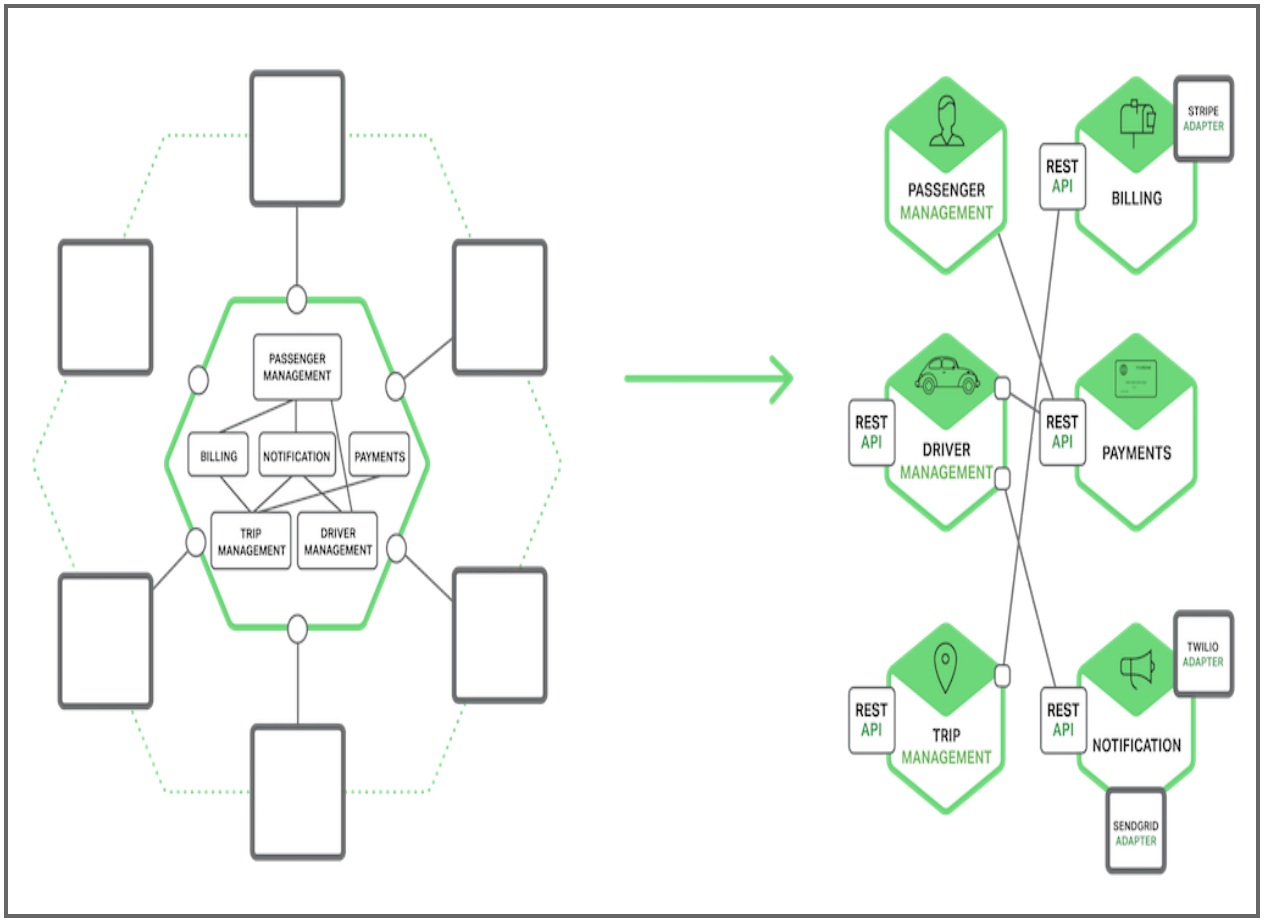
Inter-Process Communication in a Microservices Architecture

Introduction

In a monolithic application, components invoke one another via language-level method or function calls. In contrast, a microservices-based application is a distributed system running on multiple machines. Each service instance is typically a process. Consequently, as the following diagram shows, services must interact using an inter-process communication (IPC) mechanism.



Later on we will look at specific IPC technologies, but first let's explore various design issues.

Interaction Styles

When selecting an IPC mechanism for a service, it is useful to think first about how services interact. There are a variety of client⇔service interaction styles. They can be categorized along 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 instance.
- One-to-many – Each request is processed by multiple service instances.

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 while waiting for a response, and the response, if any, isn't necessarily sent immediately.

The following table shows the various interaction styles.

	One-to-One	One-to-Many
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

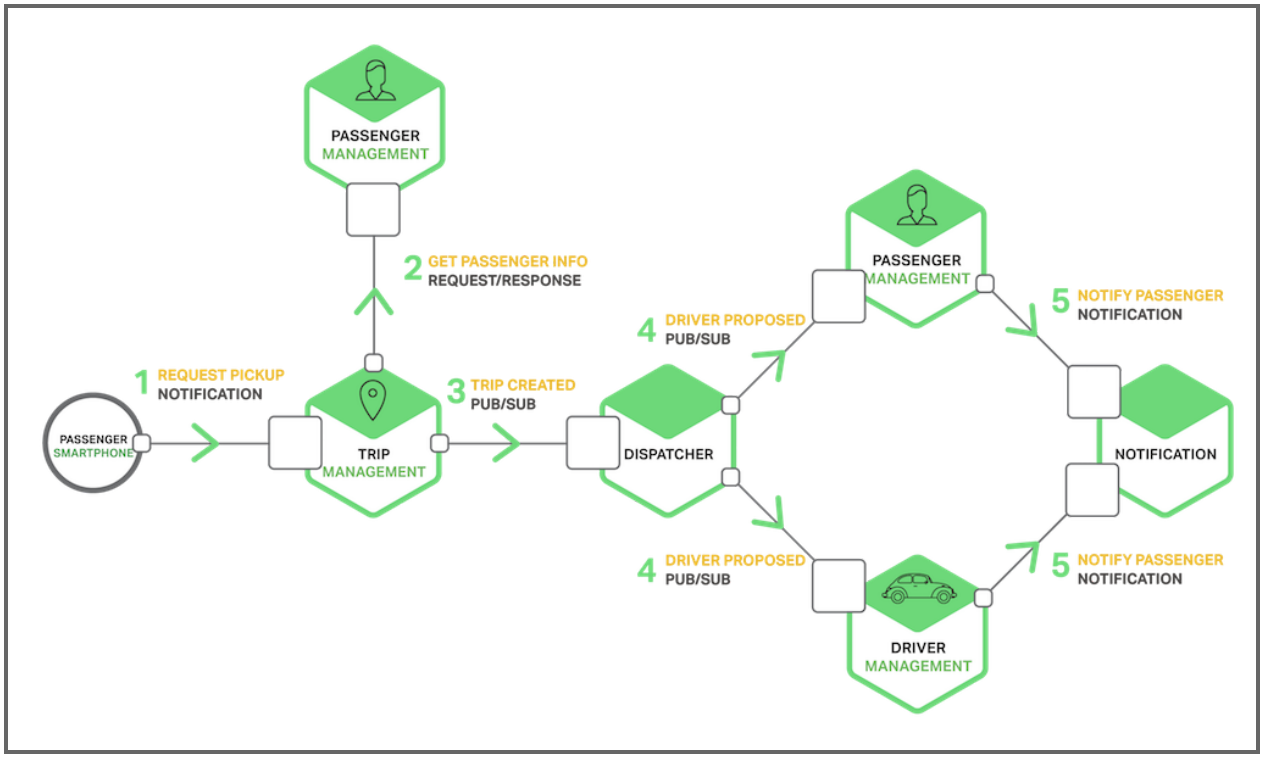
There are the following kinds of one-to-one interactions:

- Request/response – A client makes a request to a service and waits for a response. The client expects the response to arrive in a timely fashion. In a thread-based application, the thread that makes the request might even block while waiting.
- Notification (a.k.a. a one-way request) – A client sends a request to a service but no reply is expected or sent.
- Request/async response – A client sends a request to a service, which replies asynchronously. The client does not block while waiting and is designed with the assumption that the response might not arrive for a while.

There are the following kinds 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 a certain amount of time for responses from interested services.

Each service typically uses a combination of these interaction styles. For some services, a single IPC mechanism is sufficient. Other services might need to use a combination of IPC mechanisms. The following diagram shows how services in a taxi-hailing application might interact when the user requests a trip.



The services use a combination of notifications, request/response, and publish/subscribe. For example, the passenger's smartphone sends a notification to the Trip Management service to request a pickup. The Trip Management service verifies that the passenger's account is active by using request/response to invoke the Passenger Service. The Trip Management service then creates the trip and uses publish/subscribe to notify other services including the Dispatcher, which locates an available driver.

Now that we have looked at interaction styles, let's take a look at how to define APIs.

Defining APIs

A service's API is a contract between the service and its clients. Regardless of your choice of IPC mechanism, it's important to precisely define a service's API using some kind of interface definition language (IDL). There are even good arguments for using an API-first approach to defining services. You begin the development of a service by writing the interface definition and reviewing it with the client developers. It is only after iterating on the API definition that you implement the service. Doing this design up front increases your chances of building a service that meets the needs of its clients.

As you will see later in this article, the nature of the API definition depends on which IPC mechanism you are using. If you are using messaging, the API consists of the message channels and the message types. If you are using HTTP, the API consists of the URLs and the request and response formats. Later on we will describe some IDLs in more detail.

Evolving APIs

A service's API invariably changes over time. In a monolithic application it is usually straightforward to change the API and update all the callers. In a microservices-based application it is a lot more difficult, even if all of the consumers of your API are other services in the same application. You usually cannot force all clients to upgrade in lockstep with the service. Also, you will probably incrementally deploy new versions of a service such that both old and new versions of a service will be running simultaneously. It is important to have a strategy for dealing with these issues.

How you handle an API change depends on the size of the change. Some changes are minor and backward compatible with the previous version. You might, for example, add attributes to requests or responses. It makes sense to design clients and services so that they observe the robustness principle. Clients that use an older API should continue to work with the new version of the service. The service provides default values for the missing request attributes and the clients ignore any extra response attributes. It is important to use an IPC mechanism and a messaging format that enable you to easily evolve your APIs..

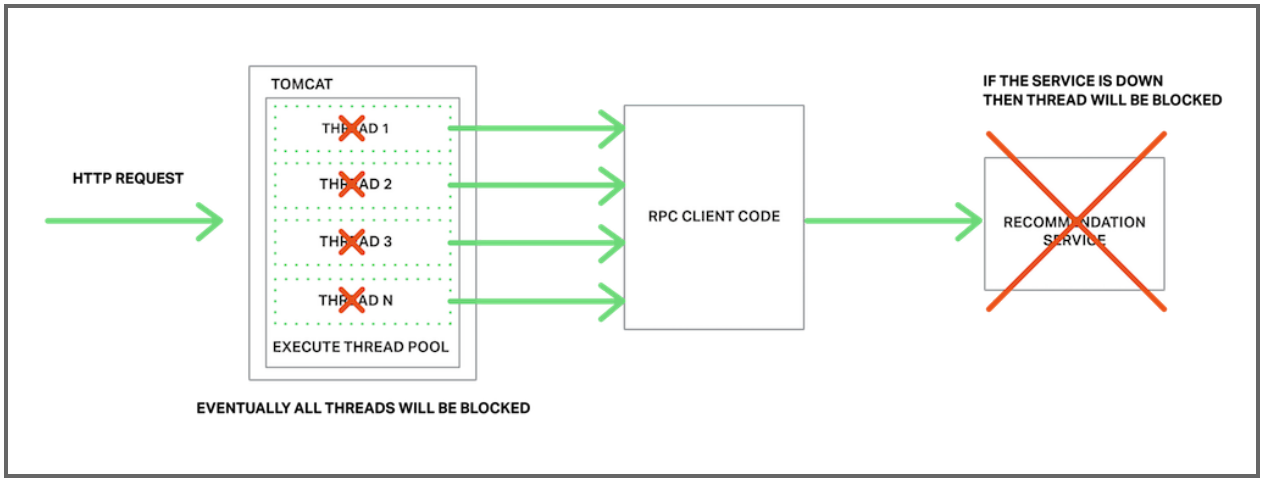
Sometimes, however, you must make major, incompatible changes to an API. Since you can't force clients to upgrade immediately, a service must support older versions of the API for some period of time. If you are using an HTTP-based mechanism such as REST, one approach is to embed the version number in the

URL. Each service instance might handle multiple versions simultaneously. Alternatively, you could deploy different instances that each handle a particular version.

Handling Partial Failure

As mentioned in the previous article about the API Gateway, in a distributed system there is the ever-present risk of partial failure. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request. A service might be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests.

Consider, for example, the Product details scenario from that article. Let's imagine that the Recommendation Service is unresponsive. A naive implementation of a client might block indefinitely waiting for a response. Not only would that result in a poor user experience, but in many applications it would consume a precious resource such as a thread. Eventually the runtime would run out of threads and become unresponsive as shown in the following figure.



To prevent this problem, it is essential that you design your services to handle partial failures.

A good to approach to follow is the one described by Netflix. The strategies for dealing with partial failures include:

- Network timeouts – Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.

- Limiting the number of outstanding requests – Impose an upper bound on the number of outstanding requests that a client can have with a particular service. If the limit has been reached, it is probably pointless to make additional requests, and those attempts need to fail immediately.
- Circuit breaker pattern – Track the number of successful and failed requests. If the error rate exceeds a configured threshold, trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.
- Provide fallbacks – Perform fallback logic when a request fails. For example, return cached data or a default value such as empty set of recommendations.

Netflix Hystrix is an open source library that implements these and other patterns. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non-JVM environment you should use an equivalent library.

IPC Technologies

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 Thrift. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP. There are also a variety of different message formats. Services can use human readable, text-based formats such as JSON or XML. Alternatively, they can use a binary format (which is more efficient) such as Avro or Protocol Buffers. Later on we will look at synchronous IPC mechanisms, but first let's discuss asynchronous IPC mechanisms.

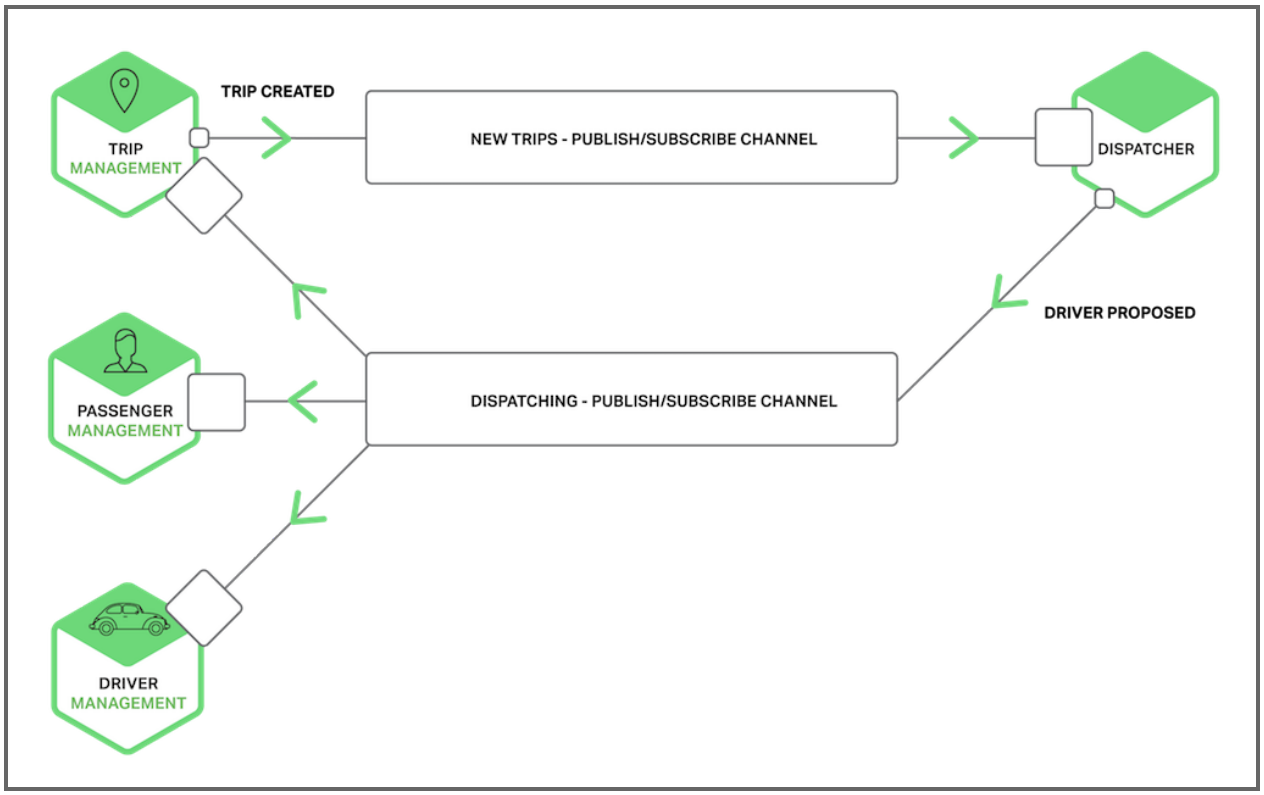
Asynchronous, Message-Based Communication

When using messaging, processes communicate by asynchronously exchanging messages. A client makes a request to a service by sending it a message. If the service is expected to reply, it does so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. Instead, the client is written assuming that the reply will not be received immediately.

A message consists of headers (metadata such as the sender) and a message body. Messages are exchanged over channels. Any number of producers can send messages to a channel. Similarly, any number of consumers can receive messages from a channel. There are two kinds of channels, point-to-point and publish-subscribe. A point-to-point channel delivers a message to exactly one of

the consumers that is reading from the channel. Services use point-to-point channels for the one-to-one interaction styles described earlier. A publish-subscribe channel delivers each message to all of the attached consumers. Services use publish-subscribe channels for the one-to-many interaction styles described above.

The following diagram shows how the taxi-hailing application might use publish-subscribe channels.



The Trip Management service notifies interested services such as the Dispatcher about a new Trip by writing a Trip Created message to a publish-subscribe channel. The Dispatcher finds an available driver and notifies other services by writing a Driver Proposed message to a publish-subscribe channel.

There are many messaging systems to chose from. You should pick one that supports a variety of programming languages. Some messaging systems support standard protocols such as AMQP and STOMP. Other messaging systems have proprietary but documented protocols. There are a large number of open source messaging systems to choose from, including RabbitMQ, Apache Kafka, Apache ActiveMQ, and NSQ. At a high level, they all support some form of messages and

channels. They all strive to be reliable, high-performance, and scalable. However, there are significant differences in the details of each broker's messaging model.

There are many advantages to using messaging:

- Decouples the client from the service – A client makes a request simply by sending a message to the appropriate channel. The client is completely unaware of the service instances. It does not need to use a discovery mechanism to determine the location of a service instance.
- Message buffering – With a synchronous request/response protocol, such as a HTTP, both the client and service must be available for the duration of the exchange. In contrast, a message broker queues up the messages written to a channel until they can be processed by the consumer. This means, for example, that an online store can accept orders from customers even when the order fulfillment system is slow or unavailable. The order messages simply queue up.
- Flexible client-service interactions – Messaging supports all of the interaction styles described earlier.
- Explicit inter-process communication – RPC-based mechanisms attempt to make invoking a remote service look the same as calling a local service. However, because of the laws of physics and the possibility of partial failure, they are in fact quite different. Messaging makes these differences very explicit so developers are not lulled into a false sense of security.

There are, however, some downsides to using messaging:

- Additional operational complexity – The messaging system is yet another system component that must be installed, configured, and operated. It's essential that the message broker be highly available, otherwise system reliability is impacted.
- Complexity of implementing request/response-based interaction – Request/response-style interaction requires some work to implement. Each request message must contain a reply channel identifier and a correlation identifier. The service writes a response message containing the correlation ID to the reply channel. The client uses the correlation ID to match the response with the request. It is often easier to use an IPC mechanism that directly supports request/response.

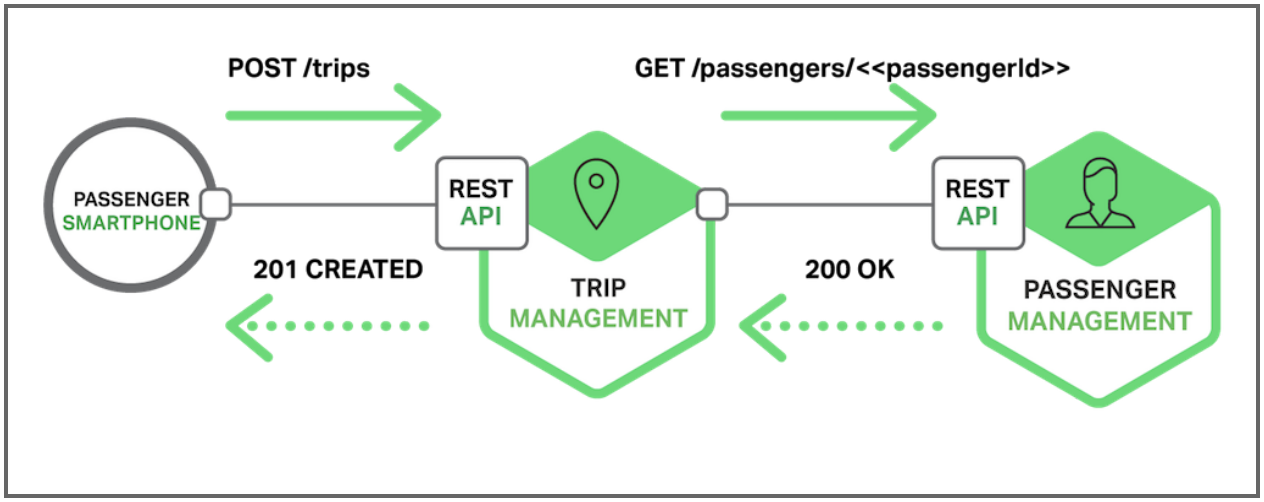
Now that we have looked at using messaging-based IPC, let's examine request/response-based IPC.

Synchronous, Request/Response IPC

When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service. The service processes the request and sends back a response. In many clients, the thread that makes the request blocks while waiting for a response. Other clients might use asynchronous, event-driven client code that is perhaps encapsulated by Futures or Rx Observables. However, unlike when using messaging, the client assumes that the response will arrive in a timely fashion. There are numerous protocols to choose from. Two popular protocols are REST and Thrift. Let's first take a look at REST.

REST

Today it is fashionable to develop APIs in the RESTful style. REST is an IPC mechanism that (almost always) uses HTTP. A key concept in REST is a resource, which typically represents a business object such as a Customer or Product, or a collection of business objects. REST uses the HTTP verbs for manipulating resources, which are referenced using a URL. For example, a GET request returns the representation of a resource, which might be in the form of an XML document or JSON object. A POST request creates a new resource and a PUT request updates a resource. The following diagram shows one of the ways that the taxi-hailing application might use REST.



The passenger's smartphone requests a trip by making a POST request to the /trips resource of the Trip Management service. This service handles the request by sending a GET request for information about the passenger to the Passenger Management service. After verifying that the passenger is authorized to create a trip, the Trip Management service creates the trip and returns a 201 response to the smartphone.

Many developers claim their HTTP-based APIs are RESTful. However, as Fielding describes in this blog post, not all of them actually are. Leonard Richardson (no relation) defines a very useful maturity model for REST that consists of the following levels.

- Level 0 – Clients of a level 0 API invoke the service by making HTTP POST requests to its sole URL endpoint. Each request specifies the action to perform, the target of the action (e.g. the business object), and any parameters.
- Level 1 – A level 1 API supports the idea of resources. To perform an action on a resource, a client makes a POST request that specifies the action to perform and any parameters.
- Level 2 – A level 2 API uses HTTP verbs to perform actions: GET to retrieve, POST to create, and PUT to update. The request query parameters and body, if any, specify the action's parameters. This enables services to leverage web infrastructure such as caching for GET requests.
- Level 3 – The design of a level 3 API is based on the terribly named HATEOAS (Hypertext As The Engine Of Application State) principle. The basic idea is that the representation of a resource returned by a GET request contains links for performing the allowable actions on that resource. For example, a client can cancel an order using a link in the Order representation returned in response to the GET request sent to retrieve the order. Benefits of HATEOAS include no longer having to hardwire URLs into client code. Another benefit is that because the representation of a resource contains links for the allowable actions, the client doesn't have to guess what actions can be performed on a resource in its current state.

There are numerous benefits to using a protocol that is based on HTTP:

- HTTP is simple and familiar.
- You can test an HTTP API from within a browser using an extension such as Postman or from the command line using curl (assuming JSON or some other text format is used).
- It directly supports request/response-style communication.
- HTTP is, of course, firewall-friendly.
- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using HTTP:

- It only directly supports the request/response style of interaction. You can use HTTP for notifications but the server must always send an HTTP response.

- Because the client and service communicate directly (without an intermediary to buffer messages), they must both be running for the duration of the exchange.
- The client must know the location (i.e., the URL) of each service instance. As described in the previous article about the API Gateway, this is a non-trivial problem in a modern application. Clients must use a service discovery mechanism to locate service instances.

The developer community has recently rediscovered the value of an interface definition language for RESTful APIs. There are a few options, including RAML and Swagger. Some IDLs such as Swagger allow you to define the format of request and response messages. Others such as RAML require you to use a separate specification such as JSON Schema. As well as describing APIs, IDLs typically have tools that generate client stubs and server skeletons from an interface definition.

Thrift

Apache Thrift is an interesting alternative to REST. It is a framework for writing cross-language RPC clients and servers. Thrift provides a C-style IDL for defining your APIs. You use the Thrift compiler to generate client-side stubs and server-side skeletons. The compiler generates code for a variety of languages including C++, Java, Python, PHP, Ruby, Erlang, and Node.js.

A Thrift interface consists of one or more services. A service definition is analogous to a Java interface. It is a collection of strongly typed methods. Thrift methods can either return a (possibly void) value or they can be defined as one-way. Methods that return a value implement the request/response style of interaction. The client waits for a response and might throw an exception. One-way methods correspond to the notification style of interaction. The server does not send a response.

Thrift supports various message formats: JSON, binary, and compact binary. Binary is more efficient than JSON because it is faster to decode. And, as the name suggests, compact binary is a space-efficient format. JSON is, of course, human and browser friendly. Thrift also gives you a choice of transport protocols including raw TCP and HTTP. Raw TCP is likely to be more efficient than HTTP. However, HTTP is firewall, browser, and human friendly.

Message Formats

Now that we have looked at HTTP and Thrift, let's examine the issue of message formats. If you are using a messaging system or REST, you get to pick your message format. Other IPC mechanisms such as Thrift might support only a small number of message formats, perhaps only one. In either case, it's important to use a

cross-language message format. Even if you are writing your microservices in a single language today, it's likely that you will use other languages in the future.

There are two main kinds of message formats: text and binary. Examples of text-based formats include JSON and XML. An advantage of these formats is that not only are they human-readable, they are self-describing. In JSON, the attributes of an object are represented by a collection of name-value pairs. Similarly, in XML the attributes are represented by named elements and values. This enables a consumer of a message to pick out the values that it is interested in and ignore the rest. Consequently, minor changes to the message format can be easily backward compatible.

The structure of XML documents is specified by an XML schema. Over time the developer community has come to realize that JSON also needs a similar mechanism. One option is to use JSON Schema, either stand-alone or as part of an IDL such as Swagger.

A downside of using a text-based message format is that the messages tend to be verbose, especially XML. Because the messages are self-describing, every message contains the name of the attributes in addition to their values. Another drawback is the overhead of parsing text. Consequently, you might want to consider using a binary format.

There are several binary formats to choose from. If you are using Thrift RPC, you can use binary Thrift. If you get to pick the message format, popular options include Protocol Buffers and Apache Avro. Both of these formats provide a typed IDL for defining the structure of your messages. One difference, however, is that Protocol Buffers uses tagged fields, whereas an Avro consumer needs to know the schema in order to interpret messages. As a result, API evolution is easier with Protocol Buffers than with Avro. This blog post is an excellent comparison of Thrift, Protocol Buffers, and Avro.