### **Communicating using the synchronous Remote procedure invocation pattern**

When using a remote procedure invocation-based IPC mechanism, a client sends a request to a service, and the service processes the request and sends back a response. Some clients may block waiting for a response, and others might have a reactive, nonblocking architecture. But unlike when using messaging, the client assumes that the response will arrive in a timely fashion.

The business logic in the client invokes a proxy interface, implemented by an RPI proxy adapter class. The RPI proxy makes a request to the service. The request is handled by an RPI server adapter class, which invokes the service’s business logic via an interface. It then sends back a reply to the RPI proxy, which returns the result to the client’s business logic.

Diagram, shape, polygon

Description automatically generated

*The client’s business logic invokes an interface that is implemented by an RPI proxy adapter class. The RPI proxy class makes a request to the service. The RPI server adapter class handles the request by invoking the service’s business logic.*

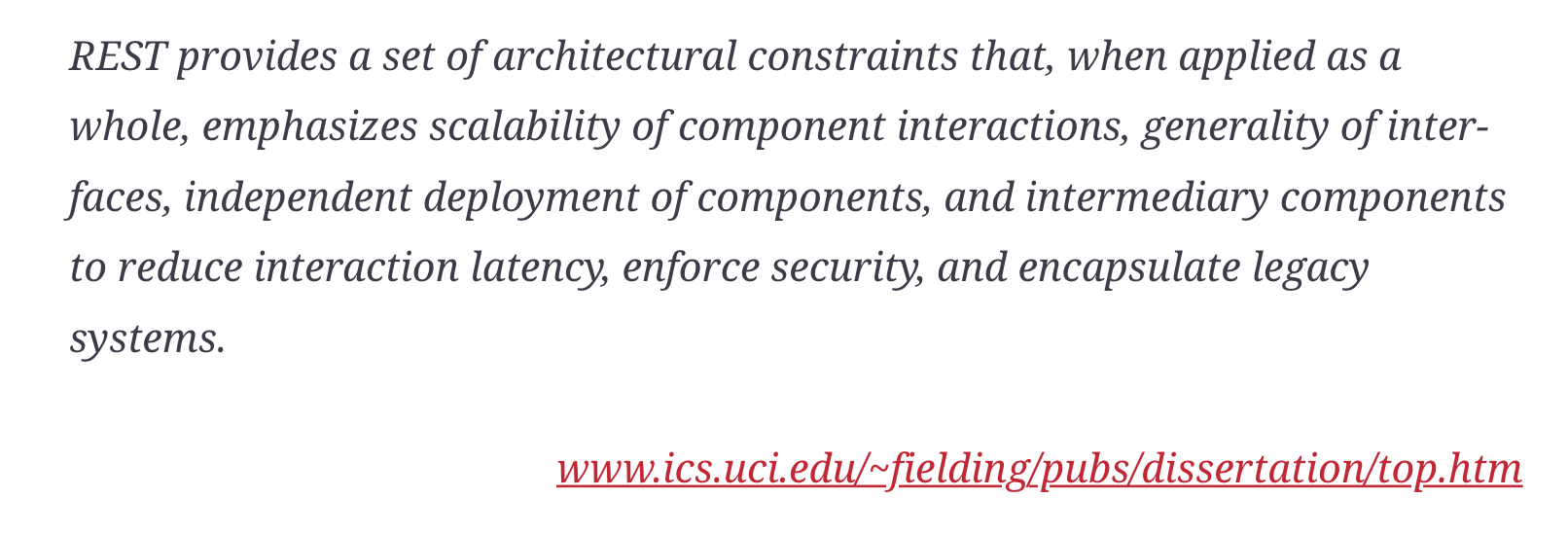
**Pattern: Remote procedure invocation**

A client invokes a service using a synchronous, remote procedure invocation-based protocol, such as REST (<http://microservices.io/patterns/communication-style/messaging.html>)

The proxy interface usually encapsulates the underlying communication protocol. There are numerous protocols to choose from. In this section, I describe REST and gRPC. I cover how to improve the availability of your services by properly handling partial failure and explain why a microservices-based application that uses RPI must use a service discovery mechanism.

**Using REST**

Today, it’s fashionable to develop APIs in the RESTful style (<https://en.wikipedia.org/wiki/Representational_state_transfer>). *REST* is an IPC mechanism that (almost always) uses HTTP. Roy Fielding, the creator of REST, defines REST as follows:

****

A key concept in REST is a resource, which typically represents a single 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 is often in the form of an XML document or JSON object, although other formats such as binary can be used. A POST request creates a new resource, and a PUT request updates a resource.

Many developers claim their HTTP-based APIs are RESTful. But as Roy Fielding describes in a blog post, not all of them actually are (<http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>). To understand why, let’s take a look at the REST maturity model.

**The REST maturity model**

Leonard Richardson (no relation to your author) defines a very useful maturity model for REST (<http://martinfowler.com/articles/richardsonMaturityModel.html>) that consists of the following levels:

**Level 0—** Clients of a level 0 service 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 (for example, the business object), and any parameters.

**Level 1—** A level 1 service 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 service 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 actions’ parameters. This enables services to use web infrastructure such as caching for GET requests.

**Level 3**— The design of a level 3 service 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 actions on that resource. For example, a client can cancel an order using a link in the representation returned by the GET request that retrieved the order. The benefits of HATEOAS include no longer having to hard-wire URLs into client code ([www.infoq.com/news/2009/04/hateoas-restful-api-advantages](http://www.infoq.com/news/2009/04/hateoas-restful-api-advantages)).

##### Specifying REST APIs

Unlike older communication protocols like CORBA and SOAP, REST did not originally have an IDL. Fortunately, the developer community has rediscovered the value of an IDL for RESTful APIs. The most popular REST IDL is the Open API Specification ([www.openapis.org](http://www.openapis.org/)), which evolved from the Swagger open source project. The Swagger project is a set of tools for developing and documenting REST APIs. It includes tools that generate client stubs and server skeletons from an interface definition.

**The challenge of fetching multiple resources in a single request**

REST resources are usually oriented around business objects, such as Consumer and Order. Consequently, a common problem when designing a REST API is how to enable the client to retrieve multiple related objects in a single request. For example, imagine that a REST client wanted to retrieve an Order and the Order’s Consumer. A pure REST API would require the client to make at least two requests, one for the Order and another for its Consumer. A more complex scenario would require even more round-trips and suffer from excessive latency.

One solution to this problem is for an API to allow the client to retrieve related resources when it gets a resource. For example, a client could retrieve an Order and its Consumer using GET /orders/order-id-1345?expand=consumer. The query parameter specifies the related resources to return with the Order. This approach works well in many scenarios but it’s often insufficient for more complex scenarios. It’s also potentially time consuming to implement. This has led to the increasing popularity of alternative API technologies such as GraphQL ([http://graphql.org](http://graphql.org/)) and Netflix Falcor (<http://netflix.github.io/falcor/>), which are designed to support efficient data fetching.

**The challenge of mapping operations to HTTP verbs**

Another common REST API design problem is how to map the operations you want to perform on a business object to an HTTP verb. A REST API should use PUT for updates, but there may be multiple ways to update an order, including cancelling it, revising the order, and so on. Also, an update might not be idempotent, which is a requirement for using PUT. One solution is to define a sub-resource for updating a particular aspect of a resource. The Order Service, for example, has a POST /orders/{orderId}/cancel endpoint for cancelling orders, and a POST /orders/{orderId}/revise endpoint for revising orders. Another solution is to specify a verb as a URL query parameter. Sadly, neither solution is particularly RESTful.

This problem with mapping operations to HTTP verbs has led to the growing popularity of alternatives to REST, such as gPRC.

**Benefits and drawbacks of REST**

There are numerous benefits to using REST:

* It’s simple and familiar.
* You can test an HTTP API from within a browser using, for example, the Postman plugin, 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 REST:

* It only supports the request/response style of communication.
* Reduced availability. Because the client and service communicate directly without an intermediary to buffer messages, they must both be running for the duration of the exchange.
* Clients must know the locations (URLs) of the service instances(s). Clients must use what is known as a service discovery mechanism to locate service instances.
* Fetching multiple resources in a single request is challenging.
* It’s sometimes difficult to map multiple update operations to HTTP verbs.

Despite these drawbacks, REST seems to be the de facto standard for APIs, though there are a couple of interesting alternatives. GraphQL, for example, implements flexible, efficient data fetching.

gRPC is another alternative to REST. Let’s take a look at how it works.

**Using gRPC**

As mentioned in the preceding section, one challenge with using REST is that because HTTP only provides a limited number of verbs, it’s not always straightforward to design a REST API that supports multiple update operations. An IPC technology that avoids this issue is gRPC (www.grpc.io), a framework for writing cross-language clients and servers (see https://en.wikipedia.org/wiki/Remote\_procedure\_call for more). gRPC is a binary message-based protocol, and this means—as mentioned earlier in the discussion of binary message formats—you’re forced to take an API-first approach to service design. You define your gRPC APIs using a Protocol Buffers-based IDL, which is Google’s language-neutral mechanism for serializing structured data.

You use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons. The compiler can generate code for a variety of languages, including Java, C#, NodeJS, and GoLang. Clients and servers exchange binary messages in the Protocol Buffers format using HTTP/2.

A gRPC API consists of one or more services and request/response message definitions. A service definition is analogous to a Java interface and is a collection of strongly typed methods. As well as supporting simple request/response RPC, gRPC support streaming RPC. A server can reply with a stream of messages to the client. Alternatively, a client can send a stream of messages to the server.

gRPC uses Protocol Buffers as the message format. Protocol Buffers is, as mentioned earlier, an efficient, compact, binary format. It’s a tagged format. Each field of a Protocol Buffers message is numbered and has a type code. A message recipient can extract the fields that it needs and skip over the fields that it doesn’t recognize. As a result, gRPC enables APIs to evolve while remaining backward-compatible.

##### An excerpt of the gRPC API for the Order Service

service OrderService {

rpc createOrder(CreateOrderRequest) returns (CreateOrderReply) {}

rpc cancelOrder(CancelOrderRequest) returns (CancelOrderReply) {}

rpc reviseOrder(ReviseOrderRequest) returns (ReviseOrderReply) {}

...

}

Graphical user interface, text, application

Description automatically generated

gRPC API for the Order Service. It defines several methods, including createOrder(). This method takes a CreateOrderRequest as a parameter and returns a CreateOrderReply.

CreateOrderRequest and CreateOrderReply are typed messages. For example, CreateOrderRequest message has a restaurantId field of type int64. The field’s tag value is 1.

gRPC has several benefits:

Graphical user interface, text, application, email

Description automatically generated

gRPC also has several drawbacks:

Text

Description automatically generated with medium confidence

gRPC is a compelling alternative to REST, but like REST, it’s a synchronous communication mechanism, so it also suffers from the problem of partial failure. Let’s take a look at what that is and how to handle it.

**Handling partial failure using the Circuit breaker pattern:**

In a distributed system, whenever a service makes a synchronous request to another service, there is an ever-present risk of partial failure. Because the client and the service are separate processes, a service may not be able to respond in a timely way to a client’s request. The service could be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests. Because the client is blocked waiting for a response, the danger is that the failure could cascade to the client’s clients and so on and cause an outage.

**Pattern: Circuit breaker**

An RPI proxy that immediately rejects invocations for a timeout period after the number of consecutive failures exceeds a specified threshold. See <http://microservices.io/patterns/reliability/circuit-breaker.html>.

A mobile client makes a REST request to an API gateway, is the entry point into the application for API clients. The API gateway proxies the request to the unresponsive Order Service.

**An API gateway must protect itself from unresponsive services, such as the Order Service**

**Diagram

Description automatically generated**

A naive implementation of the OrderServiceProxy would 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 API gateway would run out of resources and become unable to handle requests. The entire API would be unavailable.

It’s essential that you design your services to prevent partial failures from cascading throughout the application. There are two parts to the solution:

* You must use design RPI proxies, such as OrderServiceProxy, to handle unresponsive remote services.
* You need to decide how to recover from a failed remote service.

**Developing robust RPI proxies**

Whenever one service synchronously invokes another service, it should protect itself using the approach described by Netflix (<http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html>). This approach consists of a combination of the following mechanisms:

Text, letter

Description automatically generated

Netflix Hystrix (<https://github.com/Netflix/Hystrix>) is an open source library that implements these and other patterns. If you’re using the JVM, you should definitely consider using Hystrix when implementing RPI proxies. And if you’re running in a non-JVM environment, you should use an equivalent library. For example, the Polly library is popular in the .NET community (<https://github.com/App-vNext/Polly>).

**Recovering from an unavailable service**

Using a library such as Hystrix is only part of the solution. You must also decide on a case-by-case basis how your services should recover from an unresponsive remote service. One option is for a service to simply return an error to its client.

For example:

* The only option is for the API gateway to return an error to the mobile client.
* In other scenarios, returning a fallback value, such as either a default value or a cached response, may make sense.
* In other scenarios, the API gateway implements the GET /orders/{orderId} endpoint using API composition. It calls several services, aggregates their responses, and sends a response to the mobile app. The code that implements the endpoint must have a strategy for handling the failure of each service that it calls:

Diagram

Description automatically generated

It’s likely that each service’s data isn’t equally important to the client. The data from the Order Service is essential. If this service is unavailable, the API gateway should return either a cached version of its data or an error. The data from the other services is less critical. A client can, for example, display useful information to the user even if the delivery status was unavailable. If the Delivery Service is unavailable, the API gateway should return either a cached version of its data or omit it from the response.

It’s essential that you design your services to handle partial failure, but that’s not the only problem you need to solve when using RPI. Another problem is that in order for one service to invoke another service using RPI, it needs to know the network location of a service instance. On the surface this sounds simple, but in practice it’s a challenging problem. You must use a service discovery mechanism.

**Using service discovery**

Say you’re writing some code that invokes a service that has a REST API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are usually static. For example, your code could read the network locations from a configuration file that’s occasionally updated. But in a modern, cloud-based microservices application, it’s usually not that simple.

**Service instances have dynamically assigned IP addresses:**

**Diagram

Description automatically generated**

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code must use a service discovery.

**Overview of service discovery**

As you’ve just seen, you can’t statically configure a client with the IP addresses of the services. Instead, an application must use a dynamic service discovery mechanism. Service discovery is conceptually quite simple: its key component is a service registry, which is a database of the network locations of an application’s service instances.

The service discovery mechanism updates the service registry when service instances start and stop. When a client invokes a service, the service discovery mechanism queries the service registry to obtain a list of available service instances and routes the request to one of them.

There are two main ways to implement service discovery:

* The services and their clients interact directly with the service registry.
* The deployment infrastructure handles service discovery.

One way to implement service discovery is for the application’s services and their clients to interact with the service registry. As shown below:

Diagram

Description automatically generated

This approach to service discovery is a combination of two patterns. The first pattern is the Self registration pattern. A service instance invokes the service registry’s registration API to register its network location. It may also supply a health check URL. The health check URL is an API endpoint that the service registry invokes periodically to verify that the service instance is healthy and available to handle requests. A service registry may require a service instance to periodically invoke a “heartbeat” API in order to prevent its registration from expiring.

**Pattern: Self registration**

A service instance registers itself with the service registry. See <http://microservices.io/patterns/self-registration.html>.

The second pattern is the Client-side discovery pattern. When a service client wants to invoke a service, it queries the service registry to obtain a list of the service’s instances. To improve performance, a client might cache the service instances. The service client then uses a load-balancing algorithm, such as a round-robin or random, to select a service instance. It then makes a request to a select service instance.

**Pattern: Client-side discovery**

A service client retrieves the list of available service instances from the service registry and load balances across them. See <http://microservices.io/patterns/client-side-discovery.html>.

Application-level service discovery has been popularized by Netflix and Pivotal. Netflix developed and open sourced several components: Eureka, a highly available service registry, the Eureka Java client, and Ribbon, a sophisticated HTTP client that supports the Eureka client. Pivotal developed Spring Cloud, a Spring-based framework that makes it remarkably easy to use the Netflix components. Spring Cloud-based services automatically register with Eureka, and Spring Cloud-based clients automatically use Eureka for service discovery.

One benefit of application-level service discovery is that it handles the scenario when services are deployed on multiple deployment platforms. Imagine, for example, you’ve deployed only some of services on Kubernetes, and the rest is running in a legacy environment. Application-level service discovery using Eureka, for example, works across both environments, whereas Kubernetes-based service discovery only works within Kubernetes.

One drawback of application-level service discovery is that you need a service discovery library for every language—and possibly framework—that you use. Spring Cloud only helps Spring developers. If you’re using some other Java framework or a non-JVM language such as NodeJS or GoLang, you must find some other service discovery framework. Another drawback of application-level service discovery is that you’re responsible for setting up and managing the service registry, which is a distraction. As a result, it’s usually better to use a service discovery mechanism that’s provided by the deployment infrastructure.

**Applying the platform-provided service discovery patterns**

Docker and Kubernetes have a built-in service registry and service discovery mechanism. The deployment platform gives each service a DNS name, a virtual IP (VIP) address, and a DNS name that resolves to the VIP address. A service client makes a request to the DNS name/VIP, and the deployment platform automatically routes the request to one of the available service instances. As a result, service registration, service discovery, and request routing are entirely handled by the deployment platform.

The platform is responsible for service registration, discovery, and request routing. Service instances are registered with the service registry by the registrar. Each service has a network location, a DNS name/virtual IP address. A client makes a request to the service’s network location. The router queries the service registry and load balances requests across the available service instances.

Diagram

Description automatically generated

The deployment platform includes a service registry that tracks the IP addresses of the deployed services.

This approach is a combination of two patterns:

Text

Description automatically generated

**Pattern: 3rd party registration**

Service instances are automatically registered with the service registry by a third party. See <http://microservices.io/patterns/3rd-party-registration.html>

**Pattern: Server-side discovery**

A client makes a request to a router, which is responsible for service discovery. See <http://microservices.io/patterns/server-side-discovery.html>

The key benefit of platform-provided service discovery is that all aspects of service discovery are entirely handled by the deployment platform. Neither the services nor the clients contain any service discovery code. Consequently, the service discovery mechanism is readily available to all services and clients regardless of which language or framework they’re written in.

One drawback of platform-provided service discovery is that it only supports the discovery of services that have been deployed using the platform. For example, as mentioned earlier when describing application-level discovery, Kubernetes-based discovery only works for services running on Kubernetes. Despite this limitation, I recommend using platform-provided service discovery whenever possible.