### **Overview of interprocess communication in a microservice architecture**

There are lots of different IPC technologies to choose from. Services can use synchronous request/response-based communication mechanisms, such as HTTP-based REST or gRPC. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP. There are also a variety of different messages formats. Services can use human-readable, text-based formats such as JSON or XML. Alternatively, they could use a more efficient binary format such as Avro or Protocol Buffers.

**Interaction styles**

There are a variety of client-service interaction styles. The first dimension is whether the interaction is one-to-one or one-to-many:

Text

Description automatically generated

The second dimension is whether the interaction is synchronous or asynchronous:

Graphical user interface, text, application

Description automatically generated

##### The various interaction styles can be characterized in two dimensions: one-to-one vs one-to-many and synchronous vs asynchronous.

Graphical user interface, application, Teams

Description automatically generated

**Request/response**— A service client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. It might event block while waiting. This is an interaction style that generally results in services being tightly coupled.

**Asynchronous request/response**— A service client sends a request to a service, which replies asynchronously. The client doesn’t block while waiting, because the service might not send the response for a long time.

**One-way notifications**— A service client sends a request to a service, but no reply is expected or sent.

It’s important to remember that the synchronous request/response interaction style is mostly orthogonal to IPC technologies. A service can, for example, interact with another service using request/response style interaction with either REST or messaging. Even if two services are communicating using a message broker, the client service might be blocked waiting for a response. It doesn’t necessarily mean they’re loosely coupled.

Graphical user interface, text, application

Description automatically generated

#### **Defining APIs in a microservice architecture**

APIs or interfaces are central to software development. An application is comprised of modules. Each module has an interface that defines the set of operations that module’s clients can invoke. A well-designed interface exposes useful functionality while hiding the implementation. It enables the implementation to change without impacting clients.

A service’s API is a contract between the service and its clients. The challenge is that a service API isn’t defined using a simple programming language construct. By definition, a service and its clients aren’t compiled together. If a new version of a service is deployed with an incompatible API, there’s no compilation error. Instead, there will be runtime failures.

Regardless of which IPC mechanism you choose, it’s important to precisely define a service’s API using some kind of interface definition language (IDL). Moreover, there are good arguments for using an API-first approach to defining services (see [www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10](http://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10) for more). First you write the interface definition. Then you review the interface definition with the client developers. Only after iterating on the API definition do you then implement the service. Doing this up-front design increases your chances of building a service that meets the needs of its clients.

**API-first design is essential**

Even in small projects, I’ve seen problems occur because components don’t agree on an API. For example, on one project the backend Java developer and the AngularJS frontend developer both said they had completed development. The application, however, didn’t work. The REST and WebSocket API used by the frontend application to communicate with the backend was poorly defined. As a result, the two applications couldn’t communicate!

The nature of the API definition depends on which IPC mechanism you’re using. For example, if you’re using messaging, the API consists of the message channels, the message types, and the message formats. If you’re using HTTP, the API consists of the URLs, the HTTP verbs, and the request and response formats. Later in this chapter, I explain how to define APIs.

**Evolving APIs**

APIs invariably change over time as new features are added, existing features are changed, and (perhaps) old features are removed. In a monolithic application, it’s relatively straightforward to change an API and update all the callers. If you’re using a statically typed language, the compiler helps by giving a list of compilation errors. The only challenge may be the scope of the change. It might take a long time to change a widely used API.

In a microservices-based application, changing a service’s API is a lot more difficult. A service’s clients are other services, which are often developed by other teams. The clients may even be other applications outside of the organization. You usually can’t force all clients to upgrade in lockstep with the service. Also, because modern applications are usually never down for maintenance, you’ll typically perform a rolling upgrade of your service, so both old and new versions of a service will be running simultaneously.

**Use semantic versioning**

The Semantic Versioning specification (http://semver.org) is a useful guide to versioning APIs. It’s a set of rules that specify how version numbers are used and incremented. Semantic versioning was originally intended to be used for versioning of software packages, but you can use it for versioning APIs in a distributed system.

The Semantic Versioning specification (Semvers) requires a version number to consist of three parts: MAJOR.MINOR.PATCH. You must increment each part of a version number as follows:

Graphical user interface, text

Description automatically generated

There are a couple of places you can use the version number in an API. If you’re implementing a REST API, you can, as mentioned below, use the major version as the first element of the URL path. Alternatively, if you’re implementing a service that uses messaging, you can include the version number in the messages that it publishes.

##### Making minor, backward-compatible changes

Ideally, you should strive to only make backward-compatible changes. Backward-compatible changes are additive changes to an API:

Graphical user interface, text, application

Description automatically generated with medium confidence

If you only ever make these kinds of changes, older clients will work with newer services, provided that they observe the Robustness principle (<https://en.wikipedia.org/wiki/Robustness_principle>), which states: “Be conservative in what you do, be liberal in what you accept from others.” Services should provide default values for missing request attributes. Similarly, clients should ignore any extra response attributes. In order for this to be painless, clients and services must use a request and response format that supports the Robustness principle.

##### Making major, breaking changes

Sometimes you must make major, incompatible changes to an API. Because you can’t force clients to upgrade immediately, a service must simultaneously support old and new versions of an API for some period of time. If you’re using an HTTP-based IPC mechanism, such as REST, one approach is to embed the major version number in the URL. For example, version 1 paths are prefixed with '/v1/...', and version 2 paths with '/v2/...'.

Another option is to use HTTP’s content negotiation mechanism and include the version number in the MIME type. For example, a client would request version 1.x of an Order using a request like this:

Text

Description automatically generated

In order to support multiple versions of an API, the service’s adapters that implement the APIs will contain logic that translates between the old and new versions.

**Message formats**

The essence of IPC is the exchange of messages. Messages usually contain data, and so an important design decision is the format of that data. The choice of message format can impact the efficiency of IPC, the usability of the API, and its evolvability. If you’re using a messaging system or protocols such as HTTP, you get to pick your message format. Some IPC mechanisms—such as gRPC, which you’ll learn about shortly—might dictate the message format. In either case, it’s essential to use a cross-language message format. Even if you’re writing your microservices in a single language today, it’s likely that you’ll use other languages in the future. You shouldn’t, for example, use Java serialization.

There are two main categories of message formats: text and binary. Let’s look at each one.

##### Text-based message formats

The first category is text-based formats such as JSON and XML. An advantage of these formats is that not only are they human readable, they’re self describing. A JSON message is a collection of named properties. Similarly, an XML message is effectively a collection of named elements and values. This format enables a consumer of a message to pick out the values of interest and ignore the rest. Consequently, many changes to the message schema can easily be backward-compatible.

A downside of using a text-based messages format is that the messages tend to be verbose, especially XML. Every message has the overhead of containing the names of the attributes in addition to their values. Another drawback is the overhead of parsing text, especially when messages are large. Consequently, if efficiency and performance are important, you may want to consider using a binary format.

**Binary message formats**

There are several different binary formats to choose from. Popular formats include Protocol Buffers and Avro. Both formats provide a typed IDL for defining the structure of your messages. A compiler then generates the code that serializes and deserializes the messages. You’re forced to take an API-first approach to service design! Moreover, if you write your client in a statically typed language, the compiler checks that it uses the API correctly.

One difference between these two binary formats is that Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages. As a result, handling API evolution is easier with Protocol Buffers than with Avro.