% is predicted, resulting in a global market worth about $55.8 billion. Taking a closer look at the European market, an increase from $6.8 billion in 2022 to $13.2 billion in 2030 is expected[2]. The numbers for Europe are in-line with expectations from the *European Commission*, who estimate a market size of at least €10 billion in 2035, with the service industry as the primary driver for growth. One example of potential growth is the increase of UAS-deliveries of small and premium goods (e.g. pharmaceuticals) in the e-commerce sector[3]. This is not unrealistic considering that that drones can be cost-competitive compared to traditional ground-bound delivery systems, albeit only for light-weight packages and short distances[4]. There is even a real world example in Zurich, Switzerland. Here, the company *Matternet* has set up a drone delivery route for transporting samples between hospitals and laboratories[5]. Yet examples like this are rare in densely populated areas.
For an increased rate of adoption technological developments and new regulations play a crucial role. Because of the aforementioned market potential, initiatives for new developments are taken. One such initiative is the definition of a common *U-space service*.

[1] *Find the Top Drone Application |Drone Industry Insights 2022* 2022.
[2] *Industry Leading Drone Market Analysis 2022-2030 | Droneii* 2022.
[3] SESAR Joint Undertaking 2017a.
[4] **GuestEditorialCan2014**.
[5] *Matternet Launches World's Longest Urban Drone Delivery Route Connecting Hospitals and Laboratories in Zurich, Switzerland* 2022.

U-space can be defined as a set of highly digitalized services and procedures with the aim of providing safe, efficient and secure access to airspace for all types of missions, users and drones. It is planned to roll-out *U-space* in four stages with increasing levels of automation. This is done by adding more and more services until the highest level of automation (autonomously operating UAS) is reached[6]. Services provided by *U-space* considered in this thesis include[7]:

- A digital logbook for creating user reports based on the legal recording information.
- Creating legal recordings of the system's state at any moment, allowing to investigate incidents. Additionally, legal recordings could serve as a source of information for future research and training.
- A network identification service for tracking and reporting all flight movements in real-time. It is essential to know where the UAS is at any moment in time.
- A tactical conflict resolution that checks for potential incidents in real-time and provides instructions (e.g. speed reduction, change of direction) to prevent potential incidents.

To reach a higher level of automation by adding the above mentioned services, extensive amounts of data must be exchanged (between the UAS and a ground-based control system) and processed in real-time continuously. This means that the data is not available all at once but rather as a continuous data stream. Here, a data stream can be defined as a real-time, continuous series of items that are ordered by either their inherent timestamp or their time of arrival. It is not possible to alter the order of the items' arrival, nor is it practical to assume that the entire data stream can be stored locally[8]. A system capable of handling multiple data streams that produces answers continuously and promptly is referred to as *DSMS (Data Stream Management System)*[9]. The main aim of this thesis is to develop and test a prototype for such a DSMS to support *U-space services*.
Various frameworks are available to help with the implementation of a DSMS. For instance, *Apache Kafka* is a highly scalable and fault-tolerant open-source framework for real-time data ingestion[10]. *Apache Spark* and *Apache Flink* are open-source frameworks used for processing data streams in real time to do some analysis or apply machine learning techniques[11][12]. These are just some of the most common frameworks utilized by

[6]SESAR Joint Undertaking 2017b.
[7]CORUS XUAM Consortium 2022.
[8]Golab and Özsu 2003.
[9]Motwani et al. 2002.
[10]*Apache Kafka* 2023.
[11]*Apache Spark - Unified Engine for Large-Scale Data Analytics* 2023.
[12]*Apache Flink — Stateful Computations over Data Streams* 2023.

many leading technology companies (e.g., *Apache Kafka* is used by *PayPal*, *Netflix*, *Oracle*[13]). Of course, just because large technology companies use these frameworks does not necessarily mean that they are applicable in any scenario. After all, none of these frameworks are without their downsides. Depending on the use scenario an alternative framework might yield better results. The large amount of available frameworks provides a lot of opportunities, while making it difficult to select the right ones based on criteria like manageability, support for different programming languages[14], fault-tolerance[15] or latency and throughput[16]. These criteria strongly correlate with the *U-space* system requirements, which form the basis for the selection of frameworks in this thesis.

1.2. Research question

The main goal of this thesis is to develop and test a prototype for a DSMS that provides U-space services like legal recording, tactical conflict resolution etc. These services should be provided in real-time with 1Hz as reference value. The DSMS should be built using open-source frameworks where the selection of these frameworks is based on the system requirements of U-space (e.g. fault-tolerance and throughput). In addition to the DSMS, a storage system for long-term storage of recordings must be designed. All decisions behind design and implementation choices must be documented. Interfaces from the existing system must be considered to provide a compatible prototype. Also, the prototype must be in line with all current regulations and standards. In the end, a test should show the limits of the implemented prototype (e.g. number recordings processed per second), where the results are validated through the use of a customized simulator.

1.3. Structure

Chapter 2 of the thesis focuses on defining system specifications and proposing a general direction of the components needed to build the system. Specifications are defined based on regulations, problem-specific requirements and general challenges for developing software. In order to understand what a DSMS looks like at a high level of abstraction, fundamental principles from stream processing are presented and their relevance for this thesis is discussed. Finally, all system specifications and design choices are summarized.
Chapter 3 presents the frameworks that are selected based on the system specifications defined in Chapter 2. We discuss our choices for a data distribution framework, a database technology and a stream processing framework. Moreover, we give brief explanations on

[13]*Apache Kafka Powered By* 2023.
[14]Isah et al. 2019.
[15]Van Dongen and Van den Poel 2021.
[16]Nasiri et al. 2019.

why these frameworks meet our requirements.
Chapter 4 presents the general architecture of our system at a high level of abstraction. In order to understand how the prototype can be set up for execution, an overview of its deployment is provided. Finally, the dataflow for the stream processing is presented, before a design for a database is proposed.
Chapter 5 describes the machine the prototype is executed on, explains the most important settings of Kafka and all the producers, topics, and stream processors that are created. Moreover, a test procedure is proposed before the results are elaborated. Finally, the results and the limitations of the prototype are discussed.
In the last part, the thesis is concluded and an outlook for further improvements and research is provided.

2. System specifications

In this chapter, regulatory requirements from U-space as well as general specifications that affect many software systems are introduced. This should help to better understand the requirements the DSMS has to meet. Furthermore, streaming data architectures for a DSMS at a high level of abstraction are presented, before a more detailed look on a potential architecture for IoT devices is offered. Based on these architectures the main parts of the DSMS, their functions, and potential difficulties are explained. To be able to follow the reasoning of subsequent decisions, essential concepts from stream processing are outlined. Finally, all specifications are summarized in a table and the decisions made are documented.

2.1. Regulations and general specifications

First, this section explores regulatory requirements from the *European Commission* for U-space. Second, problem-specific requirements based on the regulations and the existing system are discussed. Finally, an overview of general specifications of software systems is presented and their relevance for this thesis explained.

2.1.1. Regulatory requirements

The *European Commission*[17] has defined a set of regulations for U-space. The target for these regulations are UAS operating beyond the visual line of sight because they are considered a risk to safety, security, privacy and the environment. Not included are, e.g. UAS with a take-off mass of less than $250g$ that operate within the visual line of sight. The regulations cover a wide range of topics such as flight authorization, application for a U-space certificate and capabilities of competent authorities. Regulations considered in this thesis include:

- Existence of a network identification service, which allows authorized users to receive messages containing, among other things, geographical position, height above surface and time of creation of the message. The update frequency for presenting this information to the users is defined by the competent authority. Authorized users are, among other things, the general public and other U-space service providers.

[17]European Commission 2021.

- Existence of a conformance monitoring service, which checks if the UAS operates in conformance with other regulations. Any deviations should be reported immediately to the UAS operator.
- Existence of U-space service providers, who have to keep a storage of all records that allows to retrace all their activities. The exchange of information has to meet the data quality, latency and safety requirements. Thereby, all metadata must be kept, data quality maintained and verification and validation should ensure data is not corrupted.

2.1.2. Problem-specific requirements

Based on the regulatory requirements and the existing system it is possible to divide problem-specific requirements into two categories. The first category refers to the processing tasks that must be executed, the second category concerns the storage requirements and how historical data is used.

Processing tasks

The prototype must be able to execute the following tasks (or at least provide means to implement them at a later point in time):

- C_1 *Enrich telemetry data*
  Each telemetry message must contain information referring to the altitude of the UAS. The task of C_1 is to update this altitude information for each message so it is in line with the required standard.
- C_2 *Flight plan correlation*
  Each UAS operation must be correlated with a flight plan (FP), otherwise it is categorized as an unknown UAS operation. The task of C_2 involves checking for each telemetry message if it can be correlated with a FP or not. Moreover, we have to check for a loss of signal. If we do not receive telemetry messages from a UAS for a while we must emit a warning and mark the telemetry messages accordingly as soon as the signal is re-established.
- C_3 *Conformance monitoring*
  For all messages that can be correlated with a FP it is necessary to verify that they comply with this FP at any point in time. Moreover, any kinds of geofences (geometry that restricts the airspace e.g. at an airport) must not be violated. This is important in e.g. emergency cases like police surveillance where temporarily set up geofences can lead to conflicts with existing FPs. The task of C_3 is to monitor

the conformance with FPs and geofences and to alert the system in case there is a violation.

- C_4 *Conflict detection*
  Collisions of UAS must be avoided at all costs, therefore it is essential to predict potential collisions based on the current trajectories of UAS. Here all telemetry messages are taken into consideration, even the ones from unknown UAS. The task of C_4 is to monitor for potential collisions and alert the system in time, so precautions can be taken.

- C_5 *Spatial clustering*
  With relation to C_4, it is not reasonable to check for potential collisions of UAS in zones that are far apart from a geographical point of view. The task of C_5 is to divide the incoming telemetry messages into spatial clusters in order to simplify C_4.

Storage requirements

The following list includes a few use cases describing the requirements for the database technology; it is by no means an exhaustive list:

- For reporting and visualization, data on the whole operations must be retrieved. This includes all telemetry messages, any correlated FPs or any alarms sent by the system. It should be possible to filter data based on e.g. geographical location, time range, pilot operator.

- Analysis of historical data is done only sporadically. Typical analysis relates to latency, flight performance and alert KPIs. Similar to reporting and visualization, it should be possible to filter the data based on time range, location etc.

- It is essential to keep a log of all historical data to the extent and for the timespan defined by current regulations. Easy retrieval of historical data is less important than compression in order to minimize storage requirements.

2.1.3. General specifications

Designing software depends on many system-specific factors such as the above mentioned regulations. However, there are three factors that are not just relevant for this software prototype, but for many other software systems as well[18]:

- *Reliability*
  Reliability of a software system is about making sure that a systems continues to

[18]Kleppmann 2017, p. 3 ff.

work correctly even in the event of a fault. Thereby it is important to distinguish between a fault and a failure. A system has a failure if it stops providing the user the service defined in the system's specification. A fault is a failure of one of the system's components, where the system as a whole still provides the desired service. A system that is able to continue to provide the desired service in the event of a fault is called *fault tolerant*[19]. Faults can happen due to hardware, software or human errors. In our use case, an example for a hardware error is the failure of a sensor on one of the UAS. Even if one sensor fails, the DSMS should be able to keep processing incoming data from other UAS sensors. Software errors are harder to detect, as bugs may lie undetected for quit some time. For instance, due to a software error it could happen that identical sensor recordings from an UAS are written to a database multiple times, resulting in a congested database. Examples for human errors include any mistakes made by the UAS operator such as entering the wrong update frequency for messages. Ultimately, we want a fault tolerant DSMS that can deal with any of the mentioned faults, as it is impossible to guarantee that no fault will ever happen. (How fault tolerance is achieved in a DSMS will be explained in Section 2.3.3).

- *Scalability*
  Scalability is defined as the ability of a software system to deal with future growth (i.e. increased data volume or complexity) by adding more computational resources. Thereby the first step is to define the current load of the system with so-called *load parameters*. In our case, the expected number of UAS multiplied by the expected frequency of messages is a viable choice. Next, the load parameters can be used to investigate what happens to the DSMS if the load increases. Of course, this cannot be done without performance metrics. Typical performance metrics are *throughput* or *response time*. Throughput is defined as the number of transactions that can be computed within a given amount of time. It strongly depends on the complexity of the transactions. Response time is frequently considered to be identical to *latency*, but it is not. Whereas response time describes the amount of time it takes to process a request from outside, latency is the least amount of time required to receive any response, even when no work needs to be done[20].

  As stated in Chapter 1, $1Hz$ is the reference value for providing U-space services, as we want messages from UAS (requests from outside) to be processed by the DSMS within one second. Consequently, it is reasonable to use the response time as the main performance metric. Unfortunately, response time and throughput do have a positive correlation. If the number of active UAS (throughput) is increased,

[19]Walter Heimerdinger, Charles Weinstock 1992.
[20]Fowler 2002, p. 16 f.

response time increases too. Further tests have to show how many UAS can operate at the same time while keeping the response time at $1Hz$.

Other scalability issues concern the way scaling is done. Computational resources can be added through *vertical scaling* (switch from the current machine to a more powerful one) or *horizontal scaling* (split the load among multiple smaller systems). Furthermore, the scaling could be done either *elastically* (system automatically takes the resources when the load increases) or *manually* (a human checks the load and scales the system accordingly).

- *Maintainability*
  During the life cycle of a software system, many different people are going to work on it in one way or another (e.g. add new functionality, fix bugs). Maintainability is about ensuring that any work on the system can be carried out without losing time while dealing with system peculiarities. Three design principles help to make this possible.

  A system with good *operability* simplifies routine tasks such as monitoring and documentation to support all engineers working on the system. In general, some form of monitoring (for response time, data volume, etc.) should be implemented right from the start. It is not recommendable to wait for deployment, as mistakes would potentially remain undetected for too long[21].

  Simplicity is about avoiding *complexity* wherever possible. Thereby, a distinction is made between two types of complexity. *Essential complexity* is inherent in the problem solved by the software, whereas *accidental complexity* is due to flaws in the implementation of the software[22]. Accidental complexity can be avoided through *abstractions* that hide implementation details to make the concepts easier to understand. This will be important when selecting a streaming data architecture in Section 2.2. Another aspect of simplicity is deciding between a *monolithic* and a *modular* approach for building the system. A monolithic system integrates a lot of functionality into one system as opposed to a modular system where each component is specialized to do one task only. Consequently, a monolithic system is easier to deploy at the beginning, yet more difficult to adapt if new technologies emerge[23]. This is why this thesis will follow the modular approach, while trying to keep the system as simple as possible.

  Evolvability is concerned with keeping the system changeable, as requirements will (most likely) change. Moreover, it is closely linked to complexity. As systems grow,

[21]Reis and Housley 2022, p. 266.
[22]Moseley and Marks 2006.
[23]Reis and Housley 2022, p. 139.

their complexity increases, making them more difficult to evolve[24]. An interesting thought on this is building the system around an *immutable technology* at its core. It is difficult to determine which technologies will prevail, but finding an immutable technology gives stability as the transitory tools built around it change with evolving technologies[25]. In order to develop a future-proof prototype, the notion of immutability is important for the framework selection in Chapter 3.

2.2. Streaming data architecture

At a high level of abstraction it is possible to define general streaming data architectures that help building a DSMS. This section introduces the *Lambda architecture* and the *Kappa architecture*, as they are the two most common streaming data architectures. Further, advantages and disadvantages of each architecture are elaborated. These advantages and disadvantages serve as basis for decision-making for the selection of one architecture as blueprint for the prototype DSMS.

2.2.1. Lambda architecture

The general idea behind the *Lambda architecture* (cf. Figure 2.1) is to run two systems simultaneously: The *batch (processing) layer* yields accurate results from the whole dataset, whereas the *speed (stream processing) layer* processes data as it arrives to produce immediate but potentially inaccurate results with low latency. In cases where the fast approximation from the speed layer yields inaccurate results, results from the slower batch layer (using all historic data) can be utilized to correct the stream layer at a later point in time. The results from the speed and the batch layer are combined in a special database called the *serving layer*. This serving layer simplifies creating queries and is updated each time new results (from speed or batch layer) are available[26].

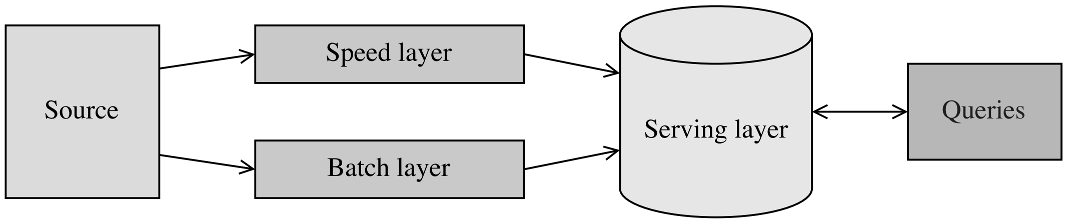


Figure 2.1.: Lambda architecture

Unfortunately, the Lambda architecture is not without its disadvantages. First, maintaining two different systems (possibly created with two different frameworks) requires a

[24]Breivold et al. 2008.
[25]Reis and Housley 2022, p. 122.
[26]Marz and Warren 2015, p. 14 ff.

lot of additional work. Moreover, the different results created by speed and batch layer must be merged in order to respond to stakeholder requests. For some computations (e.g. aggregation) this might be easy, but it becomes increasingly difficult for more complex operations (e.g. joins). Finally, using all (historic) data to compute a result in the batch layer is problematic if extensive amounts of data are available, which is why the batch processing can be done incrementally. If, however, the processing is done incrementally, the lines between the speed and the batch layer blur, which makes it questionable whether the batch layer is needed at all[27].

2.2.2. Kappa architecture

The *Kappa architecture* is an alternative to the Lambda architecture first proposed by *Kreps*[28]. He argues that if simplicity is more important than efficiency, the Kappa architecture is the better choice. As opposed to a main assumption from the Lambda architecture that the speed layer can deliver inaccurate results, Kappa assumes that stream processing does not yield inaccurate results. Consequently, all data is processed as a stream (the relation between batch and stream processing is explained in Section 2.3.1) and the batch layer is omitted to create a simpler system, as seen in Figure 2.2. Another difference to the Lambda architecture is that batch processing is done only if the requirements change; Lambda requires to do batch processing all the time in order to correct results from the speed layer.

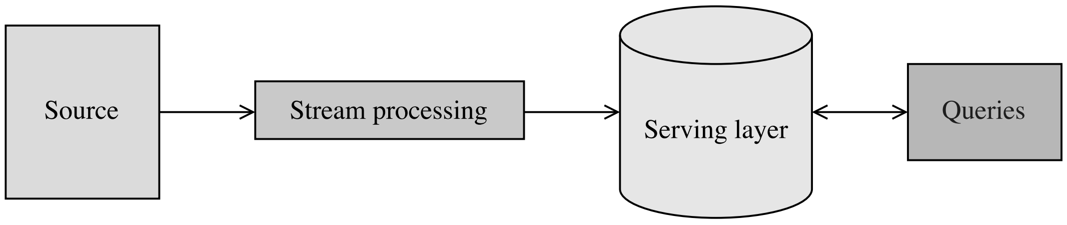


Figure 2.2.: Kappa architecture

Depending on the use case, the assumption that stream processing does not yield inaccurate results is wrong, which is why results from the Lambda architecture are more accurate. Furthermore, the batch processing performance of Kappa is not as good as Lambda's either, as dedicated systems for batch processing run more efficiently. Yet, saying that one architecture is better than the other is not reasonable, as they serve different purposes. Ultimately, which architecture to choose depends on the requirements for the system (cf. Section 2.1)[29]. However, considering the additional effort required to implement a system following the approach of the Lambda architecture and the limited

[27]Kleppmann 2017, p. 497 f.
[28]Kreps 2014b.
[29]Feick et al. 2018.

time frame of this thesis, it is reasonable to stick with the Kappa architecture. A second pipeline for batch processing can be added in the future, if more complex batch processing jobs have to be executed.

2.3. Challenges and concepts from stream processing

This section offers a general overview of important challenges and concepts from stream processing. First, the relevance of stream processing for this thesis is explained and the differences to batch processing are highlighted. Next, a more detailed view on how to ingest data into a DSMS is given, before concepts for stream processing and their necessity are expounded. Finally, important storage concepts are presented and the synchronization of different storage locations is explained.
Please note that the presented list of terms and principles is not exhaustive, as they are selected with the problem setting of this thesis in mind. Furthermore, the descriptions of the presented principles focus on the general concepts underlying them and their purpose. In order to gain a more detailed understanding of how these principles work, we refer to the cited literature.

2.3.1. UAS and stream processing

In general, we distinguish between two forms of data: *bounded* and *unbounded* data (cf. Figure 2.3). Unbounded data is similar to data in reality, a continuous (or sporadic) flow of events without a known end. Bounded data is created by splitting the data into finite pieces (so-called *batches*) by introducing a boundary such as time. It can be assumed that most data is received in an unbounded form before it becomes bounded. Defining where to set boundaries closely relates to the frequency with which data is ingested into a system for processing. If the frequency is low (e.g. data for quarterly financial statements) data can be collected and processed as one huge batch; this is called *batch processing*. However, batch processing is not an option if the frequency of arrival is high, and data is to be processed in real time[30]. Collecting data to create a batch before processing results is a delay that is intolerable for real time processing. In order to evade any unacceptable delay the processing must be done continuously by processing each newly incoming message immediately. This is the general concept of *stream processing*[31].
For large parts, this thesis will deal with stream processing. To understand how stream processing, DSMS, and UAS are related, consider e.g. a fleet of UAS where each UAS is equipped with multiple sensors. Thereby, each sensor sends messages continuously and at a high frequency during operation. In order to monitor this fleet in real time,

[30]Reis and Housley 2022, p. 235 ff.
[31]Kleppmann 2017, p. 439 f.

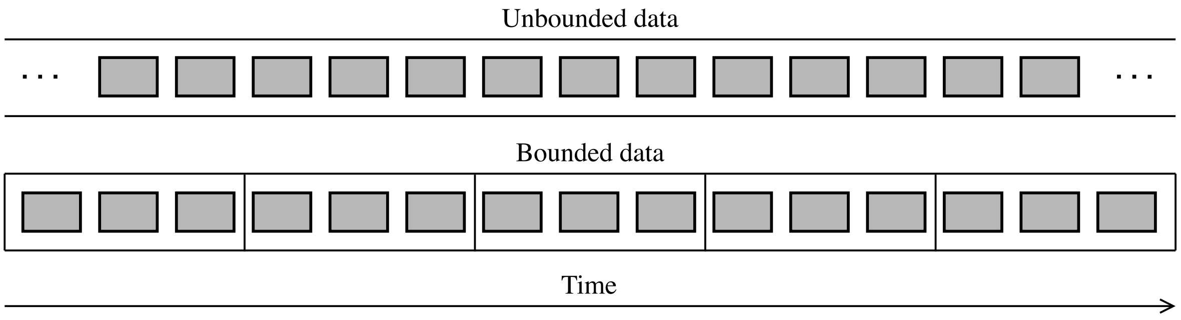


Figure 2.3.: Illustration comparing unbounded and bounded data

sensory data must first be ingested into a system that can handle the incoming messages. This can be done through something we will (for now) refer to as *ingestion black box* (cf. Section 2.3.2). The ingestion black box takes the incoming messages and prepares the stream so that it can be consumed by the *processing black box* (cf. Section 2.3.3). In the processing black box the data is processed to create the required real time insights for monitoring and to store the data to resilient storage. In addition to resilient long term storage, all messages have to be cached in the *streaming storage* (cf. Section 2.3.4) throughout the whole process. An illustration of this process can be seen in Figure 2.4. The tasks carried out by the two black boxes and the streaming storage, while having to meet all requirements from Section 2.1, is what this thesis refers to as stream processing. Because these components form a system that can handle data streams in real time, it is called a DSMS.

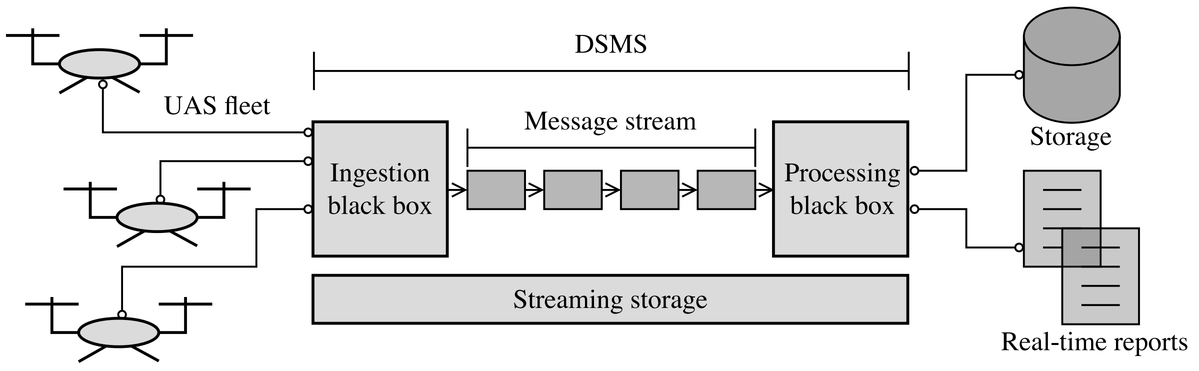


Figure 2.4.: Exemplary illustration of a fleet of UAS transmitting data as a continuous stream of messages, which are ingested into a DSMS and processed

As mentioned above, another possibility to deal with large amounts of data is batch processing. Instead of computing answers timely, newly arriving messages are buffered and results are produced at a later point in time[32]. For instance, we could wait until the end of day to collect any sensory data from the UAS before processing it. By doing so, producing any form of real time insight becomes impossible. That is why this thesis

[32]Babcock et al. 2002.

focuses on developing a prototype for stream processing. However, the notion of batch processing is not irrelevant. Imagine that after extensive testing a huge batch of test data is available. This batch data could then be used to enhance the results from stream processing, as many *machine learning* techniques still depend heavily on batch processing. Consequently, it is essential to design the prototype with the possibility in mind that batch processing is added in the future and/or the produced data is used for batch processing.

2.3.2. Ingestion black box

To shine some light on how the ingestion black box ingests data for stream processing, this subsection introduces potential difficulties and important definitions.

Events

Until now we assumed that a DSMS processes streams of messages, whereby messages can contain anything from a picture to an audio file. However, for ingestion it is not enough to assume that an ominous message with unknown content is processed. After all, we need to control the type of data in the system. Just think what would happen if messages with different or even unknown data types are saved in the same database. Consequently, for this thesis it is assumed that messages are used to transmit *events*. Thereby, an event can be defined as: *"a small, self-contained, immutable object containing the details of something that happened at some point in time"*[33]. Moreover, it is assumed that each event is subject to a *payload*. Usually, the payload is used to describe a dataset's characteristic like kind, shape, and size. However, throughout this thesis payload is used as a means of describing events. Thus, kind describes the type and the format of an event, whereby the type (e.g. text, tabular) influences how the format is expressed as file extension. Shape describes the dimensions of the event such as the number of characters if text is transmitted. Size describes how many bytes one event has[34].

Message broker

Dealing with large amounts of incoming messages introduces the following problem: What if messages are produced at a higher rate than the consumer (e.g. a stream processor) can handle? There are three possibilities how to deal with this: Either messages are dropped, buffered in a queue or back pressure is applied, which means the producer is blocked and therefore cannot send more messages. Of course, dropping messages before processing is not an option if one purpose of the system is to store all records in order to be able to retrace all activities (cf. Section 2.1). This is why the possibility to send messages directly from the producer to the consumer can be eliminated, as it is prone to message loss. To

[33]Kleppmann 2017, p. 440.
[34]Reis and Housley 2022, p. 235 ff.

prevent any message loss, so-called *message brokers* (also known as *message queues*) can be utilized[35]. They are comparable to a database class in object-oriented programming that is enhanced with methods such as *enqueue()* and *dequeue()* for handling message streams[36].
After a message is processed (thus it is saved to long-term storage), the message broker can either delete it from or keep it in the (short-term) streaming storage. Deleting it is not ideal, as that would not allow to do any re-processing if a consumer fails. If messages are deleted from the queue you cannot re-run the same queue and expect the same result[37]. In order to be able to re-process a queue of messages in the same order and yield the same results the notion of *log* is essential. A log can be defined as an append-only series of messages ordered by time. Append-only implies, that messages are immutable and therefore it is not possible to delete or update any values from a message. The ordering by time is done via a *log entry number*, which is assigned to each message when appended. Thereby, each log entry number is a unique, sequential number; it is like a timestamp, yet independent of any physical clock. Working with logs guarantees that results are deterministic (when the same messages are processed in the same order by different systems they yield the same result)[38]. The importance of re-processing is highlighted in Section 2.3.3.
Given that the message order is relevant and dropping messages is not an option, this thesis will use a *log-based message broker* as ingestion black box.

Defining time

Logs enable the processing and re-processing of message in the exact order in which they arrive. However, by definition it is assumed that the order of arrival of events in any data stream cannot be influenced (cf. Section 1.1). This means we cannot assume that the order of arrival is equal to the order of occurrence; at least not if multiple producers are connected to the system. This is problematic in cases where events are ingested at times that differ strongly from the times of their actual occurrences.
In Figure 2.5 an example of two UAS, that send messages at the same frequency but time-shifted, is illustrated. Usually, we would assume that both UAS send messages (containing events) in the same order as events occur. The message broker then assigns a unique log entry number so the order is not lost. Unfortunately, arrival time and time of occurrence will not always be equal. For instance, during the message transmission, UAS 2 could experience some kind of network delay due to network congestion. Because UAS 1 is in a different network its messages arrive earlier, although some of them actually occur later

[35]Kleppmann 2017, p. 441 ff.
[36]Gray 1995.
[37]Kleppmann 2017, p. 444 ff.
[38]Kreps 2014a, p. 1 ff.

in time. However, this is not reflected in the message queue, where the message with the log entry number 2 occurs after 3 and 4 occurs after 5.

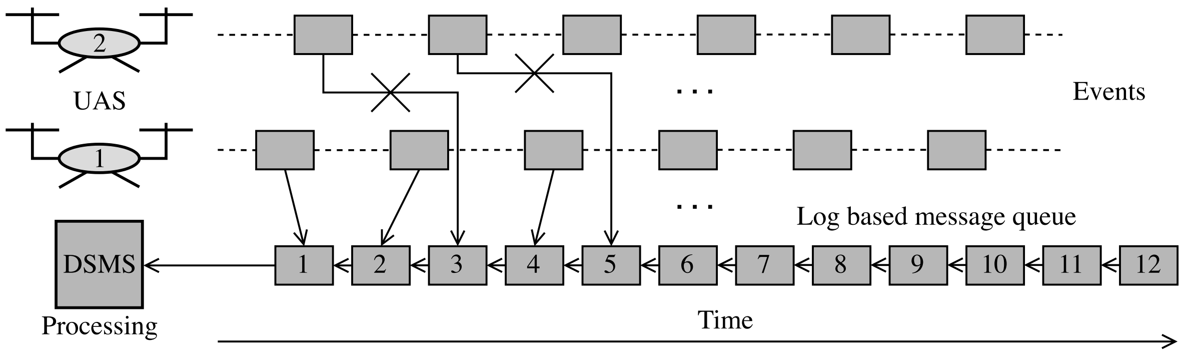


Figure 2.5.: Illustration of an example where event time and ingestion time differ strongly

Problems like this cannot be solved with a DSMS, as it cannot influence potential network delays, transmission failures, device failures etc. Consequently, it is of utmost importance to keep track of time as events are handled by different parts of the DSMS. A distinction can be made between three points in time[39]:

- *Event time* indicates when an event is generated by a producer.
- *Ingestion time* indicates when a message is put in queue by the message broker.
- *Process time* indicates the point in time after ingestion when the message is processed, whereby *processing time* describes the amount of time it takes to process it.

We note that the notion of time for stream processing is not defined uniformly in literature. For instance, Akidau et al.[40] distinguishes between event time and processing time. Whereas event time is as defined above, processing time is defined differently as *"... the time at which an event is observed at any given point during processing within the pipeline ..."*. Another example is Babcock et al.[41] who distinguish between *implicit* and *explicit* time. They define implicit time as an indicator for the order of event arrivals and explicit time as a data attribute that is selected as timestamp. In order to deal with the difficulty of different system clocks Dean[42] proposes to record the time an event occurs (according to the device clock), the time it is sent to the server (according to the device clock) and the time it is received by the server (according to the server clock). Even if all definitions are slightly different, the general idea is to keep track of time during all phases of message handling from event generation to processing.

[39]Reis and Housley 2022, p. 164 f.
[40]Akidau et al. 2015.
[41]Babcock et al. 2002.
[42]Dean 2015.

Late arrivals

It is common to define some sort of exclusion time for data that arrives late with respect to event time. After this exclusion time late data is dropped and will not be processed[43]. However, in this thesis, defining an exclusion time is not feasible, because we still need to save all messages to resilient storage. If we drop messages before saving them we would not have a full record of all activities available after the UAS operations finish. Of course, we still need to make sure, that late events do not delay our stream processing results for too long. How late data can be handled during stream processing is explained in more detail in Section 2.3.3.

Ingestion task summary

The tasks of the ingestion black box can be summarized as follows: A producer creates an event, which is an immutable object that represents something that happened. This event is sent to a consumer as a message. Now, it could happen that messages are sent at a higher rate than the consumer can process. Since message loss is not acceptable, the messages are queued, which is done by a message broker. Considering the possibility that a consumer fails, it is important to allow re-processing of messages. To guarantee deterministic results, a log is used to keep the exact order of message arrivals. However, the order of message arrivals must not equal the order of event occurrences. Given that the DSMS cannot influence the order of arrival, it is important to track different times during all stages of message handling, such as event time, ingestion time and process time. Regardless of when events are received, they must be saved to resilient storage to guarantee complete recordings of all UAS operations.

2.3.3. Processing black box

In Section 2.3.1 the processing black box is described as a means for processing a stream of data to gain insights; there is no mention of how this is done. In this thesis it is assumed that the processing is done by one or more *stream processors*. Thereby a stream processor takes a stream as input, applies a transformation and produces another stream as output. Typical tasks carried out by a transformation are manipulation (e.g calculating an average), enhancement (e.g. adding additional information from a database) and saving data for downstream use[44]. Based on the problem definition the output stream can then be consumed by another stream processor, written to a database, or used to create real time reports. The unbounded nature of data streams introduces a whole range of new challenges for stream processors that require a number of new concepts, both of which are discussed in this subsection.

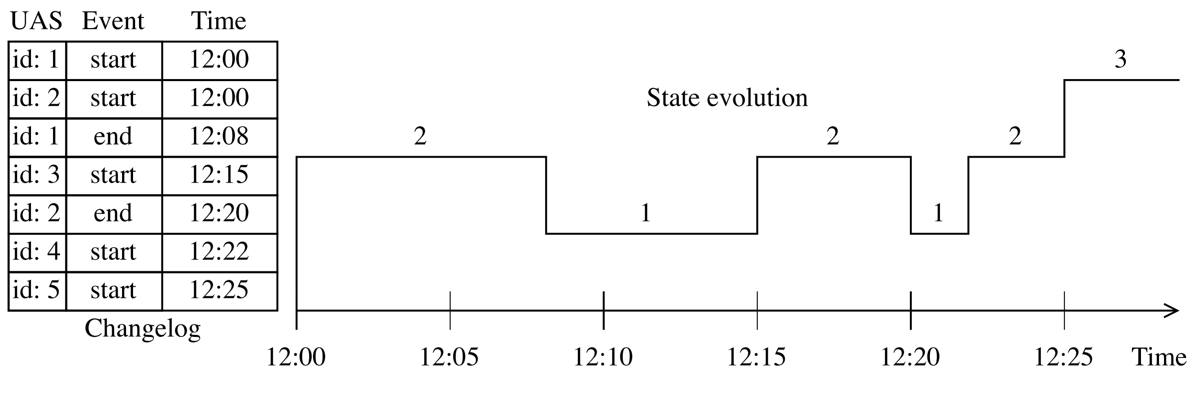
43 Reis and Housley 2022, p. 248 f.
44 Reis and Housley 2022, p. 309.

State

In stream processing the notion of a mutable *state* is essential to do non-trivial computations with data streams. This kind of processing is referred to as *stateful stream processing* whereby state can be defined as a local variable that stores the result of a computation carried out with incoming, immutable events. When a new event arrives the updated state is computed based on the current state and the new event. In other words: The result of a stateful calculation depends on all previous events and the newly arriving event[45]. This means if the log of all events is viewed, it is possible to reconstruct how the state evolved over time. An analogy from mathematics is to define state as the integral of a stream of events over time from $t = 0$ until the present $t = now$; if the state is differentiated by time at any t you get the stream of events that changed the state until t (cf. Equation (2.1))[46].

$$state(now) = \int_{t=0}^{now} stream(t)\, dt \qquad stream(t) = \frac{d\, state(t)}{dt} \tag{2.1}$$

A simple example for state is monitoring the number of operating UAS. As soon as a UAS sends a message that a new operation is started, the number of operating UAS is increased. After the UAS completes its operation it sends a message to inform the system, decreasing the number of active UAS. If the log of all messages (often referred to as *changelog)* is viewed, it is possible to reconstruct how many UAS were operating at any point in time. An example how the state evolves based on the changelog from multiple UAS starting and ending operations can be seen in Figure 2.6. Moreover, this example highlights the importance of state: If each event is viewed individually without considering the other events there is no possibility to know the number of active UAS. Of course, there are operations in stream processing that do not require a state (e.g. simply view a message), yet without any state it is not possible to do any complex computations. Consequently, it is important to build a DSMS that can handle stateful stream processing.



UAS	Event	Time
id: 1	start	12:00
id: 2	start	12:00
id: 1	end	12:08
id: 3	start	12:15
id: 2	end	12:20
id: 4	start	12:22
id: 5	start	12:25

Changelog

Figure 2.6.: Example of how state evolution in counting can be modeled with a changelog

[45]Hueske and Kalavri 2019, p. 53 ff.
[46]Kleppmann 2017, p. 459 ff.

State and fault tolerance

In Section 2.3.2 the log based message broker, that allows to re-process messages if a consumer fails, has been presented. However, it is not discussed where exactly messages (or state) are stored for re-processing and why it is important to be able to re-process them. Storing messages or state to remote storage is not ideal, as the data would not be available locally in the system. A possibility that provides locality would be to store the changelog to a local table as an incremental record-for-record backup of the state[47]. The importance of re-processing originates from using a backup to restore the state in order to achieve fault tolerance (cf. Section 2.1.3). For instance, considering the example in Figure 2.6, if processing fails at 12:10 the current state of counting $s = 1$ is lost. Not knowing any state, the system is restarted at 12:14 before a new UAS operation is started at 12:15 with the result that the system now wrongly thinks $s = 1$ not $s = 2$. With an available changelog the system would be able to replay all events and restore the correct state.
This, however, leads to another problem: If a long-running system fails it is not feasible to re-process all events, as it would require too much time and storage. If the state is changed by events identified by a key (i.e. IDs for different UAS), so-called *log compaction* can help to reduce storage requirements. Thereby, only the most recent updates by a key are kept, allowing to re-construct the latest state only, not its complete history[48]. Let us return to the example mentioned above: A system without log compaction would have to store $m = 3$ messages to restore the state at 12:10. With log compaction only the most recent updates are kept, which are *start* for *UAS 2* and *end* for *UAS 1*, so $m = 2$. Knowing these last two states, it is easy to figure out that $s = 1$ at 12:10, because there is only one UAS that started an operation but did not end it. Certainly, reducing m from 2 to 3 does not seem like a huge achievement and yet it could have a significant impact on systems dealing with large UAS fleets. For example, a fleet with n UAS would have to keep at most $m = n$ messages to restore the current state. Assuming that each UAS carries out x operations in one day, a system without log compaction would have to keep $m = n * x$ messages. That means, the number of messages that must be stored and re-processed to restore the state is reduced by a factor of x.
Saving the state as log compaction in a local table is a valid option to create a backup for restoring the state. However, when to create a backup remains to be answered. A simple solution would be to create backups at fixed time intervals. Yet, depending on the framework, there are different concepts available that define when the state is saved. For instance, *Apache Flink* uses a variant of a checkpointing algorithm called *Asynchronous Barrier Snapshotting*, which stores the global state (snapshot) to resilient storage[49];

[47]Kreps 2014a, p. 37 f.
[48]Kreps 2014a, p. 38 f.
[49]*Fault Tolerance via State Snapshots* 2023.

it is based on the *Chandy-Lamport algorithm*[50]. The creation of checkpoints is done by continuously inserting *barriers* into the data stream. A checkpoint is created for all messages received between two consecutive checkpoints. If a process fails, all messages that were received after the last checkpoint are re-processed. This means that all messages before the last checkpoint can be discarded[51]. Another approach is *microbatching* utilized by *Apache Spark*[52]. Microbatching partitions a continuous data stream into finite pieces based on small time intervals. Whenever an interval is complete according to the measured time, an intermediate state is created and saved to resilient storage. If processing fails, only the latest, incomplete interval must be re-processed[53].
In order to guarantee fault tolerance in line with system requirements, how and where to store messages and state is an important consideration when selecting frameworks for deployment (cf. Chapter 3).

Exactly once

By resetting the computation and its input, messages can be re-processed to restore the state after a fault. However, it is not possible to reset any output produced by the stream processor. Particularly not if it is already consumed by a downstream processor or written to a database. In order to prevent wrong results, it is essential that re-processing of messages must not influence results[54]. Consequently, mechanisms are needed to guarantee *exactly once processing* of messages, whereby a message only affects a result once. Please note that *exactly once processing* and *exactly once delivery* are not equivalent. Whereas exactly once processing allows to deliver a message *at least once*, exactly once delivery strictly forbids redeliveries. Technically, however, exactly once delivery is impossible, as you can never be absolutely certain that a message will arrive. This means messages can only be are delivered *at most once* or *at least once*[55]. The issue with exactly once delivery is closely related to the *Two generals problem*, where two parties have to communicate to coordinate a potential attack. Because neither the message to start the attack nor the acknowledgment for the message arrive with absolute certainty, no party ever really knows when the other one will attack, thus no one attacks[56].
The following example highlights why exactly once processing is important in this thesis: In Figure 2.7 the situation from Figure 2.6, where a stream processor tracks the count of active UAS, is extended by a downstream stream processor that calculates a moving average of active UAS per time interval. The first processors receives the start/end

[50]Chandy and Lamport 1985.
[51]Carbone et al. 2015.
[52]*Structured Streaming Programming Guide - Spark 3.4.1 Documentation* 2023.
[53]Zaharia et al. 2012.
[54]Kleppmann 2017, p. 477 f.
[55]Treat 2015.
[56]Archer Brown 2023.

messages from the UAS and counts the number of active UAS every two minutes. Thereby a new message format is created; it contains a new ID, the timestamp and the UAS count. These messages are sent to the second processor who collects all messages in order to calculate the moving average of active UAS. This moving average is updated each time a new message from the first processor is received. Now, it could happen that the count processor fails at 12:05 like in our example. As a consequence, the count processor reprocesses all start/end messages received after 12:05 in order to recreate the state before the failure. Assuming that no results should get lost we send each message at least once. Since we cannot discard the messages we already sent, the second processor receives some messages twice and computes a wrong average (in our example the message with $id = 4$ is included twice).

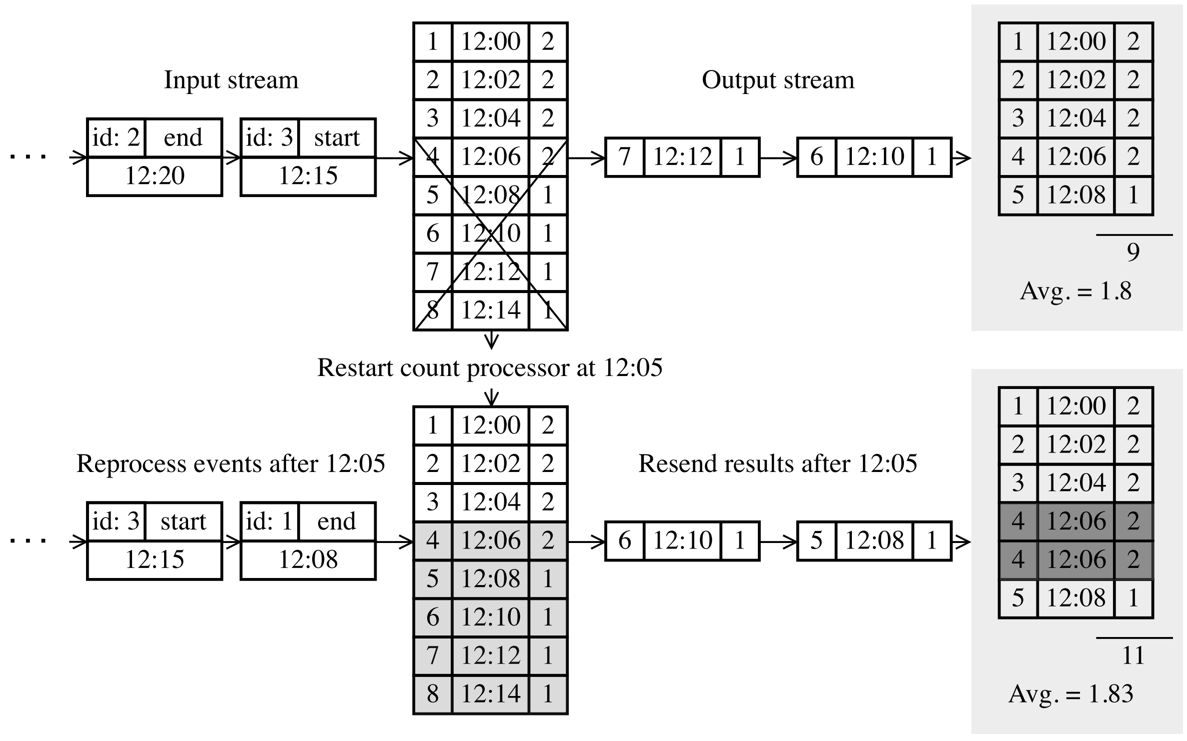


Figure 2.7.: Illustration why exactly once can be important for stream processing

A possibility to guarantee exactly once processing is to make the computations *idempotent*. The term *idempotence* means that running an operation multiple times yields the same results as running it exactly once. A simple example for an idempotent operation is cleaning a floor. No matter how often the floor is cleaned, the result remains a clean floor. An operation that is not idempotent is withdrawing money from a bank account[57]. Naturally, the computation done in the above mentioned example is not idempotent (i.e. increasing a counter without decreasing it yields a different result). Yet, it is possible to make the processor idempotent via the log entry number (cf. Section 2.3.2)[58]. Given

57 Helland 2012.

58 Kleppmann 2017, p. 478.

that it is a unique sequence number, the processor can check if a number has already been processed. If so, the message is discarded and does not affect the result again[59].
The need for exactly once processing to guarantee correct results in all downstream operations highlights the importance of a log based message broker for data ingestion (cf. Section 2.3.2).

Windowing

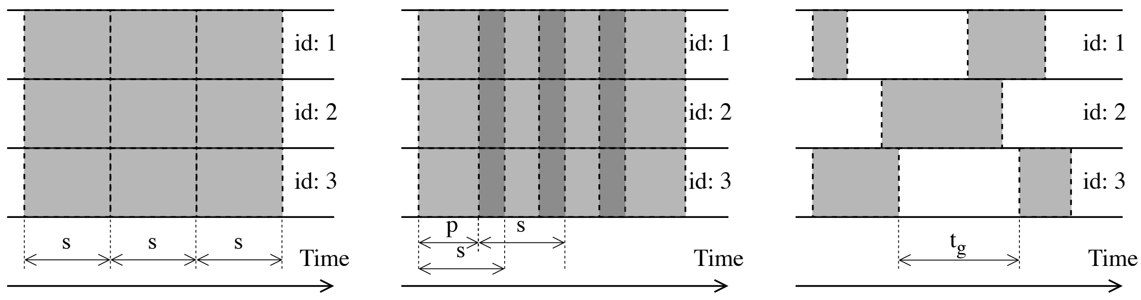
Windowing is the process of dividing a data stream into smaller, finite subsets called *windows* and calculating a result for each window (you could also refer to windows as micro-batches). Thereby a window is specified by a set of parameters such as size s and slide period p (which delimits the beginnings of different windows)[60]. When to use windows strongly depends on the use case. For instance, if you want to filter or map some data, it is not mandatory to utilize windows. However, any form of e.g. time-bound operations require some form of windowing. There are many different forms of windows, the three most common ones are[61][62]:

- *Fixed windows*
  Fixed windows, also known as *tumbling windows*, have a static window size s, which is defined by either time or the number of events. For fixed windows, the slide period $p = s$. Usually, fixed windows are aligned, which means that a window applies to all messages, no matter which id they are assigned to (e.g. a windows applies to all UAS independent of the id). An illustration of fixed windows can be seen in Figure 2.8a.

- *Sliding windows*
  Sliding windows are defined by a static window size s and a slide period p (time or number of events). If $p < s$, the windows overlap, i.e. sliding windows are a special case of fixed windows where the windows can overlap. Sliding windows are illustrated in Figure 2.8b, here you can see that they are aligned like fixed windows.

- *Sessions*
  Sessions are used to capture periods of activity defined by a timeout gap t_g. All events that take place within a span of time $< t_g$ are grouped together, filtering periods of inactivity. Unlike fixed or sliding windows, sessions are always unaligned, i.e. a window applies to one id only. An illustration can be seen in Figure 2.8c.

[59]Fernandez et al. 2014.
[60]Li et al. 2005.
[61]Akidau et al. 2015.
[62]Reis and Housley 2022, p. 283. ff.



(a) Fixed windows (b) Sliding windows (c) Sessions

Figure 2.8.: Illustrations of the three most common types of windows

Windowing provides a lot of flexibility for processing data streams. For instance, with windows it would be possible to compute and monitor the average speed of a UAS in five second intervals. So, we receive messages containing the speed of a UAS at a given rate, collect these messages for five seconds and compute a result for the average speed of the UAS in the last five seconds. Thus, if any unexpected deviations are detected actions can be taken immediately. Of course, without windows it would still be possible to monitor the speed by computing a moving average (similar to Figure 2.7). However, with a moving average outliers can be missed easily, because it is hard to tell exactly when they happened. Consequently, when selecting frameworks in Chapter 3, it is important to include support for some sort of windowing as criterion.

Watermarks Although it is possible to define a window's size based on the number of events it should contain, the size of a window is mostly determined by time. As explained in Section 2.3.2, there are many different times to keep track of in a DSMS. This raises the question of what time should be used to define a window's size. If windows are defined based on ingestion or process time, results are computed with low latency exactly when and in the same order the data arrives. The computed results, however, would not take into account any events that are late with respect to event time[63]. A mechanism used to help defining windows based on event time is called *watermark*. Thereby a watermark can be defined as a threshold used to determine whether events belong to a window or are considered late[64]. Unfortunately, watermarks are not perfect in the sense that they could be *too fast* by emitting the threshold before the arrival of a late event or *too slow* by waiting too long for a single late event, delaying subsequent processing. In general, no system can wait indefinitely long for late data, so no system can ever be 100% certain that no data is missing[65]. It is important to find the right balance between confidence in the results and the delay introduced by waiting for late data.

[63]Hueske and Kalavri 2019, p. 31.
[64]Reis and Housley 2022, p. 285.
[65]Akidau et al. 2015.

Given that not all computations need to define windows based on event time, it is essential to evaluate any computation beforehand and take into account how late data would influence the result. If the influence is substantial, mechanisms like watermarks must be implemented in order to yield correct results. Maybe this is not important for the first prototype of the system, but more complex computations in the future might require watermarks. Consequently, the availability of mechanisms like watermarks is an important factor for framework selection (cf. Chapter 3).

Stream joins

Typically, a record in a dataset is associated with another record through some reference (e.g. a *foreign key* in relational database). If you want to access information from the record holding the reference and the record being referenced a *join* is needed. Usually, joins are used on static batch datasets, but by maintaining a state it is possible (yet more difficult) to apply them to unbounded data streams as well[66]. Three different types of stream joins can be distinguished[67][68][69]:

- *Stream-stream join*
  You can use stream-stream joins to combine events from two different data streams into a single, enhanced data stream. It is possible to do an *inner join* or an *outer join*. With the inner join an output event is only created if events are received from both streams. For the outer join an output is created if one event arrives, regardless of the arrival of an event from the other stream. Please note that the (potentially) different arrival times of the streams requires buffering of events. For instance, if the first stream is delayed for 5 seconds, the second stream has to buffer events for 5 seconds until the two streams match. The buffer size depends on the expected delay and the available storage. An application example for stream-stream joins is the joining of streams from different sensors, e.g. one sensor monitors the speed of the UAS, whereas the other one monitors wind speeds.

- *Stream-table join*
  With a stream-table join it is possible to enrich a data stream with information stored in a table. It works similarly to stream-stream join, whereby one of the streams is known from the beginning in its entirety. For example, the events from the data stream might contain a UAS' unique ID but no information about its model, age, etc. Given that this information is (mostly) static it can be stored in its entirety in a table. Every time a new event arrives the system can check the table for the UAS ID to enrich the event with additional details if needed.

[66]Kleppmann 2017, p. 403, 473 ff.
[67]*Samza - State Management* 2023.
[68]*Introducing Stream-Stream Joins in Apache Spark 2.3* Tue, 03/13/2018 - 07:59.
[69]Reis and Housley 2022, p. 286 ff.

- *Table-table join*
  If the changelogs of data streams are saved to tables you can use table-table joins to create a new table. Doing this join in a DSMS has the advantage that whenever data changes in one table, it is automatically synchronized with the most recent data for the same key from the other table to create a joined output.

Joining streams and tables in a DSMS is a powerful tool to enhance a stream with additional information. The notion of stream joins could become even more relevant, the more data on UAS operations is accumulated. A framework's ability to do stream joins will be considered for the framework selection in Chapter 3.

Processing task summary

The tasks of the processing black box can be summarized as follows: First, a stream of messages enhanced with logs and timestamps is received from the ingestion black box. Next, the received stream is transformed by one or more stream processors to produce the desired output. In order to produce results that depend on changes made by previous events, the system has to maintain a state. This state must be retrievable in the event of a fault, otherwise the system would not be fault tolerant. The state can be retrieved through a changelog that stores all changes made to the state by the events. Because storing all changes could require too much storage, mechanisms like log compaction can help to reduce storage requirements. Now, if events are re-processed, it must not happen that results are affected twice. This can be evaded by guaranteeing exactly once processing whereby computations can be made idempotent with the help of the log entry number. Furthermore, a concept for making stream processing more flexible is called windowing, thereby unbounded data is split into finite pieces called windows before processing. While windowing is very powerful, it can become challenging, if a correct order of events based on event time is required. In this case mechanisms like watermarks need to be implemented. Finally, another essential concept from stream processing are stream joins. They can be used to enhance a stream with additional details if needed.

2.3.4. Database technology

Whenever an event is generated during an UAS operation it must be saved to a database for long-term storage to comply with EU regulations (cf. Section 2.1). Additionally, parts of the data stream need to be saved in cache (streaming storage), where the data stream can be accessed quickly, in case parts of it must be replayed after a fault (cf. Section 2.3.3). This section outlines general storage concepts and elaborates how to keep different databases and caches synchronized during stream processing. Moreover, popular concepts like the CAP-theorem and ACID transactions are presented and discussed.

Hot, warm, and cold data

Depending on how frequently data needs to be accessed it is possible to distinguish between hot, warm and cold data[70]:

- *Hot data* is data that is accessed frequently and must be cheap to retrieve (i.e. requires little time to retrieve). Typically, the storage required for hot data is very expensive, as it is saved to *Random Access Memory (RAM)* (as it is the fastest form of storage) or a fast *Solid State Drive (SSD).* The notion of hot data is important for replaying data streams in the event of an error. If retrieving the data after an error takes too long, the performance of the DSMS suffers severely, as the processing of any subsequent events has to be postponed until the old events are replayed. In general, it can be said that hot data is the only form of data that is saved in a cache, not in a database.

- *Cold data* is data that is accessed only infrequently, whereby costs for retrieval are high but storage itself is cheap. Typically, cold data is saved on a cloud or on a *Hard Disk Drive (HDD).* Considering the limited read and writes speeds of a HDD, and the limited bandwidth for cloud storage, retrieval of data is expensive in the sense that it takes very long. If accessing legal recordings is a rare occasion, it makes sense to store them as cold data in a database.

- *Warm data* is data that is not accessed as often as hot data but more often than cold data; storage costs are cheaper than for hot data, but slightly more expensive than for cold data. Typically, warm data is stored on a slower SSD. If, e.g. the legal recordings need to be accessed frequently, it is reasonable to store them as warm data. Of course, it would be possible to store the most recent recordings as warm data first and as cold data later.

The bottom line here is that we need different kinds of storage to build a DSMS (with caches and databases) that fulfills all performance requirements. This also means that we have to think of a way to keep all these different kinds of storage synchronized during operations.

Event sourcing and change data capture

Data must be synchronized across multiple databases and caches for two reasons: Performance requirements must be met, and data must be available to different stakeholders (e.g. operators and authorities). There are two important concepts related to synchronization:

[70]Reis and Housley 2022, p. 223 ff.

- *Event sourcing* is a concept used to describe how data is structured in a database (or a cache). Thereby every write to a database is considered an immutable event that is saved to a log. An identical database can be constructed by consuming this log in sequential order. Errors during log consumption do not yield wrong results, as keeping track of the sequential log number helps to resume consuming the log after an error. The log of all changes acts as the single source of truth[71].

- *Change data capture (CDC)* is a concept similar to event sourcing, but at a different level of abstraction; CDC does not require to keep track of all writes to a database but the most recent ones. It pairs well with the concept of log compaction explained in Section 2.3.3, whereby the number of events that are saved depends on the number of different keys for the events. CDC is often used if an existing database is in place. Thereby, a snapshot of the current database is taken and all subsequent changes are recorded as a stream of events. If we select one database as the *leader* (the only database where writes are possible), we can track all changes applied to this database to replicate it in any of the downstream databases (the *followers*). Thereby the followers can consume the changes at their own pace without affecting the performance of the leader[72].

Although logs are a simple concept, they are very powerful. Utilizing logs throughout a DSMS is not only a requirement to build a fault tolerant system, but also helpful to correctly synchronize changes across multiple databases and caches. Both, event sourcing and CDC, rely heavily on a log as a single source of truth for synchronization. Whether the prototype uses event sourcing or CDC strongly depends on the existing system and its interfaces. Either way, the final system (maybe the prototype not yet) will require multiple synchronized databases and caches with different kinds of storage to meet all performance and stakeholder requirements, regardless of which of the above mentioned concepts is used.

CAP-theorem

Deciding on how to build a database for longterm storage does not only depend on choosing the right storage technology, but also on balancing trade-offs and providing requirement-based guarantees. The relation between the most important guarantees and the trade-offs between them can be explained by the *CAP-theorem* (also known as *Brewer-theorem*)[73]. CAP is the abbreviation for:

- *Consistency C* - Strong consistency provides the capability to execute updates while offering the same view of data to all consumers.

[71]Kleppmann 2016, p. 1 ff., 52 ff.
[72]Kleppmann 2016, p. 81 ff.
[73]Fox and Brewer 1999.

- *Availability A* - High availability is given if a consumer can access any replica of the given data at any point in time. Availability can be provided by creating redundancies through data replication.
- *Partition-resilience P* - Partition-resilience is given if the entire system can outlast a partition between data replicas.

The theorem states that in the presence of partition-resilience it is not possible for any networked shared-data system to provide both, strong consistency and high availability. In other words, if partition-resilience is required you have to find a trade-off between C and A. Based on our requirements, we can choose between three different types of systems (CA, CP and AP). However, it is not that one type is better than the other in general, they simply serve different purposes.
We note that some sources criticize the applicability of the CAP-theorem to database classification. Most notably, *Kleppmann*[74] argues that CAP is too vague and ignores a wide range of other essential trade-offs. Examples include not taking latency or node failures into account. Moreover, he argues that many technologies that claim to offer CA, CP, or AP do not obey to the original definitions, which leads to a lot of confusion and misunderstandings. He proposes a different framework called *delay-sensitivity* and recommends to abandon the CAP-theorem.
Although there are arguably strong limitations to the CAP-theorem, it still finds wide application in classifying database technologies today. Given the vast amount of available technologies (cf. Section 3.2), we require an easy, widely applied classification to guide us in the right direction. CAP offers this guidance, which is why the decision is made to apply it in spite of its limitations. Albeit, we are aware of its limitations and do consider them.

ACID vs BASE

A series of interactions with a database (typically reads and writes) that should happen as one operation is called a *transaction*. Safety guarantees related to transactions can be described with the *ACID acronym (Atomicity, Consistency, Isolation and Durability)*[75].

- *Atomicity* means that transactions must either be completed or undone entirely (i.e. all-or-nothing).
- *Consistency* means that a transaction shifts a database from one consistent state to another (i.e. transactions can commit only legitimate results).

[74] Kleppmann 2015.
[75] Haerder and Reuter 1983.

- *Isolation* refers to hiding transactions from each other so they can occur concurrently without leading to inconsistent states.
- *Durability* refers to making sure that once a transaction is committed, its results are stored permanently (i.e. results are not lost when the system fails).

While ACID describes the way transactions are handled in traditional transactional databases, there are alternatives. A popular alternative that weakens ACID-guarantees is called *BASE (Basically Available, Soft state, and Eventual consistency)*[76]. Its definitions are very vague, and basically mean that anything that is not ACID is BASE. ACID is focused on providing consistency above all else, while no guarantees concerning availability are provided. BASE relaxes some of the guarantees in order to shift the focus to availability. Which one wants to utilize comes down to the use case.

SQL vs NoSQL

A popular alternative to traditional SQL-based relational databases are so-called *NoSQL* databases (which stands for *Not only SQL*). NoSQL emerged from the trade-offs discussion sparked by the CAP-theorem. In general, NoSQL solutions can be divided into four different categories: document databases, wide-column databases, key-values databases and graph databases. Some of these options are discussed briefly in Section 3.2. For now it is only important to know that there are alternatives to SQL-based relational databases.

Database technology summary

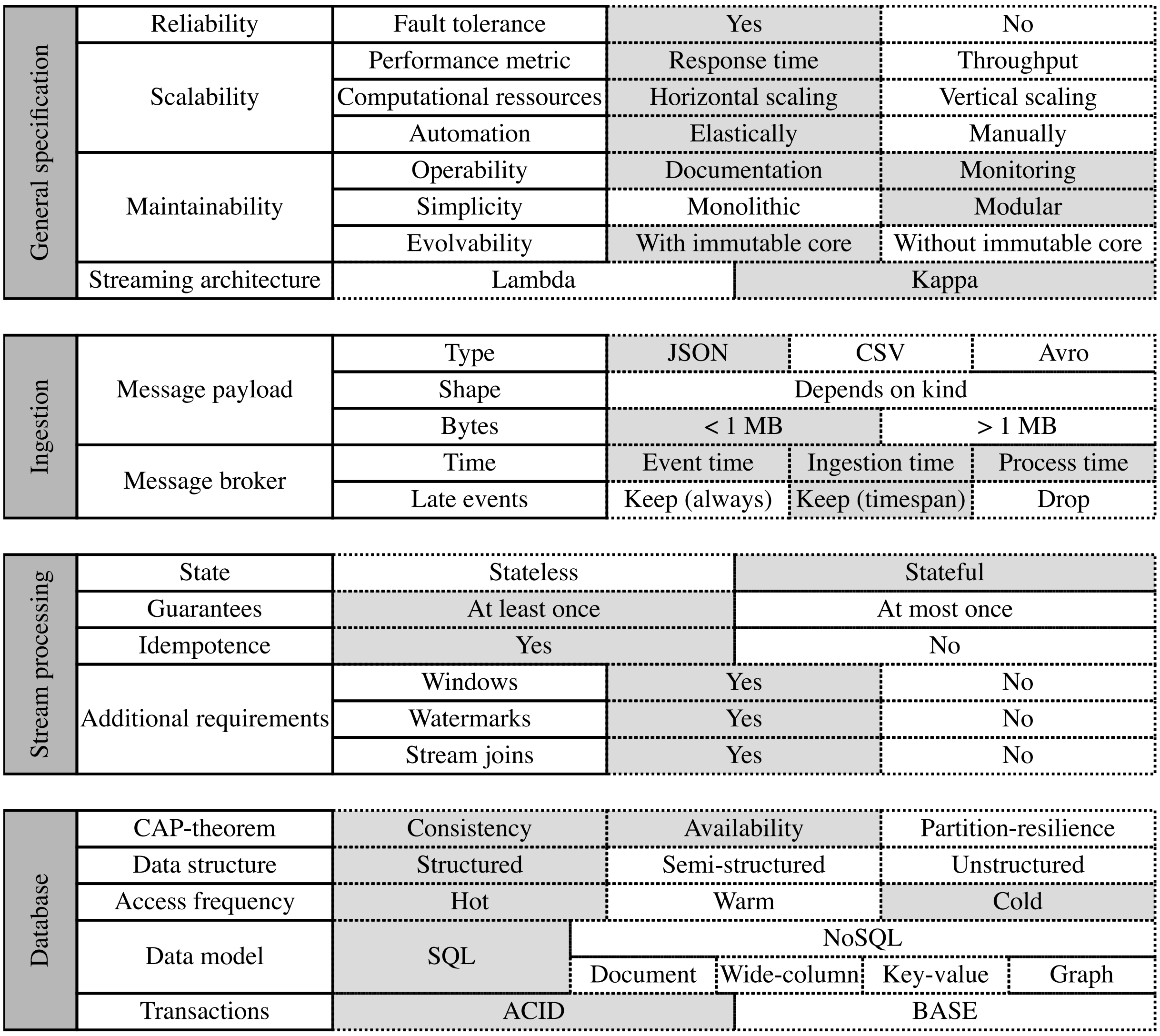
How to choose the right database technology strongly depends on the specific use case. First, it is important to consider how often the stored data is accessed by differentiating between hot, warm and cold data. This distinction alone shows that we require at least two different kinds of storage: one for the "hot" data from the stream processing, and one for the "cold" data from logging all UAS activities. Another important concept involves synchronization across multiple databases. Here, event sourcing and CDC are two popular concepts that are based on observing the database's log of changes. In order to classify databases based on the guarantees they provide the CAP-theorem can be utilized. Although it has its limitations, it still is a widely-known framework that can help to get a broad overview of the available technologies. Guarantees for database transactions can be classified in ACID and BASE transactions. Whereas ACID strictly obeys rules to provide consistency, BASE relaxes the rules and provides more flexibility. Finally, a popular alternative to SQL databases are NoSQL databases.

[76]Fox, Gribble, et al. 1997.

2.4. Specification and decision summary

Starting with the general specifications, the decision is made to make the system fault tolerant. This is inevitable, as a complete failure of the whole system is not acceptable due to regulatory safety requirements. Related to scalability, the most important performance metric is response time, because we need to be able to respond to alerts from the system immediately. Moreover, all scaling should be done horizontally and elastically, as the system will be subject to changing workloads throughout its runtime. In order to keep the system easily maintainable, a documentation should be provided and monitoring of the most important parameters should be implemented right from the start. The system itself will be modular with an immutable framework at its core. This attempts to make the system future proof as much as possible, while keeping a certain degree of flexibility. Considering that there is no immediate need for complex batch processing jobs, the general architectural approach is based on the Kappa architecture.
The data ingested into the system is exclusively available in JSON format. The shape of the data cannot be precisely defined, as the messages vary in terms of their content, but it is certain, that messages will not exceed a message size of 1MB. Considering the monitoring requirements for the system and how important the correct order of messages is, event, ingestion and process time must be tracked for each message. Late events will always be saved to resilient storage, no matter when they arrive. However, they will only be considered relevant for processing for a limited timespan.
Stream processing will involve almost exclusively tasks that require the maintenance of a state. Moreover, we do not tolerate the loss of any messages, as this could lead to safety related issues. Consequently, we have to implement a system that guarantees that messages are sent at least once, whereby all processing is made idempotent through unique message identifications. In order to deal with more complex processing jobs related to conflict detection, windows are a must have. Also, mechanisms like watermarks are required to deal with late events. Knowing that incoming messages must be correlated with existing data like flight plans, the selected framework must provide stream join capabilities.
Selecting a database technology comes (mostly) down to the CAP-theorem. As the expected workload for the system is not in the magnitude of hundreds of thousands of concurrent writes, it is decided to focus on consistency and availability rather than partition-resilience. Moreover, it is assumed that all data ingested into the system is structured. The stored messages will only be accessed sporadically for analysis purposes, which is why it makes sense to put them in cold storage. However, the streaming storage requires to be easily accessible, which is why hot storage must be considered, too. Furthermore, it is necessary to be able to perform different operations like filtering, mapping etc. on the data. Considering SQL's wide-spread application it is only reasonable to implement a

database that supports SQL queries. Finally, it is defined that transactions must comply with the ACID-schema.
A summary of all decisions described above made for the specifications and concepts presented in Section 2.1, Section 2.2 and Section 2.3 can be seen in Figure 2.9.



General specification	Reliability	Fault tolerance	[✓ Yes]	No		
	Scalability	Performance metric	[✓ Response time]	Throughput		
		Computational ressources	[✓ Horizontal scaling]	Vertical scaling		
		Automation	[✓ Elastically]	Manually		
	Maintainability	Operability	[✓ Documentation]	[✓ Monitoring]		
		Simplicity	Monolithic	[✓ Modular]		
		Evolvability	[✓ With immutable core]	Without immutable core		
	Streaming architecture		Lambda	[✓ Kappa]		

Ingestion	Message payload	Type	[✓ JSON]	CSV	Avro	
		Shape	Depends on kind			
		Bytes	[✓ < 1 MB]	> 1 MB		
	Message broker	Time	[✓ Event time]	[✓ Ingestion time]	[✓ Process time]	
		Late events	Keep (always)	[✓ Keep (timespan)]	Drop	

Stream processing	State		Stateless	[✓ Stateful]		
	Guarantees		[✓ At least once]	At most once		
	Idempotence		[✓ Yes]	No		
	Additional requirements	Windows	[✓ Yes]	No		
		Watermarks	[✓ Yes]	No		
		Stream joins	[✓ Yes]	No		

Database	CAP-theorem	[✓ Consistency]	[✓ Availability]	Partition-resilience		
	Data structure	[✓ Structured]	Semi-structured	Unstructured		
	Access frequency	[✓ Hot]	Warm	[✓ Cold]		
	Data model	[✓ SQL]	NoSQL			
			Document	Wide-column	Key-value	Graph
	Transactions	[✓ ACID]	BASE			

Figure 2.9.: Summary of all decisions made for the framework selection based on the specifications and concepts presented in Section 2.1, Section 2.2 and Section 2.3

3. Framework selection

Looking at the proposed concept from Section 2.3.1 while taking all requirements from Section 2.4 into consideration, it becomes clear that it is highly unlikely to find an all-in-one framework that fulfills every requirement. Consequently, the framework selection is split in three parts. First, the technology at the heart of the system is selected. It assumes the role of the message broker (cf. Section 2.3.2), i.e. the data ingestion and data distribution between the various frameworks. The next step is to select a database technology for longterm storage (which is not to be confused with streaming storage). Finally, a framework for implementing the tasks of stream processing is selected. In order to justify the choice of frameworks, each of the selected frameworks is described briefly and it is explained how these frameworks fulfill the given requirements and specifications.

3.1. Data distribution framework

There are numerous open-source frameworks for data distribution available. Popular examples include frameworks like *RabbitMQ*[77] (which implements the *Advanced Messaging Queuing Protocol (AMQP)*), *ActiveMQ* (which implements the *Java Message Service (JMS)*)[78] and *Apache Kafka*[79]. All requirements considered, of the three, Apache Kafka is the most viable option for the system we are trying to build. Frameworks like RabbitMQ and ActiveMQ can be excluded, because neither AMQP nor JMS allows the reprocessing of messages. They both require the discarding of messages after acknowledgement, which is not the case for Kafka, where logs allow deterministic reprocessing.
Of course, there are lots of other, lesser known frameworks available that (potentially) meet all our requirements for a message broker. Especially as handling large amounts of data becomes more and more important the number of smaller, more specialized frameworks increases. These frameworks are focused on solving very specific problems in niches where Kafka might not be the best choice. However, these frameworks are (often) not as mature and do not (yet) have a huge community for supporting and developing the framework further. Given the importance of resources like demo projects, known error messages, community support, etc. for learning and deploying an unknown framework,

[77] *RabbitMQ: Easy to Use, Flexible Messaging and Streaming — RabbitMQ* 2023.
[78] *ActiveMQ* 2023.
[79] *Apache Kafka* 2023.

smaller, less mature frameworks are excluded from consideration.
The following sections will offer a brief overview of Kafka's most important components and give a few simple examples for data flows in Kafka. This should help to better understand why Kafka is our framework of choice.

3.1.1. Introduction to Apache Kafka

Apache Kafka is a distributed *publish and subscribe* messaging framework for event streaming. Here, publish and subscribe messaging can be described as a framework for asynchronous distribution of messages between publishers (sending messages) and subscribers (receiving messages). This means that, in order to send and receive messages, publishers and subscribers do not have to be aware of each other. This decoupling of publishers and subscribers allows to build highly scalable and reliable applications[80].
In order to understand how Apache Kafka works and how it meets our requirements, first the different components and their interaction have to be explained. We note that all information in this subsection is either from the website to the Apache Kafka documentation[81] or the guide book *Kafka: The Definitive Guide, 2nd Edition*[82]. An overview of how all components, that are presented in the following, are connected from source to sink is illustrated in Figure 3.1.

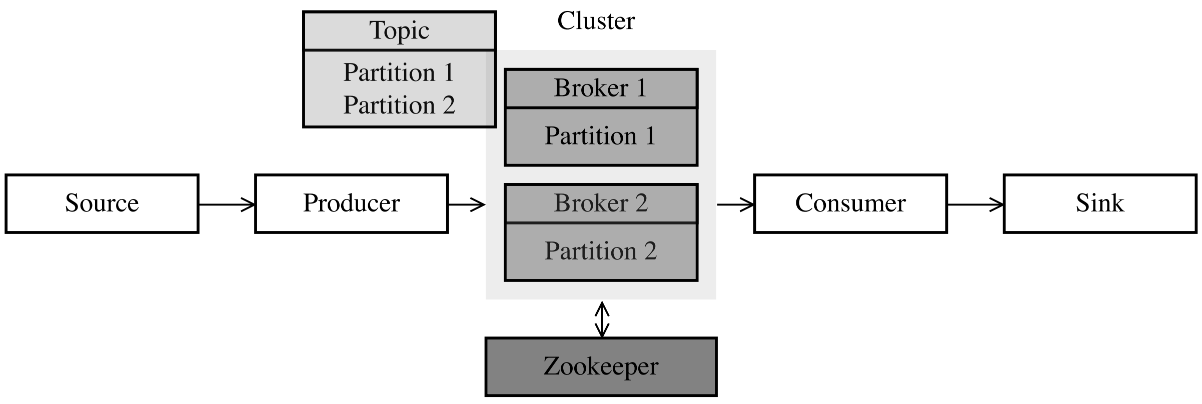


Figure 3.1.: Simple illustration of the connection between essential Kafka components

- *Topics* and *partitions*
  A *topic* in Kafka is used to categorize incoming messages (i.e. data streams). For instance, we could have one topic for the drones' sensory data, one for external information like weather, geofences etc. Within a topic data is split into so-called *partitions*, whereby the partitions are a way for Kafka to provide scalability and redundancy. How many partitions are created and how the data is distributed to

[80] *Publish-Subscribe - Intro to Pub-Sub Messaging* 2023.
[81] *Apache Kafka Documentation* 2023.
[82] Shapira et al. 2021.

the different partitions in a topic must be defined before starting the application. Usually, data is distributed to the partitions based on a key (e.g. the ID of a drone). The number of partitions depends strongly on how the rest of Kafka is configured. In order to allow deterministic reprocessing of messages in topics, a log (cf. Section 2.3.2) guarantees that all messages within a partition are ordered. It is important to understand, that the order guarantee does not apply to the whole topic, as each partition has its own log (otherwise the system would not be scalable).

- *Kafka Clients*
  The most common *Kafka Clients* are the *producer* and the *consumer*. The producer writes new messages to a topic (i.e. in a publish/subscribe system it assumes the role of the publisher). Moreover, the producer implements the partitioning logic needed to distribute messages between different partitions based on e.g. a key. In order to guarantee that messages with the same key are sent to the same partition, the key is hashed and mapped to the specific partition (this guarantee expires, if the number of partitions changes). Another task provided by the producer is message serialization. Internally, Kafka handles messages as a byte representation, not the message in its original format (e.g. JSON).
  After a producer pushed data into Kafka, a consumer can pull the data from Kafka and read it (i.e. it assumes the role of the subscriber). Thereby the consumers read data from partitions from one or more topics in the order provided by the log. With the log entry number [83] (cf. Section 2.3.3) the consumers can keep track of the messages they already consumed. Whenever a consumer fails, it just resumes consuming messages from the latest offset. In order to know the latest offset, it must be submitted to be stored by Kafka. How often to submit the offset depends on the required message delivery guarantee, at least once or at most once (cf. Section 2.3.3). Furthermore, each consumer operates as part of a consumer group. Each consumer in a group consumes different partitions of a topic (i.e. two distinct consumers in the same group cannot consume the same partitions). If one consumer fails, the load is spread across the remaining consumers (the submitted offset for the failed consumer is available for the whole group). Similarly, if the load increases to a point where the existing consumers cannot handle it, new consumers can be added to the group. This allows for horizontal scaling of consumers. Of course, as messages are serialized by the producer, there must also be a mechanism for deserialization that converts the byte representation from Kafka back into its original format.
  In general, Kafka enables the decoupling of producers and consumers, so either side can be scaled according to the current needs. For instance, it is possible to have

[83]The Kafka documentation often refers to the log entry number as *offset*

multiple consumer groups that consume the topic of a single producer for different purposes (e.g. stream processing, saving in a database).

- *Broker* and *cluster*
  A *broker* is a single server that contains partitions, handles the messages created by the producer, assigns offsets to the messages, serves the requests from consumers etc. By taking multiple brokers, with each broker having a unique ID, you can form a *cluster* (it is also possible to create multiple clusters). Having multiple brokers is essential for creating a fault tolerant system, which requires to share partitions between multiple brokers. For each partition one broker assumes the role of the *leader*. Brokers that contain the same partitions, but are not elected leader, are called *followers*. However, reading data from and writing data to a partition is only possible through the leader. If a leader is down (e.g. server is down due to a power failure), one of the followers must assume its place as leader. By doing so the system can continue to provide its functionality, even in the event of a broker failure. How many followers a leader has depends on the *replication factor (RF)*. For instance, if we set $RF = 3$, there are three replications of each partition available. In this case we can tolerate the failure of two brokers, as we only require one functioning replica to continue operations. However, fault tolerance comes at the cost of latency. The partition replication between the different brokers consumes time, because each new message that is written to the leader has to await confirmation from all its followers. Consequently, it is important to find the right balance between fault tolerance and the latency requirements of the system. Which broker assumes the role of leader is determined by one *controller* broker per cluster. Usually, the controller is the first broker to be started in a cluster. Managing the brokers in a cluster and electing a controller is done by a different framework called *Apache Zookeeper*[84]. It is important to understand that Zookeeper is not utilized to manage Kafka Clients, as this is done by Kafka itself. Moreover, Zookeeper will be replaced by *Kafka Raft* in Kafka 4.0 (for more details we refer to the official documentation[85]).
  An example for a Kafka set up with three brokers in a single cluster managing two different topics with two partitions each, can be seen in Figure 3.2. In this example, each broker is the leader for a different partition (*Broker 1* for *T1_1*, *Broker 2* for *T1_2* and *Broker 3* for *T2_1* and *T2_2*) and *Broker 1* is the controller. It is not common to make one broker the leader for all partitions, because if this leader-brokers fails you would have to elect a new leader for all partitions (which takes too much time). In order to make the system fault tolerant we set $RF = 3$, which is why each of the three brokers has the data from all four partitions (if we had more brokers, not every broker would contain all partitions). We note that it is possible

[84] *Apache ZooKeeper* 2023.
[85] *KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum* 2023.

that a broker is the leader for partitions from different topics, yet it is not possible to have two leaders for the same topic at the same time.

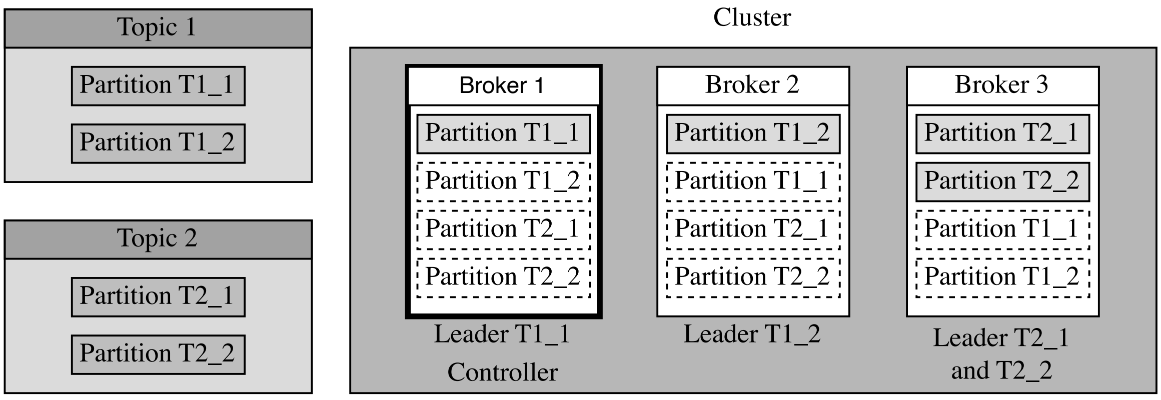


Figure 3.2.: Exemplary illustration how partitions can be distributed and replicated between three different brokers in a single cluster

3.1.2. How data flows through Kafka

One of Kafka's biggest advantage is its vast ecosystem and how flexibly components can be aligned around it. In Figure 3.3 an example for a system built around Kafka is illustrated. The block in the center represents all topics that are stored within Kafka. We note that all topics are partitioned and distributed across multiple brokers in the same way as illustrated in Figure 3.2. On the left side we have two data sources: One is a simulator creating messages at a fixed rate (i.e. a data stream); the other one is a database containing legacy records. On the right side we have two sinks: messages are either saved to a database, or forwarded to a monitoring system. The three data flows illustrating how data can flow from source to sink through Kafka are discussed in the following:

1. The first data flow illustrates a simple example of a data stream from source to sink through Kafka without any processing in between. Thereby a producer writes messages to a topic; in our example *legal records*. The consumer then reads messages from this topic and writes them to a database for e.g. historical records.

2. The second data flow illustrates how a stream of alerts for an external monitoring system can be created by observing a different data stream. The *telemetry* producer pushes messages from the simulator to the topic *tracking*. In order to observe a stream and make predictions, stream processing is required. Because Kafka offers lots of options for connecting stream processors (here we use *Kafka Streams*, cf. Section 3.3), it is easy to implement a stream processor for writing alert messages

to a new topic *alerts*. The consumer *alert tracker* can then read messages from this topic so that they can be monitored by a warning system.

3. The third data flow combines historical and real-time messages to create an enriched stream that can be used for machine learning. Reading messages from an existing database can be done by implementing CDC (cf. Section 2.3.4). Assuming that for our purposes a standard implementation is sufficient, we can re-use existing implementations through a framework called *Kafka Connect*. This Kafka Connect client pushes data to the topic *legacy* before the topics *legacy* and *tracking* are joined via a stream join (cf. Section 2.3.3). Stream joins can be implemented with Kafka Streams, whereby a new stream is created and written to the topic *ML*. From there it can be used for any kind of machine learning we want to implement.

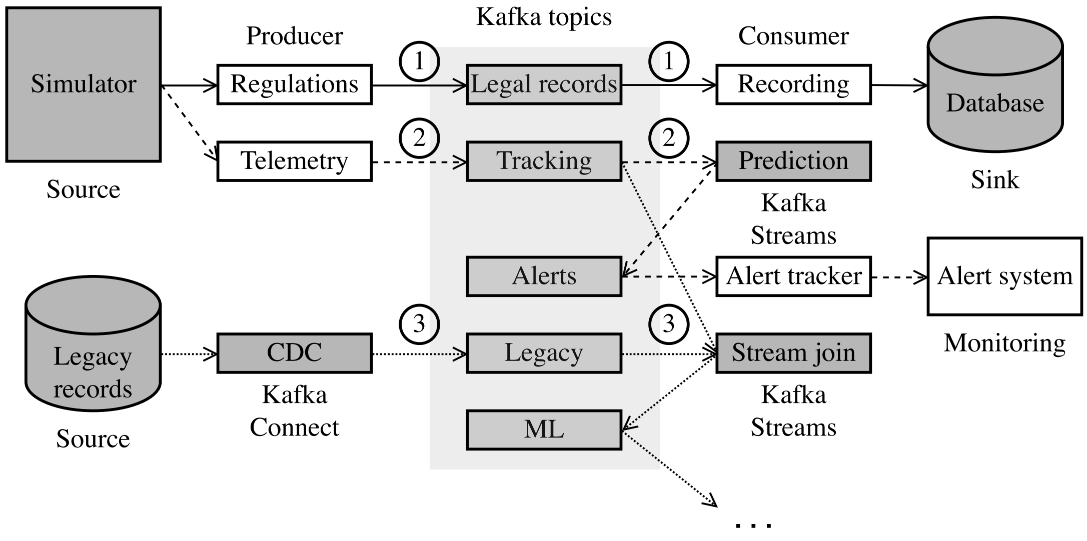


Figure 3.3.: Example for a system with Kafka at its core

As illustrated in the example above, Kafka provides a lot of options for creating a system that is highly flexible. If we require a second database we can simply add a new consumer that reads the desired topic and writes it to the database. If we need to set up a new source we create a new producer that writes to a new topic. In the same manner we can add new processing operations, connect existing databases, use pre-configured connectors from Kafka Connect etc. All this connectivity and flexibility makes Kafka a future proof choice. Additionally, all data is partitioned and distributed between brokers automatically (based on the user's settings) to create a fault tolerant and highly scalable system. The sum of all these features is why Kafka is our framework of choice.

3.2. Database technology

Selecting a database technology depends strongly on the problem-specific requirements, even more so than selecting a data distribution framework. This is due to the large number of open-source options available. On one hand, there are lots of big frameworks for all kinds of different needs. On the other hand, there are also smaller even more specialized frameworks that are often based on the big frameworks. However, like with data distribution frameworks, smaller frameworks are (usually) more difficult to deploy given their lack of community support and demo projects. Consequently, we do not consider them to be viable options for building our database. In the following we will discuss a selected list of potential frameworks before presenting our decision.

- *Apache Cassandra*[86]
  Apache Cassandra is a popular NoSQL, column-based database. With its peer-to-peer architecture it is highly scalable for millions of concurrent writes. Referring to the CAP-theorem, Cassandra is usually associated with AP systems. Considering that we do not require to handle millions of concurrent writes and that we prefer a system that is as close to AC as possible, Cassandra can be excluded from our list of potential database technologies.

- *MongoDB*[87]
  MongoDB is a popular document-based NoSQL database. Due to its master-slave architecture it is highly scalable and it does not require a schema. As opposed to Cassandra, MongoDB typically favors consistency over availability; it is associated with CP systems. Considering that we want to store time series data, using a document-based database does not seem like the most feasible option, as it does not offer support for SQL-queries. As a consequence we exclude MongoDB from our list of considerations.

- *PostgreSQL*[88]
  PostgreSQL is one of the most widely utilized database frameworks for relational databases. It supports SQL queries and ACID transactions. Typically it is associated with CA systems. In total, it checks all our boxes for a database technology (cf. Section 2.4). A potential downside is its steep learning curve. This, however, is not a reason to exclude it from consideration entirely.

- *Influxdb*[89]
  Researching popular database technologies showed the possibility of utilizing time

[86] *Apache Cassandra* 2023.
[87] *MongoDB* 2023.
[88] Group 2023.
[89] *InfluxDB | Real-time Insights at Any Scale* Sat, 15 Jan 2022 15:32:09 +0000.

series databases for data storage. Knowing that all data stored in our database will have a time stamp, this is indeed a viable option. *Influxdb* is the most popular representative of time series databases. It is highly scalable for reads and writes and can store historical and real-time data in the same place. Referring to the CAP-theorem, it is usually associated with CP systems. However, a big limitation of the open-source version of Influxdb is its lack of scalability. This might not be a problem for the first prototype, but a production ready system would (most likely) require the paid version for proper scaling. Consequently, Influxdb can be excluded as potential database technology.

- *Timescale*[90]
  Timescale is a time series database that is based on PostgreSQL (i.e. it is like a PostgreSQL database that is optimized for storing time series data). Consequently, it offers support for SQL queries and ACID transactions. Referring to the CAP-theorem, it is mostly assumed a CP system. As it is based on PostgreSQL, Timescale has an even steeper learning curve, especially without previous experience with PostgreSQL.

Ultimately, the choice for a database technology comes down to either PostgreSQL or Timescale. Assessing the advantages of Timescale over PostgreSQL for our use case is hardly possible without any previous experience. For the moment using PostgreSQL provides more flexibility, as we do not have to stick exclusively to time series data. Considering that Timescale is based on PostgreSQL, it would be possible to change from PostgreSQL to Timescale at a later point in time. Consequently, the decision is made to opt for a traditional relational database built using PostgreSQL.

3.3. Stream processing framework

Our stream processing "framework" of choice is *Kafka Streams*, as it is already included in Kafka (so technically it is more a feature of Kafka than a framework) and provides all functionality we require. It is possible to create windows, maintain a state, execute stream joins and deal with out-of-order data. Of course, there are other, more advanced dedicated stream processing frameworks like *Apache Flink*[91] and *Apache Spark*[92]. However, adding yet another framework would increase the complexity of deploying the prototype significantly. For the moment, Kafka Streams provides a tight integration within Kafka of all techniques we require to implement our stream processing tasks (cf. Section 2.1.2). If more complex processing is required or requirements change, it is still possible to add a

[90] *PostgreSQL ++ for Time Series and Events* 2023.
[91] *Apache Flink — Stateful Computations over Data Streams* 2023.
[92] *Apache Spark - Unified Engine for Large-Scale Data Analytics* 2023.

dedicated framework at a later point in time. For this purpose Kafka provides interfaces for the most common stream processing frameworks.
The following subsections will give a quick overview of scaling in Kafka Streams, and how state can be maintained. We note that all information in these subsections is either from the the official Kafka Streams documentation[93] or the guide book *Kafka: The Definitive Guide, 2nd Edition*[94].

3.3.1. Scaling in Kafka Streams

In order to understand how Kafka Streams can scale, it is essential to know what a stream processing application is and what it does. A stream processing application is a Java application that is built using Kafka Streams' library. The underlying processing logic can be described through a graph consisting of nodes and edges. Nodes represent stream processors (cf. Section 2.3.3) that describe a processing step in the data transformation process (e.g. filtering, mapping). These nodes are connected through edges that represent the data stream. A stream processing application reads data from topics and transforms a data stream with one or more stream processors before the last node writes the output to a new topic or an external system.
An example of how one stream processing application that consumes two topics can be scaled across two threads is illustrated in Figure 3.4. In general, scaling is done by splitting a stream processing application into multiple tasks. The tasks share the processing workload by reading from different partitions from one or more topics. This means that the maximum number of tasks is determined by the maximum number of different partitions in the consumed topics. In our example the maximum number of partitions in all topics we consume is 3 in *topic 1*. Because *topic 2* with two partitions is consumed by the same stream processing application, we can split the stream processing application into three tasks, not five. Depending on the number of threads available, Kafka Streams tries to balance the workload as good as possible. In our example *thread 1* processes *task1_1* and *task1_2*, whereas *thread 2* processes *task1_3*. Because Kafka can scale horizontally, threads do not have to be in the same machine. This means that in our example we have the option to add a third thread from a different machine if we require better performance (i.e. one task from *thread 1* is assigned to the new thread). Having more threads than tasks (and therefore partitions) would not make any sense from a performance point of view. The additional threads would remain in idle mode until another thread fails and they have to pick up its work. The dependence of multi-threading on the maximum number of partitions highlights how important it is to define a suitable partitioning logic in Kafka.

[93]*Kafka Streams Documentation* 2023.
[94]Shapira et al. 2021.

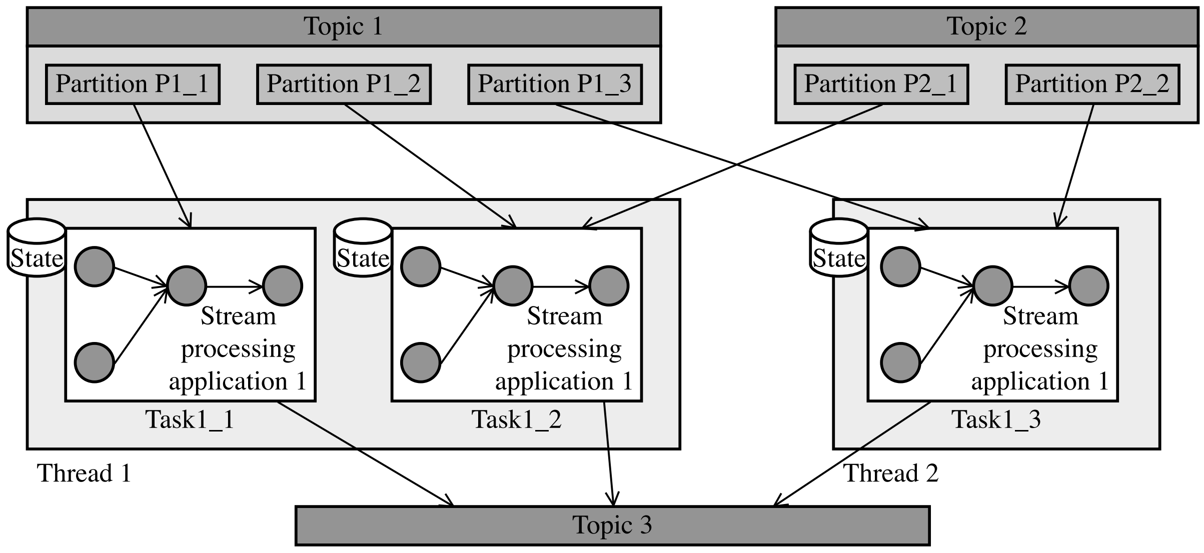


Figure 3.4.: Illustration how stream processing of two different topics can be scaled across multiple tasks and threads in Kafka Streams

3.3.2. Stateful stream processing in Kafka Streams

In general, there are two options for saving state in Kafka Streams. One option is to save the state locally in an in-memory database; the other one is to store it in an external system. Because saving state externally would require to set up yet another system, our prototype will work exclusively with local state. Local state has the advantage of being way faster than external state (cf. Section 2.3.4). However, storage for local state is limited, and memory spilling to disk would cause significant performance losses. This is something to keep in mind when testing the performance of the system. Of course, if necessary it is always possible to complement Kafka with an external state storage system later.

Coming back to Figure 3.4, we can see that each task requires its own local state. This local state is automatically saved by Kafka Streams in-memory utilizing the embedded *RocksDB*[95]. To ensure that local state is not lost in case a stream processor fails, changes to the local state are stored in a topic. Like all other topics in Kafka, they are partitioned and distributed across multiple brokers. This means that after the failure of a stream processor local state can be restored by reading all changes from the topic. This is done automatically, which is why it is reasonable to argue that the processing is fault tolerant. Moreover, log compaction (cf. Section 2.3.3) makes sure that topics do not grow too big. Finally, it is important to understand that the state of the entire system is distributed across all local state storages. It is not possible (nor feasible) to keep only one local state for all tasks.

[95] *RocksDB | A Persistent Key-Value Store* 2023.

4. System architecture and dataflow

This chapter begins with a section that describes the general architecture, and the data flow between the different components of the prototype. Further, it is explained, how a test environment for the prototype is deployed, and how all components within the test environment can communicate. The following section elaborates the data flow specific for stream processing in more detail and provides insights into the implementation of the stream processors. Finally, a design for a database for long term storage is presented.

4.1. Architecture

As explained in Section 2.3, the three main components of our prototype are data ingestion, stream processing, and storage. However, in order to build our system we require additional supporting components for certain tasks. In our case supporting components include a schema registry, Kafka Connect and monitoring frameworks. The schema registry is explained in more detail in Section 4.1.1. Kafka Connect can be used to connect existing databases to Kafka, or to get data from Kafka into a database. Because we have decided to implement a relational database for longterm storage with PostgreSQL, it is sound to use an existing solution built for Kafka Connect (cf. Section 4.1.2). Moreover, monitoring is not built into Kafka, yet it is essential to keep an overview of all topics, the number of written messages, etc. Similarly, we have to keep track of the data written to our relational database. For this reason we can use existing external solutions that simplify any monitoring. A general overview of all components and their connections can be seen in Figure 4.1. We start by producing data from data streams (telemetry messages) and existing databases (e.g. registered missions, geofences) into Kafka. To process the data we use Kafka Streams, whereby the data is consumed from Kafka, processed, and then written back into Kafka. To get the data from Kafka into our relational database we utilize an existing connector from Kafka Connect. All de-/serialization between the different components is handled with the help of the schema registry. Finally, because Kafka and our relational database are two distinct components, we implement a monitor for each of them.

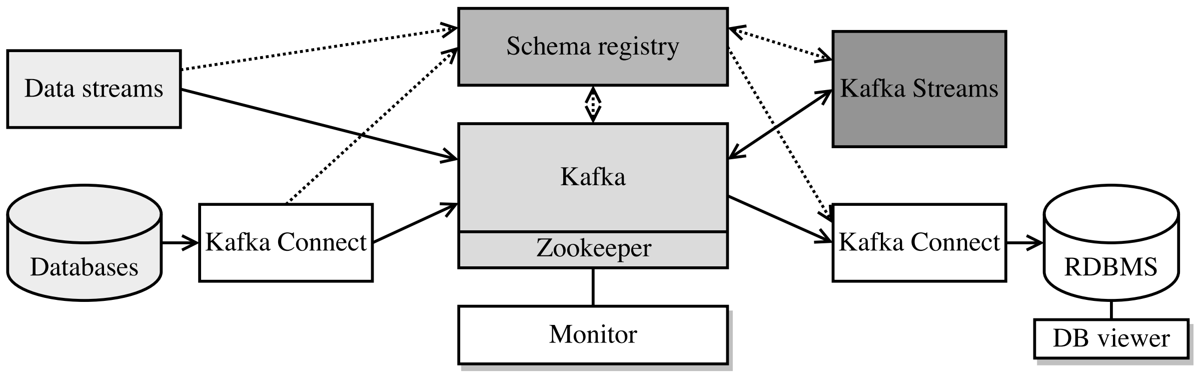


Figure 4.1.: Simplified overview of the prototype's architecture

4.1.1. Schema registry

The *schema registry* is an external (to Kafka) component that is utilized to simplify the process of de-/serialization of data sent between different components and Kafka. As explained in Section 3.1.1, Kafka handles all data as a byte representation, which makes de-/serialization an absolute necessity. By default, Kafka implements a set of de-/serializers for generic data types such as *String*, *Integer*, etc. However, we are working exclusively with JSON messages. For de/-serializing JSON messages there are two options: Either the JSON message, is converted into a *String* which is then de/-serialized with a standard de-/serializer, or a custom de/-serializer working with a schema (describing the structure of the JSON) is utilized. The advantage of using a custom de-/serializer is the ability to control which data gets into the system, and how the data evolves. For instance, if a new field is added to our JSON messages, our system would recognize that it does not comply with the existing schema. Consequently, we can update our schema, so the system knows that the messages look different now. Without a schema, we would not know that the message structure changed. Upon deserialization we would be greeted with an additional field, not knowing what to do with the data. This means that in order to de-/serialize JSON messages while controlling evolving data, we require some schema that describes how our messages are structured. For this purpose we utilize a popular tool called *JSON schema*[96]. Alternatively, Kafka also offers support for *Apache Avro*[97], and *Protobuf*[98]. However, as we will work exclusively with JSON, it is reasonable to select a JSON schema that has been specifically developed for this purpose.
In order to use a schema for de-/serialization it must be stored in a place where it is available to all consumers and producers. One possibility is to attach the schema to each message. In this way, it is available throughout the whole data flow. However, this creates an additional storage overhead because the schema is included in each message

[96] *JSON Schema* 2023.
[97] *Apache Avro* 2023.
[98] *Protocol Buffers* 2023.

that is sent to Kafka. So if we send 1000 messages we have to save the same schema a thousand times. For this reason we use the schema registry for storing one schema upon serialization. When one component (e.g. a consumer) reads data from Kafka it simply retrieves the same schema and re-uses it for deserialization. How the schema registry can be utilized is exemplified in Figure 4.2. We can see that upon producing a message, first, the schema is sent to the schema registry. If the same schema already exists in the registry, nothing happens, otherwise the updated schema is saved as a new version. In this way, we only have to store one schema for each version of a message. Next, the serialized data, in the form of a byte representation, is then sent to Kafka. From Kafka the serialized data is read by a consumer, who retrieves the schema from the schema registry to deserialize the data for further use.

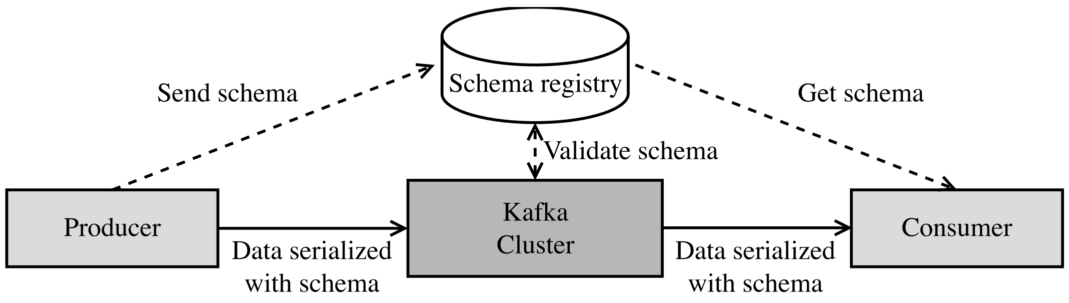


Figure 4.2.: Illustration of the schema registry's functionality

We note that it is also possible to upload a pre-defined schema to the schema registry to have more control over the schema uploaded by the producer. Moreover, in case the structure of the message should change at some point, it is possible to evolve the schema while maintaining backwards compatibility with older schemas. Regarding fault tolerance, the schema registry (basically) is a Kafka topic for storing schemas. Consequently, it is possible to apply the same replication techniques as for ordinary topics (i.e. define a replication factor $RF \geq 2$).

4.1.2. Kafka Connect

In order to get data from external databases into Kafka, and from Kafka into external databases in a scalable manner *Kafka Connect*[99] can be utilized. Kafka Connect uses so-called *connectors* (for source or sink) to define where the data comes from and goes to. An instance of a connector can be viewed as a logical job for the exchange of data between Kafka and an external storage system. Depending on our database schema we can start multiple instances of the same or different connectors (i.e. we require at least one connector instance for each table in our database). Because we are working with a widely applied framework like PostgreSQL we can choose between numerous existing

[99] *Kafka Connect | Confluent Documentation* 2024.

connectors that can be installed as plug-ins. In our case we can use a *JDBC-connector (Java Database Connectivity)*[100] for source and sink that is compatible with PostgreSQL. If however a more customized database solution is implemented it is also possible to develop a customized connector. Each connector instance is divided in to a set of *tasks*. By defining a maximum number of tasks we can determine how many tasks are available for load balancing; i.e., for tables where we expect a high number of consecutive writes (or reads) we allow more tasks. Of course, the tasks are fault-tolerant as their state is stored in Kafka topics. In order to execute connectors and tasks Kafka Connect utilizes processes called *workers*. Thereby it is possible to execute workers either in standalone or distributed mode. In standalone mode one worker executes all connectors and tasks. However, this does not allow for any scalability. Consequently, we work in distributed mode where many worker processes execute tasks and connectors. In order to handle the de-/serialization between Kafka and Kafka Connect, Kafka Connect provides converters that are compatible with the schema registry. In cases where small changes must be applied to a message, Kafka Connect provides a series of simple transformations (e.g. replace the existing key with fields from the record value). However, the possibilities of these transformations are limited and should not be viewed as substitute for Kafka Streams (cf. Section 3.3).
We note that all information from this subsection is sourced from the official Kafka Connect documentation from *Confluent*[101].

4.2. Deployment

Deploying Kafka brokers natively on one or more servers is a (presumably) long and intensive process. Because time and resources are limited, it does not make sense to set up a native broker deployment for testing the prototype. For this reason we require a technology that enables Kafka brokers to run on a local machine for development, while also allowing to easily switch to a different machine (i.e. a server) for performance testing. A popular option for this purpose is *Docker*[102], a platform that can be used to build and run containerized applications. Here, a container can be described as an isolated entity that includes everything required to run an application. In order to prepare a container for running an application it can be set up manually, or existing *image*-files can be downloaded from the *Docker-hub*[103]. In general, a container is similar to a *virtual machine*, yet a container does not have its own OS kernel, therefore it requires less resources, thus simplifying the development process and testing. Moreover, when switching to a different machine, we require only a Docker installation on that machine to

[100] *JDBC Connector (Source and Sink) for Confluent Platform* 2024.
[101] *Kafka Connect | Confluent Documentation* 2024.
[102] *Docker* 2022.
[103] *Docker Hub* 2023.

run all containerized applications without having to apply any changes. Applications with multiple containers can be defined in a single *Docker-compose* file (with the file ending *.yaml* or *.yml*). The communication between the different containers is managed through ports (how all containers are connected is defined in Section 4.2.1). In our case, we only need one Docker-compose file to define containers for all components such as brokers, schema registry, monitors, etc.
Of course, preparing Docker containers alone is not enough as we still need to implement our own coding logic in *Java* (an overview of all applications implemented is given in Section 4.2.2). All our producers and Kafka stream processors are Java programs written in an *Integrated Development Environment (IDE).* Within this IDE we can initialize all our containers, and run our Java programs. Like the interaction between different containers, the interaction between our Java producers/processors and containers is done through ports (cf. Section 4.2.1). Because the Java programs rely on the containers it is important to start the containers before the programs. We note that producers and processors are not the only applications that interact between Docker and Java programs. Other typical use cases include an admin client that can create new topics, a schema uploader for the schema registry, or a customized consumer.
In general, the process of developing, and testing our prototype within an IDE can be described as follows:

1. The first step is to define containers for all components in a Docker-compose .yaml-file. It is important to correctly specify in which sequence the containers are started (e.g. you must start zookeeper before the broker; cf. Section 4.2.1), which ports are open, and to define the properties accordingly. Of course, to prepare the container for running the application, the correct images must be downloaded. The specified image is downloaded automatically upon starting the container.

2. The next step is to write all Java programs required (e.g. stream processors, producers, admin client, ...). Moreover, *.properties* files must be defined to specify the settings for the producers, processors, as well as topic names, ports etc. Defining properties in external files greatly simplifies the change process for settings that are re-used for multiple Java applications (e.g. broker connection, schema registry). If they were defined directly in the Java applications, every small change would require to manually adapt all Java applications to the new setting.

3. After the preparation of all containers and Java programs, the containers are started. Upon the first start of a container instance the image is pulled from the Docker-hub. If no settings are changed container instances can be stopped and started effortlessly. Albeit, the dependencies between the containers have to be considered. If, however, settings for a container are updated, it is necessary to delete the old container instance (and all depended instances of other containers) before re-building

it. We note that the images are not downloaded again after deleting an instance of a container, as images are stored in a separate directory.

4. Finally, all Java programs can be started. Again, it is important to stick to the right sequence (e.g. do topic creation and schema registration before producing and processing data).

4.2.1. Docker set-up

All communication between components within Docker and components external to Docker is done via ports. However, because of numerous different components in our set-up (cf. Figure 4.1) it is difficult to keep track of all connections. For this reasons, an overview of our Docker set-up with an exemplary external connection is illustrated in Figure 4.3.
We can see that the broker is the most important component, as every external component and many of the Docker-internal components connect to it. In order to allow external and internal connections we provide two different ports. In our case, port 19092 is used for external, 9092 for internal connections. Furthermore, we open port 9997 to provide access to the *Java Management Extension (JMX)* metrics which can be utilized for monitoring Java applications.
In general, to establish connections with Docker, externally *localhost:port-number* and internally *container-name:port-number* must be accessed. Of course, instead of *container-name* and *port-number* we have to enter the values defined in Docker.
Another component that has internal and external connections is the schema registry. As it is essential for de-/serialization, it is reasonable to connect it externally to producers, stream processors, and internally to Kafka Connect, and the broker. Yet, unlike the broker due to technical specifications the schema registry can operate with only one open port for internal and external connections.
The last component in Docker with external connections is *Prometheus*[104]. We require Prometheus to gather all data from the open JMX ports of our components for monitoring. In order to make metrics available at the ports we require a *JMX-exporter*[105] that is executed together with the Java applications we want to monitor. Without a jmx-exporter Prometheus would not be able to collect any data at the specified ports. Because the jmx-exporter runs together with the Java applications, it is not illustrated in Figure 4.3. However, it would be possible to run the jmx-exporter within Docker. Yet, for reasons of simplicity, the jmx-exporter is executed together with the Java applications. Finally, to visualize our metrics, we utilize a tool called *Grafana*[106]. Grafana takes the data gathered by Prometheus and visualizes it in customized dashboards. One big advantage

[104]Prometheus 2023.
[105]*Prometheus/Jmx_exporter* 2023.
[106]*Grafana* 2023.

of Grafana is the availability of many pre-designed dashboards. Of course, if required, it is possible to design dashboards that are entirely customized. However, to create a customized dashboard, it is essential to have an overview of all available metrics. It is possible to view all metrics available by accessing the *JConsole*[107] (this tool is included in Java and does not require further steps of installation). As the JConsole is used only in the beginning to examine available metrics, it is not illustrated in Figure 4.3.
The rest of the components in our Docker-compose set-up are related to the interaction with Kafka and our relational database. In order to interact with all our components we utilize the tool *Kafka UI*[108] which is connected directly to nearly all internal components. As explained in Section 3.2, we use PostgreSQL for implementing a relational database that stores our historical data. The content of this PostgreSQL database can be accessed with the help of *PgAdmin4*[109]. Because it is mandatory to define user-name and password, some generic placeholders are entered.
We note that *graphical user interfaces (GUIs)* for Prometheus, Grafana, Kafka UI, and PgAdmin4 can be accessed through any web-browser in the same network by calling *localhost:port-number*. For instance, Grafana is available on port 3000, so the GUI can be accessed by entering *localhost:3000* in a browser on our machine. Of course, all the other components can be accessed the same way. However, they do not have a dedicated GUI.

Components start sequence

All Docker containers must be started in the right sequence to comply with different container dependencies. The correct start sequence for all our components from Figure 4.3 running in Docker containers is illustrated in Figure 4.4. We can see that we have three (mostly) independent start sequences: *Kafka*, *database*, and *monitoring*. This is reasonable, as we do not want our database, monitoring service, and Kafka set-up to depend on each other. By running them independently we (try to) evade complete system failures. For instance, if Kafka fails it is important that our monitoring service is still up and running to detect the failure immediately.
The individual start sequences themselves are pretty straightforward. In Kafka brokers depend on Zookeeper (cf. Section 3.1.1), which is why Zookeeper must be started first. The schema registry stores schemas in Kafka topics, therefore it requires a working broker. Similarly, Kafka Connect requires a schema for de-/serialization; consequently it is started after the schema registry. Because Kafka UI interacts with all these components, it is started last. For the database sequence, Postgres must be started before PgAdmin4 (you cannot use the database viewer without a database). Finally, Grafana requires data from Prometheus to create visualizations, so Prometheus must be started first.

[107] *Using JConsole - Java SE Monitoring and Management Guide* 2023.
[108] *Kafka UI Provectus* 2023.
[109] *pgAdmin4* 2023.

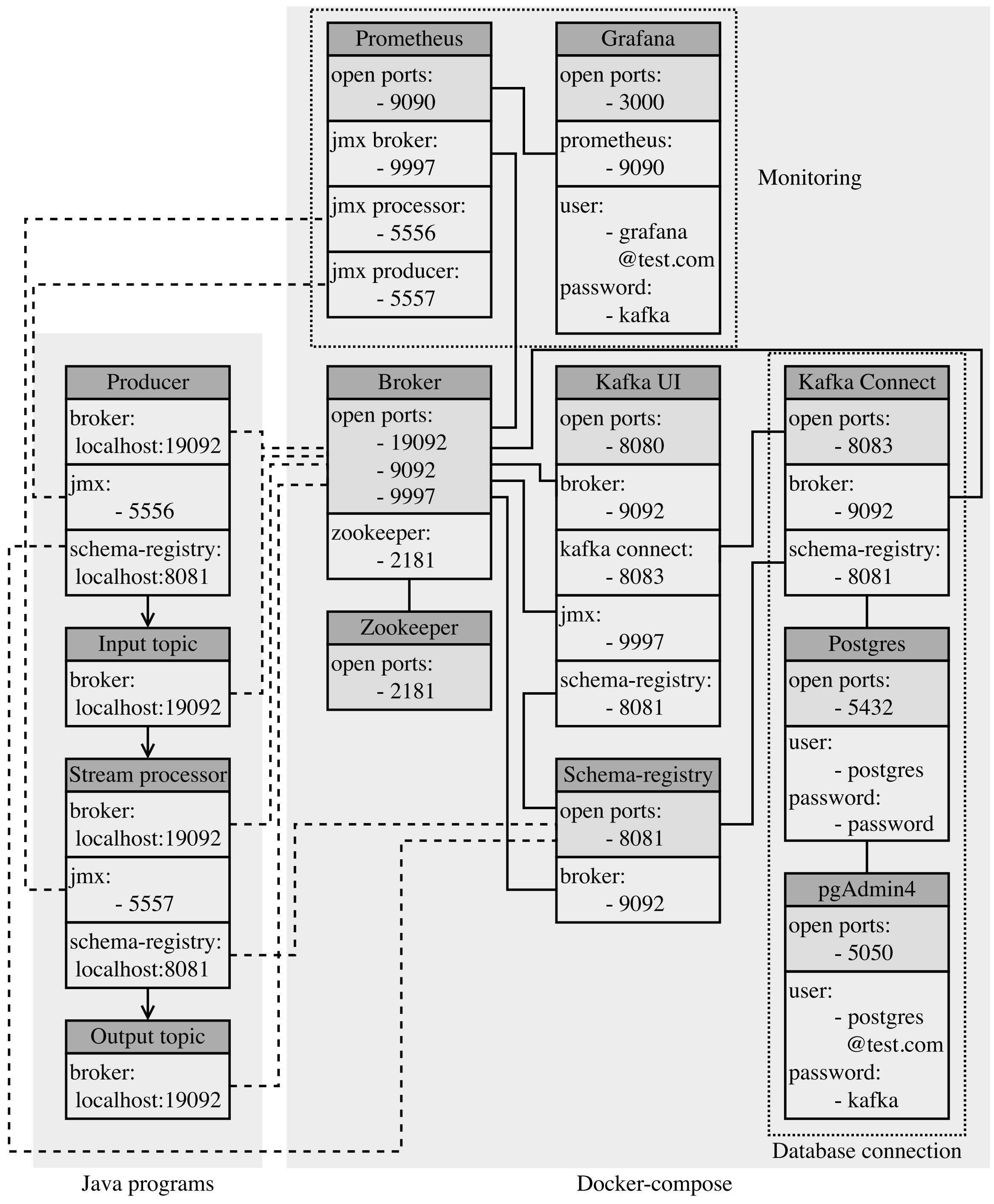


Figure 4.3.: Illustration how all components from the prototype are connected through ports

We note that despite the independence of the different sequences, it is recommendable to start all of them before executing any Java applications for producers, stream processors, or consumers. Even if the most important sequence is Kafka itself, the simultaneous start of all sequences before the execution of Java applications helps to reduce the susceptibility to errors.

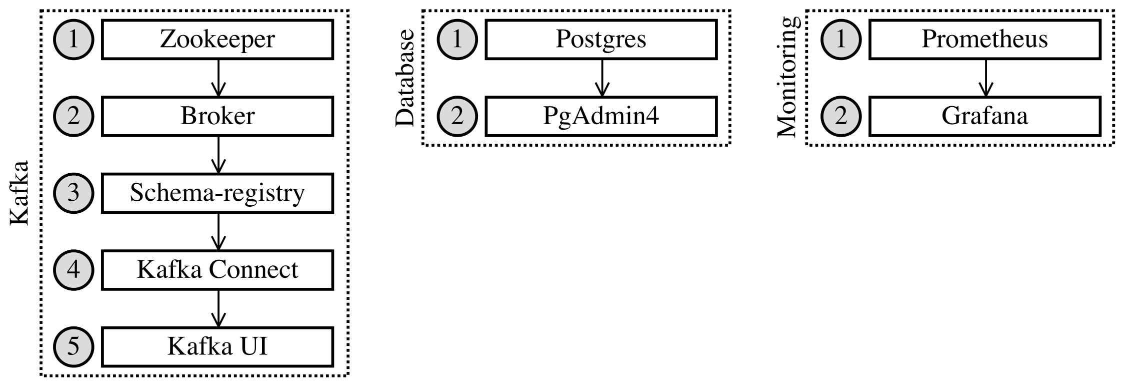


Figure 4.4.: Start sequence dependencies of all components running in Docker containers

4.2.2. Java applications

As mentioned in the previous sections, the logic for our producers, stream processors, and consumers is implemented in Java applications. An overview of all classes and their methods is given in Appendix A. These class diagrams do not include any information about the simulator used (cf. Section 4.2.3), except for the interface it is connected to. In general, our prototype includes Java packages for producers (cf. Appendix A.3), stream processors (cf. Appendix A.4), and one for a variety of tools (e.g., uploading a schema, creating a new topic; cf. Appendix A.1). In addition to our main, we use two supporting classes; one for the simulator and one for starting Kafka with all connectors and topics (cf. Appendix A.2). Each package contains all the relevant classes associated with the given task. We are not giving an explanation for each package with all its classes, as the comments in the code should suffice to understand the coding logic.
At this point we want to note that *ChatGPT* is used extensively throughout the development process to create simple scripts or get a general idea for how to structure Java programs that are created with previously unknown frameworks. However, the structure of the files, the design of the dataflow, in general, all decision that concern the design of the system are made without ChatGPT.

4.2.3. Simulator

In order to test the prototype it is connected to a simulator. With this simulator we can create any number of drones, operators, missions, and telemetry messages based on the requirements of our tests. The interaction with this simulator is managed through an interface that includes function calls to our Kafka producers and our connected database. For instance, if a new telemetry message is created by the simulator, it is handed to the Kafka producer through the interface, which produces the message to the target topic. Similarly, if the simulator creates, updates, or deletes a mission, we send an SQL-statement to our database which executes the update on the corresponding row in the table.

In general, any interaction between the prototype and the simulator is managed via the interface. Apart from the settings controlling e.g. the number of telemetry messages no changes are made to the simulator. As designing a simulator is not part of this thesis' scope, neither the simulator nor any of its modules are described in more detail.

4.3. Stream processing

The first part of this section describes the general design of the dataflow between the data sources, Kafka, and the stream processors. This part offers an overview of the created topics and their relation with the respective stream processors. Also, the dependencies between the different stream processors are highlighted. The second part focuses on the implementation of the stream processors, describing the general processing logic, encountered difficulties, and how they are solved.
We note that the first subsection describes the whole system, whereas the second part describes only the tasks that are actually implemented (i.e. C_1, C_2, and C_3) for the tests in Chapter 5. The reason behind this is that the implementations of C_1, C_2, and C_3 are considered to provide enough evidence for the feasibility of our prototype, as the require us to deal with stateless and stateful operations, data partitioning, source and sink database connections etc.

4.3.1. Stream processing data flow

Our prototype has to provide five stream processing tasks (C_1 enrich telemetry data, C_2 FP correlation, C_3 conformance monitoring, C_4 conflict detection, C_5 spatial clustering; cf. Section 2.1.2). Knowing that these five tasks receive data from four different sources (a telemetry data stream, as well as geofence, flight plan, and unregistered UAS databases), we can construct an abstraction of our streaming dataflow within Kafka. In Figure 4.5 we can see an illustration of the dataflow centered around Kafka. The general procedure for processing always remains the same: the initial data is produced to topics, a stream processor reads data from one or more topics, processes it, and writes it to one or more new topics.
Although the general design of the dataflow cannot be altered, there are two possible variations that could lead to different results in performance (something that remains to be tested).

1. The output from C_1 is written to the topic *Enriched telemetry*. Because the only consumer for this topic is C_2 it would be possible to combine C_1 and C_2 into one stream processing application that is executed by the same threads (cf. Section 3.3.1). The way it is sketched in Figure 4.5 C_1 and C_2 are two distinct stream processing applications, which means that they are executed by different threads. If running these

tasks as separate threads is advantageous can only be shown by performance tests. The same idea applies to C_4 and C_5, because the output topic from C_5 is consumed only by C_4.

2. The second alternative is not so much related to the dataflow, but to the partitioning of the telemetry data stream. As explained in Sections 3.1.1 and 3.3.1, the question of how to partition data in Kafka is an essential one. Depending on the data at hand there are different options for the initial partitioning available (e.g. based on ID, operator, spatial partitioning). Of course, the initial partitioning can be modified by stream processing, yet for some tasks it might be beneficial to apply a certain partitioning strategy right from the beginning. Our default partitioning will be based on a unique UAS identification number. However, which partitioning strategy is ideal (or if we require more than one strategy) for our prototype remains to be tested.

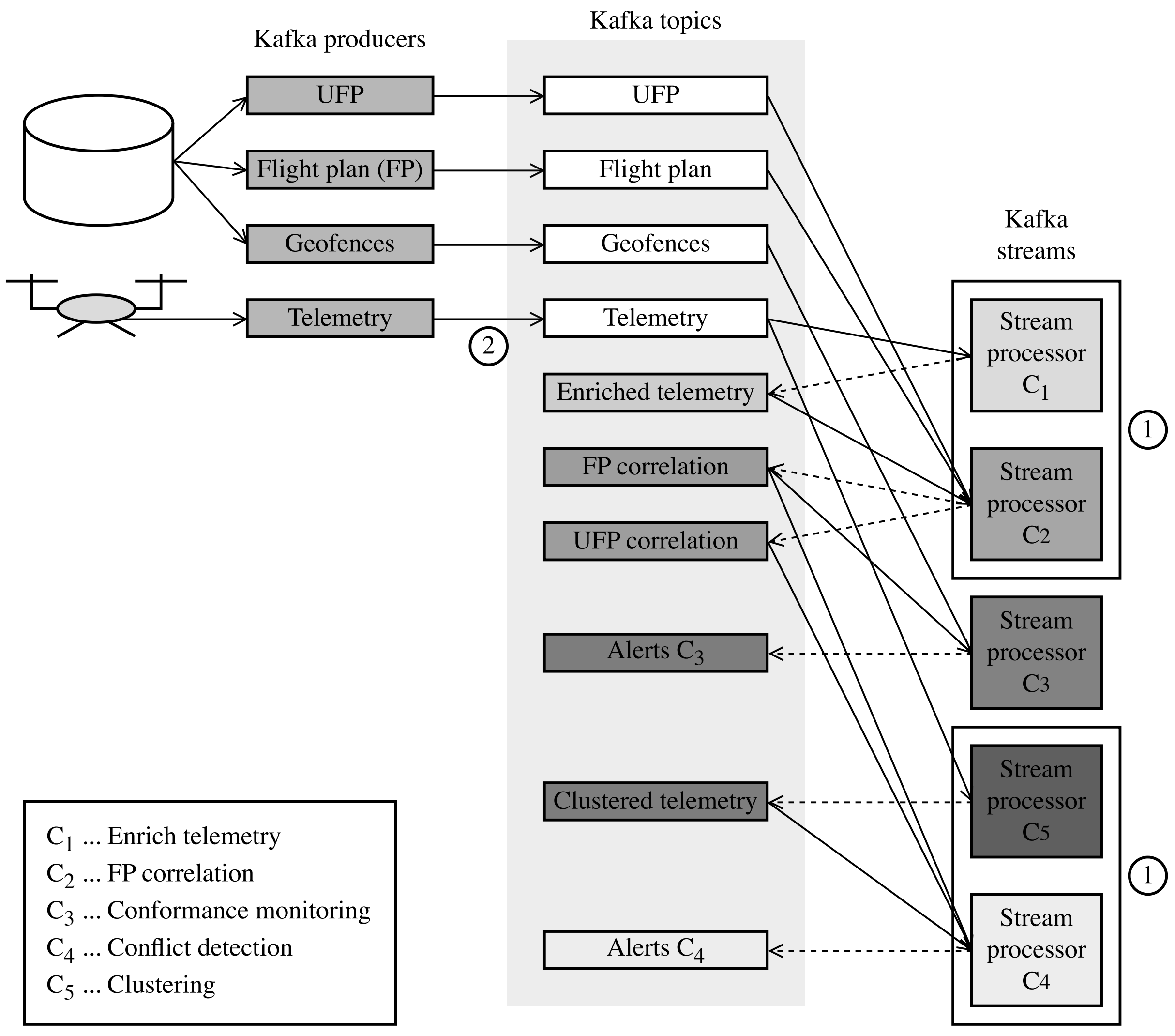


Figure 4.5.: Illustration of the streaming dataflow centered around Kafka

An alternative illustration of the dataflow can be seen in Figure 4.6. Here, the dataflow is not built around Kafka, but illustrated as a *directed acyclic graph (DAG)*. A DAG is a graph consisting of nodes and edges, where all nodes are connected through edges with a given direction. Furthermore, a DAG does not allow any cycles (i.e. it is not possible to find a directed path where the start and end node are the same). The illustration as DAG emphasizes the flow of data in a single direction; it is not feasible that a stream processor consumes data from and writes data to the same topic. Similarly, no stream processor consumes data from any downstream stream processors. Moreover, this illustrations shows more clearly, how the different processing tasks depend on each other. For instance, the processing tasks C_3 and C_4 depend on the results from C_2, which in turn depends on the results from C_1.
We note that for this illustration we distinguish between three kinds of nodes. This, however, serves only the purpose of highlighting similarities with Figure 4.5; all kinds of nodes are to be considered equivalent.

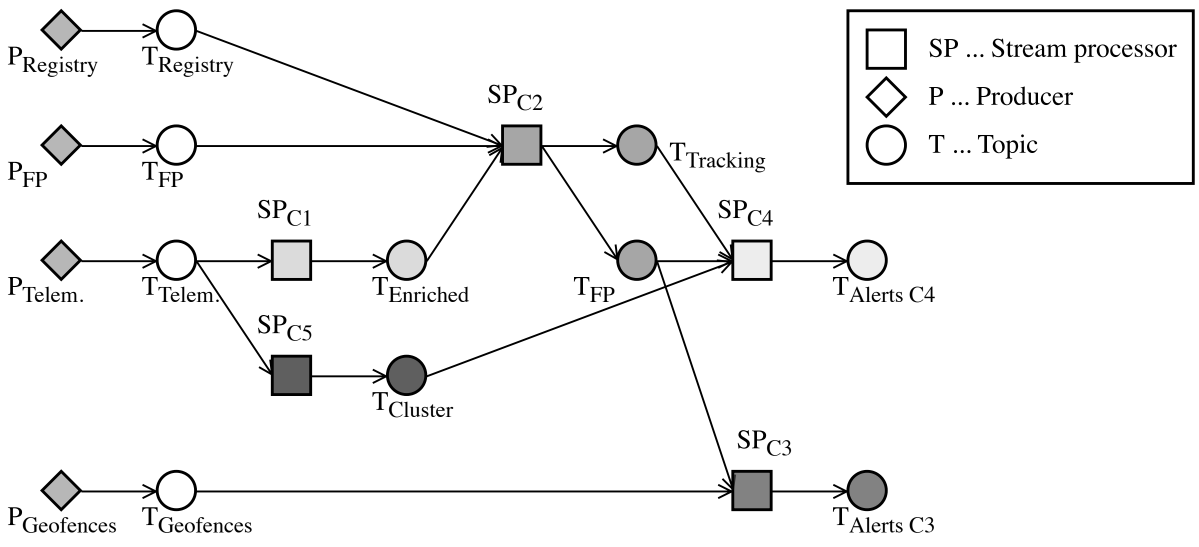


Figure 4.6.: Illustration of the streaming dataflow as DAG

4.3.2. Stream processor implementations

Kafka Streams offers two levels of *API (Application Programming Interface)*. At a high-level API, the *Kafka Streams DSL (Domain Specific Language)* provides a series of common functions (e.g. *map*, *filter*, *join*) that are easy to implement, but do not offer a lot of customization options. At a low-level API, the *Processor API* offers more flexibility than the Kafka Streams DSL, especially for working with state (cf. Section 2.3.3). Because the Processor API is the basis for the Kafka Streams DSL it is possible to use a combination of both[110]. This is ideal for our prototype which requires standard functions for simple

[110] *Apache Kafka Documentation* 2023.

processing operations like C_1 , but needs to handle state for more complex operations like C_2 and C_3:

- C_1 *Enrich telemetry data*
  Updating the altitude for each telemetry message individually is a stateless transformation that can be implemented with the high-level Kafka Streams DSL. The stream is read from the topic that is fed with telemetry messages from the simulator. This stream is then processed into a new stream with updated messages that are saved to a new topic for downstream use. All messages are updated via a module from an external library.

  We note that all modules from the external library used for the computations of either C_1, C_2, or C_3 are not developed as part of this thesis.

- C_2 *FP correlation*
  In order to correlate FPs with telemetry messages we need to connect an existing database that contains all FPs. We save all FPs in a PostgreSQL database and keep a copy of the whole database in a Kafka topic. To get the data into Kafka we utilize Kafka Connect. Because we have to perform a look-up for a corresponding FP for each telemetry message we save the whole topic in a state store which can be queried by our stream processor. This state store keeps only the most recent key-value pairs; in our case the ID of the FP (key) and the FP itself (value). To access the state store during processing we have to use a customized processor implemented with the Processor API. In this customized processor we access the current state of our FP database and call the external module to perform the correlation between a FP and a telemetry message based on the ID.

  We note that it is also possible to query a Kafka topic directly without using a state store. A popular option besides Kafka Streams that allows to query a data stream with SQL-like syntax is *ksqlDB*[111]. However, in our case this implementation would not be feasible as we are bound to an existing module through an interface. This interface explicitly demands a *Java Collection<E>*[112] of FPs as input. Yet, when querying from a topic we cannot create collections easily, as this is not the intended way of using streaming queries. Queries of streams are usually executed continuously as new messages of a potentially unbounded stream arrive. Turning them into a Collection is possible, but requires a work-around. We could take the topic as input and turn it into a stream. We filter this stream and save the results to another topic; then we consume this topic with a different stream processor that saves each message into our Collection. This method is overly complicated and not sustainable for large numbers of UAS as it would create lots of one-use-only topics in

[111] *ksqlDB: The Database Purpose-Built for Stream Processing Applications.* 2024.
[112] *Collection (Java Platform SE 8)* 2024.

Kafka. Consequently, we have to work with a customized solution that is compatible with the existing module. For future iterations of the system it is recommendable to consider implementing the external modules natively using the API provided by Kafka Streams.

The second task performed by C_2 is checking for a loss of signal. In order to do this we have to keep the latest message received from each UAS in a state store. When a new message arrives, we access the state store to check when the latest message was received. We compare the times of the latest and the current message (t_{latest} and $t_{current}$) to see if $t_{\Delta max} \geq t_{current} - t_{latest}$. If so we mark the signal of the current message as reset, so we know it is either the first message received, or there was a loss of signal. Moreover, we have to check the state store regularly to automatically send a warning if for some period of time t (which does not have to equal $t_{\Delta max}$ necessarily) no signal is received. By implementing a *Punctuator* we schedule automatic monitoring of our state store based on the system time. If an old message with a correlated FP is received the Punctuator sends a message to the corresponding topic in Kafka. Like performing a look-up for a FP, checking for a loss of signal is performed through a customized processor with a state store. It would be possible to perform both computations with one processor, but for the sake of modularity two separate processors are implemented.

One difficulty with using a state store in combination with a schema registry is that the first access to the state store must be a write because schemas are registered upon the first write. If we try to read from a state store without writing to it first the serialization fails because the serializer cannot find a schema in the schema registry. For cases where it is not possible to write to the state store first a schema must be registered in the schema registry manually.

- C_3 *Conformance monitoring*
  Checking if telemetry messages are in conformance with an existing FP requires to keep the flown trajectory for each UAS in state. Similar to the implementation of C_2, the interface for the module for C_3 requires a the flown trajectory as Collection as input. Consequently, we use the Processor API to create another customized processor. This processor reads only from the topic with correlated telemetry messages to check the adherence. For each correlated telemetry message another message is written to a different topic to see for each message if adherence exists. In order to automatically create alerts this topic can be monitored by another processor to send an alert if adherence is lost.

4.4. Database connections

The prototype can connect tables in our PostgreSQL database as source or sink for Kafka through Kafka Connect (cf. Section 4.1.2). An overview of all tables in our database can be seen in Figure 4.7. All tables are created automatically upon the initialization process of PostgreSQL when a script with SQL commands is executed. For each table in our database we require a JDBC-connector from Kafka Connect (cf. Section 4.1.2); in our case we can use multiple instances of the same connector, as all our tables are in PostgreSQL. In the source tables we store all FPs and UFPs; geofences are not included, as they are not implemented in the simulator. In the sinks we store all telemetry messages that are correlated with FPs, and any alarms (i.e. adherence with the FP, loss of signal) sent by the system (cf. Section 2.1.2). Also, all telemetry messages that cannot be correlated with a FP are saved because of regulatory requirements (cf. Section 2.1.1). The content of the tables is based on the attributes of the Java objects that are provided by the simulator. Our table for the correlated telemetry includes references to the FP or UFP, as well as an information about the signal (i.e. if we receive a message after a signal loss we set signal_reset to true). For the signal loss table we store references to the FP or UFP and the signal boolean which is false as soon as no signal has been received for some time. In the adherence table we keep only the reference for the FP as we cannot check the adherence for an uncorrelated UAS. For each telemetry message with a correlated FP we check if it has adherence, if not we calculate the delay. It is essential that the structure of these objects matches the tables' columns exactly. This also applies to the data types; if structure and data types do not match Kafka Connect cannot save any data in the table. This is where the schema registry (cf. Section 4.1.1) comes in handy, as it allows us to define schemas for the structures and data types of all our objects. If, however, the schema from the registry does not match the object or the table in the database, the de-/serialization fails immediately. Neither Kafka Connect nor PostgreSQL provides a lot of flexibility for altering objects or schemas (except for e.g. changing the primary key). The only way to correct any mismatch between objects, schemas, or tables is to transform the objects via stream processing. In order to evade this inconvenience, it is important to have consistently defined objects, schemas, and tables along the whole dataflow from source to sink.

We note that in order to use foreign keys with PostgreSQL and Kafka correctly the design of the dataflow is important; the parent table with the original data must be populated before the child tables that hold the references.

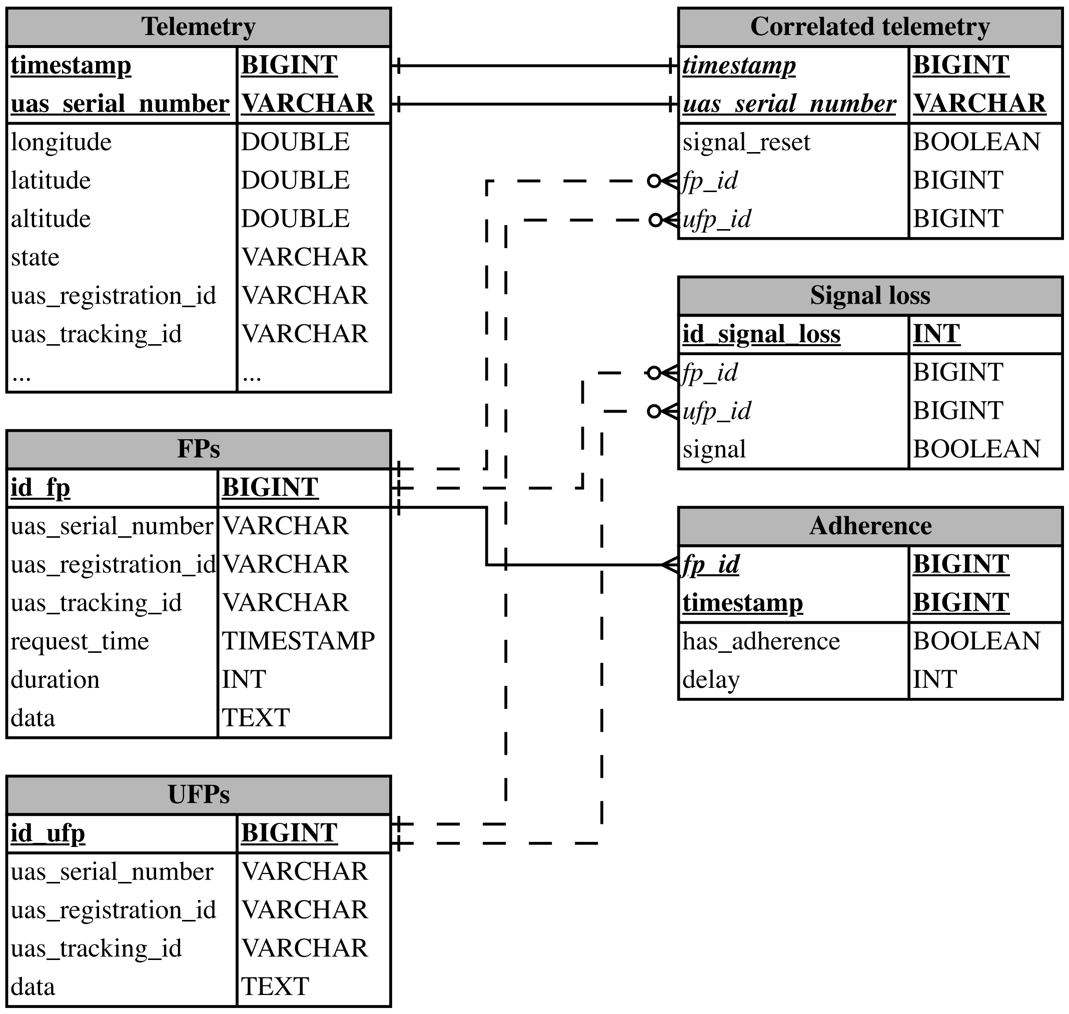


Figure 4.7.: Overview of all tables (source and sink) in our PostgreSQL database

5. System implementation tests

In this chapter the results from testing the prototype with data provided by a simulator are described and discussed. First, the environment and the machine that the prototype is running on are described. Next, the test scenarios and settings from the simulator are elaborated, before the results are presented. Finally, the results are discussed and an overview of the limitations of the implementation is given.

5.1. Runtime environment

All tests are conducted on an *Apple MacBook Pro 14" 2021* with a *8-core M1 Pro chip*, and *16GB unified memory* [113] . The operating system running on this machine is *macOS Sonoma 14.2.1*. However, as most of our components are executed as Docker containers, it is possible to run the same tests on any other operating system (i.e. *Linux* and *Windows*) that has an installation of Docker-desktop. In Docker it is possible to adjust the maximum resource allocation for the number of CPU-cores, unified memory, swap memory, and memory of the virtual disk. Because we do not yet know how this resource allocation affects our results, we have to try different configurations, especially for the number of CPU-cores and the unified memory. We refrain from allocating all resources to Docker by default, as our stream processors are running as Java applications that are not part of any Docker containers.

5.2. Test procedure, settings, and scenarios

The tests for our prototype focus mostly on the stream processing, as it is the core of our system. However, before testing the stream processing logic, we have to ensure that our dataflow in Kafka is handled correctly. In order to control that topics are created correctly, messages are produced to the right topic etc. we can use Kafka UI (cf. Section 4.2.1). Kafka UI allows us to interact with our Kafka cluster through a GUI running in a web-browser. With this GUI we can view all our brokers, topics, consumers, Kafka Connect

[113]Unified memory is a marketing term used by Apple. It describes a combination of RAM and VRAM that is soldered directly on the CPU, which, in theory, should greatly reduce memory accessing speeds. We do not assume that this difference in architecture has a big impact on performance, yet, for the sake of completeness, this information is included.

connectors, and the schema registry. Moreover, we can inspect topics more closely by viewing all partitions and the messages in them individually. It is also possible to observe all schemas created in the schema registry. For consumers and connectors there is an overview of all stable connections; so we know when e.g. a connector fails to write to our database. Of course, we also have to check that our data gets from Kafka into our PostgreSQL database (or from PostgreSQL into Kafka). Here, we can use PgAdmin4, which, like Kafka UI, provides a GUI through a web-browser. This GUI allows us to interact with our database in order to control that all tables are created correctly, and that all events from our Kafka topics arrive in the correct tables. By inspecting the stability of the connectors and the events that arrive in the database we can control that the de-/serialization with the schema registry works correctly (if it does not work the connector would fail immediately).
Two things that are difficult to test with our prototype are fault-tolerance and scalability. For fault-tolerance it is easy to simulate failures, like by stopping and restarting the broker to see if any events are lost. However, we do not have a multi-broker set-up running across multiple servers that would allow us to simulate more complex failures. The mechanisms provided by Kafka to provide fault-tolerance are discussed throughout this thesis (cf. Section 3.1.1), yet in order to say the system is fully faul-tolerant more extensive testing of a native implementation would be necessary. For scalability we would require more computational resources (e.g. CPU and RAM) that we can add to see how the system scales. For now we want to establish a basis of what kind of performance to expect from Kafka and Kafka Streams running on a machine that is not very powerful. Due to the scaling mechanisms available (cf. Sections 3.1.1 and 3.3.1) we are certain that a more powerful system would lead to better results in performance.
The first set of tests to run for our Kafka Streams implementation concern the functionality. Testing the functionality includes verifying for each stream processor individually if the messages are transformed according to the requirements (cf. Section 2.1.2) and stored in the right topic(s) (cf. Section 4.3.1). This verification can be done easily with Kafka UI and PgAdmin4. The second set of tests is concerned with the performance of the stream processors using different parameter settings. We want to know if our prototype can deliver all services (C_1, C_2, and C_3) within 1Hz. If yes, we want to know how many UAS can be simulated in parallel before the threshold of 1Hz is exceeded. Because Kafka and all other components offer an extensive amount of configuration options, we refrain from changing and discussing all of them. Also, most configurations deliver the desired results with the default settings. Yet, the partitioning of data and the number of tasks possible for the stream processors greatly influence scalability, therefore we will test different options to see how the results change.

Kafka set-up

The following tables offer an overview of the most important topics (cf. Table 5.1), producers (cf. Table 5.2), connectors (cf. Table 5.3), and stream processors (cf. Table 5.4) created and utilized by our prototype. In Table 5.1 we can see a list of all topics that we create manually. This table does not include any topics that are created automatically to e.g. store log numbers or when using Kafka Connect. The maximum number of partitions we assign is 3. Partitions are important for scaling the system, yet if we have too many partitions we could create an unnecessary overhead for coordinating these partitions. For our tests we align the number of partitions with the maximum number of stream threads from our stream processors (cf. Table 5.4). We will only change the number of partitions for the topics where we expect the most traffic (i.e. telemetry, enrichedTelemetryMessage, correlated, adherence). To all other topics we assign only one partition; we will not change this, unless we discover a bottleneck because of these topics. Because we are working with a single broker set-up, the replication factor (RF) is set to 1 for all topics. This means that the only replication we keep is the original topic. Consequently, the number of in-sync replicas (ISR) is set to 1 as well, because only the original topic is synchronized. We note that for a typical multi-broker set-up $RF = 3$ and $ISR = 2$. A set-up like this could tolerate the failure of $RF - ISR = 1$ brokers. In order to get more information on the available configurations for topics in Kafka and their meaning we refer to the documentation from *Confluent*[114].

Topics

topic_names	replication factor (RF)	partitions	in-sync replicas (ISR)
telemetry	1	3	1
jdbc-source-fps	1	1	1
ufps	1	1	1
enrichedTelemetryMessage	1	3	1
correlated	1	3	1
uncorrelated	1	1	1
signal	1	1	1
adherence	1	3	1

Table 5.1.: Overview of all manually created topics in Kafka

As we can see in Table 5.2, we require three different producers in our Kafka set-up. The TelemetryProducer produces a stream of messages from the simulator to our telemetry topic; the UfpProducer creates a new UFP in our topic when an uncorrelated UAS is discovered; the SignalLossProducer produces a message to our signal topic if for some time no messages are received. Except names and key serializers, the settings for all producers are identical. The listed configurations indicate that our producers send messages

[114] *Kafka Topic Configuration Reference | Confluent Documentation* 2024.

immediately, are idempotent (cf. Section 2.3.3), and that schemas are created automatically. For our tests we leave these settings at their default values. We will not change any producer settings in our test scenarios, as the scaling for the producer depends on the number of available partitions for the topics. To get more information on the available configurations for Kafka producers and their meaning we refer to the documentation from *Confluent*[115].

Producers

producer_names	key_serializer	value_serializer
TelemetryProducer	uas_serial_number (String)	KafkaJsonSchema
UfpProducer	id_ufp (Long)	KafkaJsonSchema
SignalLossProducer	id_signal_loss (Integer)	KafkaJsonSchema

configurations equal for all producers	
linger.ms	default (0)
batch.size	default (16384)
compression	snappy
idempotence	true
auto.register.schemas	true
retries	default (2147483647)
buffer.memory	default (33554432)

Table 5.2.: Overview of all producers and their settings in Kafka

In Table 5.3 we can see a list of all source and sink connectors utilized (this list matches the tables given in Section 4.4). The most important configurations are the key and value converters and the maximum number of tasks. For the key and value converters it is important that they match the tables in the database (cf. Section 4.4). For instance, in our adherence table, fp_id and timestamp form the primary key. Consequently, we have to set the same key in our connector and serialize it as String. As explained in Section 4.1.2, the maximum number of tasks is important for scaling the connectors. However, we do not plan to change this number for any connector, unless we discover there is a bottleneck that decreases the performance of the whole system.
In Table 5.4 we can see a list of the three stream processors that are implemented to provide C_1, C_2, and C_3 (cf. Section 4.3.2). The key and value de-/serializer [116] settings are not that important here, as they are just default settings and can be configured manually during the stream processing. However, the number of threads is essential for the parallelization of our stream processing (cf. Section 3.3.1). For our tests we can allocate up to eight physical CPU-cores. We favor C_2 and C_3 over C_1 for core allocation, because they are more computationally intense.

115 *Kafka Producer Configuration Reference | Confluent Documentation* 2024.
116 A common abbreviation for a combination of a serializer and a deserializer is serde.

Connectors (consumers)

connector_name	key_converter	value_converter	tasks
sink-adherence	fp_id + timestamp (String)	JsonSchema	1
sink-correlated	timestamp + uas_serial_number (String)	JsonSchema	1
sink-raw-telemetry	timestamp + uas_serial_number (String)	JsonSchema	1
sink-signal	id_signal_loss (Integer)	JsonSchema	1
sink-ufps	id_ufp (Long)	JsonSchema	1
source-fps	id_fp (Long)	JsonSchema	1

Table 5.3.: Overview of all connectors from Kafka Connect

Stream processors

stream_processor_name	key_serde	value_serde	threads
C1_EnrichTelemetry	uas_serial_number (String)	telemetrySerde	2
C2_FPCorrelation	uas_serial_number (String)	correlatedSerde	3
C3_ConformanceMonitoring	uas_serial_number (String)	correlatedSerde	3

Table 5.4.: Overview of all stream processors implemented with Kafka Streams

We note that the schemas used for the value serializers of the producers, the value converters of the connectors, and the serdes of the stream processors must be equal and match design of the tables in our database (cf. Section 4.4). For instance, our TelemetryProducer utilizes a telemetry-message-schema for the KafkaJsonSchema serializer, the same schema must be used for the sink-raw-telemetry connector with the JsonSchema converter, and the telemetrySerde of the C1_EnrichTelemetry stream processor.

Test scenarios

At first we will run some test scenarios to determine the best Kafka set-up for our machine. Once the best set-up is established we will run a more extensive test scenario with this set-up. For our initial tests we use two different Docker resource allocations. In our *Full Docker (FD)* scenario (cf. Table 5.5) we allow Docker to access all eight CPU-cores and up to 12GB of memory. In our *Limited Docker (LD)* scenario (cf. Table 5.6) we limit the CPU-cores to four and cap memory at 8GB. We want to see if the performance of the prototype changes if Kafka has more computational resources available. For each Docker scenario we run two tests. One of the tests sets the number of partitions for all topics to 1 and only one stream thread is used per stream processor. The other test allows parallelization of producers and stream processors by creating three partitions for the topics with the most traffic (i.e. telemetry, enrichedTelemetryMessage, correlated, adherence). For all test scenarios that can be seen in Tables 5.5 and 5.6 we use the same simulator settings where the simulator gradually creates new UAS until a total of 50 UAS is reached and 50 operations are started. Although the simulated routes can differ, the total number of messages produced to our system remains approximately the same for each simulation run. When all UAS are created we let the simulator run a while longer to see how it performs. The duration for each test run is 400s. The metric we use to compare the scenarios is the latency $\Delta t_{xi} = t_{xi} - t_i$ between the different topics x. Here, t_{xi} is the timestamp when a record i is created in a topic x and t_i is the original timestamp when the same record i is created by the simulator; t_i is equal for all topics x. That way we can calculate our Δt_{xi} for all available topics x and records i and see how Δt_{xi} progresses. After the best set-up is established, we use this set-up to run the same test again. This time, however, we create up to 100 UAS and use a longer simulation time of 600s.

Full Docker (FD) scenario

Scenario	Nr. of UAS	Partitions	C1_threads	C2_threads	C3_threads
FD_1	50	1	1	1	1
FD_2	50	3	2	3	3

Table 5.5.: Test configurations for the a Docker set-up where all CPU-cores and more memory are available to Docker

Limited Docker (LD) scenario

Scenario	Nr. of UAS	Partitions	C1_threads	C2_threads	C3_threads
LD_1	50	1	1	1	1
LD_2	50	3	2	3	3

Table 5.6.: Test configurations for the a Docker set-up where memory and CPU resources are limited

We note that in order to create comparable results it is absolutely necessary to delete any previous files from PostgreSQL or offset storages from the C_2 and C_3 stream processors before restarting the Docker container. Moreover, it does take some time for each stream processor to start and to assign the partitions from the topics they consume. Consequently, all stream processor Java applications must be started first and the logs must be checked. As soon as the logs indicate that the partitions are assigned the simulator can be started.

Hypothesis

In general, we expect our prototype to provide functionally correct services within the required latency for somewhere between 10 to 100 UAS operating in parallel. For a higher number we still expect correct results, but an increased latency due to the limited performance of the machine available. Moreover, we expect the latency to increase gradually with each stream processing step. For our initial test scenarios we expect the scenarios with parallelization (i.e. FP_2 and LD_2) to outperform the ones without (i.e. FP_1 and LD_1), because of better CPU utilization. Also, the two LD scenarios should outperform the FP scenarios, because more computational resources are free for the stream processing.

5.3. Results

After each test run we check the current state of our set-up to see if the results are correct. By inspecting the created topics, messages, stream processors, databases etc. with Kafka UI we can say that the data distribution and the stream processing work correctly most of the time. The only problem is that in the beginning, when new UAS are started regularly, some messages are not correlated with FPs although it is defined explicitly in the simulator to send only known UAS. Storing the data in our tables works flawlessly, as all data is stored in the right table with the correct primary keys, foreign keys, and datatypes.

Initial test scenarios

In Figure 5.1 we can see a comparison of the mean latency for all the different test scenarios and topics. Each mean latency is calculated from more than 10000 records (raw, enriched, or correlated telemetry messages, and adherence records) in the respective topic. The results show that the values for all test scenarios are in a similar range (between 450 and 700ms) and follow the same pattern. The biggest increase in latency comes from ingesting the date from the simulator into the telemetry topic. The subsequent processing of C_1 for the enriched topic and C_2 for the correlated topic do not impact the latency at all. The final spike in latency comes from executing the stream processing of C_3 and saving the data to the adherence topic. We can see that LD_1 outperforms all other topics for the final latency, although it has the highest latency for all other topics. Moreover, both scenarios with parallelization (LD_2 and FD_2) offer a better latency for the first three topics, while the scenarios without parallelization (LD_1 and FD_1) deliver better performance for last topic.
In the Tables 5.7 and 5.8 we can see a more detailed overview of the results with min/max values, the median, and the standard deviation. The results indicate that our latency values are not stable. For instance, for FD_2 in the adherence topic we have a difference of $2696 - 26 = 2670ms$ between the min and max value with a standard deviation of 311.725. Also, we observe that the max values for LD_2 and FD_2 in the adherence topic are more pronounced than for the other two scenarios. As we can see in Table 5.8, for all other topics the difference between min and max is not as striking. The values for the standard deviation are similar across all topics and test scenarios. However, LD_1 is

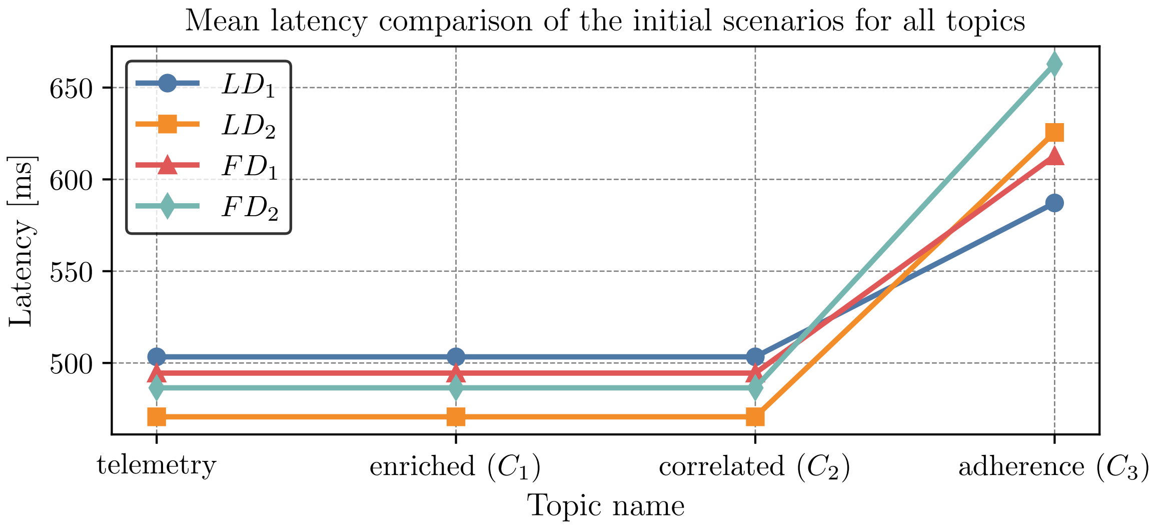


Figure 5.1.: Mean value comparison for the latency of LD_1, LD_2, FD_1, and FD_2 for the topics with the highest throughput

an exception, here the standard deviation is about 50ms lower compared to the other scenarios. Finally, comparing mean and median values shows that the outliers do not distort the latency results strongly. We note that the results for the topics telemetry, enriched, and correlated are equal, which is why we include only Table 5.8 here.

Adherence topic

Test_scenario	Mean	Median	Min	Max	Std
LD_1	587.160	567	154	1223	244.064
LD_2	625.552	581	75	1391	302.370
FD_1	612.812	624	28	1224	304.119
FD_2	662.825	602	26	2696	311.725

Table 5.7.: Comparison of the results from the initial tests for the adherence topic

Telemetry, enriched, and correlated topic

Test_scenario	Mean	Median	Min	Max	Std
LD_1	503.345	482	2	999	241.547
LD_2	470.690	442	0	1001	294.690
FD_1	494.505	461	1	999	301.247
FD_2	486.468	441	0	1000	299.692

Table 5.8.: Comparison of the results from the initial tests for telemetry, enriched, and correlated topic

Based on these results it is difficult to make a decision about the best configuration, as they all perform similarly, but neither offers perfect results. Consequently, we do not expect vastly different results for running our extensive test with either configuration. Ultimately, our configuration of choice is the one from LD_2, because it offers the lowest mean latency for the first three topics.

Extensive test

For the extensive test the prototype does not deliver entirely correct results, as the stream processor for C_3 fails after about seven minutes and cannot be restarted. Also, like for the initial tests, some messages are not correlated with a FP as expected. The rest of the system works as expected and C_1 and C_2 continued to provide correct services even after C_3 is failed. In Figure 5.2 we can see a comparison of the mean latency values over the run time of the whole simulation between the correlation and the adherence topic. For this plot the mean latency is calculated for each second runtime of the simulator for all active drones. We can see that the latency is not constant, but does vary strongly throughout the whole simulation. About four minutes into the simulation we can observe the lowest mean latency, from this point on the latency continues to increase linearly for about four minutes as more drones are started. It is during this time when the processing for C_3 fails. Another observation to make is that the latency is higher in the beginning than in the end.

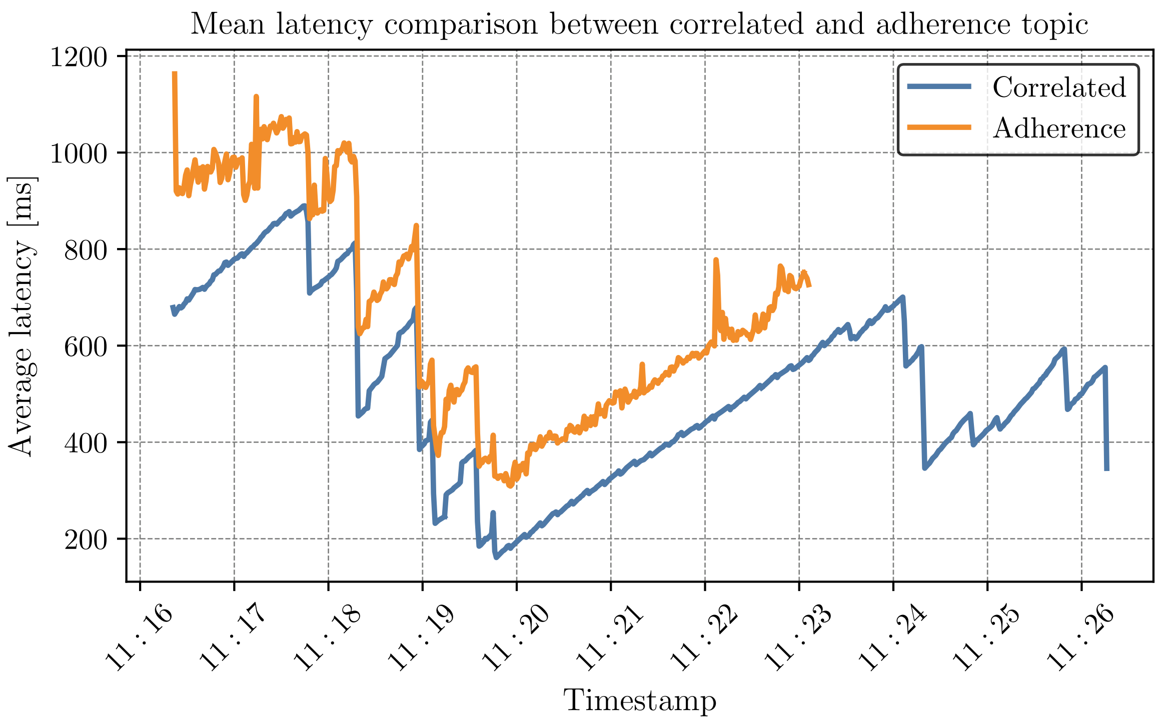


Figure 5.2.: Mean comparison over the run time of the simulation between the correlated topic and the adherence topic

For the topics telemetry, enriched, and correlated we also get similar results as for the initial test scenarios. As illustrated in Figure 5.3, there is no difference in latency between telemetry, enriched, or correlated. There are only two latency spikes. One occurs, when the message is first produced to Kafka by the simulator, and then when it is processed for C_3.

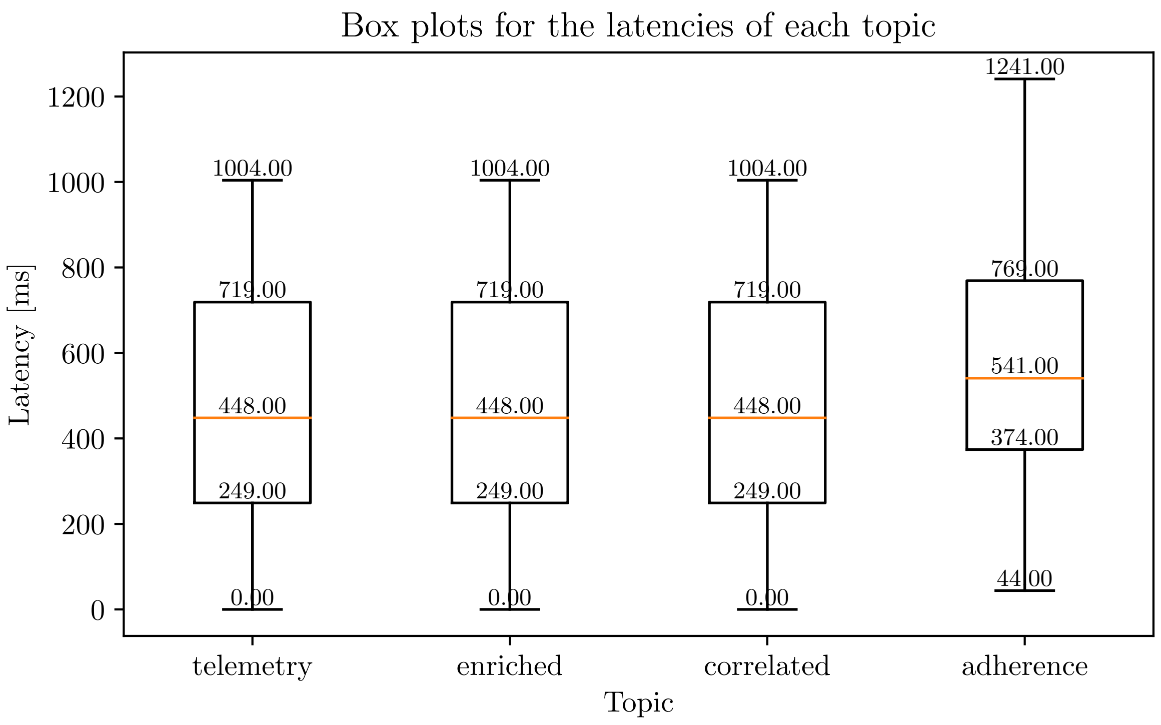


Figure 5.3.: Box plots for the topics telemetry, enriched, correlated, and adherence

Looking at the differences between the creation times from the telemetry and the enriched topic (cf. Figure 5.4) and the enriched and correlated topic (cf. Figure 5.5) shows that for all 30000 messages processed, not a single one has even a slight difference in creation time between these topics.

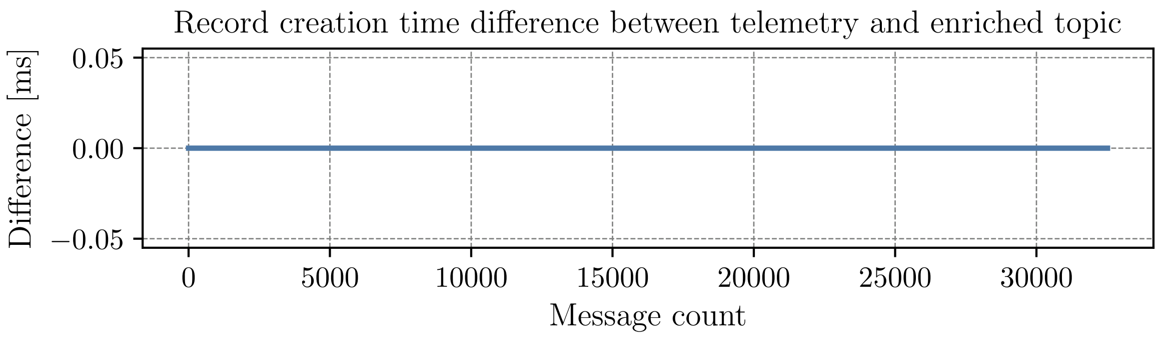


Figure 5.4.: Calculating the difference in creation time between the telemetry and the enriched topic for all more than 30'000 messages

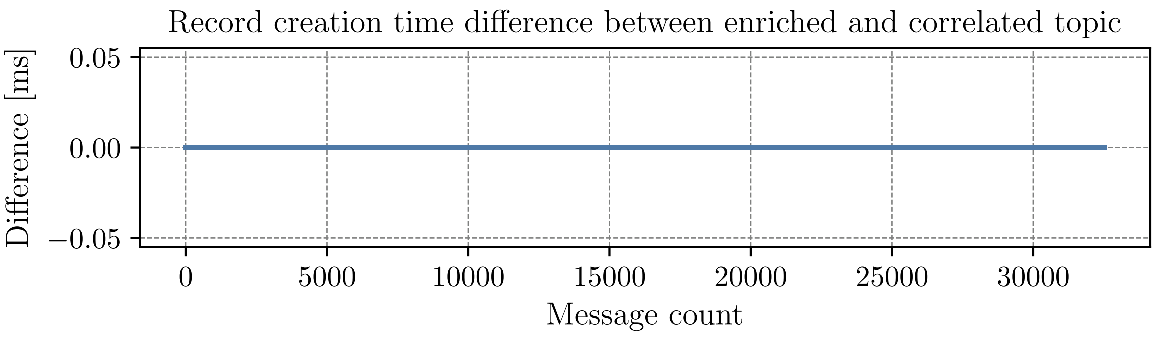


Figure 5.5.: Calculating the difference in creation time between the enriched and the correlated topic for all more than 30'000 messages

5.4. Discussion

As expected, our prototype delivers correct results for all initial tests, with the exception of some uncorrelated messages. However, this can be explained by viewing the logs of the simulator. The problem here is that, occasionally, the creation of a FP and the start of the UAS mission happen at nearly the same point in time. Because we ingest our FPs through an external database, the interface of the simulator must write the FP to the database before it is synchronized with the Kafka topic that backs our state store. Meanwhile, the received telemetry message is sent to the topic for processing directly. So, when the processing starts, the topic state store with the FPs does not contain the FP yet, so the message is not correlated. In order to fix this it would require to redesign the interface by defining a short waiting period between submitting a FP and starting the mission in the simulator.
Knowing that all data is stored correctly in our database tables proves that our de-/serialization logic works correctly and that the correct schemas are created and registered in the schema registry. If there were any incompatibilities the Kafka Connectors would fail immediately and no data would be stored in the tables. Also, if an incompatible schema were registered, Kafka would throw an error.
For our initial tests, the LD scenarios outperform the FD scenarios, conforming our hypothesis that more computational resources for the stream processing helps to improve performance. However, the differences are not as big as expected and it is unreasonable to state that the LD scenarios perform better in any case. The fact that the latency for the first three topics is a little bit lower for the partitioned scenarios when compared to the unpartitioned scenarios is in accordance with our expectations. It is interesting to see that more threads do not help to improve the performance of the adherence topic and the C_3 stream processor. For the adherence topic both unpartitioned scenarios perform better. The only explanation we can think of is that backing up a partitioned state store introduces an overhead that results in a worse performance than for an unpartitioned

state store. A point from our hypothesis we cannot confirm is the gradual increase in latency for each stream processing step. We see a steep increase for the ingestion of the data and when doing the C_3 processing, but no increase at all for C_1 and C_2. This shows that the ingestion of data into Kafka and the way the state store is handled have the biggest impact on the overall latency in our prototype.
The results from the extensive performance test indicate some problems in our prototype. Most notably, the stream processor of C_3 fails before the simulation finishes. Inspecting the error logs shows that the failure is related to our state store. Each state store has a topic in Kafka which contains the log of all changes applied to the state store. If the system fails the state store can be restored from this topic. For our state store we keep a list of the flown trajectories (i.e. all received telemetry messages) for each UAS. However, as some operations have long durations, the lists in the state store can get very long, so the message size exceeds the allowed maximum of 1MB. To evade this error we must redesign our state store by either not saving complete trajectories in one list (maybe start a new list before 1MB of size is exceeded) or by using a stronger compression for our state store.
The other big problem is the behavior of the latency. The latency is not constant at all and we cannot find a pattern or reasons behind it. It is not the number of UAS operating in parallel that is responsible for the behavior of the latency, as the lowest point of latency is reached at about four minutes into the simulation when more then 30 operations are already started. Also, we cannot say it is the fault of the simulator, because when we stop creating new UAS one minute before ending the simulation, we still do not see a stable latency. Assuming that Kafka requires some time in the beginning to rebalance the load is also wrong, as we see more drops of latency in the last two minutes of the simulation.
We conclude that the data ingestion is an important part of the system's design and that more thought must be put into designing the ingestion of data into Kafka. We assume that a standardized interface could be the solution to our problem, as our customized interface does not seem to be optimized. For future iterations of the prototype more research on how the data gets into Kafka needs to be done.
Although these results show some strong limitations of our prototype, there are also some points the highlight why Kafka and especially Kafka Streams is the right tool to provide U-space services. In our results we can see that there is no latency increase from the ingestion to the enriched telemetry and from the enriched telemetry to the correlated telemetry. This means that once the data is ingested into Kafka there is hardly any latency for providing our services, no matter if we starte one UAS or 100. It hints that Kafka and Kafka Streams are highly scalable frameworks used to build systems that handle far bigger amounts of data that are not in the hundreds of concurrent operations but in the thousands or millions. However, it is in contrast to the expectations from our hypothesis, as we expected the latency to increase gradually the more UAS operations are started. Moreover, the services for C_1 and C_2 are provided within the 1Hz goal (except for

a few outliers at the beginning of the simulation). For C_3 we have way more outliers, but the average value is still way below 1Hz. These results are impressive, considering that our prototype runs on a not very powerful machine. In order to make a future iteration of our prototype production-ready we need to work on the ingestion to have a more constant latency and work on the state store of C_3 to evade failures due to the message size.
In summary we can say that the dataflow within our prototype works as expected and provides the required results, with the exception of some design issues regarding the state store and optimization issues with the interface. As soon as we get the data into Kafka, Kafka Streams offers a lot of possibilities to flexibly process the messages without a big increase in latency. However, any stateful computations must be designed carefully to not exceed parameters like the maximum message size of 1MB. At its current state our prototype is not ready for production, but it shows a lot of potential. If the aforementioned problems with the ingestion and the state store are resolved we have a powerful, scalable, fault-tolerant system at our hands, capable of providing U-space services in real-time.

5.5. Limitations

We note that our prototype does not have the claim to be a production-ready implementation of Kafka, but a means for testing the feasibility of such a system concerning the problem setting. For this reason a few simplifications must be considered when viewing the results explained in the previous sections.
The biggest limitation is that our prototype is tested only in a controlled environment that eliminates potential sources for failures (e.g. connectivity issues, incompatibility), or places them on purpose through the simulator (e.g. time delay to create a loss of adherence). Moreover, our prototype is not running natively on a server, but within a Docker container. Of course, this set-up does not allow us to fully test the performance capabilities of our implementation, yet the advantages of a greatly simplified development process cannot be denied (cf. Section 4.2). In our set-up we do not have multiple brokers up and running to replicate the data between different brokers to create a fault tolerant system. Setting up multiple brokers in our test environment would require to work with yet another framework called *Kubernetes*[117]; a framework that can be used to orchestrate different container applications. Thus, any settings regarding fault tolerance must be revised for a production-ready implementation. Also, the replication between different brokers introduces a delay that is not considered throughout any tests. Any replication must be viewed as a potential source for failures (e.g. if the replication is interrupted due to network problems).
Concerning the database, like Kafka, our implementation of PostgreSQL runs in a Docker container. Here, all data is stored in a local folder, not a dedicated database. A dedicated,

[117] *Kubernetes* 2024.

fault tolerant database that can deal with old data vs new data and implements useful data compression mechanisms would require a lot of additional work that is not within the scope of this thesis.
Because the prototype is tested in a controlled environment with non-critical data, for reasons of simplicity, no safety mechanisms (e.g. encryption) are implemented. However, for any production-ready implementation of Kafka a proper implementation of safety mechanisms is indispensable.
In general, our prototype must cover a lot of different functionalities. Because the time-frame for this thesis is limited, the main focus is on designing the system as a whole. Due to the large number of components it is difficult to fully develop each component equally. For this reason a lot of pre-defined and pre-build frameworks are used as placeholders for components that require more customization. For any production-ready implementation it is necessary to evaluate the needs and replace the components step-by-step with production-ready, customized versions.

6. Conclusion

In order to continue to provide safe operations of UAS in a growing market, further development of regulations and technologies is necessary. U-space services like flight tracking and reporting should facilitate a save and efficient access to the common airspace. However, these services must be provided in real-time, which requires the handling of vast amounts of continuous data. Handling continuous data in real-time is challenge in itself and demands purpose-built systems. Through the application of open-source frameworks like Apache Kafka it is possible to develop a DSMS that in theory can handle data streams and provide insights in real-time. This poses the question if and how a prototype of a DSMS can be fit to provide U-space services in real-time.

Before designing the system it is essential to evaluate the specifications and get a better understanding for the requirements of our system. Our specifications are based on a mix of regulatory, problem-specific, and general requirements. The regulatory requirements specify the direction of our problem-specific requirement, which in turn describe the U-space services that should be provided by our prototype. The general requirements are mostly concerned with the fault-tolerance and the scalability of our system. These are two very important characteristics of a DSMS. We need fault-tolerance, as we cannot evade all faults from happening. Without scalability our prototype would not be able to adjust to the potential load increase that comes with the growing UAS market.
Implementing a prototype with these specifications introduces a series of technical challenges. These challenges can be categorized in three groups: Ingestion, processing, and storage. For the ingestion it is essential to correctly deal with the order in which events arrive. This order must be kept through the use of a log in order to provide deterministic results if events have to be re-processed in cases of failures. The processing is concerned with transforming streams through stream processors. Thereby it is important to maintain the current state of the processing. To evade losing the state due to a failure we require a changelog of the state for recreation. During re-processing it must not happen that results are affected twice. With the log from the ingestion we can make our computations idempotent and guarantee that each message is processed exactly once. We must not lose messages or provide wrong results due to the regulatory requirements. For the storage we distinguish between cold storage use for long-term legal recordings and hot memory used during stream processing that is accessible immediately after a failure

to restore e.g. the state. In order to get some general guidance for selecting a database framework we classify frameworks with the CAP-theorem. Although CAP has its limitations, it is enough to get an overview of the vast amounts of different database frameworks available.

In order to develop our prototype we have to select numerous different frameworks for deployment, as there exists no single framework that provides all the required functionalities. The most important framework at the core of our prototype is Apache Kafka. It is selected based on the big community behind it for support and its log-based approach for message distribution. Additionally, fault-tolerance can be guaranteed through message replication and the system is highly scalable due to its possibilities for distributed computing through partitioning. Also, as the different components of Kafka are decoupled, we have a lot of options to flexibly shape our system and add and remove components as we require. Finding a framework for the database technology proved to be more difficult, as there are lots of very specialized solutions available. Without any prior experience it is difficult to select one for the problem at hand. Consequently, the choice is made for PostgreSQL, an established, flexible framework with a big community behind it called. Finally, for the stream processing we opt for Kafka Streams. Technically, it is not a stand alone framework, but its tight integration with Kafka and the rich feature set for stream processing with state support our choice.

Applying these frameworks to develop our prototype demands additional components. Most importantly, a schema registry has to make sure that we control the form of all JSON data that gets into and out of our system. Without a JSON schema we would run into compatibility issues between the different components, because data in Kafka is always serialized as byte-representation. Now, to get data from Kafka into an external database or vice versa we require Kafka Connect. Kafka Connect in combination with the schema registry are the two key components of our prototype to get the data from source to sink. In order to comfortably test and debug our prototype we implement all our components in Docker-containers. This limits our possibilities for extensive performance tests, but greatly reduces the workload compared to a native implementation of Kafka and all other components. However, all stream processors are executed as native Java applications that do not run as part of Docker. They communicate with the Docker set-up through ports, but are mostly independent. Designing the dataflow for the stream processing is crucial for a successful implementation, as it is invaluable to understand where the data comes from and where it is going to. For the implementation dealing with customized state is the most challenging part, as it requires the use of the lower level Processor API. Implementing the database connection is not difficult if the schema of the data and the data types are handled consistently throughout the whole system, otherwise it is not possible to save data to the tables in our database.

As the results from the tests suggest, our prototype is not ready for production yet. The latency for ingesting the data into Kafka may not be too high, but is not linear either, and one of the stream processors has problems with the maximum message size due to a design issue of the customized state store. However, for two of the three implemented stream processors, no latency is measured for processing, indicating the potential of Kafka Streams if applied correctly. Moreover, the results for the average latency are way below the threshold of 1Hz and outliners are sparse. Considering that these test results are from a laptop with limited computational resources, not a dedicated server, it is fair to say that a DSMS built around Kafka can be capable of providing U-space services in real-time.

Research outlook

For a production-ready implementation of Kafka more thought has to be put into the data ingestion into Kafka. For now we are working with a simulator running in Java, but in a real UAS network data is received from many IoT devices that are connected through standardized protocols like MQTT. For future iterations of the prototype this is the best starting point for improvements, as our results suggest that the ingestion introduces the biggest latency. Concerning the latency, we definitely require more insight into why the latency is not constant, or why it does not follow a clear pattern. Now that a basic implementation exists it is possible to try different variations for the ingestion. From an economical point of view it should not remain unmentioned that lots of paid, highly-scalable, mostly cloud-based services for DSMS are available in the web. Of course, trusting an external party with sensitive data is always critical and should be done only with utmost care. However, compared to the effort it takes to deploy a production-ready version of Kafka it should be evaluated when it is reasonable to develop a complex system internally, or if there are options for relying on external providers.
To summarize, we see two possible ways to continue research in the area of DSMS built around Kafka. On the one hand, it is possible to dive deeper into the technical part of the implementation and explore Kafka's behavior with different ingestion interfaces, state stores etc. On the other hand, it is possible to examine the economic aspect related to the costs of developing this system as opposed to paying for existing solutions; a point that is not discussed in this thesis. Either way Kafka is a powerful framework with a lots of fields of application. It has proven its capabilities across many industries and most certainly will find more adopters in the future.

Bibliography

Books

Fowler, Martin (01/01/2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional. ISBN: 0-321-12742-0

Hueske, Fabian and Vasiliki Kalavri (04/11/2019). *Stream Processing with Apache Flink.* First Edition. O'Reilly Media, Inc. ISBN: 978-1-4919-7429-2

Kleppmann, Martin (05/2016). *Making Sense of Stream Processing.* O'Reilly Media, Inc. ISBN: 978-1-4919-3728-0

– (03/01/2017). *Designing Data-Intensive Applications.* O'Reilly Media, Inc. ISBN: 978-1-4493-7332-0

Kreps, Jay (09/2014a). *I Heart Logs.* First Edition. O'Reilly Media, Inc. ISBN: 978-1-4919-0933-1

Marz, Nathan and James Warren (02/2015). *Big Data: Principles and Best Practices of Scalable Realtime Data Systems.* 1st ed. USA: Manning Publications Co. 425 pp. ISBN: 978-1-61729-034-3

Reis, Joe and Matt Housley (07/2022). *Fundamentals of Data Engineering.* First Edition. O'Reilly. ISBN: 978-1-09-813980-3

Shapira, Gwen et al. (11/2021). *Kafka: The Definitive Guide, 2nd Edition.* O'Reilly Media, Inc. ISBN: 978-1-4920-4308-9

Articles

Akidau, Tyler et al. (08/01/2015). "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data

Processing". In: *Proceedings of the VLDB Endowment* 8, pp. 1792–1803. DOI: `10.14778/2824032.2824076`

Chandy, K. Mani and Leslie Lamport (02/01/1985). "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Transactions on Computer Systems* 3.1, pp. 63–75. ISSN: 0734-2071. DOI: `10.1145/214451.214456`.
`https://dl.acm.org/doi/10.1145/214451.214456`
(Visited on 08/24/2023)

Fox, Armando, Steven D. Gribble, et al. (10/01/1997). "Cluster-Based Scalable Network Services". In: *ACM SIGOPS Operating Systems Review* 31.5, pp. 78–91. ISSN: 0163-5980. DOI: `10.1145/269005.266662`.
`https://dl.acm.org/doi/10.1145/269005.266662`
(Visited on 11/14/2023)

Golab, Lukasz and M. Tamer Özsu (06/01/2003). "Issues in Data Stream Management". In: *ACM SIGMOD Record* 32.2, pp. 5–14. ISSN: 0163-5808. DOI: `10.1145/776985.776986`.
`https://doi.org/10.1145/776985.776986`
(Visited on 08/08/2023)

Haerder, Theo and Andreas Reuter (12/02/1983). "Principles of Transaction-Oriented Database Recovery". In: *ACM Computing Surveys* 15.4, pp. 287–317. ISSN: 0360-0300. DOI: `10.1145/289.291`.
`https://dl.acm.org/doi/10.1145/289.291`
(Visited on 11/14/2023)

Helland, Pat (04/01/2012). "Idempotence Is Not a Medical Condition". In: *Queue* 10, pp. 30–46. DOI: `10.1145/2181796.2187821`

Isah, Haruna et al. (10/10/2019). "A Survey of Distributed Data Stream Processing Frameworks". In: *IEEE Access* 7, pp. 1–1. DOI: `10.1109/ACCESS.2019.2946884`

Kleppmann, Martin (09/17/2015). "A Critique of the CAP Theorem". In

Motwani, Rajeev et al. (12/14/2002). "Query Processing, Resource Management, and Approximation in a Data Stream Management System". In

Nasiri, Hamid, Saeed Nasehi, and Maziar Goudarzi (06/11/2019). "Evaluation of Distributed Stream Processing Frameworks for IoT Applications in Smart Cities". In: *Journal of Big Data* 6. DOI: `10.1186/s40537-019-0215-2`

Van Dongen, Giselle and Dirk Van den Poel (06/28/2021). "A Performance Analysis of Fault Recovery in Stream Processing Frameworks". In: *IEEE Access* PP, pp. 1–1. DOI: `10.1109/ACCESS.2021.3093208`

Reports

CORUS XUAM Consortium (12/23/2022). *Advanced U-space Definition*

European Commission (04/22/2021). *COMMISSION IMPLEMENTING REGULATION (EU) 2021/664 of 22 April 2021 on a Regulatory Framework for the U-space.* Official Journal of the European Union

Gray, J. (12/1995). *Queues Are Databases.*
`https://www.semanticscholar.org/paper/Queues-Are-Databases-Gray/d6ba65fa74469cef98fbbe342465034cb5108301`
(Visited on 08/21/2023)

SESAR Joint Undertaking (2017a). *European Drones Outlook Study Unlocking the Value for Europe.*
`https://www.sesarju.eu/sites/default/files/documents/reports/European_Drones_Outlook_Study_2016.pdf`
(Visited on 08/08/2023)

– (06/09/2017b). *U-Space Blueprint.*
`www.sesarju.eu`
(Visited on 08/08/2023)

Walter Heimerdinger, Charles Weinstock (1992). *A Conceptual Framework for System Fault Tolerance.* CMU/SEI-92-TR-033. Software Engineering Institute, Carnegie Mellon University.
`https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11747`
(Visited on 08/16/2023)

Conference papers

Babcock, Brian et al. (06/03/2002). “Models and Issues in Data Stream Systems”. In: *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’02. New York, NY, USA: Association for Computing Machinery, pp. 1–16. ISBN: 978-1-58113-507-7. DOI: 10.1145/543613.543615.
https://doi.org/10.1145/543613.543615
(Visited on 08/08/2023)

Breivold, Hongyu Pei, Ivica Crnkovic, and Peter J. Eriksson (07/2008). “Analyzing Software Evolvability”. In: 2008 32nd Annual IEEE International Computer Software and Applications Conference, pp. 327–330. DOI: 10.1109/COMPSAC.2008.50

Feick, Martin, Niko Kleer, and Marek Kohn (2018). “Fundamentals of Real-Time Data Processing Architectures Lambda and Kappa”. In: Gesellschaft für Informatik e.V. ISBN: 978-3-88579-448-6.
https://dl.gi.de/handle/20.500.12116/28983
(Visited on 09/25/2023)

Fernandez, Raul Castro et al. (06/19/2014). “Making State Explicit for Imperative Big Data Processing”. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC’14. USA: USENIX Association, pp. 49–60. ISBN: 978-1-931971-10-2

Fox, Armando and Eric Brewer (03/28/1999). “Harvest, Yield, and Scalable Tolerant Systems”. In: Hot Topics in Operating Systems, pp. 174–178. ISBN: 978-0-7695-0237-3. DOI: 10.1109/HOTOS.1999.798396

Li, Jin et al. (06/14/2005). “Semantics and Evaluation Techniques for Window Aggregates in Data Streams”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS05: International Conference on Management of Data and Symposium on Principles Database and Systems. Baltimore Maryland: ACM, pp. 311–322. ISBN: 978-1-59593-060-6. DOI: 10.1145/1066157.1066193.
https://dl.acm.org/doi/10.1145/1066157.1066193
(Visited on 08/29/2023)

Moseley, Ben and Peter Marks (2006). “Out of the Tar Pit”. In: Software Practice Advancement (SPA).
https://www.semanticscholar.org/paper/Out-of-the-Tar-Pit-Moseley-

`Marks/41dc590506528e9f9d7650c235b718014836a39d`
(Visited on 08/17/2023)

Zaharia, Matei et al. (06/12/2012). “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters”. In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’12. USA: USENIX Association, p. 10

Internal documents

Carbone, Paris et al. (06/29/2015). “Lightweight Asynchronous Snapshots for Distributed Dataflows”. DOI: `10.48550/arXiv.1506.08603`. arXiv: `1506.08603 [cs]`.
`http://arxiv.org/abs/1506.08603`
(Visited on 08/24/2023)

Internet sources

ActiveMQ (2023).
`https://activemq.apache.org/`
(Visited on 11/08/2023)

Apache Avro (2023). Apache Avro.
`https://avro.apache.org/`
(Visited on 12/04/2023)

Apache Cassandra (2023).
`https://cassandra.apache.org/_/index.html`
(Visited on 11/15/2023)

Apache Flink — Stateful Computations over Data Streams (2023).
`https://flink.apache.org/`
(Visited on 08/09/2023)

Apache Kafka (2023). Apache Kafka.
`https://kafka.apache.org/`
(Visited on 08/09/2023)

Apache Kafka Documentation (2023). Apache Kafka.
`https://kafka.apache.org/documentation/`
(Visited on 11/10/2023)

Apache Kafka Powered By (2023). Apache Kafka.
`https://kafka.apache.org/powered-by`
(Visited on 08/09/2023)

Apache Spark - Unified Engine for Large-Scale Data Analytics (2023).
`https://spark.apache.org/`
(Visited on 08/09/2023)

Apache ZooKeeper (2023).
`https://zookeeper.apache.org/`
(Visited on 11/13/2023)

Archer Brown, Seth (2023). *The Two Generals Problem.*
`https://haydenjames.io/the-two-generals-problem/`
(Visited on 08/25/2023)

Collection (Java Platform SE 8) (2024).
`https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`
(Visited on 01/29/2024)

Dean, Alex (09/15/2015). *Improving Snowplow's Understanding of Time.* Snowplow.
`https://snowplow.io/blog/improving-snowplows-understanding-of-time/`
(Visited on 08/23/2023)

Docker Hub (2023).
`https://hub.docker.com/`
(Visited on 12/05/2023)

Docker (05/10/2022). *Docker: Accelerated Container Application Development.*
`https://www.docker.com/`
(Visited on 12/05/2023)

Fault Tolerance via State Snapshots (2023).
`//nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/fault_`

`tolerance/`
(Visited on 08/24/2023)

Find the Top Drone Application |Drone Industry Insights 2022 (04/27/2022).
`https://droneii.com/top-drone-applications`
(Visited on 08/07/2023)

Grafana (2023). *Grafana: The Open Observability Platform*. Grafana Labs.
`https://grafana.com/`
(Visited on 12/18/2023)

Group, PostgreSQL Global Development (2023). *PostgreSQL*. PostgreSQL.
`https://www.postgresql.org/`
(Visited on 11/15/2023)

Industry Leading Drone Market Analysis 2022-2030 | Droneii (09/20/2022).
`https://droneii.com/drone-market-analysis-2022-2030`
(Visited on 08/07/2023)

InfluxDB | Real-time Insights at Any Scale (Sat, 15 Jan 2022 15:32:09 +0000). InfluxData.
`https://www.influxdata.com/home/`
(Visited on 11/15/2023)

Introducing Stream-Stream Joins in Apache Spark 2.3 (Tue, 03/13/2018 - 07:59). Databricks.
`https://www.databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html`
(Visited on 08/31/2023)

JDBC Connector (Source and Sink) for Confluent Platform (2024).
`https://docs.confluent.io/kafka-connectors/jdbc/current/`
(Visited on 01/31/2024)

JSON Schema (2023).
`https://json-schema.org/`
(Visited on 12/04/2023)

Kafka Connect | Confluent Documentation (2024).
`https://docs.confluent.io/platform/current/connect/index.html`
(Visited on 01/31/2024)

Kafka Producer Configuration Reference | Confluent Documentation (2024).
`https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html`
(Visited on 02/05/2024)

Kafka Streams Documentation (2023). Apache Kafka.
`https://kafka.apache.org/36/documentation/streams/`
(Visited on 11/16/2023)

Kafka Topic Configuration Reference | Confluent Documentation (2024).
`https://docs.confluent.io/platform/current/installation/configuration/topic-configs.html`
(Visited on 02/05/2024)

Kafka UI Provectus (2023).
`https://docs.kafka-ui.provectus.io/overview/readme`
(Visited on 12/05/2023)

KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum (2023).
`https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum`
(Visited on 11/13/2023)

Kreps, Jay (07/02/2014b). *Questioning the Lambda Architecture.*
`https://www.oreilly.com/radar/questioning-the-lambda-architecture/`
(Visited on 09/25/2023)

ksqlDB: The Database Purpose-Built for Stream Processing Applications. (2024).
`https://ksqldb.io/`
(Visited on 01/29/2024)

Kubernetes (2024).
`https://kubernetes.io/`
(Visited on 01/07/2024)

Matternet Launches World's Longest Urban Drone Delivery Route Connecting Hospitals and Laboratories in Zurich, Switzerland (12/12/2022).
`https://www.businesswire.com/news/home/20221212005097/en/Matternet-Launches-World%E2%80%99s-Longest-Urban-Drone-Delivery-Route-Connecting-Hospitals-and-Laboratories-in-Zurich-Switzerland`
(Visited on 08/07/2023)

MongoDB (2023). *MongoDB*. MongoDB.
`https://www.mongodb.com/de-de`
(Visited on 11/15/2023)

pgAdmin4 (2023).
`https://www.pgadmin.org/download/`
(Visited on 12/05/2023)

PostgreSQL ++ for Time Series and Events (2023).
`https://www.timescale.com`
(Visited on 11/15/2023)

Prometheus (2023). *Prometheus - Monitoring System & Time Series Database.*
`https://prometheus.io/`
(Visited on 12/18/2023)

Protocol Buffers (2023).
`https://protobuf.dev/overview/`
(Visited on 12/04/2023)

Publish-Subscribe - Intro to Pub-Sub Messaging (2023). Confluent.
`https://www.confluent.io/learn/publish-subscribe/`
(Visited on 11/23/2023)

RabbitMQ: Easy to Use, Flexible Messaging and Streaming — RabbitMQ (2023).
`https://www.rabbitmq.com/`
(Visited on 11/08/2023)

RocksDB | A Persistent Key-Value Store (2023). RocksDB.
`http://rocksdb.org/`
(Visited on 11/16/2023)

Samza - State Management (2023).
`https://samza.apache.org/learn/documentation/0.10/container/state-management.html`
(Visited on 08/31/2023)

Structured Streaming Programming Guide - Spark 3.4.1 Documentation (2023).
`https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html`
(Visited on 08/24/2023)

Treat, Tyler (03/25/2015). *You Cannot Have Exactly-Once Delivery.* Brave New Geek.
`https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/`
(Visited on 08/28/2023)

Using JConsole - Java SE Monitoring and Management Guide (2023).
`https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html#`
(Visited on 12/18/2023)

A. Java class diagrams

This part of the appendix offers an overview of all Java classes implemented for the prototype. Each section contains the Java class diagram of an individual package in the Java class path. Below each class diagram there are lists of the methods where the input variables are not clearly described in the diagram.

A.1. Tools Java class diagram

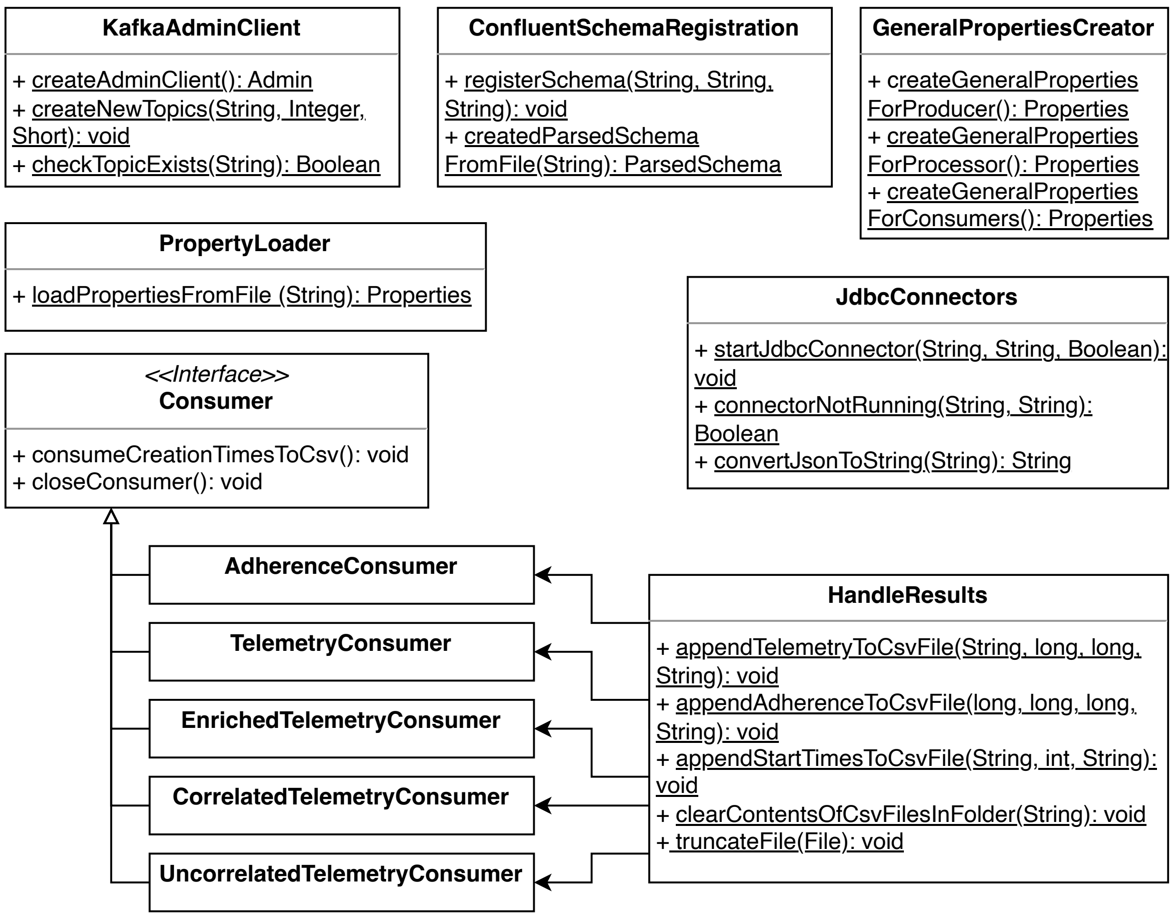

KafkaAdminClient
+ createAdminClient(): Admin
+ createNewTopics(String, Integer, Short): void
+ checkTopicExists(String): Boolean
ConfluentSchemaRegistration
+ registerSchema(String, String, String): void
+ createdParsedSchemaFromFile(String): ParsedSchema
GeneralPropertiesCreator
+ createGeneralPropertiesForProducer(): Properties
+ createGeneralPropertiesForProcessor(): Properties
+ createGeneralPropertiesForConsumers(): Properties
PropertyLoader
+ loadPropertiesFromFile (String): Properties
JdbcConnectors
+ startJdbcConnector(String, String, Boolean): void
+ connectorNotRunning(String, String): Boolean
+ convertJsonToString(String): String
<<Interface>>
Consumer
+ consumeCreationTimesToCsv(): void
+ closeConsumer(): void
AdherenceConsumer
TelemetryConsumer
EnrichedTelemetryConsumer
CorrelatedTelemetryConsumer
UncorrelatedTelemetryConsumer
HandleResults
+ appendTelemetryToCsvFile(String, long, long, String): void
+ appendAdherenceToCsvFile(long, long, long, String): void
+ appendStartTimesToCsvFile(String, int, String): void
+ clearContentsOfCsvFilesInFolder(String): void
+ truncateFile(File): void

Figure A.1.: Java class diagram of all tools that were implemented for the prototype

KafkaAdminClient:

- createNewTopics(String topicName, int partitions, short replicationFactor): void
- checkTopicExists(String topicName): Boolean

ConfluentSchemaRegistration:

- registerSchema(String subject, String pathToSchema, String schemaRegistryUrl, String compatibility): void
- createParsedSchemaFromFile(String pathToSchema): ParsedSchema

PropertyLoader:

- loadPropertiesFromFile(String filePath): Properties

JdbcConnectors:

- startJdbcConnector(String kafkaConnectUrl, String pathToConnectConfig, Boolean source): void
- connectorNotRunning(String kafkaConnectUrl, String connectorName): Boolean
- convertJsonToString(String filePath): String

HandleResults:

- appendTelemetryToCsvFile(String uasSerialNumber, long timestamp, long creationTimestamp, String filePath): void
- appendAdherenceToCsvFile(long fp_id, long telemetry_timestamp, long creationTimestamp, String filePath): void
- appendStartTimesToCsvFile(String uasSerialNumber, int timestamp, String filePath): void
- clearContentsOfCsvFilesInFolder(String directoryPath): void
- truncateFile(File file): void

A.2. Main Java class diagram

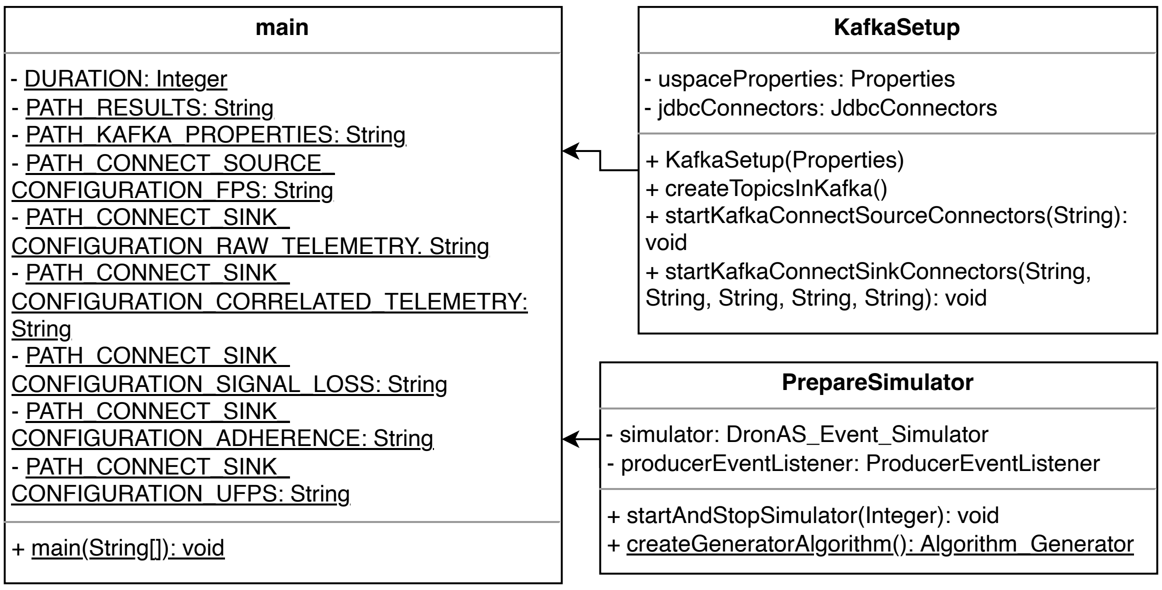

main
- DURATION: Integer
- PATH_RESULTS: String
- PATH_KAFKA_PROPERTIES: String
- PATH_CONNECT_SOURCE_CONFIGURATION_FPS: String
- PATH_CONNECT_SINK_CONFIGURATION_RAW_TELEMETRY. String
- PATH_CONNECT_SINK_CONFIGURATION_CORRELATED_TELEMETRY: String
- PATH_CONNECT_SINK_CONFIGURATION_SIGNAL_LOSS: String
- PATH_CONNECT_SINK_CONFIGURATION_ADHERENCE: String
- PATH_CONNECT_SINK_CONFIGURATION_UFPS: String
+ main(String[]): void
KafkaSetup
- uspaceProperties: Properties
- jdbcConnectors: JdbcConnectors
+ KafkaSetup(Properties)
+ createTopicsInKafka()
+ startKafkaConnectSourceConnectors(String): void
+ startKafkaConnectSinkConnectors(String, String, String, String, String): void
PrepareSimulator
- simulator: DronAS_Event_Simulator
- producerEventListener: ProducerEventListener
+ startAndStopSimulator(Integer): void
+ createGeneratorAlgorithm(): Algorithm_Generator

Figure A.2.: Java class diagram of the main class and the set-up classes used for Kafka and the simulator

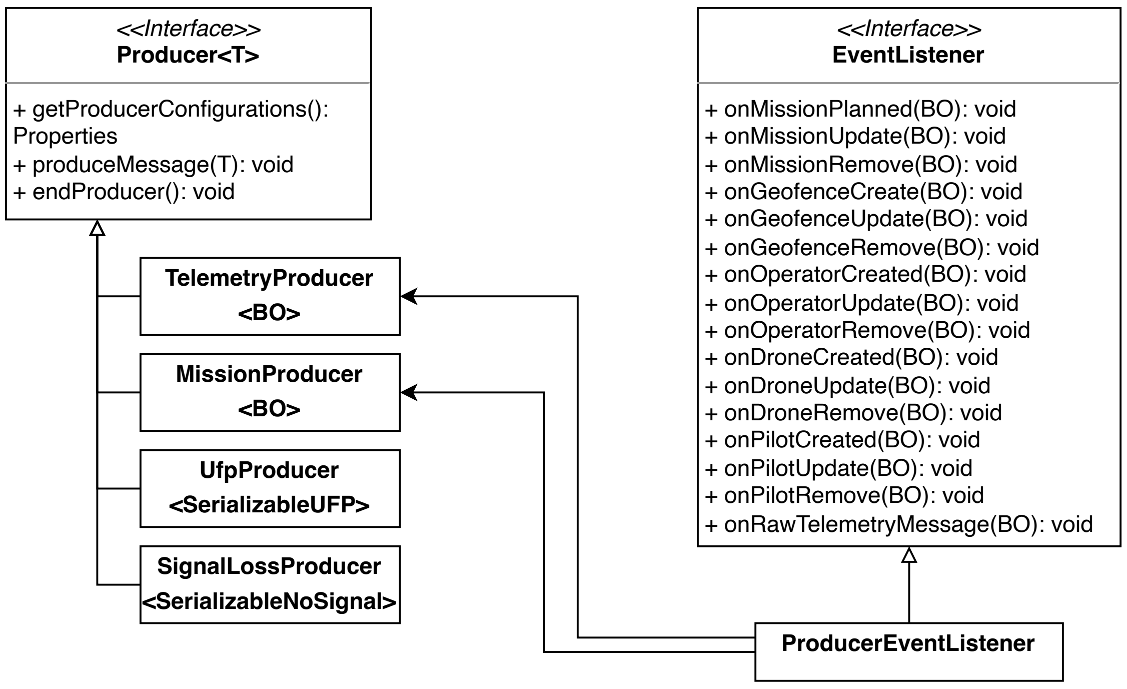
KafkaSetup:

- startKafkaConnectSourceConnectors(String pathFP): void
- startKafkaConnectSinkConnectors(String pathUfps, String pathRawTelemetry, String pathCorrelatedTelemetry, String pathSignalLoss, String pathAdherence): void

PrepareSimulator:

- startAndStopSimulator(int duration): void

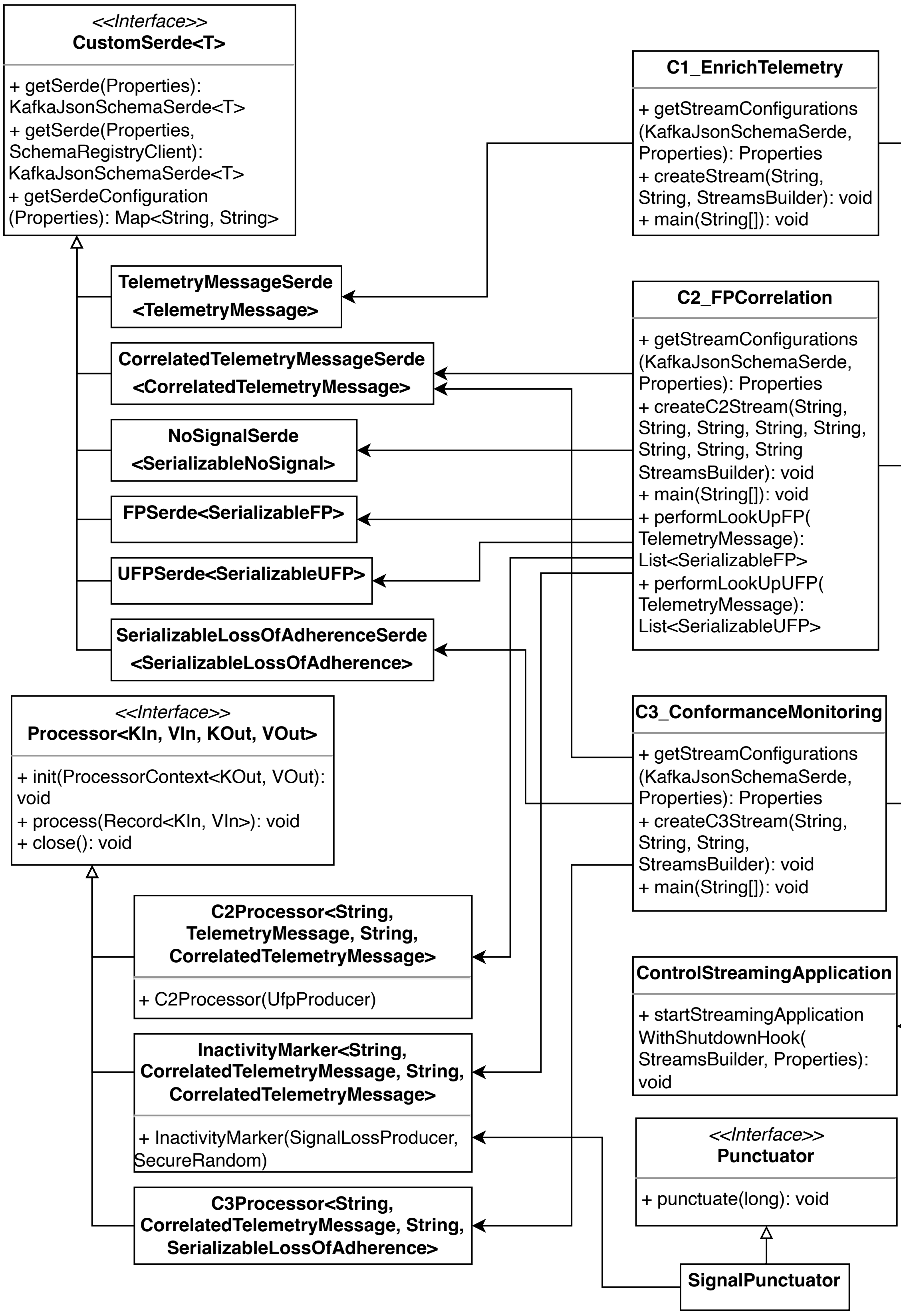
A.3. Producers Java class diagram



<<Interface>>
Producer<T>
+ getProducerConfigurations(): Properties
+ produceMessage(T): void
+ endProducer(): void
<<Interface>>
EventListener
+ onMissionPlanned(BO): void
+ onMissionUpdate(BO): void
+ onMissionRemove(BO): void
+ onGeofenceCreate(BO): void
+ onGeofenceUpdate(BO): void
+ onGeofenceRemove(BO): void
+ onOperatorCreated(BO): void
+ onOperatorUpdate(BO): void
+ onOperatorRemove(BO): void
+ onDroneCreated(BO): void
+ onDroneUpdate(BO): void
+ onDroneRemove(BO): void
+ onPilotCreated(BO): void
+ onPilotUpdate(BO): void
+ onPilotRemove(BO): void
+ onRawTelemetryMessage(BO): void
TelemetryProducer <BO>
MissionProducer <BO>
UfpProducer <SerializableUFP>
SignalLossProducer <SerializableNoSignal>
ProducerEventListener

Figure A.3.: Java class diagram of all producer classes and the interface that communicates with the simulator

A.4. Stream processing Java class diagram



<<Interface>>
CustomSerde<T>
+ getSerde(Properties): KafkaJsonSchemaSerde<T>
+ getSerde(Properties, SchemaRegistryClient): KafkaJsonSchemaSerde<T>
+ getSerdeConfiguration (Properties): Map<String, String>
C1_EnrichTelemetry
+ getStreamConfigurations (KafkaJsonSchemaSerde, Properties): Properties
+ createStream(String, String, StreamsBuilder): void
+ main(String[]): void
TelemetryMessageSerde <TelemetryMessage>
CorrelatedTelemetryMessageSerde <CorrelatedTelemetryMessage>
NoSignalSerde <SerializableNoSignal>
FPSerde<SerializableFP>
UFPSerde<SerializableUFP>
SerializableLossOfAdherenceSerde <SerializableLossOfAdherence>
C2_FPCorrelation
+ getStreamConfigurations (KafkaJsonSchemaSerde, Properties): Properties
+ createC2Stream(String, String, String, String, String, String, String, String StreamsBuilder): void
+ main(String[]): void
+ performLookUpFP(TelemetryMessage): List<SerializableFP>
+ performLookUpUFP(TelemetryMessage): List<SerializableUFP>
<<Interface>>
Processor<KIn, VIn, KOut, VOut>
+ init(ProcessorContext<KOut, VOut>): void
+ process(Record<KIn, VIn>): void
+ close(): void
C3_ConformanceMonitoring
+ getStreamConfigurations (KafkaJsonSchemaSerde, Properties): Properties
+ createC3Stream(String, String, String, StreamsBuilder): void
+ main(String[]): void
C2Processor<String, TelemetryMessage, String, CorrelatedTelemetryMessage>
+ C2Processor(UfpProducer)
ControlStreamingApplication
+ startStreamingApplication WithShutdownHook(StreamsBuilder, Properties): void
InactivityMarker<String, CorrelatedTelemetryMessage, String, CorrelatedTelemetryMessage>
+ InactivityMarker(SignalLossProducer, SecureRandom)
<<Interface>>
Punctuator
+ punctuate(long): void
C3Processor<String, CorrelatedTelemetryMessage, String, SerializableLossOfAdherence>
SignalPunctuator

Figure A.4.: Java class diagram of the Kafka Streams implementation with all de-/serializers (serdes) and processors

C1_EnrichTelemetry:

- createC1Stream(String inputTopic, String outputTopic, StreamsBuilder builder, C1_EnrichTelemetry_module c1): void

C2_FPCorrelation:

- createC2Stream(String inputTelemetry, String inputSerializedFP, String inputSerializedUFP, String outputCorrelated, String outputUncorrelated, String stateStoreLatestMessage, String stateStoreSerializableFP, String stateStoreSerializableUFP, StreamsBuilder builder): void

C3_ConformanceMonitoring:

- createC3Stream(String inputTopic, String outputTopic, String flownTrajectoryStateStore, StreamsBuilder builder): void