Microservices in the Apache Kafka™ Ecosystem

ABSTRACT

This paper provides a brief overview of how microservices can be built in the Apache Kafka ecosystem. It begins with a look at event-driven services, exploring the tradeoffs involved in implementing the various protocols available for service-to-service communication. This is followed by a brief introduction of Confluent Enterprise, detailing a unique approach to meeting the challenges inherent in implementing and managing a services-based environment.

1. Microservices Background

When you build microservices in the Apache Kafka™ ecosystem you're blending two quite disparate worlds: the world of business systems (that is, of service-based systems) and the world of stream processing. Historically, these two domains have been separate.

Microservice architectures were originally designed to provide flexibility and adaptability to business systems, by splitting roles into lightweight, independently deployed services. Stream processing, on the other hand, addresses the problem of continually reacting to and processing data as it flows through a business.

Despite originating in separate worlds, there is much crossover between these two domains. We can expect this to continue. As business systems have become increasingly data-centric, the links between different services have become denser and more unwieldy. This makes it hard to change and adapt such architectures as they grow. Simple request response protocols are quite limiting in this regard, but there are several ways to piece together service-based systems. Brokered technology changes the dynamics of interaction by decoupling sender and receiver. The use of retentive broker technology and stream processing takes this even further, providing an alternate solution that better weathers the inevitable increase in service complexity.

Figure 1, below, shows how microservices can split the roles within a system into discrete units. These are often termed bounded contexts. Each of the services shown — UI, Order, Fulfillment etc. — match to an underlying business function.

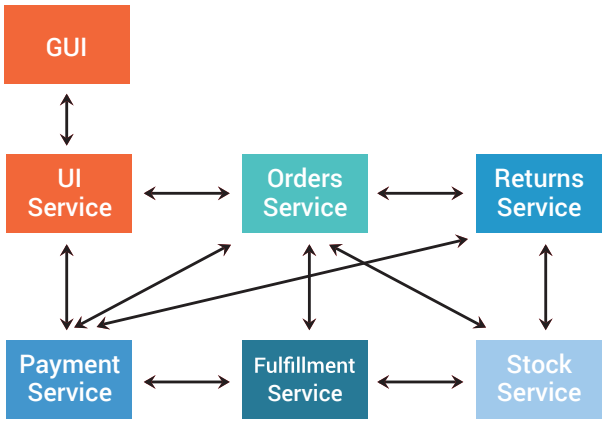


Figure 1: A simple business system built on a microservices architecture

Microservice architectures come with a number of benefits. They provide a convenient abstraction for reuse. Breaking services up makes them more fungible. It provides the potential for increased scalability and fault tolerance. But the main benefit is that each service is deployed independently, and can evolve independently, of the others. This independence is key.

The monolithic approach is quite different. A well-managed monolithic system can provide several of the benefits named above: reuse, encapsulation, and some degree of scalability. But as the system grows, it becomes increasingly difficult to integrate the contributions of a rapidly growing staff of engineers. Over time, a monolithic system struggles to provide the agility that business users typically need.

A second problem with monolithic architectures is that they typically assume some form of central, shared-state management, such as a central database. Having different teams collaborating over a shared database is known to come with a variety of risks and organizational headaches. Monolithic systems tend to lead to tight temporal and data couplings. Using a service-based approach, on the other hand, tends to reduce these problems by having each of the services effectively own its own data.

Request Response and Event-Driven Approaches

Service Oriented Architectures (SOA), which preceded today's microservice-based approaches by more than a decade, used a range of synchronous and asynchronous protocols from SOAP (Simple Object Access Protocol) to various broker technologies of the time. These have been largely eclipsed in recent years by REST (Representational State Transfer) as the most popular protocol for assembling services. REST is both lighter weight and faster than SOAP, and can be implemented using a wide variety of tools. But while it is well established, the request response approach assumes a very different set of tradeoffs to brokered or event-driven protocols.

In an event-driven architecture, services raise events. These events typically map to the real-world flow of the business they represent. For example, a user makes a purchase, which defines an event. This in turn triggers a series of downstream services (payment, fulfillment, fraud detection, etc.)

These events flow through a broker (e.g. Apache Kafka). So, downstream services *react* to business events, rather than being "called" directly by other services (as in the request response approach). The broker provides the level of decoupling needed to do this. Producing services are decoupled from consuming services that sit downstream, and vice versa. The result is an architecture that is more fungible. As long as the services leverage a simple mechanism to ensure communication remains compatible, each one can be coupled only to business events, rather than specific services. This makes it easier to either add new services or change existing ones. As we'll discuss later, this decoupling is made further possible when the solution can ensure compatibility of data across the pipeline at all times through a mechanism for governing schemas across the system.

Figure 2, below, shows two variations on the same business flow as discussed above. The first iteration shows the interactions built using a request response model, while the second shows the same flow based on the event-driven approach.

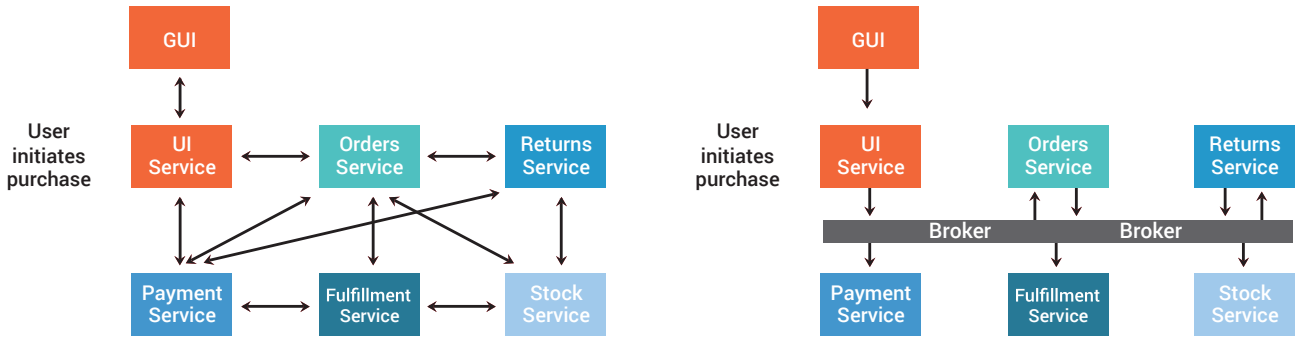
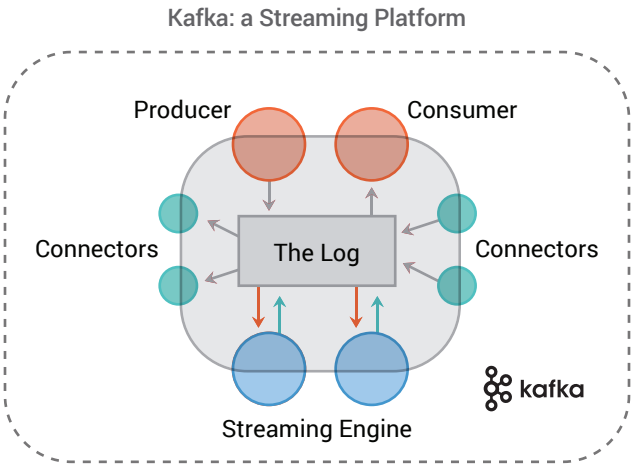


Figure 2: The different interactions that arise when comparing request response and event-driven approaches.

So a microservice-based environment can be built either with event-based or request response approaches. Another option, quite commonly used, is to apply a hybrid of the two. The hybrid approach provides a simple protocol for looking things up, or invoking synchronous actions, while also providing the fungibility, scalability and future-proofing of asynchronous, event-driven flows.

2. Confluent for Microservices

Apache Kafka is an ideal solution for these microservice-based environments. The core toolset, in combination with its built-in Streams and Connect APIs make it well suited for highly scalable microservice architectures.



3. Kafka's Distributed, Retentive Broker

At the heart of Confluent Enterprise is Apache Kafka, a distributed stream processing engine for building real-time data pipelines and streaming applications. Central to Kafka is a log that behaves, in many ways, like a traditional messaging system. It is a broker-based technology, accepting messages and placing them into topics. Any service can subscribe to a topic and listen for the messages sent to it. But as a distributed log itself, Kafka differs from a traditional messaging system by providing improved properties for scalability, availability and data retention.

As shown in Figure 3, below, the overall architecture for an Apache Kafka environment includes producing services, Kafka itself, and consuming services.

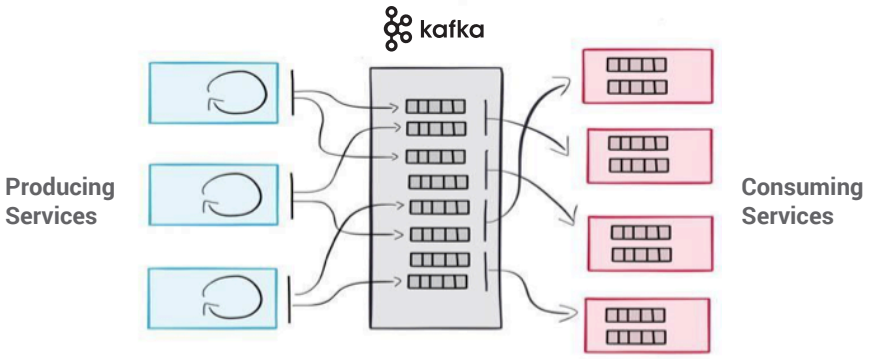


Figure 3: The three levels of scale out: producers, brokers, consumers

What differentiates this architecture is that it's completely free of bottlenecks in all three layers. Kafka receives messages and shards them across a set of servers inside the Kafka cluster. Each shard is modelled as an individual queue. The user can specify a key. This controls which shard data is routed to, thus ensuring strong ordering for messages that have the same key. On the consumption side, Kafka can balance data from a single topic across a set of consuming services. For example, in the configuration shown in Figure 3, Kafka balances the load of consuming a single topic across all four services, greatly increasing the processing throughput for that topic.

The result of these two architectural elements is a linearly scalable cluster, both from the perspective of incoming and outgoing datasets. This is often difficult to achieve with conventional message-based approaches.

Kafka also provides high availability. If one of the services fails, the environment will detect the fault and re-route shards to another service, ensuring that processing continues uninterrupted by the fault. Moreover, because Kafka provides data retention, much like a database, users have the option to rewind and replay as required. This means a consuming service can seek back through the log to read historical messages. This capability provides a powerful tool for both handling failure as well as implementing state regeneration patterns such as Event Sourcing.

Unlike conventional messaging services, Kafka has the the ability to "compact" a log. The compaction process turns a stream of all events into a stream that contains only the latest events for each key. This is akin to the difference between an audit table and a regular table in a database. If you provide a key with your messages, compaction will remove old messages, for that key, which have been superseded.

When such compacted streams are brought into the Kafka Streams APIs, they become what are termed KTables. These behave very much like tables in a database, except the view in the Kafka Streams client is constantly kept up to date. We'll dip into these next.

4. Kafka Streams API: Kafka's Stream Processing Engine

The Kafka Streams engine itself is just an API for Kafka, so it's easy to embed in any JVM-based service. But behind this deceptively simple abstraction sits a wealth of rich, declarative features for joining, filtering, transforming, and processing event streams. As such it is the toolset services need to handle the complex graph of interactions that flow from service to service.

The Kafka Streams API is what is known as a Stateful Stream Processing Engine. This is essentially a database engine that allows users to query a group of streams continuously. The queries take a form similar to SQL but are applied to an infinite stream rather than a bounded table. The query language also includes operators not found in traditional SQL. For example, the time window the query will cover or the frequency with which it should emit results. These queries provide a continuously updated view.

Data streams themselves are apportioned into windows. Windows allow the engine to reason about an indefinite and ongoing data stream. As with tables in a database, the Kafka Streams API enables users to perform joins on various data streams. Defining windows bounds the amount of data considered when doing such joins (strictly speaking this is termed a retention period, rather than a window, but the concept is equivalent). Users can also perform transformations, aggregations, apply filters, and perform ad hoc functions.

The Kafka Streams API is termed a stateful stream processing engine as it includes the concepts of tables as well as streams. A table differs from stream in representing an entire historic dataset, directly accessible from the Kafka streaming engine. This is particularly useful for service-based applications which want to do joins. A traditional Stream-Stream join has to consider the window as part of the join. This means that late data on one side of a join may not be matched if the window (retention period) is not configured for long enough. This is not the case with Stream-Table joins (or Table-Table joins), which can retain full history and hence have predictable accuracy.

When joining tables Kafka Streams caches data locally, allowing for fast, in-process or disk-backed look-ups on the joined data. As an example, consider a simple enrichment task that must operate on a large-scale stream of input data: a user has a stream of orders coming into a service and needs to join it with a set of customers. By representing the customers as a KTable, the join will be fast, accurate, and local to the machine, rather than overwhelming a remote database with massive query loads.

The Kafka Streams API also inherits all its distributed processing capability and fault tolerance from Kafka, making it ideal for managing a large service architecture or streaming data pipeline without the need for separate orchestration infrastructure. It is also just an embeddable JAR (Java Archive), a library which users can install and operate in the JVM (Java Virtual Machine). This means it's very easy to embed in any JVM-based microservice and does not require a separate cluster.

5. Kafka Connect API: Capturing and Releasing Streams in Apache Kafka

Even with the retentive properties of Kafka, it is sometimes necessary to physically move and materialize data. This may be because it is the only way to get data out of a legacy system. Or it might be because the user needs to get some data into a relational database to interface with a querying tool. Or a user may need to make data available to Elasticsearch for free-format queries.

The Kafka Connect API supports replication of data from source to any number of destinations. This makes it useful for approaches such as CQRS (Command Query Responsibility Segregation).

Further, because Kafka is retentive, meaning the streams can retain history, it is possible to regenerate target databases should anything go wrong. This empowers consuming services by allowing them to regenerate at will. Such capabilities are also useful for organizations looking to manage data in an agile way. For example, rather than having to import an entire, large schema into a service to query it, the Kafka Connect API can be used to take just a subset initially, with the option to incrementally evolve coverage over time as requirements for the dataset broaden. This can be particularly helpful when building microservices architectures in that you can use samples of data with limited effort.

Confluent: Enterprise Streaming for Microservices

When it comes to building large scale microservices architectures, Apache Kafka alone may not suffice. As the number of services and data pipelines grow, maintaining connections and data flows across them all can become challenging. Confluent Enterprise builds on Apache Kafka and makes it easy to run a streaming platform at scale. Confluent offers connectivity and data compatibility capabilities to simplify operating and maintaining a Kafka cluster in support of a microservices architecture:

- **REST API:** The Confluent REST API provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of your cluster, and perform administrative actions with the user of clients or native Kafka code.
- **Pre-built connectors and clients:** Utilizing the Kafka Connect API, certified connectors have been developed making it easy to connect to leading systems and applications. Available as either source (ie. import data from another system) or sink (export data to another system) connectors, Confluent simplifies connecting to third party tools including Elasticsearch, Hadoop, Splunk, and many more.
- **Schema registry:** Confluent makes it easy to guarantee that data that flows through your data pipeline is well-formed. The Confluent Schema Registry provides a central registry for the format of the data in each Kafka topic and provides a central service that helps to make changes to data formats easy and backwards compatible.

Confluent Enterprise is the complete solution for Apache Kafka in the enterprise, containing all of Kafka's capabilities and enhancing it with integrated, tested, and packaged features that make architecting and managing large scale streaming pipelines easier and more reliable.

Visit **www.confluent.io/download** to download the latest version of Confluent Enterprise.