Communication in Microservices, The Concepts

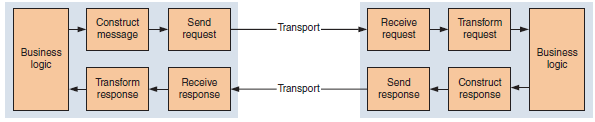
In this document we talk about methods for microservices to instruct and request action from other microservices. Most mature microservice applications will mix both synchronous and asynchronous interaction styles.

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.

# What you should read Later-on

Chapter 5 of the book Microservices in action.

Section 6.3.5 of the book on asynchronous communication.