Communication in Microservices, The Concepts

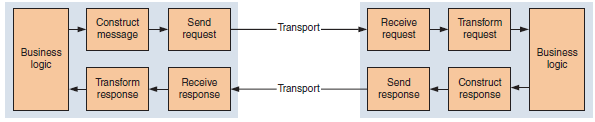
In this document we talk about methods for microservices to instruct and request action from other microservices. Most mature microservice applications will mix both synchronous and asynchronous interaction styles.

What we want is smart endpoints and dumb pipes. Take the following with a grain of salt but, I think what the mean by this is that the actual services must know who they should communicate with etc. but communication brokers as a part of the platform layer for example must be agnostic to who is using them.

# When to Use Synchronous Messaging

Synchronous messages are often the first design approach that comes to mind. **They’re well-suited to scenarios where an action’s results — or acknowledgement of success or failure — are required before proceeding with another action.**

The figure bellow illustrates a **request–response pattern** for synchronous messages. The first service constructs an appropriate message to a collaborator, which the application sends using a **transport mechanism**, such as HTTP. The destination service receives this message and responds accordingly.



## Choosing a Transport

The choice of transport — **RESTful HTTP**, an **RPC library**, or something else — will impact the design of your services. Each transport has different properties of **latency**, **language support**, and **strictness**. For example, gRPC provides generated client/server API contracts using **Protobufs**, whereas HTTP is agnostic to the context of messages.

Across your application, **using a single method of synchronous transport** has **economies of scale; it’s easier to reason through**, **monitor**, and **support with tooling**. Separation of concerns within microservices is also important. You should separate your choice of transport mechanism from the business logic of your service, which shouldn’t need to know about HTTP status codes or gRPC response streams. Doing so makes it easier to swap out different mechanisms in the future if your application’s needs evolve.

## Draw-Backs of Synchronous Communication

Synchronous messages have limitations:

* They create tighter coupling between services, as services must be aware of their collaborators.
* They don’t have a strong model for **broadcast or publish-subscribe** models, **limiting**

**your capability to perform parallel work.**

* They block code execution while waiting on responses. In a thread- or process-based server model, this can exhaust capacity and **trigger cascading failures**.
* Overuse of synchronous messages can build **deep dependency chains**, which increases the overall **fragility** **of a call path**.

# When to Use Asynchronous Messaging

An asynchronous style of messaging is **more flexible**. **By announcing events**, you make it easy to extend the system to handle new requirements, because **services no longer need to have knowledge of their downstream consumers**. New services can consume existing events without changing existing services.

* **Events** represent **post-hoc state changes**. OrderCreated, OrderPlaced, and OrderCanceled are examples of events that an orders service might emit.

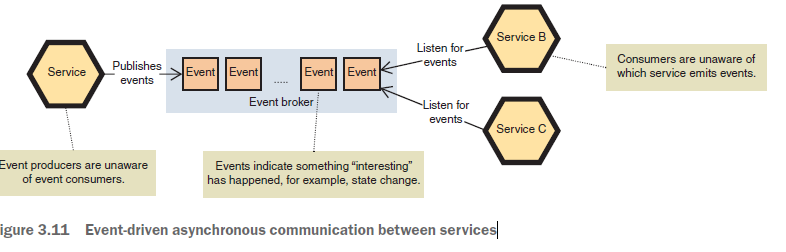
This style enables more fluid evolution and creates looser coupling between services. This does come at a cost: **asynchronous interactions are more difficult to reason through**, **because overall system behavior is no longer explicitly encoded into linear sequences**. System behavior will become increasingly *emergent (*arising unexpectedly*)*— developing unpredictably from interactions between services — requiring investment in monitoring to adequately trace what’s happening.

Asynchronous messaging typically requires a ***communication broker***, **an independent system**

**component that receives events and distributes them to event consumers.** sometimes called an *event backbone*, which indicates how central to your application this component becomes (figure below). Tools commonly used as brokers include **Kafka**, **RabbitMQ**, and **Redis**. The semantics of these tools differ: **Kafka specializes in high-volume, replayable event storage**, whereas RabbitMQ provides higher level messaging middleware (based on the AMQP protocol (<https://www.amqp.org/)>).

**a communication broker becomes a single point of failure that will require careful attention** for you to scale, monitor, and operate effectively.

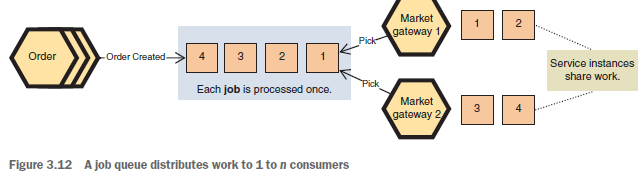
**NOTE** Events enable different styles of persistence and querying, such as **event sourcing** and **command query responsibility segregation (CQRS)**. These aren’t a prerequisite for microservices but have some synergies with a microservice approach. We’ll explore them in chapter 5.



## *Asynchronous Communication Patterns*

two most common event-based patterns are: job queue and publish-subscribe. **most higher-level interaction patterns are built on one of these two primitives.**

### Job Queue



In this pattern, **workers take jobs from a queue and execute them** (figure above). A job

should only be processed once, regardless of how many worker instances you operate. This pattern is also known as **winner takes all**.

This pattern is useful when:

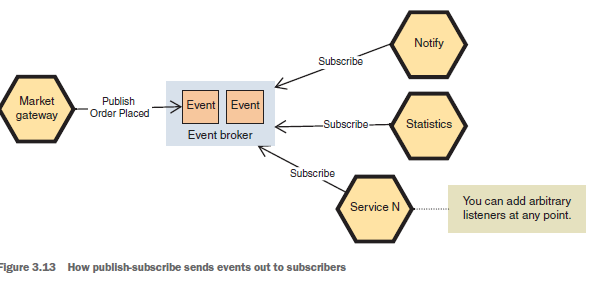
* **A 1:1 relationship exists** between an event and work to be done in response to that event.
* The work that needs to be done is **complex or time-consuming**, so it should be done out-of-band from the triggering event.

By default, this approach **doesn’t require sophisticated event delivery**. Many task queue libraries are available that use commodity data stores, such as **Redis** (Resque, Celery, Sidekiq) **or SQL** databases.

### Publish-Subscribe

In publish-subscribe, services trigger events for arbitrary listeners. All listeners that receive the event act on it appropriately. In some ways, this is the ideal microservice pattern: a service can send arbitrary events out into the world without caring who acts on them.

For example, imagine you need to trigger other downstream actions once an order has been placed. You might send a **push notification to the customer** or **use it to feed your order statistics** and **recommendation feature**. These features can all listen for the same event.



# Locating Other Services

We have discussed about this in detail in a separate document