Ramindur Singh

26 July 2024

Simple Message Exchange (SIMEX) API and Simex Oriented Asynchronous Architecture (SOAA)

Introduction

Network communication is, at its very basic level, an exchange of data between two systems. In most systems, the message's intent is encoded in the network protocol or in the software code that handles the message. One of the most common protocols used on the Internet is the HTTP protocol, used by web browsers, Single Page Applications (SPA) and mobile devices. Since the "Architectural Styles and the Design of Network-based Software Architectures" paper by Roy Fieldings, Representational State Transfer (REST) has become widely used with mobile apps and SPAs to communicate with backend services. However, REST is not only used for data exchange between mobile apps/SPAs and backend services, it is also used to exchange data within the backend services in private networks firewalled off the Internet.

However, additional network protocols are also used within the backend services, such as message queueing software (MQ), RabbitMQ and KAFKA being examples of such.

Following is a list of some problems that can arise from such a software architecture:

1. The intent of the message is not clear in the data exchanged but has to be gleaned from meta-data (i.e. the URL and the HTTP method in REST, the queue name in MQ software);
2. The data cannot be bridged from one protocol to another without first transforming it (i.e. a REST message cannot be sent across MQ unchanged as some information is contained in the meta-data);
3. With HTTP-based APIs, any changes to the set of available APIs (i.e. a new URL) require changes to the gateway or load-balancers for forwarding messages to appropriate HTTP endpoints;
4. HTTP-based APIs are not asynchronous; the client makes a request and then waits for a response before proceeding with the next tasks.

This paper proposes a software architecture, a messaging API and a network protocol that, when combined:

1. The interactions are completely decoupled or asynchronous;
2. Eliminates the requirement to make changes to the gateways when there are changes to the HTTP endpoints or additional services are added;
3. The data contains all the information and can be passed from one service to another without making changes to prepare the response;
4. It is network protocol agnostic.

The three elements are:

1. Simple Message Exchange API (Simex) - a way of constructing the messages that are used to communicate between clients and services.
2. Simex Oriented Asynchronous Architecture (SOAA) - an architecture that uses Simex API to communicate between network devices and services.
3. Simex Exchange Protocol (SEP) - a UDP based network protocol for passing Simex messages using encryption.

Whereas Simex messages and SOAA are mandatory to achieve these goals, SEP is optional. The benefits of using Simex/SOAA are:

1. No changes need to be made to the gateway when there are changes in the backend (i.e. new services or some old services are removed);
2. Clients, such as mobile apps and SPAs, can send a SIMEX request, do some other work, and then pick up the response as and when needed, freeing up the frontend application rather than just waiting for the response;
3. The whole system, from the frontends to the backends, becomes truly asynchronous.

Chapter 1: Simex Oriented Asynchronous Architecture (SOAA)

1.1 Introduction

In most applications that involve HTTP, whether using REST or not, clients will send a request and then wait for a response. HTTP is a connection-oriented protocol, in which involves a network handshake with each request. One of the aims of SOAA is a complete de-coupling of request-response between clients and services that handle requests.

Imagine a cloakroom in a concert hall where a show is presented over several days. Perhaps in such a hall, very few will stay from beginning to end. As many will be dropping off coats whilst others are collecting theirs, a single queue for both drop-off and collection would be long and customers can end up missing part of the show due to delays. The management may decide to streamline this by having two desks, a drop-off point and a collection point. The SOAA architecture proposes such a system. This has the advantage that client apps can send a request to backend services and, rather than waiting for a response for the data, can carry out other tasks and then pick up the response when required. Such a system makes the client/backend interaction asynchronous.

In this paper, the following definitions are used:

- **Service Owner** - the entity that provides services to third parties who own and manage the services.
- **Gateway** - this is not necessarily a single entity but can be composed of more than one entity - for example, the gateway can be a combination of a firewall and a load balancer. It's responsibility is to protect the internal network and forward incoming requests from the external network to appropriate services in the internal network.
- **Backend services** - a set of services that are behind the gateway and provide a set of APIs.
- **External Network** - network outside of the gateway, usually the Internet from where client mobile apps, SPAs and external entities access services.
- **Internal Network** - network behind the gateway, controlled and configured by the service owner.

In addition to decoupling the HTTP request-response, a network protocol is also proposed to replace the TCP three-way handshake, using UDP.

1.2 Drop-Off Point

A drop-off point service is where all requests, usually HTTP, are received from clients in the external network through the gateway. As the drop-off handles all incoming requests, the gateway configuration will rarely, if ever, change with changes in the services. The drop-off service will:

1. Validate that the message received is a valid SIMEX message;
2. Check that the destination of the message is supported;
3. Check the security credentials of the message;
4. Forward the message to the destination orchestrator.

1.3 Collection Point

A collection point service is the point from which clients in the external network collect the responses. This service, once set up, should rarely, if ever, change. When it receives a request for a response collection, it will:

1. Validate that the message received is a valid SIMEX message;
2. Check that it holds the response for this request;
3. Check the security credentials of the request;
4. Reply with the response if it passes all of the above steps.

1.4 Gateway

As mentioned in the introduction, the gateway is the door between the external and the internal network. It can be a single service, such as a load balancer, or a combination of network devices, such as firewalls and load balancers. Its main role should be to provide network security and to forward incoming messages from the external network to either the drop-off point or the collection point. Hence, it only supports two URLs, if using HTTP. The gateway configuration should never change with changes in the backend services.

1.5 Service Orchestrators

The backend services can be composed of micro-services, each providing a specific service, or a monolith providing a set of services. Each of these services can be viewed as orchestrators that has the responsibility to prepare a response. Once the response has been prepared, it will then send the response to the collection point.

Service orchestrators will only receive requests from the drop-off service or other orchestrators. Hence, it does not need to validate that the request is a valid SIMEX request. However, it may check the security credentials and level to prepare the response.

If the concept of orchestrators can also be used in mobile apps and SPAs. These would be the equivalent to a service that handles a particular API.

1.6 Simex Exchange Protocol (SEP)

As mentioned in the introduction, SOAA proposes a network protocol to reduce network latency by using UDP (User Datagram Protocol) over TCP (Transmission Control Protocol). There are a number of benefits to using UDP over TCP as UDP:

- UDP does not require the network layer to open up communication channels prior to sending data; this reduces the network latency as TCP requires a 3-way handshake before transmitting messages.
- UDP does not keep track of what it has sent; in SOAA, each requests are tracked and managed by the application.

However, in order to ensure that the message was received without corruption, the client (a client is defined as one sending a request) does check the response (this effectively becomes a 2-way handshake with data exchange which is still less than TCP (which requires a 3-way handshake, followed by data exchange, and then a 4-way connection termination).

As SOAA is network protocol agnostic, SEP is supported along with all the other network protocols. SEP is described in detail in chapter 3.

Chapter 2: Simple Message Exchange (SIMEX) API

2.1 Overview

As mentioned in the introduction, a SIMEX message contains information rather than just data. Although this may seem semantic, in computer science, information and data have different meanings. Data is simply raw facts, such as a string or a number. When that data has a context to make it meaningful, then it becomes information. A SIMEX message is composed of information, i.e. data and context. A SIMEX message has the following pieces of information:

1. Destination information - who should handle the message and how;
2. Client information - Who made the request;
3. Origin information - Who made the original request;
4. Data for the destination services for it can fulfil the request.

As there is a single standard message structure, message API can be used across different message handlers and network protocols. The message itself can be transmitted over HTTP protocols, messaging APIs and any other network protocols. However, there is no stipulation on whether this data should use XML, JSON or some other data format, including binary, as long as both the sender and the receiver agree on the format.

In this paper, we are going to use JSON format, as it is fairly easy to read, and Scala Language class objects as the initial library has been developed in Scala. The Scala library, simex-messaging, has transformers for converting Scala classes to JSON format and vice-versa.

2.2 The SIMEX Message

The SIMEX message format has the following structure:

```
{
  "destination": {},
  "client": {},
  "originator": {},
  "data": []
}
```

In Scala, the class is defined as:

```
case class Simex(
    destination: Endpoint,
    client: Client,
    originator: Originator,
    data: Vector[Datum]
)
```

In the following description of the different fields, when a value type is defined, then this is a required field and must be present. If it defined as "… or null" then this is an optional field and does not need to be present.

2.3 Destination Attributes

The destination section defines who handles the request. In DoC parlance, this is the orchestrator service that receives the request and prepares the response. It may send this request to other services for it to prepare a response for the collection point service.

The destination section has the following structure:

```
"destination" : {
  "resource" : string,
  "method" : string,
  "entity" : string or null,
  "version" : string, default “v1”
}
```

RESOURCE

The resource is the identification of the orchestrator that will handle the response. As such, each orchestrator in a system, both in the external and in the internal network, has a unique identifier. The HTTP REST equivalent would be a URL that points to a resource.

METHOD

The method defines the message's intent - the action that should be carried out on the supplied data. This is equivalent to the “Method” in REST request.

The following methods are defined in SIMEX:

- SELECT - a GET or read operation;
- UPDATE - an update to existing data operation;
- INSERT - save a new piece of data;
- DELETE - delete an existing data;
- PROCESS - run a particular process/function on the system;
- RESPONSE - the message is a response to a request (any other method than this is a request)

It should be apparent that this is similar to database SQL semantics.

ENTITY

This is the business entity that the orchestrator handles. A particular orchestrator can handle more than one type of information. For example, a database orchestrator may deal with fetching data from multiple tables or an authentication service that handles authentication requests, password resets, etc. Entity setting can be used to differentiate between these different requests.

VERSION

There are two ways in which this value can be used:

1. The version of the Simex message format. Should the message format change, then this field can be used to indicate this.
2. To address changes, usually to handle changed data but within the same “entity,” the service owners can use this field. An equivalent in REST would be to use ***/entity/v1*** and ***/entity/v2*** URL.

The main point here is that the business can decide how this value, if at all, should be used. One strategy would be to combine the Simex version with the API version using the format <Simex Version>-<Business Version>, such as in “v1-1” and “v1-2” but, as mentioned before, there is flexibility on how this is used.

RATIONALE

In the Scala class, the destination is defined by the Endpoint.scala:

```
case class Endpoint(
    resource: String,
    method: String,
    entity: Option[String]
)
```

The terminology, ***endpoint***, ***resource***, and ***method***, should be familiar to software developers as these have been borrowed from REST and SQL.

Whereas both ***resource*** and ***method*** are required fields, ***entity*** is optional, especially if the orchestrator handles only one business logic.

2.4 Client Attributes

The client section is defined in JSON as follows:

```
"client" : {
  "clientId" : string,
  "requestId" : string,
  "sourceEndpoint" : string,
  "authorization" : string
}
```

In Scala, the Client.scala is defined as:

```
case class Client(
    clientId: String,
    requestId: String,
    sourceEndpoint: String,
    authorization: String
)
```

CLIENTID

In order to understand who sent this message, a unique client ID is used. This can be the host ID, an IP address, or some other form for identifying the device/host that sent this request. As such, this ID should be unique across the whole system, both in the internal and external network.

REQUESTID

Every time a new request is sent, a request ID should be generated. This needs to be unique only for the host. Two or more different devices or hosts can use the same request ID, as long as the combination of client ID and request ID are unique.

SOURCEENDPOINT

The SourceEndpoint is the actual service that sent the request - the orchestrator in the DoC parlance.

AUTHORIZATION

The security token is used to verify both authentication and authorisation of the request. This is similar to the JWT token sent in REST requests, but it can also be a security key for backend service communication.

RATIONALE

As can be seen from the above description of the client section, this holds all the information required in order to understand who sent the request.

As there may be multiple requests generated from the orchestrator to the original request, the orchestrator can combine the client ID and request ID of the original request. It can then use client IDs of the responses to match the response with the request.

2.5 Originator Attributes

The JSON format for the originator is defined as:

```
"originator" : {
  "clientId" : string,
  "requestId" : string,
  "sourceEndpoint" : string,
  "originalToken" : string,
  "security" : string,
  "messageTTL" : number or null
}
```

In Scala, Originator.scala is defined as:

```
case class Originator(
    clientId: String,
    requestId: String,
    sourceEndpoint: String,
    originalToken: String,
    security: String,
    messageTTL: Option[Long]
)
```

This section holds the original request information and set by the client of the request. It never changes by the receiver, and the receiving orchestrator will use these values in any subsequent requests to other orchestrators.

It is used by the collection point service to return the correct response for requests from clients in the external network.

CLIENTID

The ID of the original client.

REQUESTID

The request ID of the original request.

SOURCEENDPOINT

The orchestrator or service that generated this request.

ORIGINALTOKEN

The original security authorisation token.

There are two secure ways in which clients can ask for a response:

1. Use the same security token as in the original request and the collection point will match these.
2. Use a new security token in the client section and the collection point will match client and request ID.
3. As in (2) but with the original security token in the data section that collection point will match.

SECURITY

In the Simex messaging library, three levels of security are defined:

1. “Basic” - Collection Point service only checks the client ID
2. “Authorized” - Collection Point service checks the authorization token is valid
3. “Original Token” - Collection Point service checks the authorization token and that the data in the request for the response has the original token

This is simply an example of security defined in the simex-messaging library. Others can defined their own security levels by extending the trait ***Security***:

```
trait Security extends StringEnumEntry {
  def value: String
  def level: String
}
```

MESSAGETTL

The message TTL (Time To Live) is defined as Long in the Scala code. It is a number and it is up to the implementation as to how this is used. Some examples of message TTL are:

1. Number of seconds that the response should be cached.
2. The number of hops (as in IP addresses) across orchestrators that a request can make before the request is dropped (to avoid perpetual messages).

RATIONALE

The rationale for ***Originator*** is fairly simple - we need to know who originally sent the request. This should be set by the originator of the request and subsequent orchestrators should simply copy the contents into the response.

This serves two purposes:

- It assists with tracking a request and it’s response through the various systems.
- It is used by the collection point service to match the original request to the response.

2.6 Data Attributes

The data attribute holds all the data for the service to process the request. The data (plural) is made up of an array of datum (singular). The datum is defined as:

```
{
  "field" : "username",
  "check" : null,
  "value" : "tester@test.com"
}
```

From the above datum, the data field has the following format:

```
"data" : [
  {
    "field" : string,
    "check" : string or null,
    "value" : string or array of datum
  }
]
```

The value of ***field*** is the name of the field that the data value refers to, with the ***value*** field containing the actual value. The ***check*** can be null or a string. This is user-defined and can indicate to the service the check that should be carried out. For example, in a ***SELECT*** method, the check can be used to indicate a boolean operator that should be carried out in the query.

By defining the ***value*** field as a string or another array, this enables Simex messages to support values that are a composition of multiple values. An example of this is:

```
"data" : [
 {
  "field" : "person",
  "check" : null,
  "value" : [
    {
     "field": "title",
     "value": "Mr"
    },
    {
     "field": "forename",
     "value": "John"
    },
    {
     "field": "surname",
     "value": "Smith"
    },
    {
     "field": "address",
     "value": [
       {
        "field": "street",
        "value": "1 Some Street"
       },
       {
        "field": "town",
        "value": "Sometown"
       },
       {
        "field": "county",
        "value": "Somecounty"
       },
       {
        "field": "postcode",
        "value": "PP1 1PP"
       }
     ]
    }
  ]
 }
]
```

As can be noticed, the values are always represented by a string. As the value field is always a string or an array, a standard library can be developed for use across different applications. Otherwise, different Simex message data format has to be used for each specific API. Using strings does result in two stage transformations. For example, converting from JSON string to Simex message and then converting simex message to data classes that the system can use. The same is also true for serialization. However, as the simex-messaging library handles the Simex message/JSON serialization/deserialization, application developers only need to develop transformers between Simex message and application specific data classes.

Using strings does not have too much of a draw back as some systems, such as the Java Virtual Machine (JVM) is optimised for strings. Also, using strings to represent values avoids ambiguity that sometimes can arise. Perhaps the classic example of this is how time is serialized and deserialized across time zones. By using strings, contracts can be agreed upon on how the string should be interpreted.

DATA REPRESENTATION IN SCALA

The code for representing data in Scala uses the ***Datum*** class:

```
case class Datum(
    field: String,
    check: Option[String],
    value: Xor[String, Vector[Datum]]
)
```

The ***value*** can contain either a string, to represent a value of the field, or a vector (a list) of ***Datum***. This allows for recursion enabling embedded data values. The ***Xor*** class is further explained below.

XOR

The Xor class, from the slogic library, is a generic sealed trait that can contain either of two values but not both, Xor[A, B]. It's implemented by a private class, XorImpl. It is similar to the Scala class Either, but Either[A, B] is right-oriented, meaning that, in practice, the right-hand side value, of type A, is the result, and if there were errors, it would hold only the left-hand side value, of type B, contained in Right[A] or Left[B] respectively. Xor[A,B] is not oriented in either right or left, both are valid results, where it can contain either the right hand side, of type A, or the left hand side, of type B, but not both nor neither. It is always contained in XorImpl class. It is not quite a Monad, but has some of the common wrapper methods, such as fold, map, flatMap, etc. These methods always require two functions, one for A type and the other for B type.

Using Xor class inside Datum enables the value to be either a single string, or an embedded class.

Research

Chapter 3 SEP

Chapter 4 Lessons Learnt from EDA/SIMEX Architecture

1. Very little changes in the infrastructure - mostly static
2. Make changes in backend and then the front-end (avoid versioning)
3. Pass SIMEX in methods - simplifies
4. FEs can make multiple requests and pull responses as required
5. Much easier to diagnose problems with SIMEX
6. The role of orchestrator is critical in determining how the message is handled
7. The concept of orchestrator can be used to replace the service layer even in frontends