Overview of InterProcess Communication in Microservices

# Introduction

within a monolithic application like FTGO, modules invoke one another via language-level method or function calls. FTGO developers generally don’t need to think about IPC unless they’re working on the REST API **or the modules that integrate with cloud services** such as the Twilio messaging service and the Stripe payment service.

In contrast, the microservice architecture structures an application as a set of services. Those services must often collaborate in order to handle a request. Because service instances are typically processes running on multiple machines, they must interact using IPC.

The choice of IPC mechanism is an important architectural decision. **It can impact application availability.** What’s more, as I explain in this chapter and the next, IPC even intersects with transaction management. **I favor an architecture consisting of loosely coupled services that communicate with one another using asynchronous messaging**. **Synchronous protocols such as REST are used mostly to communicate with other applications.**

There are lots of different **IPC (Inter-Process Communication)** 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

Interaction Styles are a **technology-independent way of describing how clients and services interact.**

* Thinking first about the interaction style will help you focus on the requirements and avoid getting mired in the details of a particular IPC technology.
* The Interaction style also impacts availability of the application.
* It helps you determine the appropriate integration testing strategy.

There are a variety of client-service interaction styles. they can be categorized in two dimensions:

The first dimension is whether the interaction is one-to-one or one-to-many:

 *One-to-one*—Each client request is processed by exactly one service.

 *One-to-many*—Each request is processed by multiple services.

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

 *Synchronous*—The client **expects a timely response from the service** and **might even block while it waits.**

 *Asynchronous*—**The client doesn’t block**, and the response, if any, isn’t necessarily sent immediately.

The following are the different types of **one-to-one interactions:**

 ***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.

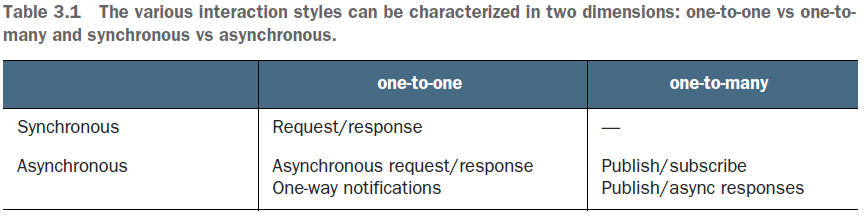
It’s important to remember that the **synchronous request/response interaction style is mostly orthogonal to IPC technologies.** A service can, for example, interact with another service using request/response style interaction with either REST or messaging**. Even if two services are communicating using a message broker, the client service might be blocked waiting for a response. It doesn’t necessarily mean they’re loosely coupled.** That’s something I revisit later in this chapter when discussing the impact of inter-service communication on availability.

The following are the different types of **one-to-many interactions:**

 ***Publish/subscribe***—A client publishes a notification message, which is consumed by zero or more interested services.

 ***Publish/async responses***—A client publishes a request message and then waits for a certain amount of time for responses from interested services.

**Each service will typically use a combination of these interaction styles**. Many of the services in the FTGO application have both synchronous and asynchronous APIs for operations, and many also publish events.



# Defining APIs in a Microservices Architecture

A service’s API consists of **operations, which clients can invoke**, **and events, which are published by the service.**

**An operation has a name, parameters, and a return type**. **An event has a type and a set of fields and is, as described later on, published to a message channel.**

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:

**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.

**A service’s API is rarely set in stone.** It will likely evolve over time. Let’s take a look at how to do that and consider the issues you’ll face.

# 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.**

**How you handle a change to an API depends on the nature of the change.**

## 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:

 MAJOR—When you make an incompatible change to the API

 MINOR—When you make backward-compatible enhancements to the API

 PATCH—When you make a backward-compatible bug fix

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.** The goal is to properly version APIs and to evolve them in a controlled fashion. Let’s look at how to handle minor and major changes.

### Making Minor, Backward Compatible Changes

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

 **Adding optional attributes to request**

 **Adding attributes to a response**

 **Adding new operations**

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.”** : In other words, programs that send messages to other machines (or to other programs on the same machine) should conform completely to the specifications, but programs that receive messages should accept non-conformant input as long as the meaning is clear.

* 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. Later in this section, I describe **how text-based formats such as JSON and XML generally make it easier to evolve APIs.**

**But I read about schema evolution in protobuffs and Avro and they can handle them too but maybe there is an overhead to do so?????????**

### 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:

GET /orders/xyz HTTP/1.1

Accept: application/vnd.example.resource+json; version=1

...

This request tells the Order Service that the client expects a version 1.x response.

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.

Also, as described in chapter 8, the API gateway will almost certainly use versioned APIs. It may even have to support numerous older versions of an API.

Now we’ll look at the issue of message formats, the choice of which can impact how

easy evolving an API will be:

# Message Formats in IPC

**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—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. the advantage of these formats is that:

* They are human readable
* they’re self-describing: the names (keys) of properties act as a meta data for the values and **they are directly embedded in the document** so you can use a json/xml document **without a strict need** for an external meta data describing the document unlike something like protobuffs where you need a .proto file to describe the serialized data.
* 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.**

The structure of XML documents is specified by an XML schema (www.w3.org/XML/Schema). Over time, the developer community has come to realize that JSON also needs a similar mechanism. One popular option is to use the JSON Schema standard (http://json-schema.org). A JSON schema defines the names and types of a message’s properties and whether they’re optional or required. As well as being useful documentation, a JSON schema can be used by an application to validate incoming messages.

The down side of using text-based formats:

* **the messages tend to be verbose, especially XML**(you have closing tags, etc.). Every message has the overhead of **containing the names** of the attributes **in addition to their values**.
* Each character requires 8 bits for example ascii characters in UTF-8 and sending an integer in this form will make the size way larger compared to a fixed 4-byte size integer.
* **the overhead of parsing text**, especially when messages are large:

a text-based format involves **reading the text**, **interpreting its structure** (like key-value pairs in JSON or tags in XML), **and converting it into an internal data structure** (like a dictionary, list, or object).

**Parsing** a binary format involves reading the binary data and converting it into a data structure. This step **is more efficient because the data is already stored in a way that is closer to the internal representation used by the machine** (e.g., integers are already stored as integers, and strings are already stored as byte sequences).

**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** (<https://developers.google.com/protocol-buffers/docs/overview>) and **Avro** (<https://avro.apache.org>).

Both formats provide **a typed IDL** (interface definition language) 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.** This blog post (http://martin.kleppmann.com/2012/12/05/schemaevolution-in-avro-protocol-buffers-thrift.html)is an excellent comparison of Thrift, Protocol Buffers, and Avro.

Now that we’ve looked at message formats, we can look at specific IPC mechanisms that transport the messages, starting with the **Remote procedure invocation (RPI) pattern.**