1 What to consider when implementing a microservice architecture approach?

1.1 Communication protocols

Determining how microservices communicate with each other is a foundational decision in microservices architecture, significantly impacting the system's scalability, latency, reliability, and consistency. Selecting an inter-process communication (IPC) mechanism for a service's API requires considering the style of interaction between the service and its clients. This approach emphasizes understanding requirements over the specifics of IPC technologies and influences the application's availability and integration testing strategies. Interaction styles can be categorized along two dimensions: one-to-one versus one-to-many, and synchronous versus asynchronous.

1.1.1 Synchronous Communication

HTTP/REST: It offers several advantages, including simplicity, ease of testing via browsers or command-line tools like curl, support for request/response communication, being firewall friendly, and not requiring an intermediate message broker. However, it also has several limitations, such as: (1) only supporting request/response communication; (2) difficulties in fetching multiple resources in a single request. Though, a REST API can enable fetching related resources in one request using a query parameter to expand the resource (e.g: GET */orders/order-id-1345?expand=consumer*), this solution might be inadequate for complex scenarios and time-consuming to implement (Richardson 2018); (3) difficulties in mapping complex update operations to limited set of HTTP verbs. For instance, non idempotent operations such as cancel or restore order does not have a straightforward HTTP verb. Thus, HTTP/REST is only well-suited for standard web interfaces that perform CRUD (Create, Read, Update, Delete) operations, benefiting from REST's stateless operation.

gRPC: gRPC offers numerous advantages, including the ease of designing APIs with extensive update capabilities (compared to the limited set of verbs offered by HTTP/REST), a highly efficient and compact inter-process communication (IPC) system ideal for large message exchanges, support for bidirectional streaming facilitating both remote procedure call (RPC) and messaging communication styles, and compatibility with clients and services across various programming languages. However, gRPC presents challenges as well, such as increased complexity for JavaScript clients in consuming gRPC APIs compared to REST/JSON APIs, and potential compatibility issues with older firewalls that may not support HTTP/2. Additionally, the binary nature of Protocol Buffers used by gRPC as message format can make debugging more complex than with textual formats like JSON (Aksakalli et al. 2021).

Synchronous communication in microservices, despite its straightforward approach, has significant drawbacks. Firstly, it requires *service discovery* for clients to locate service instances, adding complexity due to the dynamic nature of service locations in distributed systems. Secondly, it demands both the client and service to be fully operational at the time of communication, introducing a risk of partial system failures. To address such failures and maintain resilience, patterns such as *circuit breaker* should be utilized to prevent cascading failures and ensuring system reliability in microservice architectures.

1.1.2 Asynchronous Communication

Messaging Systems (Kafka, RabbitMQ): Posta (2016) draws parallels between microservices and other complex adaptive systems like cities and ant colonies, highlighting shared characteristics such as purpose, autonomy, and environmental reaction. It advocates for a shift from traditional authority-based service calls to an autonomy-focused model where services react to events, leading to more fault-tolerant systems.

Messaging systems like Kafka and RabbitMQ support asynchronous communication, decoupling service producers from consumers. They support communication through channels without knowing service instances, eliminating the need for service discovery. Also, Applications that require handling high volumes of data with minimal latency benefit from the buffering and back-pressure management capabilities of messaging systems. However,ensuring message delivery, handling message ordering, and managing distributed data consistency require additional infrastructure and expertise. Messaging systems can introduce potential drawbacks, including the risk of becoming performance bottlenecks. Modern brokers are designed to be scalable to mitigate this. They may also represent a single point of failure, though high availability designs address this concern.

A summary of differences between the protocols can be seen in Table 1. Synchronous communication, while straightforward, can introduce latency and create tight coupling between services. Asynchronous communication, on the other hand, offers loose coupling and improved scalability but may complicate data consistency and transaction management. In conclusion, the choice of communication strategy in a microservices architecture must balance the benefits of simplicity and directness offered by synchronous communications against the scalability and resilience provided by messaging systems. This decision should be informed by a thorough analysis of the application's specific requirements, including the critical trade-offs between synchronous and asynchronous communication methods.

Criteria	**HTTP/REST**	**Messaging Systems**	**gRPC**
Model	Synchronous	Asynchronous	Synchronous
Format	Text-based	Binary or text	Binary
Performance	Moderate	High	High
Ease of Use	High	Moderate	Moderate
Compatibility	Broad	Specific setup	HTTP/2 required

Table 1: Comparison of Communication Protocols in Microservices

2 How could you use Kafka in an event-driven service-based architecture?

2.1 Event Bus

Kafka's design as a distributed streaming platform makes it exceptionally suited for functioning as a centralized event bus. Kafka's design for high throughput, fault tolerance, and distributed data streaming makes it an ideal backbone for systems that need to process large volumes of events with minimal latency. By leveraging Kafka, microservices can publish and subscribe to events, thereby decoupling event producers from consumers. This separation allows for greater architectural flexibility, making it easier to add or modify services without impacting the overall

system. Since services do not depend on each other for direct communication, the system can scale parts of the architecture independently. For example, if a particular service experiences high demand, it can be scaled without requiring changes to the services that produce or consume its events.

2.2 Advantages of using Kafka as an Event Bus

- **Scalable Event Handling:** Kafka is designed to handle high volumes of data, making it suitable for systems that need to scale dynamically. Its partitioned and distributed architecture allows for the efficient processing of streams of events across multiple consumers.
- **Decoupling of Services:** By acting as an intermediary layer, Kafka allows microservices to communicate asynchronously, reducing dependencies between them. This facilitates a more modular architecture where services can be developed, deployed, and scaled independently.
- **Fault Tolerance:** Kafka's replication mechanisms ensure that data is not lost, even in the face of hardware failures. This resilience is critical for maintaining system reliability and consistency.
- **Replayable Events:** Kafka retains messages for a configurable period, allowing services to replay events. This feature is invaluable for scenarios where late processing or reprocessing of events is required, enhancing the system's flexibility and robustness.

2.2.1 Considerations when using Kafka as an Event bus

- **Operational Complexity:** Managing a Kafka cluster introduces operational complexity, including monitoring, managing topics, configuring partitions, and ensuring cluster health. Proper setup and maintenance are critical to leveraging Kafka's full potential without introducing bottlenecks or single points of failure. However making use of fully managed Kafka service providers such as Confluent, greatly reduces the operational complexity involved.
- **Data Serialization:** Efficient use of Kafka requires careful consideration of data formats and serialization. Schema management tools like Confluent Schema Registry can help manage schemas and ensure compatibility as data evolves.
- **Consumer Offset Management:** Consumers must track their progress through event streams, requiring effective offset management strategies to ensure that events are processed in order and no events are missed or processed multiple times.
- **Eventual Consistency:** Event-driven architectures decouples service dependencies but it introduces eventual consistency. For example, when a microservice updates a data entity and publishes an event, the consuming services may not immediately process that event, leading to a temporary inconsistency across the system. Designing systems around eventual consistency requires embracing it as a feature rather than a limitation. This can involve implementing strategies such as Read-Your-Writes Consistency, where the system ensures that a user's writes are immediately visible to their subsequent reads, potentially through user-specific caching mechanisms (Kleppmann 2017).

In summary, we can see that Kafka's capabilities as a centralized event bus offer significant advantages for building scalable, resilient, and decoupled microservices architectures. Based on the discussion, we can observe that utilizing a fully managed Kafka service allows efficient inter-service communication without the operational complexity and is highly suitable for developing scalable and flexible microservices.

References

Aksakalli, I. K., Çelik, T., Can, A. B. & Tekinerdoğan, B. (2021), 'Deployment and communication patterns in microservice architectures: A systematic literature review', *Journal of Systems and Software* **180**, 111014.

Kleppmann, M. (2017), *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*, " O'Reilly Media, Inc.".

Posta, C. (2016), 'Why microservices should be event driven: Autonomy vs authority – software blog'. **URL:** *https://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/*

Richardson, C. (2018), *Microservices Patterns: With examples in Java*, Manning. **URL:** *https://books.google.co.uk/books?id=UeK1swEACAAJ*