Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Asynchronous communication in microservice architecture using Apache Kafka

Arlan Nurkhozhin

Supervisor: Ing. Kyrylo Bulat
Field of study: Software engineering and technologies
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Nurkhozhin Arlan**	Personal ID number:	**499318**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Software Engineering and Technology**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Asynchronous communication in microservice architecture using Apache Kafka

Bachelor’s thesis title in Czech:

Asynchronní komunikace v mikroservisní architektuře pomocí Apache Kafky

Guidelines:

As part of the bachelor's thesis, analyze and suggest optimizations for asynchronous communication between microservices using Apache Kafka. Compare asynchronous to synchronous communication, explore Kafka's architecture and mechanisms, and study design patterns and their implementation practices in microservices. Besides the theoretical part, focus on developing an application based on microservices, demonstrating the use of Apache Kafka for asynchronous communication. This practical component will evaluate Kafka's impact on scalability, throughput, fault tolerance, data durability, and agility.

Bibliography / sources:

Gwen Shapira, Todd Palino, Rajini Sivaram, Krit Petty: "Kafka: The Definitive Guide, 2nd Edition"
Sam Newman: "Building Microservices, 2nd Edition"
Chris Richardson: "Microservices Patterns: With examples in Java, 1st Edition"

Name and workplace of bachelor’s thesis supervisor:

Ing. Kyrylo Bulat System Testing IntelLigent Lab FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Kyrylo Bulat	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt	Student’s signature

Acknowledgements

I want to thank my supervisor Ing. Kyrylo Bulat for enhancing this bachelor thesis and providing helpful feedback. His meticulous attention to detail helped me to identify errors I had overlooked. Furthermore I want to thank my family, friends and my partner for unwavering support during creating of this thesis.

Declaration

I hereby declare that I have submitted this bachelor thesis that I have prepared this thesis independently and that I have indicated all information sources used.
Prague, 22 May 2024

Abstract

Microservice architecture is a popular choice among developers due to its scalability, robustness, and agility. However, this architecture brings the critical need for efficient communication between microservices. This thesis focuses on analyzing asynchronous communication in microservice architecture using Apache Kafka.

The study begins by analyzing microservice architecture, including its topology, pros and cons, and examples of practical applications in large-scale systems. It introduces different communication strategies, providing an overview of their impact on system performance.

Then, the study compares asynchronous and synchronous communication in microservices, highlighting when to use each, their advantages and limitations, and their impact on scalability, performance, and efficiency. It explores different optimizations for asynchronous communications by analyzing asynchronous architectures and message brokers, focusing on Kafka's architecture and mechanisms.

Accumulated theory knowledge strengthens the implemented prototype, which utilizes Apache Kafka for microservice internal communication. The study and prototype helped to evaluate Kafka's impact on fault tolerance, scalability, throughput, agility, and data durability, which are essential factors in developing systems.

Keywords: microservices, asynchronous communication, Apache Kafka

Supervisor: Ing. Kyrylo Bulat

Abstrakt

Mikroservisní architektuřa je mezi vývojáři popularní díky své škálovatelnosti, robustnosti a agilitě. Tato architektura však přináší kritickou potřebu efektivní komunikace mezi mikroslužbami. Tato práce se zaměřuje na analýzu asynchronní komunikace v mikroservisní architektuře pomocí Apache Kafka.

Studie začíná analýzou architektury mikroslužeb, včetně její topologie, výhod a nevýhod a příkladů praktického využití v rozsáhlých systémech. Představuje různé komunikační strategie a poskytuje přehled jejich vlivu na výkonnost systému.

Poté studie porovnává asynchronní a synchronní komunikaci v mikroslužbách, přičemž zdůrazňuje, kdy je třeba každou z nich použít, jejich výhody a omezení a jejich dopad na škálovatelnost, výkon a efektivitu. Zkoumá různé optimalizace asynchronní komunikace analýzou asynchronních architektur a message brokerů se zaměřením na architekturu a mechanismy Kafky.

Nahromaděné teoretické znalosti posilují implementovaný prototyp, který využívá Apache Kafka pro interní komunikaci mikroslužeb. Studie a prototyp pomohly vyhodnotit vliv Kafky na odolnost proti chybám, škálovatelnost, propustnost, agilitu a trvanlivost dat, což jsou zásadní faktory při vývoji systémů.

Klíčová slova: mikroslužby, asynchronní komunikace, Apache Kafka

Překlad názvu: Asynchronní komunikace v mikroservisní architektuře pomocí Apache Kafky

Contents

Figures

Tables

Chapter 1
Introduction

Over the past decade, the interest in microservice architecture has grown, as shown in Figure 1.1. The graph shows the search interest level for "Microservices" within a specific region and timeframe. The y-axis measures how often the term or phrase is searched compared to the highest point in the graph. The x-axis represents a time when it was searched. The graph represents the worldwide region and a ten-year timeframe to get nonbiased data. When a term scores 100, it is at its peak popularity. If it scores 50, the term's popularity is half its peak level. A score of 0 indicates a need for more data to evaluate the popularity of the chosen term.[1]

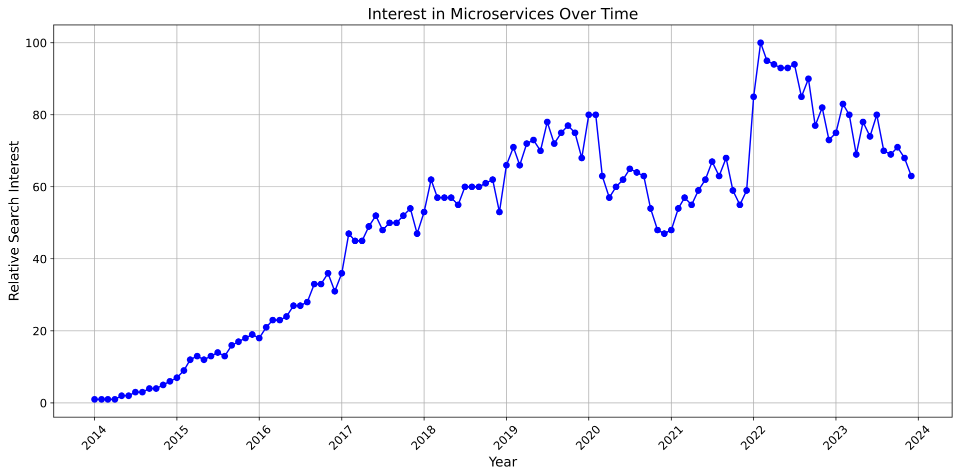


Figure 1.1: Interest over time by Google Trends for the term "Microservices".[1]

Microservice architecture has now become a technology trend, as shown in the Figure 1.2. The increased popularity of microservice architecture results from its inherent robustness, scalability, flexibility, and capability for independent deployment.[3] Communication is one of the most crucial parts of microservice orchestration. Communication between microservices can be asynchronous or synchronous, and its implementation can be realized through diverse technologies, which will be compared and discussed in this thesis.[2]

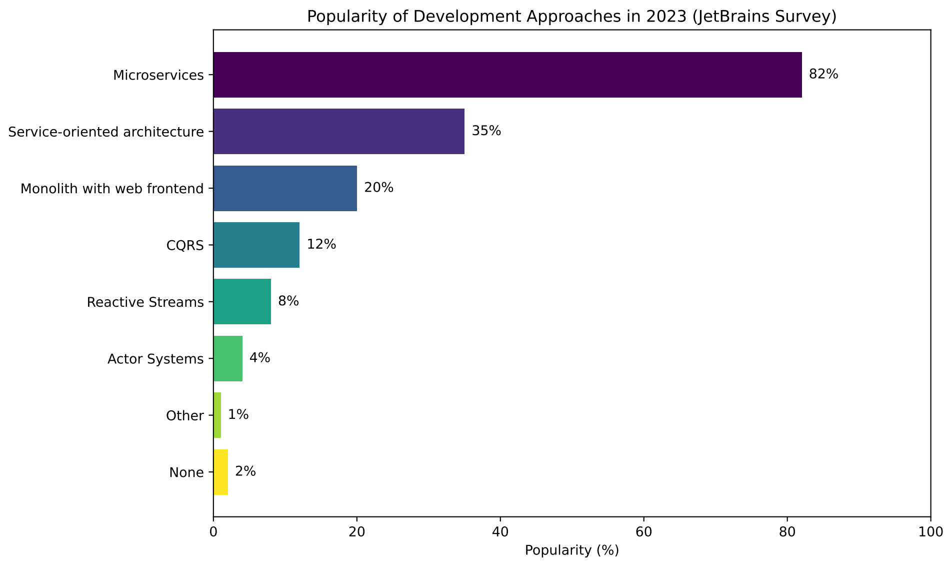


Figure 1.2: Popularity of development approaches in 2023.[2]

Chapter 2

Exploring microservice architecture and communication styles

In this chapter, we lay the groundwork for understanding microservice architecture. How it has moved from centralized monolith architecture to autonomous lightweight services, and why communication between microservices plays a key role.

2.1 Microservices at a glance

Microservice architecture is composed of small, independent components called microservices. Each microservice focuses on its specific functions, which helps to update and develop the microservice independently from other microservices. Because microservices split the whole system's functionality between each other, each of the services has its own codebase, which allows a single team of developers to focus on developing a single specific service. Additionally, independence between microservices brings technology diversity, meaning each microservice can use different technologies, such as programming languages.[4]

2.1.1 Topology

It is essential to thoroughly understand microservice and monolith topologies to understand the reasons behind the industrial shift from monolithic to microservice architecture.

As the Figure 2.1 demonstrates, the user or another server sends a request to the monolithic server. This request first reaches the presentation layer, where it is interpreted and then passed down to the business layer. The business layer processes requests and performs necessary business functions, which may include using the data access layer. The data access layer fetches, puts, updates, or deletes data based on business layer calls. Then, the response from the database travels back through the same path: first to the data access layer, then to the business logic layer, and finally back to the presentation layer, where the response is rendered and displayed to the user or server. There are direct invocations between layers, and communication is

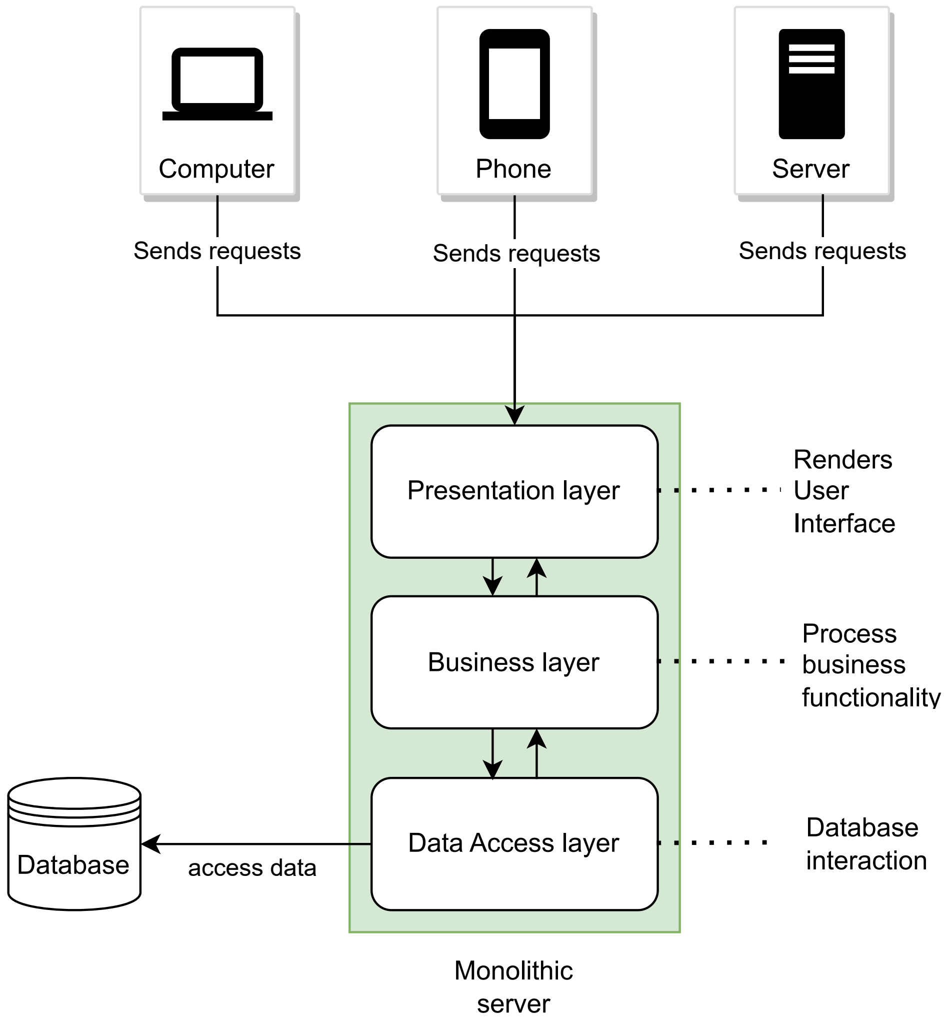


Figure 2.1: Monolithic architecture.[7][8]

synchronous. This flow leads to:[7]

- Tightly coupled and interdependent components. A change in one layer can negatively affect another layer, which could cause a whole program to crash.
- Scaling is possible by replicating the whole server again.
- Over time, the complexity of the application increases, affecting the overall difficulty of maintaining the application.
- Application is dependent on the one technological stack.

Microservice architecture effectively addresses and resolves the drawbacks inherent in monolithic systems. Figure 2.2 shows an example of this architecture.

All requests in this example go through the Application Programming Interface (API) gateway. API gateway is an entry point for incoming requests.[18] However, a gateway is not a mandatory option in a microservice architecture; for example, when microservices communicate with each other, they do not

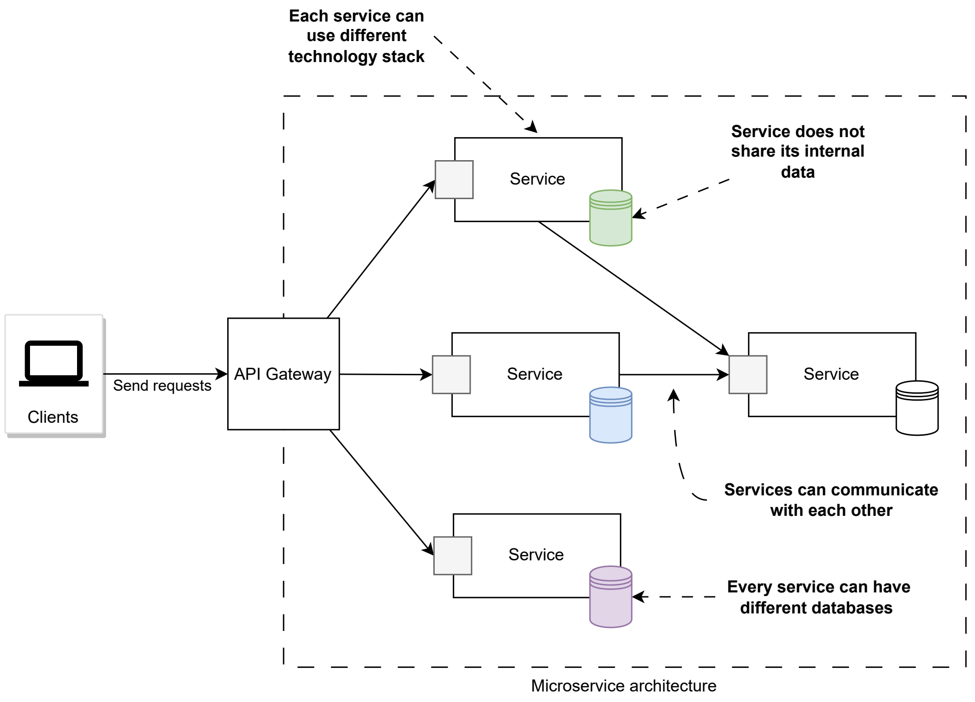

Each service can use different technology stack
Service does not share its internal data
Service
Clients
Send requests
API Gateway
Service
Service
Services can communicate with each other
Service
Every service can have different databases
Microservice architecture

Figure 2.2: Microservice architecture.[4]

need to communicate with the internet, or requests can come directly to the microservice without an API gateway. In the provided example, the API gateway then routes requests to the appropriate microservice and handles logging and the number of requests made to an API within a specified time-frame. Behind the API Gateway lies a robust collection of small, independent microservices. Each of these services solves a specific business need. For instance, one service may be responsible for authorization and another for content processing. Microservices allow developers to develop, deploy, and scale autonomously based on demand. They also promote fault tolerance and simplify maintenance, updates, and deployment cycles. Furthermore, different teams can be assigned different services, each with a unique technology stack.[4]

2.1.2 Pros and cons of microservice architecture

In the previous subsection, we compared monolithic and microservice architecture, which helps us to understand the positive and negative parts of microservice architecture. For advantages, we consider:[6][4]

- **Programming language agnostic:** Since services are independent, they can be implemented in different programming languages most suitable for the tasks.
- **Scalability:** Compared with a monolith, there is no need to scale the whole application unnecessarily. A team can choose which part of the system (microservice) to scale.

- **Robustness:** In the case of monolithic architecture, a critical error may crash the entire application. However, in microservices, it mainly affects the service itself. Keeping different parts of a system separate from one another enhances overall system robustness.
- **Easy of code maintenance:** Smaller codebases for each service is easier to understand. Updating and fixing is less risky as it will not affect the entire application.
- **Continuous deployment and integration:** Microservice architecture supports continuous deployment and integration of each service into the system without interrupting the running system.

While microservice architecture is a robust and scalable solution, there are also certain limitations:[6][15][4]

- **Complexity:** Distributed systems challenge developers in the form of additional complexity. Services must communicate with each other utilizing different internal communication mechanisms. Distributed systems are more complex than direct method calls.
- **Testing:** Testing microservices requires end-to-end testing strategies for network latency and message queues. Also, developers need to test across multiple processes handling a broad system scope. Moreover, it requires handling cases where one service dies.
- **Network latency:** Inter-service communication between microservices brings latency as it involves sending a message and serialization over the network. All these steps can result in increasing latency in the system.
- **Increased resource usage:** Due to the independence of the microservice architecture, each one needs its execution environment, which leads to increased resource usage compared to a monolith.
- **Monitoring:** Distributed systems observability involves a multifaceted approach, such as collecting logs and metrics across multiple microservices. This methodology is vital in understating the system's operational status. In addition, distributing tracing needs to be implemented to track the flow of calls, which will help investigate future errors. To ensure timely communication, it is important to set up alerts for different events in the system. These alerts should be configured in a way that notifies all relevant stakeholders. This approach to observability is instrumental in sustaining the reliability and efficiency of distributed systems.
- **Security:** In monolithic architecture, data flows within one singular process. However, data traverses over networks to other services in a microservices environment. This communication exposes data vulnerability to interception of alteration during transit. Additionally, microservices' endpoints must ensure that only authorized entities can access them.
- **Data consistency:** Ensuring data consistency in distributed systems is not as straightforward as in monolith, where data resides in a single database. Alternative approaches are required to address this problem, such as distributed transactions that maintain data consistency across different microservices. This alternative approach includes complexities.

In summary, microservice architecture can offer considerable flexibility, enabling choices in the technology stack, robustness, scalability, more manageable code maintenance, and continuous deployment and integration. These factors play a huge role in choosing this methodology. However, developers should consider increased maintenance, development, and testing complexity. For some cases, a more straightforward approach such as monolith will be sufficient, for example, when the team is small, or the application needs to be running, fulfilling all requirements quickly.

2.1.3 Microservices in practice

Having explored the foundational concepts, pros, and cons, as well as architectural designs, we are now prepared to turn our attention to the practical application of microservice architecture:

- **Netflix:** A rapidly growing user base forced Netflix to shift from monolith to microservices. Netflix began with a traditional development model, managing a monolithic DVD rental application with 100 engineers. Under the guidance of Adrian Cockcroft, the company managed to smoothly transition from monolith to microservice architecture with small teams developing hundreds of microservices. The change was compelling enough to stream digital content to millions of Netflix customers daily.[9] Based on its experience, Netflix established these best practices as demonstrated by Netflix showcases its ability to enhance flexibility, scalability, and speed in software development:[9]
  - **Creating a separate data store for each microservice:** This prevents microservices from sharing one data source, which could lead to complex dependencies.
  - **Doing a separate build for each microservice:** This allows each microservice to use component files appropriate to its specific needs.
  - **Deploying in containers:** This set one standard for deploying each microservices. The container is an independent environment where a developer can set up network and environment variables. It also ensures that failure in microservice does not crush the entire system but only a container.
  - **Keeping code at a similar level of maturity:** It advocates for handling code additions or changes by creating new microservices, thereby maintaining the stability of the existing ones.
- **Uber:** In December 2023, Uber was the largest ride-sharing company in the United States[10], having 130 million monthly active users, and the number of trips made during the first quarter of 2023 was 2.1 billion, indicating a 24 % increase year-over-year.[11] Initially, Uber's architecture struggled with inefficiencies due to its monolithic design, where all features were contained in a single, clunky system. This design made it challenging to add new functionality and fix bugs efficiently. Uber developed microservice architecture to address these issues, breaking its initial monolith into microservices.[12] This shift brought significant benefits such as:[12]

- **Improved development speed:** Teams scale up their assigned microservices without impacting other services and teams.
- **Quality and manageability of development:** High-quality software meets user requirements effectively and reliably. Because of the microservice design, the system handles bugs and errors better.

2.2 Communication strategies in microservice architecture

Transitioning from a monolithic to a microservices-based application significantly changes the communication mechanism. In a monolithic application, components communicate on function calls within one process. These calls can be either strongly coupled, where objects are directly instantiated (e.g., using `new ClassName()`), or decoupled using Dependency Injection design pattern, which references abstractions rather than concrete object instances.[13] Hence this chapter is dedicated to exploration of the pivotal part of microservice architecture - communication between microservices.

2.2.1 Communication patterns and styles

Client-service interaction in microservices can be classified along two primary axes. First axe defines if communication is one-to-one or one-to-many:[15]

- **One-to-one:** Each client request is addressed by a single service.
- **One-to-many:** Request is concurrently managed by several services.

The second axe pertains to the timing of interaction:[15]

- **Synchronous:** Client sends request and wait until response comes, often pausing its operation.
- **Asynchronous:** Asynchronous interactions do not require the client to wait, with responses potentially being deferred

One-to-one interactions can be categorized into different types:[15]:

- **Request/response:** Client sends request and wait until response comes, expecting it to be timely. This type leads to tightly coupled services.
- **Asynchronous request/response:** Client sends request to a service but does not wait for response. Response comes asynchronously which means it should not come right after sending a response.
- **One-way notifications:** The client sends a request without expecting any reply.
- **Remote Procedure Call (RPC)**: This type allows one service to call procedures or functions in other services as if they were local.

In one-to-many interactions within microservices, there are two primary types:[15]

- **Publish/subscribe:** In this interaction, a client publishes a notification message. This message is then consumed by none or multiple services that are subscribed to specific client/publisher.

- **Publish/async responses:** A client publishes a request message and waits for a predefined period for responses. Multiple services interested in this request can respond within this timeframe.

2.2.2 Impact on system's scalability and performance

In monolithic applications where communication happens in direct method invocations, this method is optimized by programming language compilers and runtime environments. Hence, these method calls have negligible overhead, often enhanced by techniques like inlining. However, the interaction between microservices involves network transmission, serialization, and deserialization of data. This leads to measurable overhead regarding resources and time compared with a monolith, where only memory pointers are typically passed inside a system. Therefore, awareness of the size of the sending data and efficient data processing becomes crucial to avoid performance bottlenecks and ensure system scalability. Consequently, developers must know these underlying communication mechanisms to optimize microservice interactions and maintain system performance.[6]

2.2.3 Ensuring system reliability and resilience

Ensuring reliability and resilience involves error handling, service self-recovery from failing, handling enormous numbers of requests, services' health checks, alerts, and logging. Unlike in single-process environments, error handling is complex in distributed systems. It involves errors inside one service and network issues such as network timeouts, unavailability of downstream services, disconnections, and resource constraints, which may lead to the service's crush. This complexity necessitates robust error-handling strategies that can address failures' unpredictable and often external nature in a distributed environment. In a single process, errors are expected and processed accordingly, or if the error is not expected, which may lead to fatal error crushing, the application error call stack is logged for future investigation.[6] Various strategies could be employed to assure resilience:[14]:

- **Retry design pattern:** Reattempting network calls that fail due to transient faults. Caution is required as retries of non-idempotent operations might lead to duplicate processing.
- **Circuit breaker:** This pattern prevents repeated attempts of likely-to-fail operations, thus avoiding resource exhaustion and cascading failures.
- **Load balancing:** Incoming requests are evenly distributed across service instances, enhancing reliability.
- **Service versioning:** Old instances are running in one line with new instances which are marked as new. It ensures continuity.
- **Security:** Implementing encryption and authentication for communication between microservices ensures security which leads to resilience.

2.2.4 Balancing trade-offs in communication choices

Choosing a specific technology or communication style is often overwhelming because the range of technology is wide. Wise consideration should be taken since the initial choice affects the future of the system or application. These considerations should be based on the following:

- **System requirements:** Functional and non-functional requirements. Functional requirements define what the system is supposed to do. One example of a functional requirement is user authentication. On the other hand, non-functional requirements define how a system should perform specific functions such as scalability, usability, or performance. Considering these two types of requirements, the technical team can choose the right tool. For instance, real-time operations are suitable for synchronous communication because they need immediate response. However, asynchronous communication may be more suitable in cases where immediate response is not important.
- **Coupling:** Desired level of coupling between microservices.
- **Team experience:** If a team is more proficient with one paradigm, it may be a good choice to stick to it as it saves time implementing the system.
- **Performance:** Latency may be an issue in some cases that require real-time processing.

In the end, a team has to understand that choice brings not only advantages but also disadvantages. For example, choosing a request/response model may be valid as it is straightforward to understand. However, it leads to a tightly coupled model, which reduces the system's scalability.

Chapter 3

Asynchronous vs synchronous communication.

Efficient communication stands as a vital pillar for building application or system. Interservice communication has two basic types of communication: asynchronous and synchronous. Both types serve the fundamental purpose of facilitating the exchange of information, yet their unique characteristics make them suitable for different scenarios.

3.1 Technical requirements

Understanding technical requirements is crucial since it affects both business and technology environments. Requirements for synchronous communication between microservices:

- **Immediate, real-time interaction:** Systems must support communication with minimum latency.
- **High bandwidth:** System must support extensive communication to support video or audio streaming.
- **Robust infrastructure:** System must be running on reliable infrastructure to provide reliable communication.
- **Service discovery:** Mechanisms to locate and communicate with other services in network.[16]
- **Load balancing:** Is device or software that distribute requests efficiently among multiple services.[17]
- **API gateway:** A single entry point for handling requests and routing them to appropriate services.[18]

In asynchronous interservice communication services are communicating via message channels. Sender writes a message into a channel and receiver reads this message from the communicating channel. Requirements for asynchronous communication:[15]

- **Event-driven architecture:** System receives various events and responds to them accordingly.
- **Message brokers:** Services in system communicate via a message broker, which acts as intermediary message channel between services.
- **Data consistency:** To ensure reliability and consistency system must

maintain data consistency across all services. It involves ensuring any update, modification or deletion of data are applied to all nodes and instances.[19]

- **Message reliable delivery:** To ensure robustness system must guarantee no messages are lost and can be proccessed eventually.

3.2 Use cases

In software, the use case refers to a hypothetical situation where a system is subjected to an external request (like user registration) and accordingly generates a corresponding response to achieve the user's goal.[20] A user, in our case, is another microservice that interacts with the system and other microservices. Use cases for synchronous communication:

- **Request/response interactions:** Where immediate response is needed. For example payment confirmation or user authentication. In this cases low latency is highly wanted.
- **Tightly coupled operations:** Situation where one service needs data or an action from another service. For instance, to process an order, system firstly need to read items in user's cart, assure these items are in stock etc.
- **Monitoring and health checks:** In cases where system needs to have an overview of services status, immediate response is needed to ensure system's robustness and fault tolerance.
- **Real-time processing:** To ensure system observability real-time processing could for example help to read and store current generating logs from the system.

Use cases for asynchronous communication:[15]

- **Decoupled operation:** In comparison with synchronous tightly coupled operations in some cases decoupled bond between microservices is more suitable. For instance, after order was placed a notification sends. Order service sends a message to a message channel. Notification service listens to the channel and waits for any message appointed to it. In this environment each service operates independently thus enhancing system's resilience as each service operates, fails and recovers without affecting other services.
- **Emails, notifications and alerts:** Emails, notification or alerts could be sent to an interested person without affecting or blocking application's main flow.
- **Message based interaction:** As described previously sender does not wait for immediate response, so sender is not blocked by awaiting for response and can continue its workflow e.g. sending emails to a lot of subscribers.
- **Load balancing:** When system is in high usage store message for later processing when system will have a capacity.

3.3 Advantages and limitations

Choice between synchronous and asynchronous communication heavily depends on system's requirements and team's experience. Selected type of interactions brings not only its advantages but also its disadvantages thus it requires a deep consideration. Advantages of synchronous communication:[13]

- **Direct response:** Synchronous protocols such as Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS) allows for immediate response from the receiver.
- **Simplicity:** Synchronous is easier to understand than asynchronous communication and it also require less work e.g team does not need to install any message channel, handle its writing, reading, scaling etc.

Challenges of synchronous communication:[13]

- **Fragility:** Since interaction is direct it leads to tightly coupled microservices. If one service fails, it can impact the entire system.
- **Performance impact:** Adding synchronous dependencies such as query requests can worsen client's experience.
- **Anti-pattern in interservice communication:** A chain of requests created between microservices in synchronous communication is considered an anti-pattern, as it can lead to inefficiencies and bottlenecks.

Asynchronous winning points:[14][21]

- **Microservices independence:** Asynchronous communication allows microservices to communicate more independently enhancing the system's overall resilience and fault tolerance. Even if one service is failed, others can operate.
- **Eventual consistency:** It facilitates replicating data across microservices to achieve eventual consistency, which is essential for maintaining data integrity across different microservices.
- **Responsiveness:** In chain of microservices, sender can respond quicker as it doesn't have to wait for receiver respond.
- **Gradual load distribution:** Message queues in message channels can be used as buffer, enabling receiver to process messages at their own pace, not forcing them for instant response.
- **Reduced coupling:** Sender does not have to know about since it communicate only with message channel, reducing dependencies between services.

Challenges of asynchronous communication:[14]

- **Complexity:** Asynchronous communication is less straightforward to implement because it involves solving challenges such as duplicated messages, request-response semantics, installing, configuring and operating this type of communication.
- **Latency issues:** If message channel's queues fill up, the end-to-end latency for operations can increase significantly.
- **Throughput limitations:** Message channels are potential bottlenecks in the system as each message may require queue and dequeue operations.
- **Messaging infrastructure coupling:** There is risk of becoming highly

coupled with a specific messaging infrastructure, making it challenging to switch to another platform later.

3.4 Scalability, performance and efficiency

After we have covered use cases and technical requirements, as well as the advantages and limitations of both communication styles, we can move to this subsection, which will highlight each contribution toward scalability, performance, and efficiency:

- **Scalability:** Asynchronous interaction offers better scalability because of discussed in above subsections: loose coupling, microservices independence in communication chains, load distribution and fault tolerance.
- **Performance:** Both types offer strong performance depending on specific context and requirements, which we described above.
- **Efficiency:** Both synchronous and asynchronous communication have their specific characteristics which can affect system's efficiency. Synchronous has direct bond between sender and receiver. It is straightforward and it is suitable for cases where instant feedback is needed. However under heavy loads in can lead to scalability issues which has negative impact on efficiency. On the other hand asynchronous interaction is loosely coupled, which means that the failure or slow performance of one service has a lesser impact on the overall system. This improves the efficiency of the system, but also brings its own limitation such as: response delays and increased complexity.

3.5 Real cases application

In this section we explore a real world scenario demonstrating successful implementation of asynchronous communication.

3.5.1 Netflix

Netflix's migration of their viewing history from synchronous request-response to asynchronous events is a notable real-world case of asynchronous communication in microservices.

Netflix is a streaming platform with more than 200 million viewers all around the world. Users can view a wide array of movies, series, TV shows, and documentaries on mobile and PC devices and on TVs. The service is designed to enhance user experience through personalized recommendations. To achieve a positive user experience while watching a movie, Netflix collects data for both operational and analytical purposes. This data supports features like "continue watching", where the user can stop the movie on one device and continue watching on another. However, it also feeds into personalization algorithms and core business analytics. The need for efficient data handling and system optimization in response to growing user demands and data landscapes was a primary aspect of migrating the viewing history feature from synchronous to asynchronous interaction.[22]

In Netflix's initial system architecture, data flow was initiated with the Gateway service, subsequently moving towards Playback API, which is responsible for managing the entire lifecycle of playback sessions. For example, at what precise time did the user stop the movie. After this stage, data was sent to the Request processor layer. The data was classified and stored in long-term and short-term viewing data. For long-term data, a relational database was used and for short-term in-memory data storage because it enables rapid data retrieval. The system worked great most of the time. However, once in a while, a strange delay occurred. Utilizing synchronous interaction in the system led to delays due to network issues or temporary shutdowns in database nodes. This situation caused a cascading slowdown from the Request processor to the Playback API and the Gateway, potentially impacting the client devices.[22] Previous architecture:

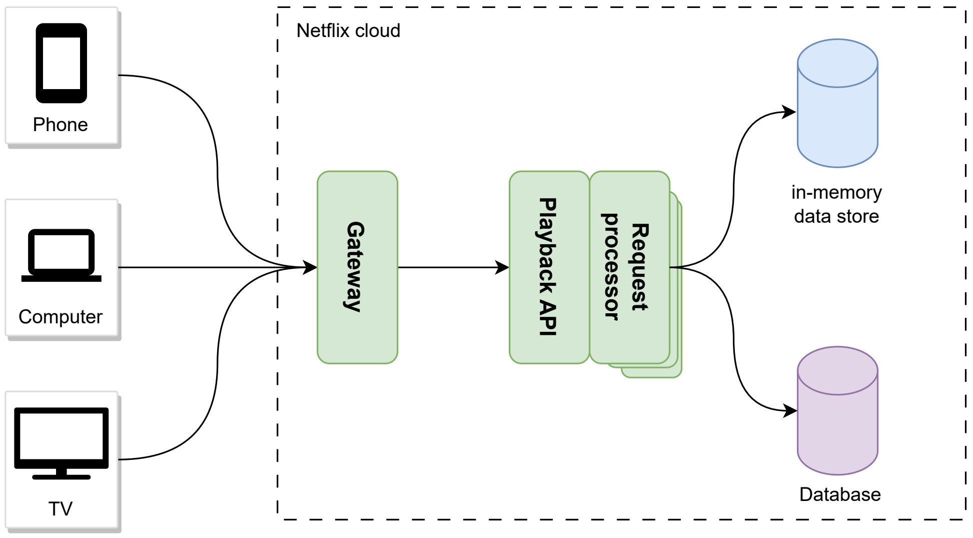


Figure 3.1: Existing system before migration.[22]

To address this, Netflix introduced asynchronous processing using a durable queue between the Playback API and the Request processor.[22]

This shift, featuring Apache Kafka, which we will discuss later, allowed for immediate client's request acknowledgement and independent processing by instances of Request processor on their own pace. This solution tackles cascading delayed responses and shows scenario where asynchronous communication is more suited. However, implementing Kafka at Netflix's scale, with challenges like data loss and processing latencies, required careful consideration and handling of complex design decisions.[22]

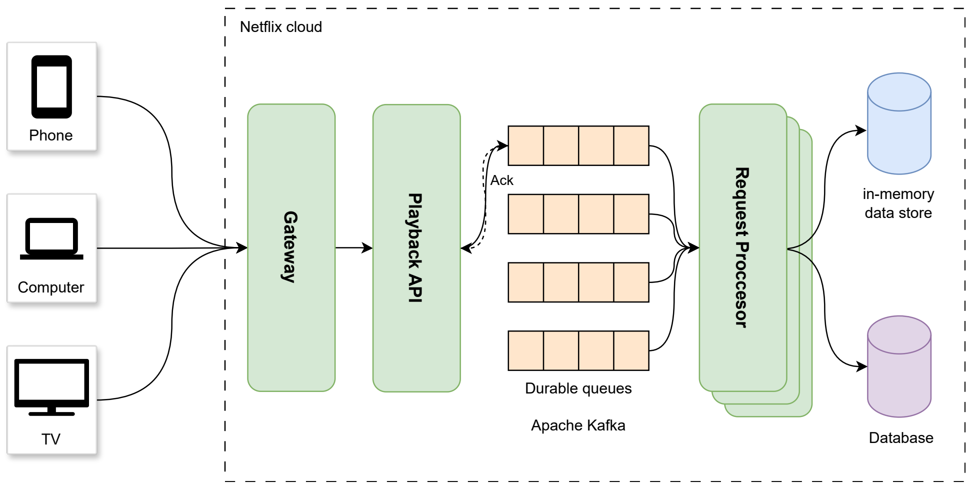


Figure 3.2: Improved system after migration.[22]

3.6 Conclusion

In conclusion, exploring asynchronous and synchronous communication in microservices architecture reveals a nuanced landscape. While synchronous interaction brings direct response and simplicity, it falls short in performance scalability fault tolerance, as highlighted in the subsections above. On the one hand, asynchronous communication offers independence of microservices, eventual consistency, responsiveness, gradual load distribution, and reduced coupling. On the other hand, it has limitations such as overall complexity, latency issues, throughput limitations, and messaging infrastructure coupling. These insights underscore choosing and aligning the communication strategy with system requirements and long-term objectives.

Chapter 4

Detailed asynchronous communication in microservices

In previous chapters, we built general knowledge to have a good overview of communication in microservices. This chapter will build a deeper understanding of asynchronous communication in microservices. By gaining a profound understanding of asynchronous communication, we can make more informed decisions and effectively implement solutions that enhance microservices-based systems' performance, scalability, and reliability.

4.1 Core architectures

4.1.1 Message queuing

A message queue is a critical element in asynchronous communication within microservices architectures. It functions as a temporary holding platform for messages that are awaiting processing. In this architecture, there is a consumer and a producer. The producer sends a message to a queue, and the consumer consumes it at its own pace. The fundamental characteristic of a message queue is that it ensures that each message is processed only once by a single consumer. This system is particularly beneficial for handling tasks that require heavy processing, managing workload buffering or batching, and moderating uneven workloads.[23]

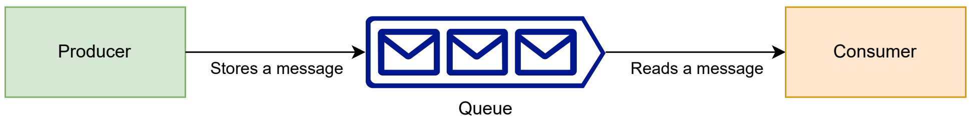


Figure 4.1: Message queuing.[23]

In modern microservice architecture, message queues help decode applications into smaller, independent units. They improve application reliability, performance, and scalability by simplifying communication between decoupled applications. Message queue provides a buffer for temporarily storing messages and endpoints for components to send and receive these messages. Additionally, message queues enable different system parts to interact and perform operations asynchronously. Messages in these queues can range from requests and responses to error messages and general information. Producers add, and consumers retrieve messages. While multiple producers and consumers can interact with the queue, each message is uniquely processed by only one consumer. Message queues can integrate with publish-subscribe messaging patterns to create a fanout design pattern for scenarios where a message needs to be processed by multiple consumers.[23]

4.1.2 Publish-subscribe architecture

This model allows messages to be broadcast to multiple subscribers from a publisher through a topic. In publish-subscribe messaging, publishers create and send messages to a specific topic; subscribers receive these messages by subscribing to that topic.[24]

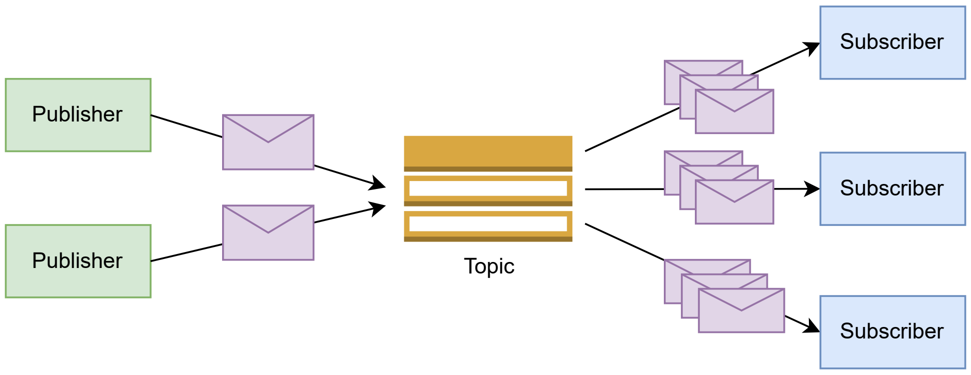


Figure 4.2: Publish-subscribe messaging.[24]

The publish-subscribe messaging system, a pivotal component in modern distributed systems, comprises four essential elements:[24]

- **Messages:** A message is communication data between publishers and subscribers. The message could be a plain string, image, video, sensor data, complex object, and other types.
- **Topics:** As an intermediary channel, a topic is associated with each message. Each topic has its list of subscribers who listen to new incoming messages.
- **Subscribers:** These are the recipients of the messages. To receive messages, subscribers must register or subscribe to their topics of interest. Upon receiving messages, subscribers may execute diverse functions or process the message in different ways, often in parallel.
- **Producers:** Producers create and send messages to appropriate topics. Consumers broadcast messages to all waiting subscribers at once. Broadcasting forms a one-to-many relationship where publishers broadcast information without needing to know the message consumers, and likewise, subscribers receive messages without needing to identify the source.

The publish-subscribe model enables working with events. For example, an event can trigger an update like adding a new item to a cart.[24]

4.1.3 Event streaming architecture

Event streaming can be likened to the central nervous system of the digital world, forming the backbone of a continuously connected, software-driven business environment. It involves capturing various data in real-time from event sources like databases, sensors, mobile devices, and other software as an event stream. These event streams are later processed, stored, manipulated, or routed to different destinations. The essence of event streaming lies in its ability to enable real-time and retrospective analysis and reaction, ensuring that relevant information is available precisely when and where it is needed.[25]

Event streaming can be used in a multitude of scenarios. Some of its key uses include processing financial transactions in real-time for banks and stock exchanges, monitoring and tracking vehicles in logistics, analyzing sensor data in industrial settings, managing customer interactions in retail, monitoring patients in healthcare, underpinning data platforms, event-driven architectures, and microservices.[25]

4.1.4 Event-driven architecture

Event-driven architecture is commonly used in microservice-based architecture to enable interservice communication between decoupled services using events. Events are either updates or changes in state, for instance a new successful payment transaction. They can carry the state such as a previous transaction example or serve as identifiers, like a notification that user received a message. An event-driven architecture has three main components: event consumers, event producers and event routes. Producer creates an event and send it event router which filters and redirect event to consumers.[26]

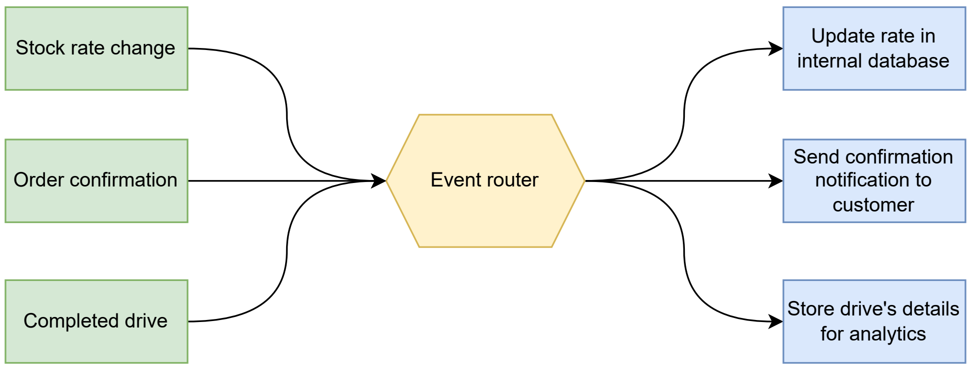

Stock rate change
Order confirmation
Completed drive
Event router
Update rate in internal database
Send confirmation notification to customer
Store drive's details for analytics

Figure 4.3: Event-driven architecture.

The benefits of an event-driven architecture:[26]

- **Independent scaling and failure management:** Because producers and consumers are decoupled and they do not have to know each other, the failure of one will not affect others. In an event-driven system, only the event router is aware of the loss of the service.

- **Agile development process:** The event router automates the filtering and delivery of events to consumers, reducing the need for extensive and manual coordination between producer and consumer services. The event router significantly accelerates the development process.
- **Efficient auditing:** An event router provides a centralized platform for auditing applications and setting policies. It is easier to manage access policies to data and who can publish and subscribe to a router.
- **Cost reduction:** Event-driven architectures operate on a push-based model, meaning actions occur as events appear in the router, avoiding the costs associated with continuous polling to check if an event appeared. Additionally, it reduces network bandwidth and CPU utilization, which reduces costs.

4.2 Message brokers

In this chapter, we will take a look at message brokers. A message broker is a software that facilitates communication and data exchange between different applications, systems, and services. It achieves this by converting messages into various messaging protocols, allowing seamless interaction between services, regardless of programming language or system platform. The message broker serves as an intermediary between services. It can validate, store, and route to various destinations.[27]

4.2.1 Apache Kafka

Apache Kafka is a distributed platform for handling large amounts of real-time data. Multiple data sources continuously produce real-time data, often simultaneously, creating a constant flow of incoming information. Apache Kafka solves the distributed platform's main problem: managing the ongoing influx of data and processing it sequentially and incrementally.[29] Kafka's use cases:[29]

- utilizing a publish-subscribe architecture. This approach helps eliminate multiple point-to-point connections, leading to system complexity. Apache Kafka also addresses publish-subscribe drawbacks such as lack of fault-tolerance and scaling issues.
- storing produced messages durably and sequentially. It allows consumers to read messages in chronological order.
- handling streams of data in real-time.

4.2.2 RabbitMQ

RabbitMQ is a general-purpose distributed message broker used in stream processing. RabbitMQ consists of three main components: exchange, binding, and queue. The exchange receives the messages and decides where to route the message. The queue receives messages from the exchange, store messages and then send them to the consumer. Binding connects the broker and exchange.[30]

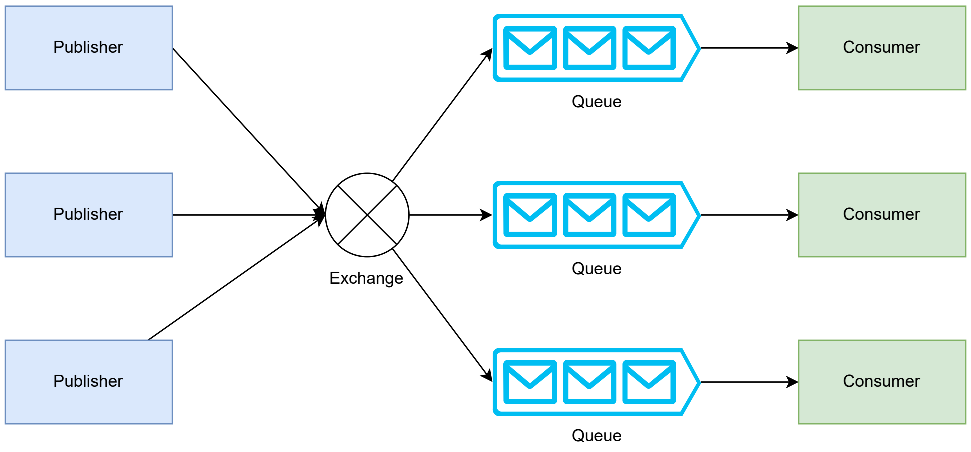


Figure 4.4: RabbitMQ architecture.[31]

Messages are high-volume, continuous, and incremental data that require rapid processing, such as sensor data streams. RabbitMQ ensures message delivery, and messages are not stored for long periods as they are in Apache Kafka. It means that when a subscriber reads a message, it sends an acknowledgment to RabbitMQ, which deletes it.[30] RabbitMQ can be used in situations where:[30]

- **Message delivering needs to be guaranteed:** Because RabbitMQ utilizes a push model, it verifies whether or not the message is delivered.
- **Complex routing architecture:** RabbitMQ has the flexibility to send messages to various destinations with bindings and exchanges.
- **Broad support:** RabbitMQ supports a broad array of programming languages compared to Apache Kafka. It also supports old protocols such as STOMP (Streaming Text Oriented Messaging Protocol).

Chapter 5

Deep dive into Apache Kafka

Every application uses data in some other, and in the evolving world of big data and distributed systems, Apache Kafka helps deliver messages from publishers to consumers. Each byte of information is valuable in every message, and servers must deliver data with minimum latency to ensure the best user experience or a competitive advantage over other enterprises. For example, e-shop platforms generate dozens of data to help provide personalized user recommendations based on user activity or real-time inventory management. In financial banks, Apache Kafka can help address fraud or market risks by analyzing transactions as they occur.[28]

5.1 Messages and batches

Message in Apache Kafka is the primary data transmission unit. From the database perspective, the message in Apache Kafka is similar to a row or record in the table. Producers create messages and send them to Kafka. Then, consumers fetch messages from Kafka. The message is an array of bytes containing the message's value and optional metadata such as key and headers.[28]

Component	Description
Key	The key is optional and decides in which partition the message should be sent. Messages with the same key will be stored in the same partition, guaranteeing the order within the partition.
Value	The value is the content of the message. It can be string, number, or complex, serialized data structures such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML).
Headers	Headers are optional key-value metadata that help provide more information about the message.

Table 5.1: Apache Kafka message structure.[28]

Publishers send messages in Kafka, and consumers read them in batches. A batch is a collection of messages sent to the same topic and partition.

This approach improves throughput and efficiency because it eliminates network round-trip. Round-trip starts with the producer sending a batch of messages over the network. Once the Kafka broker gets a batch, it sends an acknowledgment back to the producer. On the receiver side, a consumer requests to fetch incoming messages over the network. Then, the broker gets a request, and it responds with a batch of Kafka messages. Once the consumer gets a batch, a round-trip is done. So, instead of sending each message individually, the producer can send a collected group of messages, reducing the network load.[28]

5.2 Topics and partitions

Topics and partitions are foundational components in the Apache Kafka.

„*Messages in Kafka are categorized into topics. The closest analogies for a topic are a database table or a folder in a filesystem. Topics are additionally broken down into a number of partitions.*“[28]

Multiple consumers can read the topics simultaneously, and one topic can have more than one partition. Partition is a log sequence. The sequence in this term emphasizes the defined order in which the producer appends messages. Messages always append at the end of the sequence; consumers can reference every message by its position. However, Apache Kafka ensures the order of messages strictly inside the partition, not across the partitions.[28]

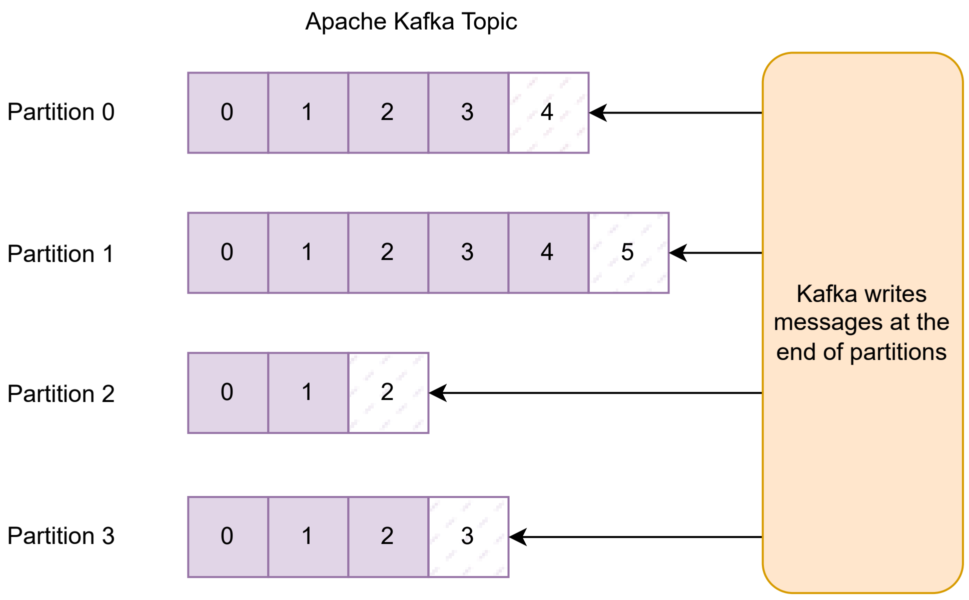


Figure 5.1: Apache Kafka appends messages from producers.[28]

Different Kafka brokers can host individual partitions, distributing the load across the server infrastructure. Additionally, Apache Kafka can replicate partitions across various brokers. It brings not only redundancy but also availability, ensuring efficient horizontal scaling.[28]

5.3 Producers and consumers

Producers and consumers are essential parts of the Apache Kafka architecture.

The producer can be an application or a service that creates and publishes messages to Kafka. The producer can determine which partition stores a message using the key metadata described in the previous chapter. If the key is not specified, Apache Kafka automatically decides where to place the message.[28]

Consumers are the entities that fetch messages from the Kafka broker. To read the message from the broker, consumers need to subscribe to one or more topics. Consumers maintain the order of the messages in which producers originally produced them within each partition. Each consumer holds an offset value to avoid repeating fetching and identify the current position in the partition. An offset in Apache Kafka is the unique value that sets the Kafka broker after receiving a message. The producer itself does not set offset. Holding offset also allows the consumer to resume fetching data from the failed position in case of interruption. This mechanism is fundamental in Apache Kafka architecture and ensures efficient and accurate message consumption.[28] Figure 5.2 shows the consumer group example.

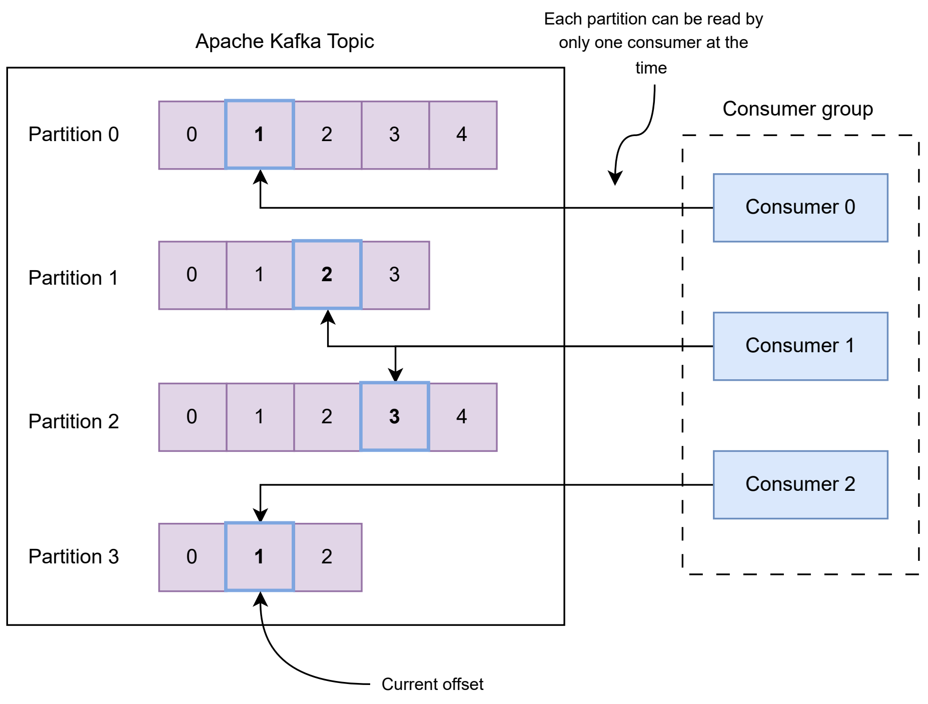


Figure 5.2: Consumer group consumes a single topic.[28]

Consumers are usually in consumer group reading data from one or more topics together. By fetching data together, consumers can consume it faster than if only one consumer was doing it. However, it is essential to note that only one consumer can read from a single partition in a Kafka topic simultaneously.[28]

5.4 Brokers and clusters

A server in Apache Kafka which stores messages is called a broker. The broker stores messages in partitions and assigns unique offsets to messages, and partitions are inside topics. The broker's liability goes well beyond storing messages. For instance, the broker is also responsible for handling incoming requests from producers sending messages and consumers fetching those messages.[28]

> „*Depending on the specific hardware and its performance characteristics, a single broker can easily handle thousands of partitions and millions of messages per second.*“[28]

A cluster is a group of brokers. Compared to a single broker, a cluster is more robust and scalable since the cluster has more brokers working together. Since several brokers are in the cluster, brokers choose a leader, called a cluster controller. The cluster controller checks the health of other brokers, and if one broker fails, others can take the failed broker's responsibility. The controller also assigns partitions to brokers, and each partition is replicated across brokers to ensure redundancy. A broker assigned a partition is called a partition leader, and brokers replicating this partition are called followers. To publish messages, producers must connect to a leader. However, consumers can connect to a non-leader to fetch messages.[28]

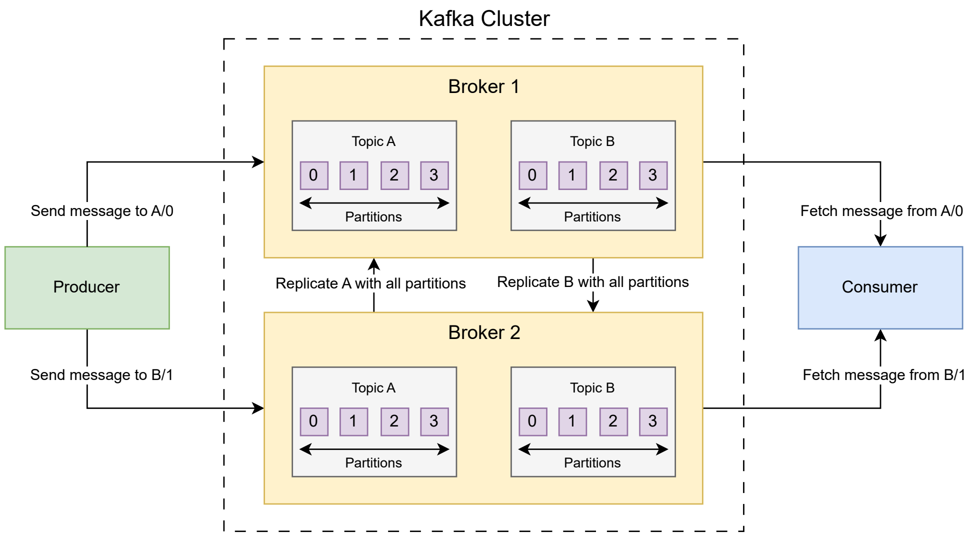


Figure 5.3: Kafka cluster architecture.[28]

Since Apache Kafka stores all messages on disk, setting up a retention policy is essential. Retention policy, in this case, is the set of rules that defines when to delete messages, and it ensures disk storage does not fill up. Retention can be in two ways:[28]

- **Time-based retention:** For instance, the timespan of the message can be seven days. It means that after seven days, Apache Kafka deletes the message.
- **Size-based retention:** For example, the maximum size of the topic or broker is 10 gigabytes. After the total size of messages reaches this value, older messages Apache Kafka deletes to free up space for new messages.

Based on an application or system's use cases, retention can be set up individually in each topic, which brings flexibility to the system.

5.5 Kafka's strengths and weaknesses compared to other message brokers

There are other technical solutions similar to Apache Kafka, however, what makes Apache Kafka unique:[28]

- **Multiple producers and consumers:** As was previously described, Apache Kafka can handle multiple producers and consumers. It allows storing incoming messages from the producers in a single place without overloading connections between producers, Kafka, and consumers. It also enables consumers to fetch data from one place without connecting to the specific producer who creates the needed data.
- **Storing messages on the disk:** Kafka consumers do not need to process data in real-time since Apache Kafka stores messages on the disk. Suppose the consumer cannot fetch messages from the topic for some reason, such as network latency or heavy traffic. In that case, it can retry from the previous place after some time without losing previous messages. However, it is crucial to manage disk space efficiently. As described in Section 5.4, Apache Kafka offers a retention policy to prevent the disk from filling up.
- **Scalable:** Apache Kafka can scale horizontally because of partitions, consumer and producer groups, replication, and the ability to add new brokers to the Kafka cluster. The broker can replicate partitions to other brokers to parallelize the processing of data from consumers. Considering brokers, Kafka can add more brokers to the cluster to handle any amount of data, and producers can send messages to different partitions in the topic concurrently.

These features combine to make Apache Kafka a high-performing publish-subscribe message broker, ensuring low latency.

5.6 Real world examples highlighting efficiency, scalability and performance

Apache Kafka is trusted by more than 80% of companies in the Fortune 100, which indicates that these companies trust Apache Kafka to provide scalable, performant, and robust business solutions.[34] The Fortune 100 list is published annually by Fortune Magazine and ranks the top 100 United States (U.S.) companies by their reported annual revenue. It contains both publicly traded and privately held companies.[35]

5.6.1 LinkedIn

LinkedIn is

> „*the world's largest professional network with more than 1 billion members in more than 200 countries and territories worldwide.*“[32]

LinkedIn initially developed Apache Kafka as an internal project, which grew into an open-source project with a high usage rate outside of the creator company, which shows the project's usefulness. Apache Kafka plays an essential role in LinkedIn's operations, along with other technologies. It has responsibility for message exchanges, metric gathering, activity tracking, and more. Kafka infrastructure in LinkedIn consists of over 100 clusters with more than 4000 brokers. Together, they have more than 100000 topics and seven million partitions. A remarkable indicator of this scale is the volume of messages processed, which in 2019 surpassed 7 trillion per day.[33]

5.6.2 The New York Times

The New York Times has a vast history in offline and recently with online journalism. It provides online access to the content, even if it was posted long ago. This content must be accessible and searchable by different applications and services at very low latency.[36] Initially, The New York Times relied on an API-based system which was:[36]

- developed by different teams, which led to incidents in the endpoint configurations and data schemas.
- challenging to manage API because services were tightly coupled.
- Each consumer service requires information at a different level of immediacy. For instance, some services require immediate availability of the latest publishing for real-time updates or a comprehensive archive of previously published content.

The new Apache Kafka-based system simplified and centralized data consumption by various services as seen in Figure 5.4 The new design addressed previous issues and brought the following advantages:[36]

- The new design standardized data flow from producers to consumers, improving the software development process for front-end and back-end services.

- Deployment simplification.
- Improved monitoring of publishing content throughout the system, from publication to end-user.

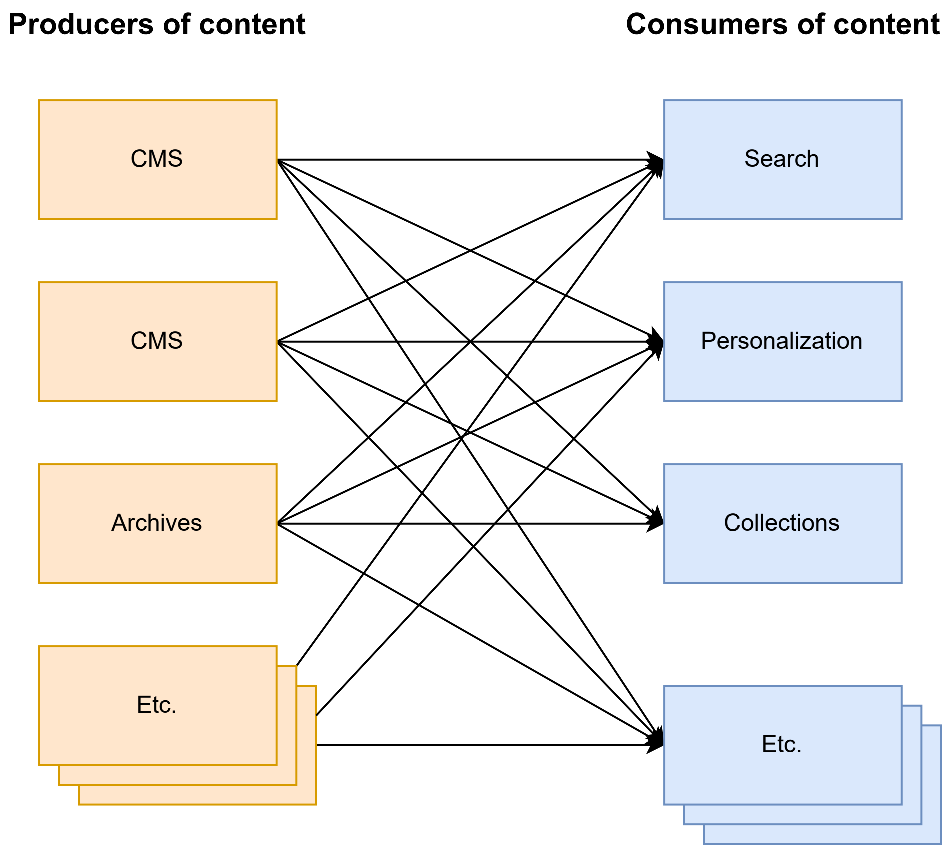


Figure 5.4: The New York Times old design.[36]

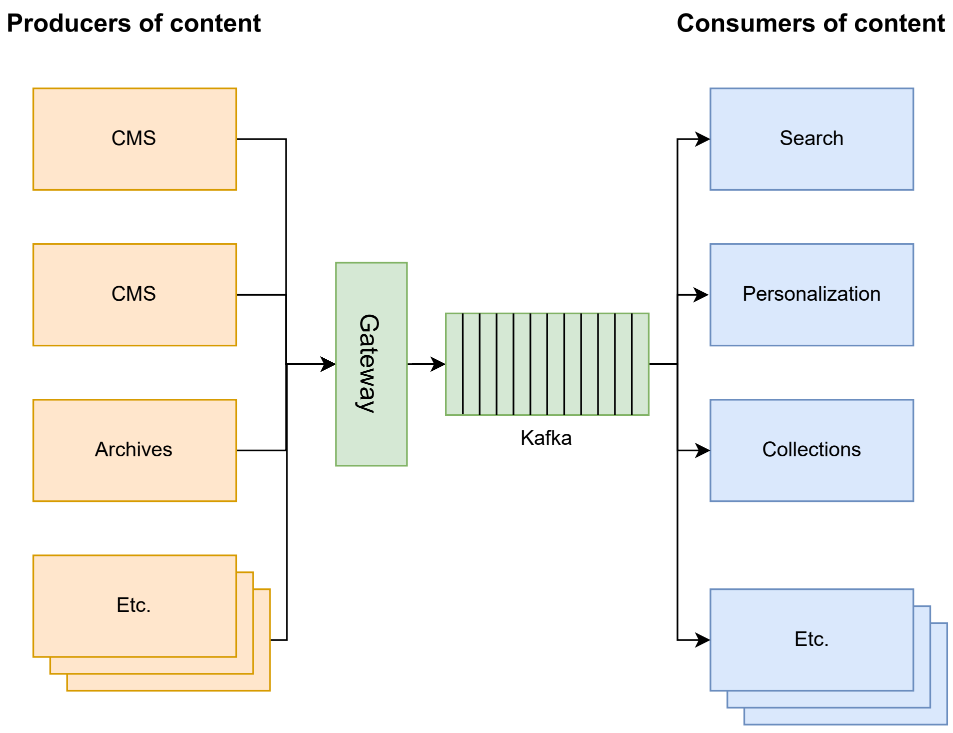


Figure 5.5: The New York Times new design.[36]

Chapter 6
Implementation of a prototype

In the previous chapters, we explored microservice architecture and its communication strategies, mainly focusing on asynchronous communication in microservices. After that, we explored Apache Kafka, which can help to utilize asynchronous inter-service communication, and this section introduces the operational details of the prototype. It includes system architecture, its components, and the required set of technologies to implement this prototype. I decided to implement a part of a ride-sharing application like the real-world ride-sharing service Uber to demonstrate the practical usage of microservices and asynchronous communication utilizing Apache Kafka.

6.1 Tools and technologies used

There are plenty of technologies to implement a ride-sharing prototype. In software development, the choice of instruments should be guided by specific requirements and the team's experience. In this case, the main requirements are scalability, throughput, fault tolerance, data durability, and agility. So, this section outlines choices for a set of technologies used to create and run prototypes.

6.1.1 Java

Java is a secure, multi-platform, and fast programming language. It is powering millions of Java applications and has been used for over two decades.[37]

In this prototype, Java serves as the primary language powering microservices. It was chosen because of the author's familiarity with this language and because it provides:[37]

- **Ton of online resources:** Because Java is a relatively old and popular language, there are many online resources to learn different aspects of Java and its libraries and frameworks, making learning Java convenient and efficient.
- **Rich built-in libraries:** Java provides plenty of useful functions and libraries.

- **High-quality development tools:** Tools that come with Java enable developers to debug, test, and deploy tools effectively.

6.1.2 Spring Boot

Spring Boot is a secure, flexible, configurable, and fast Java-based framework built on top of the Spring framework.[38] It speeds the development of microservices in general and the development of prototype because of the following:[38]

- **Eliminating boilerplate code:** Spring Boot provides auto configuration. It automatically configures Spring and third-party libraries without manually setting up configurations. For example, when developing the prototype, most of the configuration was done by Spring annotations, which sped up prototype development.
- **Starter dependencies:** It provides starter dependencies that save the developer valuable time from setting up a project. So the project can just run. For instance, the prototype uses dependencies to connect to Apache Kafka and Redis databases and manage WebSocket communication. There was no need to select and match versions of libraries manually; everything worked perfectly out of the box.
- **Dependency Injection:** Spring Boot uses Dependency Injection and Inversion of Control, which delegate the construction of dependencies to external entity. Dependency Injection is the technique that allows an object to specify its dependencies through constructor arguments or properties instead of constructing them internally. It makes the code clean and elegant.[39]
- **Easy to use and rich libraries:** Spring ecosystem provides high-quality libraries which helped during the development of the prototype. Spring Kafka and Spring Data provide tools to manage Kafka and Redis efficiently, which also helps them spend more time writing code.

6.1.3 Redis

Redis is a Not Only SQL (NoSQL) in-memory database. It operates in the Random Access Memory (RAM), being close to the applications instead of the disk, which gives the database a higher speed for reading and writing operations than those that work on the disk. Even though Redis is a key-value store, it also has advanced structures such as sets and hashes.[40] Redis was chosen as primary data storage because it offers:[40]

- **High performance:** Since Redis works with data in memory instead of hard disk, it offers high write and read database operations. It brings enhanced responsiveness and is suitable for the ride-sharing backend with many users. Particularly for managing the ride lifecycle from point A on the map to point B. Redis is an ideal candidate since a ride is often needed during its active duration. So, it allows quick access and data modification, which is crucial for real-time changes. After the trip, it can be archived in another permanent storage area. Understandably, with a small amount of test users, the difference in speed is unnoticeable. However, with the growth of test users and then real users, the difference will be noticed because of the characteristics of relational versus in-memory databases.

- **Flexible schema:** At the moment of prototype development, strict schema was not required. Dynamic schema allowed the development of the application to be done quickly without the need to manage database relations.
- **Disk persistence:** Even though Redis is an in-memory database, it can store data snapshots on the disk. So, after the database is restarted or crashed, the data will not be lost. It helps with debugging, and data can be restored in case of a crash or outage.
- **Redis data types:** It supports not only string data type but is also not limited to hashes, lists, sets, sorted sets, geospatial indexes, and binary-safe data.[41]

6.1.4 Uber H3

Uber H3 is a geospatial indexing system that Uber internally developed to enhance and optimize geographic information generated from millions of daily transactions. This information needed to be analyzed for optimized dynamic pricing in Uber's internal services. H3 uses a grid system to split the cities into hierarchically indexed hexagons. It helped Uber detect areas in the city that were presented as hexagons for high and low demand.[42]

Hexagons as grid cells in the system were explicitly selected in favor of triangles and squares because the distances between the hexagon center and neighbor centers are the same, which is one value compared to triangles and squares where the distance between neighbors varies as shown in Figure 6.1. Because the distance for hexagon neighbors is the same, it drastically simplified data analysis in Uber.[42].

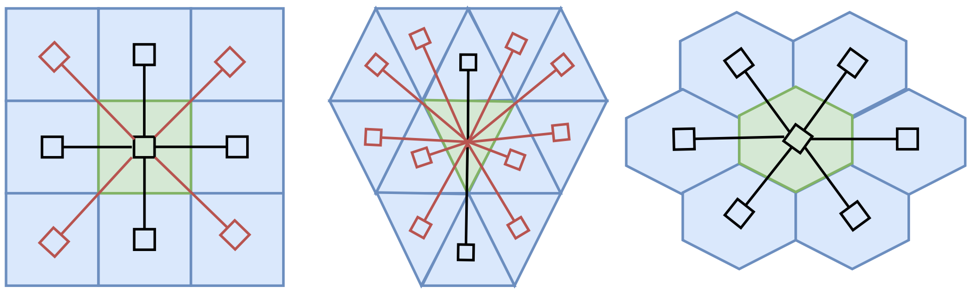


Figure 6.1: Distances between centers of math figures. The red lines have different lengths from the black lines.[42]

The H3 system was utilized in the prototype to store drivers in the Prague hexagon grids. When drivers move through the city, they change their current hexagons calculated by the H3 system. H3 also has a function to retrieve the cell's neighbor, which is used to find nearby drivers. The example of the grid-divided map of Prague is in Figure 6.2 where red dotes are drivers. Hexagon cells can have different sizes depending on the demand of the particular parts of the city. For simplicity, in the prototype, grids are the same size.

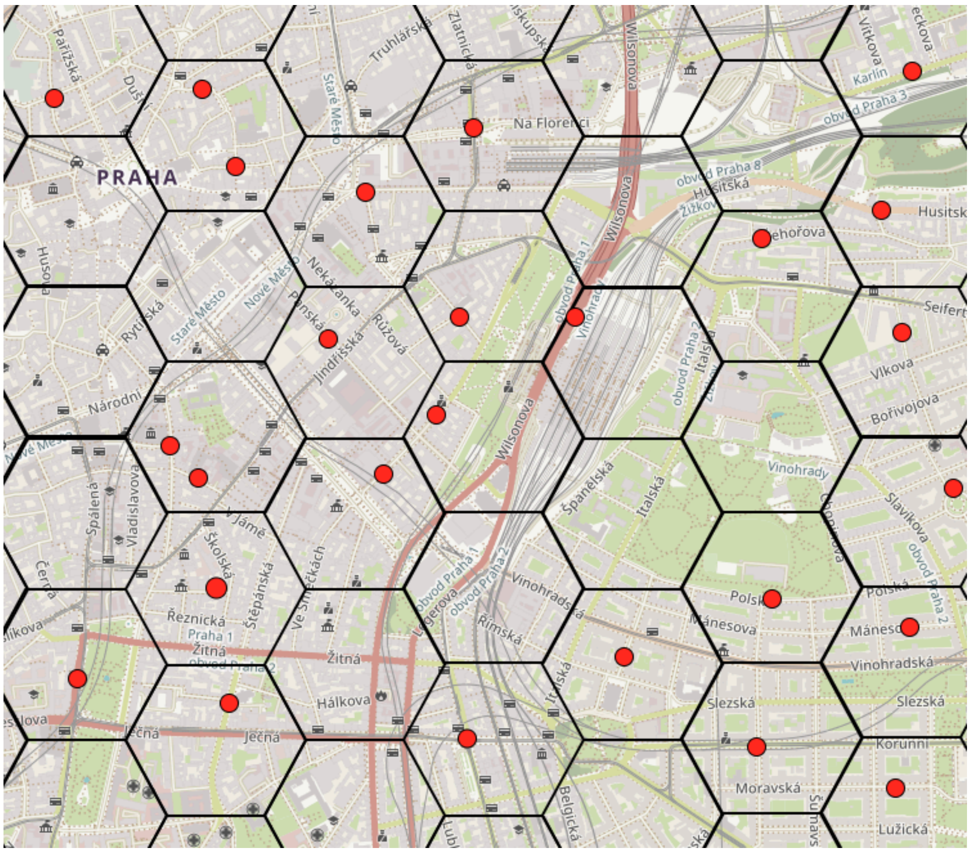


Figure 6.2: Part of Prague map divided by hexagon grid. Red dots are drivers.[42]

6.1.5 Docker Compose

Docker Compose is a tool that allows the management and definition of multi-container Docker applications. Docker itself is an open platform designed to simplify the development of software. It enables developers to define, build, run, test, and ship isolated containers. Those containers are lightweight, isolated, and do not depend on the operational system.[44] Docker Compose and Docker were chosen because they offer:[43][44]

- **Cross-platform:** Docker can be installed on multiple platforms. If the prototype is developed further with other developers, they do not have to manually download, run, and operate databases and Apache Kafka.

Redis databases and Apache Kafka are defined in Docker to eliminate the need to provision infrastructure and focus on shipping features.

- **Standartised flow:** One configuration file defines the environments for running databases and Kafka, reducing manual infrastructure setup. In the prototype, Docker is also responsible for setting up Kafka topics and their configurations, which provide a convenient way to develop applications.

- **Isolation:** Redis databases run inside containers that divide them from the main operational system. In case of data flooding and corruption, it will not impact the operational system itself. Created data can also be easily deleted in one place without manually searching for them. Not only databases but Apache Kafka is isolated from other containers, which provides modularity.

6.1.6 WebSocket

WebSocket is a TCP-based protocol that enables two-way, full-duplex, low-latency communication between sender and receiver. It is used for live chats, multiple collaboration on whiteboards, and live location tracking.[45]

Firstly, the server and client must open a WebSocket connection. It is called a handshake, which includes a request/response message between sender and receiver.[45]

After a handshake, the client and server can start transmitting messages over the WebSocket connection.[45]

The connection is established once and can be terminated by sending a close message to one of the sides.[45]

For communication between drivers/riders and the backend was chosen WebScoket protocol because:[46]

- **Real-time communication:** Communication between the driver/rider app and the server should be in real time for a better user experience. Since WebSocket allows messages to be sent at any time, user clients can receive real-time notifications and updates about rides as soon as an event occurs in the system.

- **Lower latency:** Because WebSocket maintains a persistent connection between a client and server, it does not have to establish a new connection whenever a client or server wants to send a message. Connection is established once before the communication.

- **Communication is full-duplex:** It means that the server and client can send messages simultaneously. They do not have to wait for a response from the receiving side, and because of their nonblocking nature, they can handle a high volume of messages.

In the prototype, WebSocket communication was implemented by the Spring Boot Starter Websocket library using the Simple Text Oriented Messaging

Protocol (STOMP) protocol. STOMP is a subprotocol and can work over the WebSocket protocol, providing a convenient way to pass messages between client and server.[46]

6.2 System architecture and components

In this prototype, implementation was focused on the core functionality of ride-sharing applications. Specifically, the subset of features: find ride, cancel ride, start ride, and end ride. These features were selected because they are fundamental for the operation of ride-sharing applications.

The system architecture in Figure 6.4 and in Figure 6.5 was built on microservices that utilize Apache Kafka for scalable and efficient message passing. The components used in the diagram are described in Figure 6.3. The setup allows each part of the system to operate independently yet cooperatively.

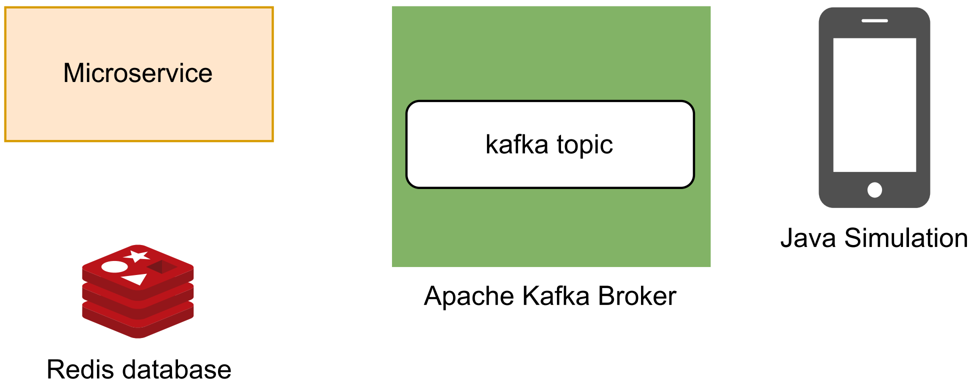


Figure 6.3: Design's components.

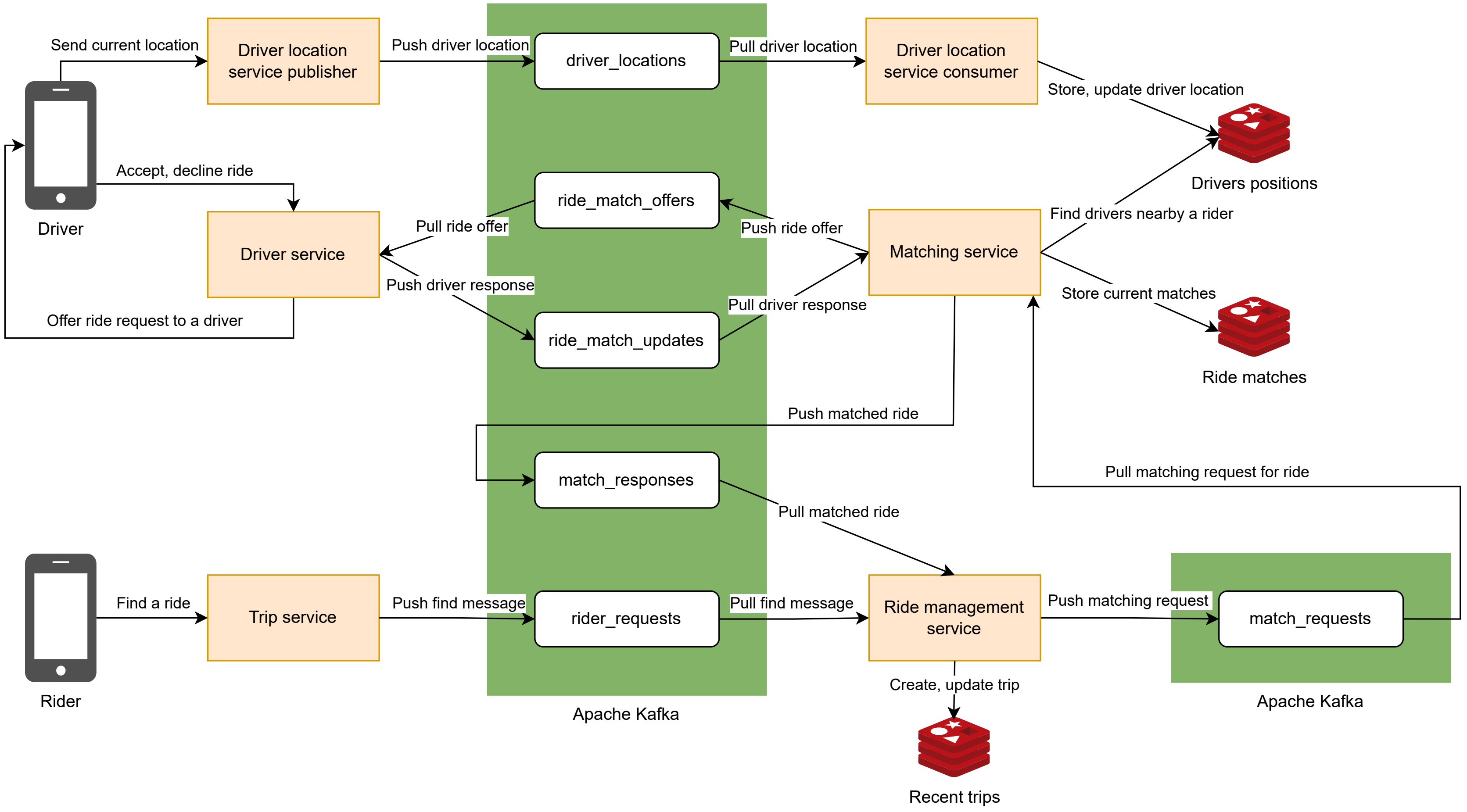


Figure 6.4: Ride sharing prototype design part 1.

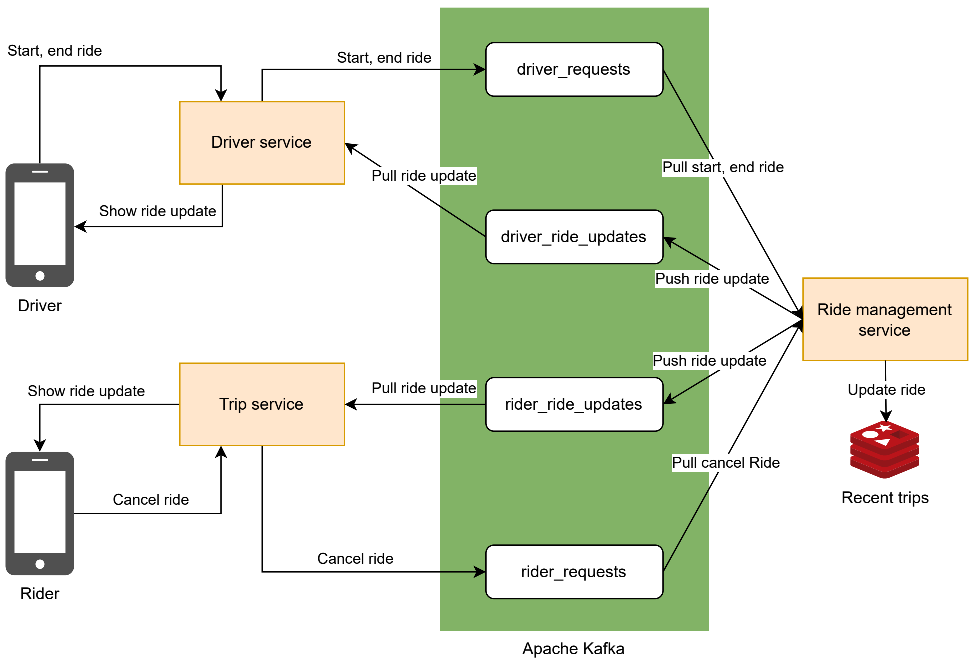


Figure 6.5: Ride sharing prototype design part 2.

6.2.1 Apache Kafka

In the ride-sharing prototype, Apache Kafka plays an important role in data flow between various microservices. It helps to achieve:

- **Asynchronous communication:** When a microservice publishes a message, it does not have to wait for a response, blocking the whole microservice. This means that consumers and producers do not interact directly but via Apache Kafka.
- **Decoupling of services:** Apache Kafka enables microservices to work independently and communicate effectively. All the microservice has to know is the port and hostname of Apache Kafka and the topic where to send messages. It does not have to manage other microservices ports and hostnames. This approach streamlines service interactions, making the overall system more scalable and easier to maintain.
- **Data storage:** Data in the topics persist in the disk, which helps with debugging and testing. Data in the topics persist in the disk, which helps with debugging and testing. For future development, it can help in case of consumer unavailability. Because data are not lost, consumers can start fetching when they connect to Kafka.

Overview of used Kafka topics:

Topic Name	Description
driver_locations	It stores recent location updates from drivers. These locations are used to track and manage driver positions in H3 system.
ride_match_offers	Contains offers sent to drivers for potential rides, based on matching logic that considers closeness to the rider.
ride_match_updates	Receives updates from drivers about whether they accept or reject the ride offers.
match_responses	Collects responses from the matching service to the ride management service about matched rides.
match_requests	It is used to submit ride requests from riders, which need to be processed by the matching service to find suitable drivers..
rider_requests	It gathers initial ride requests from riders. This topic is also used to send and consume cancel ride messages.
driver_ride_updates	Handles ride status updates from riders or drivers during the lifecycle of the ride, such as ride cancellation, start, end, or match. Used to send updates to drivers.
rider_ride_updates	Handles ride status updates from riders or drivers during the lifecycle of the ride, such as ride cancellation, start, end, or match. Used to send updates to riders.
driver_requests	Used by drivers to send messages such as start or end a ride.

Table 6.1: Detailed descriptions of Kafka topics used in the ride-sharing prototype.

6.2.2 Driver

The driver in this prototype is the Java Spring Boot application. At the very beginning of running the simulation, it creates the test driver data.

Drivers in the application can have one of the following states (`driverStatus`), which determine their availability and actions within the system:

- **AVAILABLE:** The driver is free and can accept new ride requests.
- **PICKING_UP:** The driver is on the way to pick up a passenger.
- **ON_THE_TRIP:** The driver is transporting a passenger.
- **DECIDING:** The driver sent a response for a ride offer and currently waiting for a match.

Drivers send their current locations in the time range from 6 to 12 seconds. The time range was implemented to simulate real-world scenarios such as latency and to add variability because each driver logs in at different times. Driver simulation is connected via WebSockets to Driver location service publisher and Driver service. When an update from Driver service comes, the simulation updates a specific driver's status and does actions such as start ride and end ride.

When the driver responds positively to incoming Matching service offers, the status changes from `AVAILABLE` to `DECIDING`, waiting for the ride-matched update message from the Ride management service. Then, when an update from the Ride management service comes that the ride is matched to the driver, the driver's status changes from `DECIDING` to `PICKING_UP` to mimic the driver heading to the rider. After 5-second the driver sends `START_RIDE` message to Driver service, and the driver's status changes from `PICKING_UP` to `ON_THE_TRIP`. Then, after 10 seconds, the driver sends `END_RIDE`, and the status changes from `ON_THE_TRIP` to `AVAILABLE` to simulate a situation where the driver gave the passenger a ride.

Driver entity has the following schema:

Field name	Description
id	The unique identifier for the driver.
riderId	The unique identifier for a rider who is currently with the driver
rideId	The unique identifier for a ride which driver is currently delivering
driverStatus	The current status of the driver, represented as an enumeration.
currentPosition	The current location of the driver, stored as a `MapPoint` object, which includes latitude and longitude coordinates.

Table 6.2: Database schema for the driver entities.

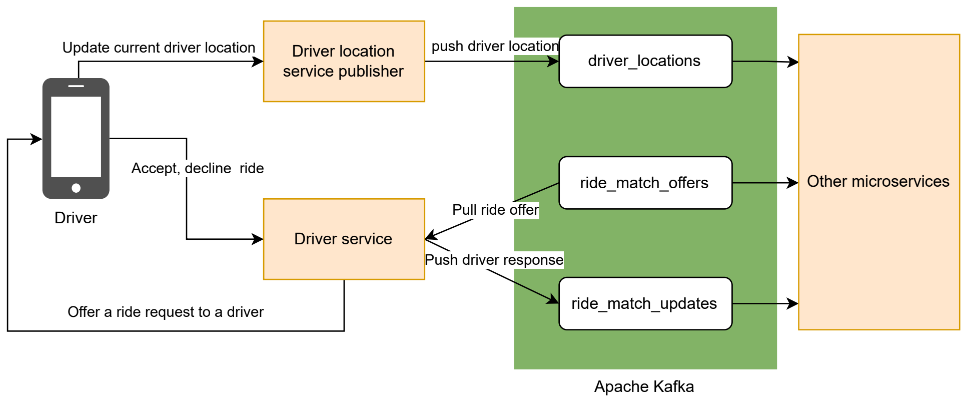
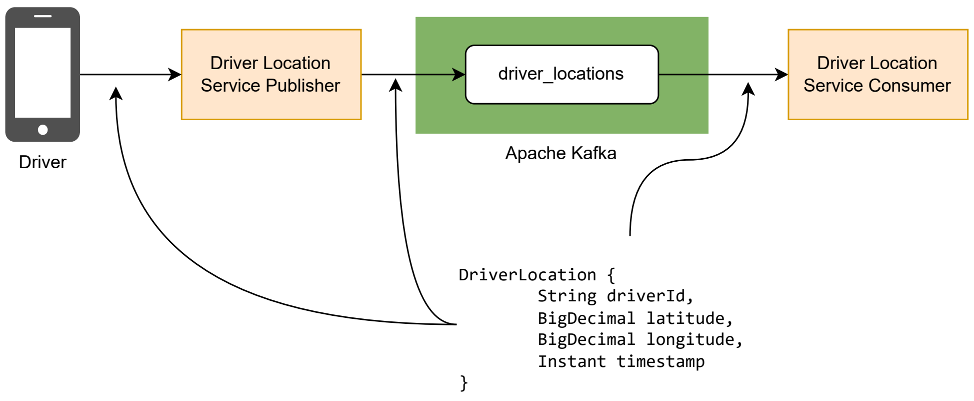


Figure 6.6: Driver workflow in a ride-sharing prototype.

6.2.3 Driver location service publisher

It utilizes WebSockets as a server to simulate real-time connection with drivers because it constantly needs to receive driver's position updates with minimal latency. It also has the role of Kafka producer. When the driver's current location arrives via WebSocket communication from the driver simulation service, the publisher forms a Kafka message as shown in Figure 6.7 that includes the driver ID, latitude, longitude, and timestamp when the message was sent. Even though microservices have different codebases, they use one message format defined in the helper library to reduce repeating code.


Driver
Driver Location Service Publisher
driver_locations
Apache Kafka
Driver Location Service Consumer

```
DriverLocation {
        String driverId,
        BigDecimal latitude,
        BigDecimal longitude,
        Instant timestamp
}
```

Figure 6.7: Driver location service publisher workflow uses one message format.

6.2.4 Driver location service consumer

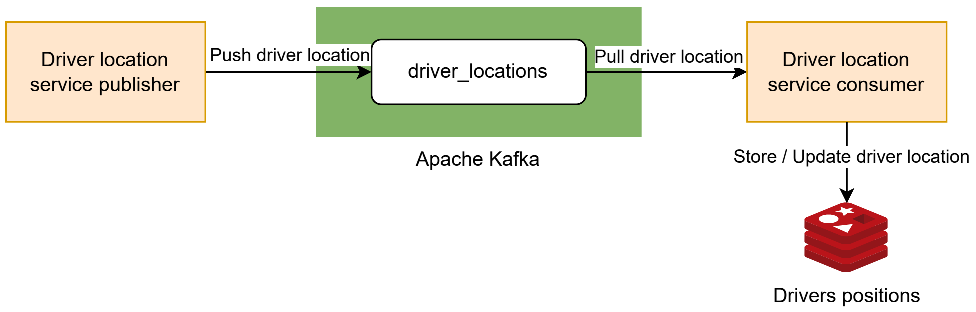

Driver location service publisher
Push driver location
driver_locations
Apache Kafka
Pull driver location
Driver location service consumer
Store / Update driver location
Drivers positions

Figure 6.8: Driver location service consumer workflow.

Driver location service consumer fulfills the Kafka consumer role by fetching incoming messages from the `driver_locations` topic. When the message arrives, it deserializes message into a Java object containing all the necessary data to store the driver's location effectively.

Then it uses H3 methods and algorithms to convert the driver's longitude and latitude into the H3 cell address where the driver is located. The address is a unique index of the cell in the grid. Drivers with different longitudes and latitudes may be stored in one cell grid if they fall within the exact cell boundaries. For example, in Figure 6.10, drivers are marked as red and blue dots on the map. All of them understandably have different geolocations, but reds are stored in one cell, and blues are stored in another, based on their respective cell locations.

In the Redis, drivers are organized by their geographic location into H3 cells as shown in Table 6.4; drivers shown as red dots on the map are stored under the same cell index due to their proximity. Meanwhile, those shown as blue are grouped into a different but similarly unified cell index.

While driver is moving and sending his location, the microservice updates the driver's current H3 cell in the `Driver positions` database as in the Table 6.3 and groups drivers in the h3 cells as in the Table 6.4 for quick driver ids retrieval in Matching service.

Driver id	H3 index cell
12345	8a1e354110effff
67890	8a2a1072b597fff
....	....
54321	8a1e354110effff

Table 6.3: Example of mapping each driver ID to their corresponding H3 index cell.

H3 index cell	Set of driver ids
8a1e354110effff	{12345, 54321}
....	....
8a2a1072b597fff	{67890, 34567, 00001}

Table 6.4: Groups of driver ids stored in set in each H3 Index Cell.

6.2.5 Driver service

Driver service's primary role is communicating with driver simulation:

- Updating ride status initiated from the driver. The driver can send requests to start, end, accept, or decline a ride. These actions must be propagated to either Ride management service or Matching service to manage the ride lifecycle.

- Update ride status initiated from riders. Driver service will send riders' ride requests to specified driver ids retrieved from `ride_match_offers` Kafka topic. Ride offer is the rider's request for a trip from point A to point B

Driver service is also a WebSocket server for communication between drivers and the ride-sharing backend. By being the communicator between other parts of the system and the driver, driver service provides:

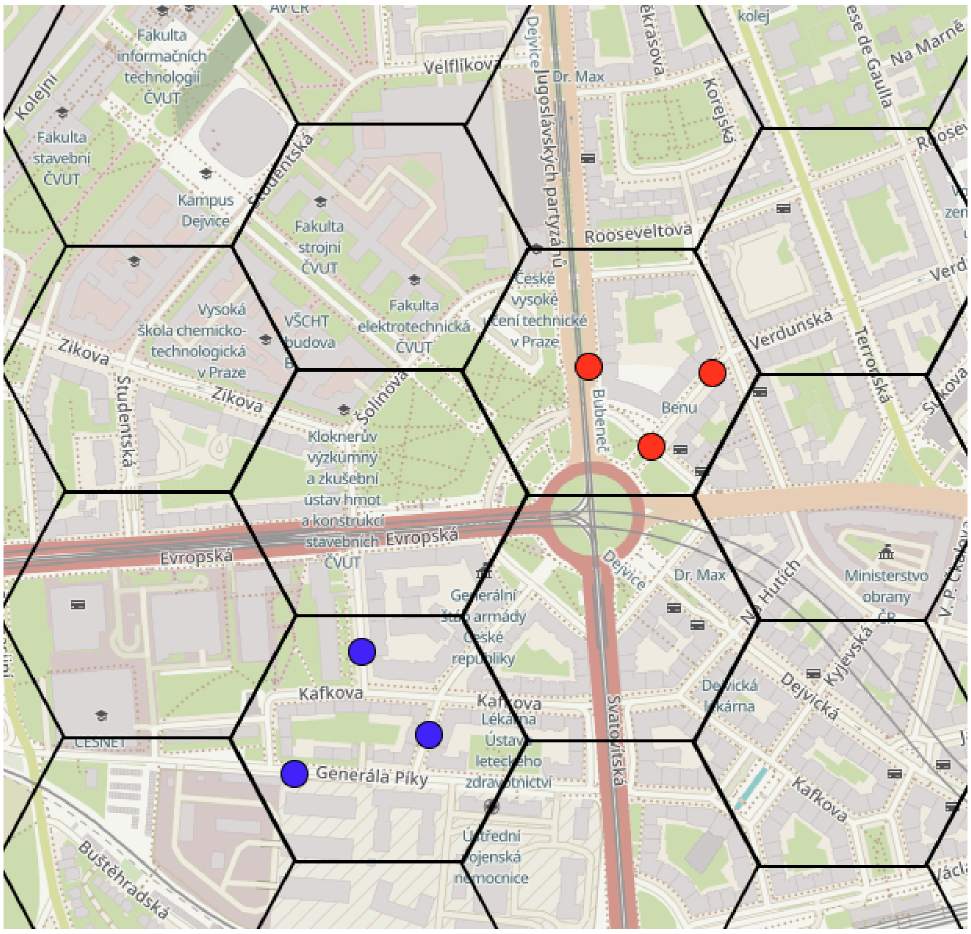


Figure 6.9: Red and blue drivers are stored in their specific unified cells.

- **Security:** Driver service can in future handle and sanitize data input before sending it to Kafka instead of direct communication between drivers and Kafka topics.
- **Clearer codebase:** Driver simulation logic is separated from driver service, which makes the overall codebase organized and clean. Each service is responsible for its domain logic.

6.2.6 Matching service

The matching service is responsible for matching drivers and riders. The rider requests a ride, and the service provides the nearest driver who accepts the offer. The matching service connects to the two Redis databases: Redis matches and Driver positions.

Firstly, the matching service receives request to match a rider to some driver from the ride management service. The message looks like this:

```
{
  "riderId": "riderIdWhoRequestedRide",
  "rideId": "assignedRideId",
  "riderPosition": {
    "longitude": 14.4378,
    "latitude": 50.0755
  },
  "startPosition": {
    "longitude": 14.4208,
    "latitude": 50.0875
  },
  "endPosition": {
    "longitude": 14.4313,
    "latitude": 50.0874
  }
}
```

Then it creates an entity in the Ride matches database with following schema:

Field name	Description
id	The unique identifier for ride matching. The same id as in Recent trips database
status	An enumeration representing the status of the matching ride: `MATCHED, NOT_MATCHED`
Stored in Redis with a time to live of 3600 seconds (1 hour).	

Table 6.5: Redis schema for managing ride matching status.

Then, the Matching service utilizes the H3 library to determine the rider's H3 cell index. After that, the service calculates the neighbor cells of the cell where the rider is located. It gets a list of H3 indexes of neighbor cells, including the cell where the rider is located, a total of seven cells. Then, using these indexes, it retrieves drivers located in those indexes. If there are some drivers, it sends them ride offers via `ride_match_offers`. If there are no drivers in the first ring of cell neighbors, the matching service increases the radius until it finds available drivers. Drivers can accept or decline the offer. If the offer was accepted, a matching ride in the Ride Matches database is marked as `MATCHED`. Next, incoming acceptances are ignored since the ride is matched. Then it sends a message to the ride management service saying that the ride is matched:

```
{
  "driverId": "matchedDriverId",
  "rideId": "rideIdThatNeedsToBeMatched"
}
```

The following needs defined the usage of the Redis database in the Matching service:

- **Consistency and integrity:** There can be a lot of incoming match requests, and information on whether the ride was already matched needs to be stored somewhere. One way was to use some data structure inside the microservice. However, it would require implementing synchronized operation and handling concurrent modifications. It would increase the overall complexity of the code base in the matching service. Redis is designed to handle an enormous load concurrently, which makes it the best candidate.
- **In-memory database:** Read and write operations in the database need the minimum possible latency to ensure a smooth user experience.

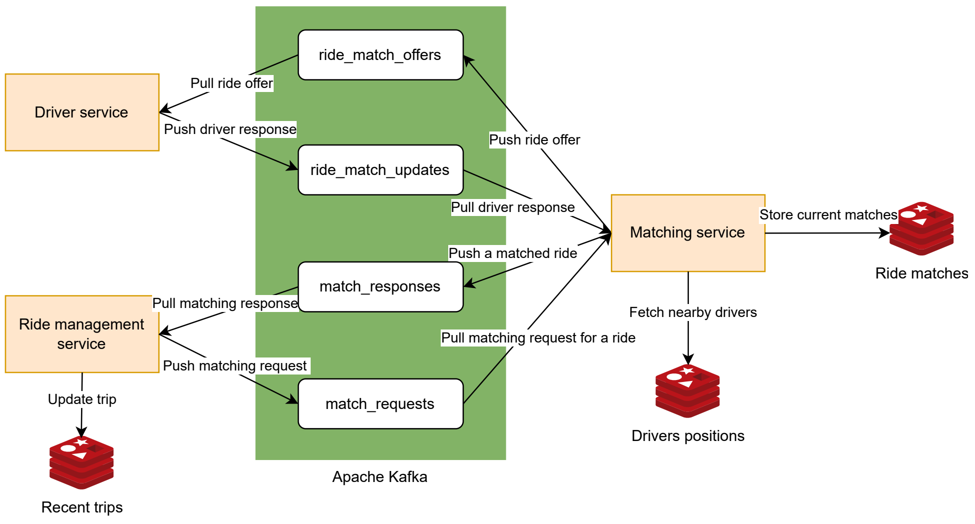


Figure 6.10: Matching service workflow.

6.2.7 Ride management service

This service is designed to manage the lifecycle of each ride, from initiation to completion.

The average flow starts at `CREATED` and goes through `MATCHING`, `MATCHED`, `STARTED` and ends in `FINISHED`. However, the ride's lifecycle can go to `CANCELED` in case the rider cancels the ride in the `CREATED`, `MATCHING` or `MATCHED` phase. Ride's lifecycle is presented in Figure 6.12.

Ride management service also use the Recent trips database to store ride information. The ride information database schema looks like in the Table 6.6:

Ride management service expects messages from riders via `rider_requests` Kafka topic. Riders can request or cancel a ride. When a rider request message arrives at the ride management service, it creates an entity in the Recent trips database with the status: `CREATED`.

After that, the service marks the ride entity in Redis as `MATCHING` indicating that the ride is currently matching and then requests the Matching service to match a rider with some available driver via `match_requests` topic.

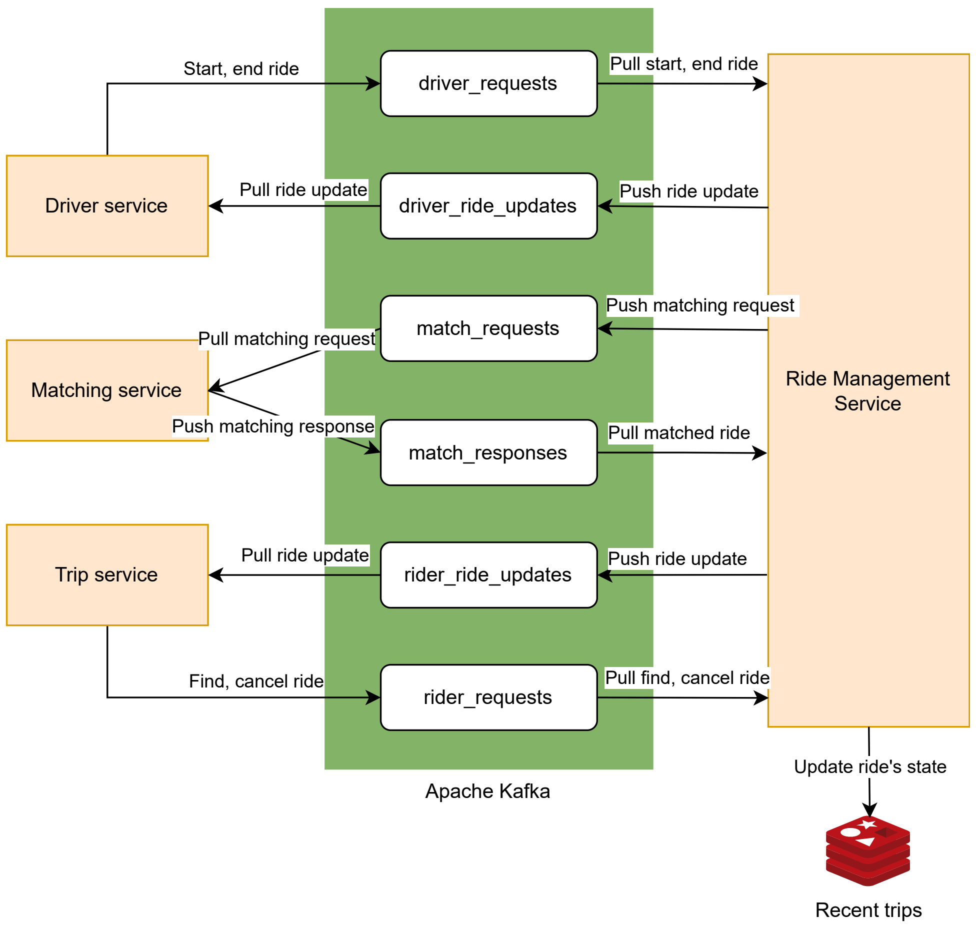


Figure 6.11: Ride management service workflow.

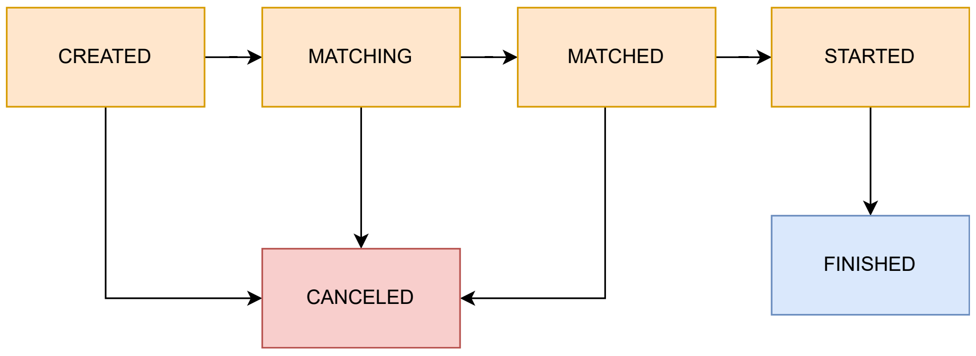


Figure 6.12: Ride statuses.

After some time, the Matching service sends a message with a found driver. Management service sets the driver id in the ride entity and changes ride status to `MATCHED`. After the change in the database, microservice sends an update that the ride is matched to the driver and rider via topics `driver_ride_updates` and `rider_ride_updates`.

If the driver starts or ends a ride, this driver action comes via `driver_requests` topic, then ride management service updated trips as `STARTED` or `FINISHED`.

If the rider cancels a ride, the service updates the ride entity status to `CANCELED`. The ride management service sends ride status updates to driver and rider so they know about ride status changes. It ensures synchronization between driver and rider.

Field name	Description
id	The unique identifier for each ride.
driverId	The unique identifier of the driver assigned to the ride.
riderId	The unique identifier of the rider who requested the ride.
rideStatus	The current status of the ride: `CREATED`, `FINISHED`, `MATCHING`, `MATCHED`, `CANCELED`, `STARTED`.
startPosition	The starting point of the ride containing latitude and longitude.
endPosition	The endpoint of the ride.

Table 6.6: Ride entity database schema in Redis.

6.2.8 Trip service

Like in the Driver service, Trip service serves as the primary communicator with the riders. It sends updates about the ride's status and receives requests from the rider to find or cancel a started ride via WebSockets. Implementing a special microservice for communicating with riders provides modularity and security. It has the role of Kafka consumer and producer at the same time. To pass the rider's request to the system, it creates a Kafka message and sends it to `rider_requests` topic. Respectively, it reads incoming messages from `rider_ride_updates` that contain updates regarding rides' statuses. After receiving updates from the Ride management service, it sends updates to the rider simulation. The update can be one of the: `MATCHED_RIDE`, `STARTED_RIDE`, `FINISHED_RIDE`, `CANCELED_RIDE` and tells the rider how to react to the update.

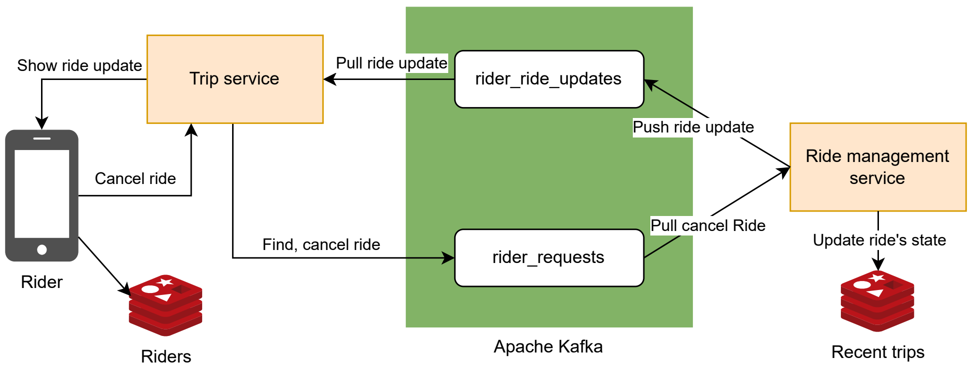


Figure 6.13: Rider simulation.

6.2.9 Rider

The Rider is a Spring Boot Java application that simulates riders' behaviors. It utilizes Redis for the internal synchronization of riders and WebSocket client to send riders' requests to the Trip service.

At the start, all riders start with the status `AVAILABLE`. Every 5 seconds, half of the riders with status `AVAILABLE` request a ride. Moreover, ten percent of drivers seeking a ride cancel their requested trip. When the trip service sends messages to the simulation, the simulation updates riders accordingly. For instance, when a trip sends an update that:

- The ride was matched, the simulation updates the rider's status who requested a ride as `WAITING_FOR_DRIVER`.
- If the ride was started, the rider's status is marked as `ON_TRIP`.
- If the ride was canceled, the rider's status returns to `AVAILABLE`.
- The ride ended. The rider is marked as `AVAILABLE` again.

This database schema defines the rider entity:

Field Name	Description
id	The unique identifier for each rider.
riderStatus	The current status of the rider: `ON_TRIP`, `AVAILABLE`, `LOOKING_FOR_RIDE`, `WAITING_FOR_DRIVER`.
currentDriverId	Stores the driver's identifier associated with the current drive.
currentRideId	Holds the identifier of the current ride the rider is involved in.

Table 6.7: Database schema for the 'Rider' entity stored in Redis.

6.3 Evaluating the impact of Kafka in the prototype

Scalability, throughput, fault tolerance, data durability, and agility are properties of the whole system, not just Apache Kafka. It means that every part of the system and prototype plays a role in those characteristics. But in this section we will focus on how Apache Kafka benefits the prototype.

6.3.1 Scalability and throughput

Integrating Apache Kafka into the ride-sharing application significantly enhances the system's scalability. Kafka's ability to handle high volumes of data through partitioning allows the system to distribute load across multiple consumers. For instance, `driver_locations` topic is expected to constantly receive messages from Driver locations service producers and fetch messages by location consumers. This situation raises possible bottlenecks in the system where consumers, producers, or the topic will be overwhelmed by the incoming driver locations.

To address this and other similar issues, we can introduce topic partitions, consumer groups, and multiple instances of microservices. In the case of the Driver location service publisher, we can introduce multiple instances of this microservice to match the expected load. Then, we can divide the topic into multiple partitions. Because the topic is divided into partitions, we must introduce a way to effectively and smartly store driver locations into relevant topics instead of choosing Kafka's automatic partitions logic. We can implement a hash function that considers geographic location, driver ID, and time when the message was sent. This function will decide in what partition to put the message. On the other side, considering this hashing information, for example, one or more consumer instances operating in the consumer groups can read driver locations from partitions they are interested in. The example is shown in Figure 6.14

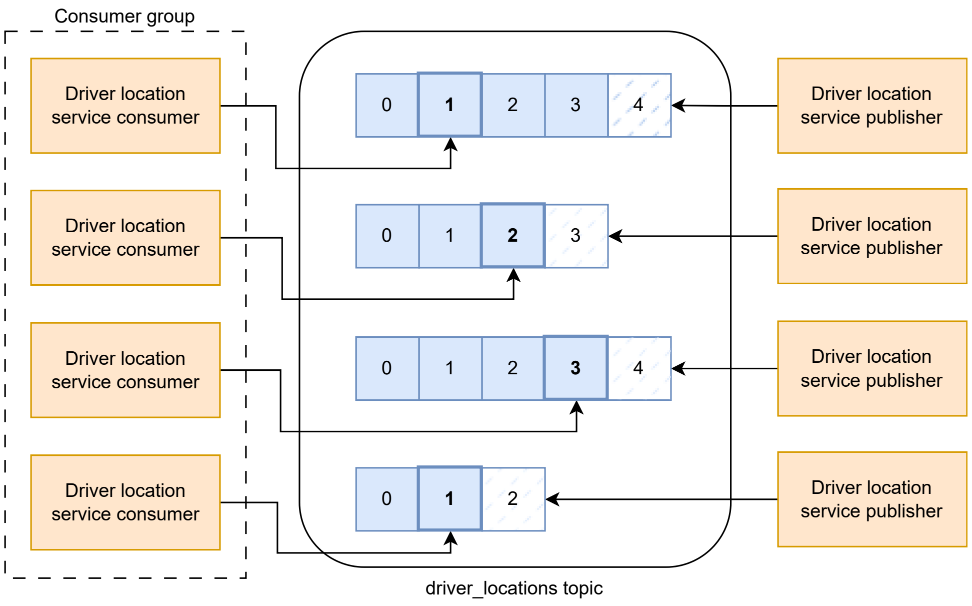

Consumer group
Driver location service consumer
Driver location service consumer
Driver location service consumer
Driver location service consumer
0 1 2 3 4
0 1 2 3
0 1 2 3 4
0 1 2
Driver location service publisher
Driver location service publisher
Driver location service publisher
Driver location service publisher
driver_locations topic

Figure 6.14: Topic partitions, multiple consumers and producers for scalability and throughput.

As traffic increases, Kafka supports the addition of more brokers to the cluster without downtime, as shown in Figure 6.15, which adds points to the system's scalability. It also ensures vertical scaling.

Using partitions, additional Kafka brokers, and multiple instances of microservices also increases overall throughput, enabling the system to handle more concurrent users even under peak loads.

Kafka internally also utilizes the zero-copy method. It is a technique that reduces CPU usage and network throughput. When Kafka sends a message to clients, it sends message as bytes directly to the network channel without intermediate buffers, unlike databases, where data are stored in a local cache before being sent to the client.[28] Kafka allows each microservice instance to process messages independently as a producer or consumer, which scales the system's ability to handle more operations.

Additionally, Kafka plays the role of buffer between consumers and producers. It means that the producer's throughput does not have to match the consumer's. Each can send or fetch Kafka messages at their own pace. To further improve the throughput of the prototype, we can enable message compression, which will reduce the size of messages sent over the network, resulting in improved throughput.

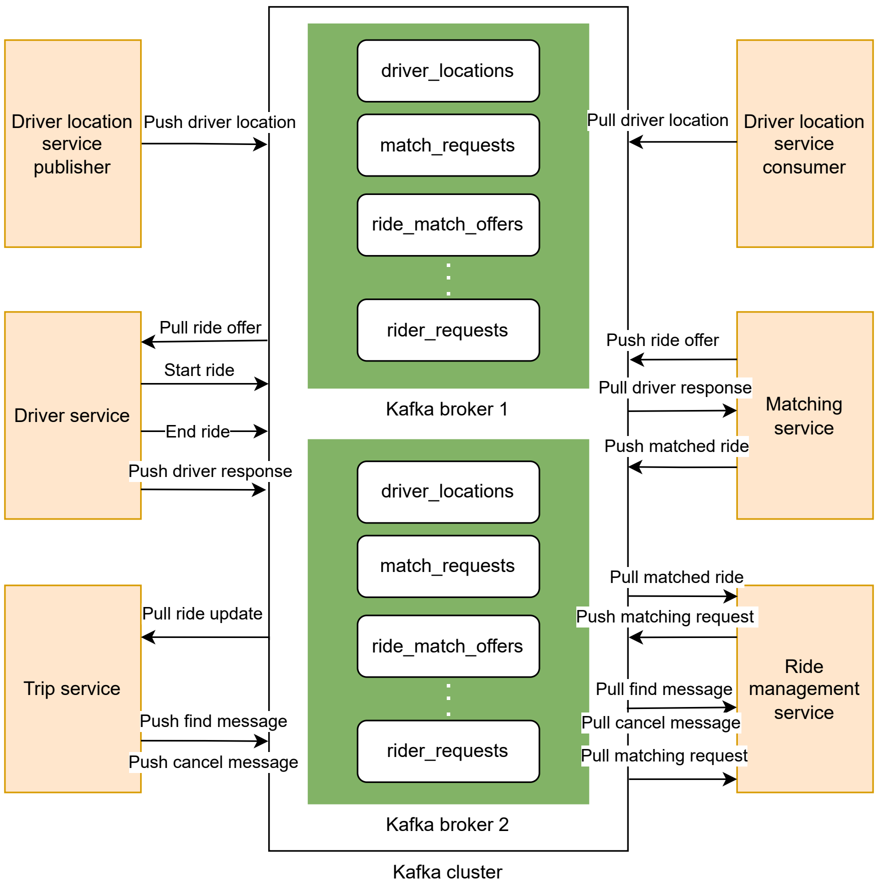


Figure 6.15: Multiple Kafka brokers inside one cluster.

6.3.2 Fault tolerance and data durability

To ensure the system continues to operate in case of the failure of one or more Kafka brokers, we could set up broker replications where messages are copied across multiple brokers. So if we set the replication factor as 3, it means that one message in the topic would be replicated across 3 Kafka brokers, so in case of the failure of one broker due to network or other issues, we still have this message in other brokers, and this message can be gotten.

Thanks to Apache Kafka, microservices in the prototype are communicating asynchronously, which means that the failure of one service does not directly affect others. For instance, if the Matching service goes down, other services, such as the Ride management service and Driver service, still function and send messages to the `match_requests` and `ride_match_updates`. These messages are stored in the topics until the Matching service is restored. This situation significantly increases the fault tolerance of the system.

Consumers keep track of processed messages using offsets. In case of consumer failure, it can return to the place where it stopped. It ensures that no message is lost or reprocessed.

Kafka brokers can be and should be hosted on different servers to eliminate dependency on hardware issues.

It is possible to set up the commitment of the messages across multiple brokers. Once the message is committed, it is considered safely stored. Uncommitted messages are not available for the consumers until they are committed.[28] For example, a ride request may be committed to reduce the risk of losing the request.

It is possible to set up producer retries.[28] There may be a situation when the producer cannot send a message to the Kafka broker because the broker does not acknowledge the delivery of the message. Losing a message from Driver service, Matching service, Trip Service, or Ride management service is intolerable. By setting up retries, we ensure user messages will not be lost. However, this approach may introduce duplicate messages. For example, a driver accepts a ride offer. After the matching service tells this information to the Ride Management Service, it attempts to send a message to the driver and rider about updating the ride status to `MATCHED` on the Kafka topic. Due to a temporary network issue, a Kafka broker is unavailable, and the message did not reach the broker. During this time, both driver and rider are unaware of the ride status updates. The producer will be sending a message until it does not receive an acknowledgment. Retries will ensure critical messages are delivered, but it introduces a new problem: duplicate messages.

We can enable producers to be idempotent by setting `enable.idempotence` to `true` in the Kafka producer settings. After the configuration change, the producer will send the sequence number together with the message. This feature guarantees that even if the ride matched message is sent multiple times due to retries, the Kafka broker will recognize and decline duplicate messages. The producer will receive harmless `DuplicateSequenceException` indicating it sent a repetitive message.[28]

6.3.3 Agility

As a communication tool, Apache Kafka brings the following benefits:

- **Improved agility in the development process:** Because Apache Kafka stores messages in the topics, debugging and seeing the message process flow was much more accessible. If direct microservice communication was used instead, it would require first manual logging of incoming requests and second jumping between running microservices and trying to find these messages between other logs. Apache Kafka allows us to see all messages in the topic. Either from the beginning or the latest (after connecting to the topic as a consumer client).

- **Decouple components in the prototype, providing agility:** Each component can operate independently. In the case of direct communication, components become dependent on each other. Suppose a developer wants to add another service. In that case, he needs to send a message to a specific endpoint, which requires opening the code base of another microservice, which may be in another programming language. This situation requires synchronizing knowledge of other microservices endpoints and their data schema. In the case of Kafka, we are good with topic names and message format.
- **A centralized, highly scalable, and clear view of data flow between microservices:** As we can see in the Figure 6.16, direct invocations bring chaos even with the small number of microservices. This prototype does not include price calculation, which could be placed in a dedicated microservice, monitoring, analytics, customer services, or payment service. Soon, this design can become super complicated and hardly maintainable. Other teams must keep in mind other microservices, which delay development and business opportunities and can have the negative consequence of losing money, investors, and the important company reputation. Much like a method's input parameters, the microservices development team needs to know the message format from the topic and, of course, the overall system design.
- **Data monitoring:** Monitoring the data flow and service communication becomes easy because the communication is centralized. For example, we can implement some fraud detector service which will be plugged in the Kafka to identify possible frauds without affecting clients. Moreover, we can check the health of Kafka and other components.

The design seems acceptable initially, but the system looks complex, even with a few microservices. In the future, there may be other microservices that would increase overall complexity and make the system even more coupled.

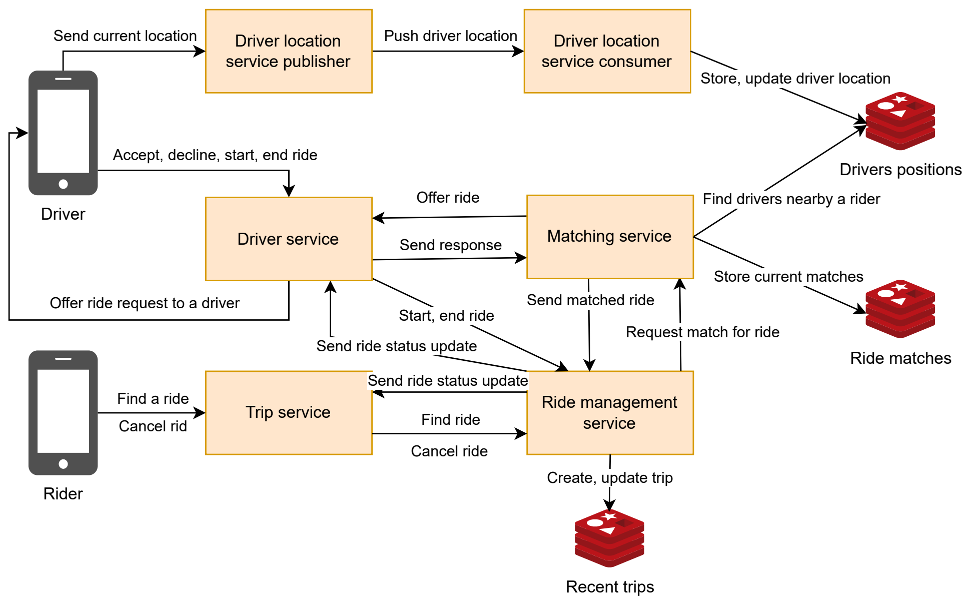


Figure 6.16: Inter service communication without Apache Kafka.

Chapter 7

Future work

There is always room for improvement, and this prototype is not an exception. Possible areas for improvement:

- **Advanced Apache Kafka:** Exploring advanced Apache Kafka topics such as broker and cluster monitoring, multiple clusters, and security, which were not used in the prototype for simplicity.
- **Completeness of the prototype:** Fully functional rider and driver simulation, estimated arrival time for a ride, price calculation, and analysis of demand and supply dynamics.
- **Reactive programming:** Implement reactive programming paradigms to improve the system's responsiveness and scalability. Reactive programming can help in building more resilient and efficiently interactive systems.
- **Benchmarks:** Provide benchmarks for response times to features such as finding a ride. Benchmarking these features would not only help in identifying performance bottlenecks but also aid in comparing the efficiency of different architectural choices and optimizations.
- **Testing:** Create extensive integration and unit testing and develop test scenarios that include simulating broker failures and network partitions. Testing these scenarios will ensure that the system can handle failures gracefully and maintain data integrity and availability under adverse conditions.

Chapter 8
Conclusion

This bachelor thesis has explored asynchronous communication in microservice architecture with Apache Kafka. We have expanded our understanding of overall communication by studying the microservice architecture, its advantages and disadvantages, and real-world examples. We studied different communication patterns, styles, and communication's influence on software systems.

Subsequently, we explored both asynchronous and synchronous communication. We studied cases when and where to use each of them and their influence on the application. Moreover, we analyzed real-world example of asynchronous communication, which gave us insight into when to implement asynchronous communication with Apache Kafka. After understanding both asynchronous and synchronous styles, we deeply analyzed asynchronous communication. Its core architectures such as message queuing, publish-subscribe architecture, event streaming architecture, and event-driven architecture. Ultimately, we looked at message brokers implementing some of the studied architectures. By exploring different brokers, we can better understand Apache Kafka itself.

Examining microservices and different interservice communications gave us enough knowledge to understand Apache Kafka. In the final chapter, the main components of Apache Kafka are explored not only by text but also visually. Additionally, the chapter highlights Apache Kafka compared to other brokers and provides real usage examples.

After understanding that Apache Kafka is a powerful and versatile tool, I described the prototype's architecture, the technologies used, each system component, and how Kafka can help to achieve scalability, throughput, fault tolerance, data durability and agility in the prototype.

Acronyms

API	Application Programming Interface. 4, 5, 11, 15, 28
HTTP	Hypertext Transfer Protocol. 13
HTTPS	Hypertext Transfer Protocol Secure. 13
JSON	JavaScript Object Notation. 23
NoSQL	Not Only SQL. 32
RAM	Random Access Memory. 32
RPC	Remote Procedure Call. 8
STOMP	Simple Text Oriented Messaging Protocol. 35, 36
U.S.	United States. 28
XML	Extensible Markup Language. 23

Bibliography

[1] "Google Trends." Accessed on December 29, 2023, Retrieved from `https://trends.google.com/trends/explore?date=2014-01-01%202023-12-29&q=%2Fm%2F011spz0k&hl=en-US`

[2] "JetBrains." *Microservices - The State of Developer Ecosystem in 2023.* Accessed on January 4, 2024, Retrieved from `https://www.jetbrains.com/lp/devecosystem-2023/development/#mcrsrvc_design_approaches`

[3] "O'Reilly." *A Quick and Simple Definition of Microservices.* Accessed on November 13, 2023, Retrieved from `https://www.oreilly.com/content/a-quick-and-simple-definition-of-microservices/`

[4] "Microsoft Learn." *Microservices: An application revolution powered by the cloud.* Accessed on November 13, 2023, Retrieved from `https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices`

[5] Richards, M., and Ford, N. *Fundamentals of Software Architecture: An Engineering Approach*, 1st Edition, O'Reilly Media Inc., 2020.

[6] Newman, S. *Building Microservices*, 2nd Edition, O'Reilly Media, Inc., 2021.

[7] "Atlassian." *Microservices vs. Monolithic Architecture.* Accessed on November 13, 2023, Retrieved from `https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith`

[8] "Baeldung." *N-Tier Architecture.* Accessed on November 13, 2023, Retrieved from `https://www.baeldung.com/cs/n-tier-architecture`

[9] "Nginx." *Adopting Microservices at Netflix: Lessons for Team and Process Design.* Accessed on November 19, 2023, Retrieved from `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`

[10] "Bloomberg Second Measure LLC" *Rideshare Industry Overview* Accessed on February 18, 2024, Retrieved from `https://secondmeasure.com/datapoints/rideshare-industry-overview/`

[11] "Uber Investor" *Uber Announces Results for First Quarter 2023* Accessed on February 18, 2024, Retrived from `https://investor.uber.com/news-events/news/press-release-details/2023/Uber-Announces-Results-for-First-Quarter-2023/default.aspx`

[12] "Dream Factory" *4 Microservices Examples: Amazon, Netflix, Uber, and Etsy.* Accessed on November 19, 2024 Retrieved from `https://blog.dreamfactory.com/microservices-examples/#examples`

[13] "Microsoft." *Communication in a Microservice Architecture.* Accessed on November 19, 2024, Retrieved from `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture`

[14] "Microsoft." *Design Interservice Communication for Microservices.* Accessed on November 27, 2024, Retrieved from `https://learn.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication#challenges`

[15] Richardson, C. *Microservices Patterns: With examples in Java*, 1st Edition, Manning, 2018.

[16] "Baeldung." *Service Discovery in Microservices.* Accessed on November 27, 2024, Retrieved from `https://www.baeldung.com/cs/service-discovery-microservices`

[17] "Baeldung." *How Does a Load Balancer Work?* Accessed on November 27, 2024, Retrieved from `https://www.baeldung.com/cs/load-balancer`

[18] "Red Hat." *What does an API gateway do?* Accessed on November 27, 2024, Retrieved from `https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do`

[19] "Baeldung." *Eventual Consistency vs. Strong Eventual Consistency vs. Strong Consistency.* Accessed on November 27, 2024, Retrieved from `https://www.baeldung.com/cs/eventual-consistency-vs-strong-eventual-consistency-vs-strong-consistency`

[20] "IBM." *Use Case.* Accessed on November 27, 2024, Retrieved from `https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-use-case`

[21] "Microsoft Learn." *Asynchronous Message-Based Communication.* Accessed on November 27, 2024, Retrieved from `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication`

[22] "InfoQ" *Migrating Netflix's Viewing History from Synchronous Request-Response to Async Events.* Accessed on November 27, 2024, Retrieved from `https://www.infoq.com/articles/microservices-async-migration/`

[23] "Amazon Web Services." *What is a Message Queue?* Accessed on January 13, 2024, Retrieved from `https://aws.amazon.com/message-queue/`

[24] "Amazon Web Services." *What is Pub/Sub?* Accessed on January 13, 2024, Retrieved from `https://aws.amazon.com/what-is/pub-sub-messaging/`

[25] "Apache Kafka." *Introduction.* Accessed on January 13, 2024, Retrieved from `https://kafka.apache.org/intro`

[26] "Amazon Web Services." *Event-driven architecture.* Accessed on January 13, 2024, Retrieved from `https://aws.amazon.com/event-driven-architecture/`

[27] "IBM." *What are Message Brokers?* Accessed on January 13, 2024, Retrieved from `https://www.ibm.com/topics/message-brokers`

[28] Shapira, G., Palino, T., Sivaram, R., and Petty, K. *Kafka: The Definitive Guide.* O'Reilly Media, Inc., 2021.

[29] "Amazon Web Services." *What is Apache Kafka?* Accessed on January 13, 2024, Retrieved from `https://aws.amazon.com/what-is/apache-kafka/`

[30] "Amazon Web Services." *What's the Difference Between Kafka and RabbitMQ?* Accessed on January 13, 2024, Retrieved from `https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka`

[31] "RabbitMQ Blog." *Interoperability in RabbitMQ Streams.* Accessed on January 13, 2024, Retrieved from `https://blog.rabbitmq.com/posts/2021/10/rabbitmq-streams-interoperability`

[32] "LinkedIn" *About LinkedIn.* Accesed on January 13, 2024, Retrieved from `https://about.linkedin.com/`

[33] "LinkedIn Engineering" *How LinkedIn customizes Apache Kafka for 7 trillion messages per day.* Accesed on January 13, 2024, Retrieved from `https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages`

[34] "Apache Kafka" *Powered By.* Accessed on May 4, 2024 Retrieved from `https://kafka.apache.org/powered-by`

[35] "Wall Street Mojo" *Fortune 100 - What Is It, Companies, Examples, Vs Fortune 500.* Accessed on May 4, 2024 Retrieved from `https://www.wallstreetmojo.com/fortune-100/`

[36] "The New York Times" *Publishing with Apache Kafka at The New York Times.* Accessed on May 4, 2024 Retrieved from https://open.nytimes.com/publishing-with-apache-kafka-at-the-new-york-times-7f0e3b7d2077

[37] "AWS". *What is Java?* Accessed on May 5, 2024 Retrieved from https://aws.amazon.com/what-is/java/

[38] "Microsoft Azure". *What is Spring Boot?* Accessed on May 5, 2024 Retrieved from https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-spring-boot

[39] "Docs Spring". *Dependency Injection.* Accessed on May 5, 2024 Retrieved from https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html

[40] "IBM". *What is Redis?* Accessed on May 5, 2024 Retrieved from https://www.ibm.com/topics/redis

[41] "Redis". *What Redis data structures look like* Accessed on May 5, 2024 Retrieved from https://redis.io/glossary/redis-data-structures/

[42] "Uber Engineering Blog". *H3: Uber's Hexagonal Hierarchical Spatial Index* Accessed on May 5, 2024 Retrieved from https://www.uber.com/en-CZ/blog/h3/

[43] "Docker Docs". *Why use Compose?* Accessed on May 6, 2024 Retrieved from https://docs.docker.com/compose/intro/features-uses/

[44] "Docker Docs". *Docker overview.* Accessed on May 6, 2024 Retrieved from https://docs.docker.com/get-started/overview/#the-docker-platform

[45] "Alex Diaconu. Ably" *The WebSocket API and protocol explained* Accesed on May 13, 2024 Retrieved from https://ably.com/topic/websockets

[46] "Baeldung" *Rest vs WebSockets* Accesed on May 6, 2024 Retreived from https://www.baeldung.com/rest-vs-websockets

Appendix A

Used Software

The following software was used during the development of the thesis:

- Draw.io for drawing all diagrams.[1]
- OpenStreetMap for images with maps. [2]
- Overpass turbo for retrieving locations data such as road and supermarkets longitudes and latitudes.[3]

[1]`https://www.draw.io`
[2]`https://www.openstreetmap.org`
[3]`https://overpass-turbo.eu/`