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

Abstract 3
1. Introduction 4
2. Complexity and Vulnerabilities 5
2.1 Increasing Complexity 5
2.1 Internet Security 5
3. Simple Message Exchange 7
4: Simex Oriented Asynchronous Architecture (SOAA) 8
4.1 Introduction 8
4.2 Drop-Off Point 9
4.3 Collection Point 9
4.4 Forwarders 9
4.5 Gateway 9
4.6 Service Orchestrators 9
4.7 Simex Exchange Protocol (SEP) 10
4.8 Benefits of SOAA 10
5: Simple Message Exchange (SIMEX) API 11
5.1 Overview 11
5.2 The SIMEX Message 11
5.3 Destination Attributes 11
5.4 Client Attributes 12
5.5 Originator Attributes 13
5.6 Data Attributes 14
6. Simex Exchange Protocol 17
6.1 Limitations of SEP 17
6.2 SEP Message Format 17
6.3 Payload Encryption 18
6.4 Cryptographic Algorithms 18
6.5 Post Quantum Computer Safe Algorithms 18
7. Implementation of SOAA 18
8. Service Orchestrator Example 19
9. Conclusion 20
Bibliography 21

Abstract

In 1986, Frederick Brooks wrote a paper "No Silver Bullet - Essence and Accidents of Software Engineering." In the paper, he discusses how complexity can arise in software engineering. A number of innovations have tried to address this issue, such as higher level programming languages, micro-service architecture, etc., but complexity is still an issue in software engineering. Since then, with the advent of the Internet and mobile applications, cyber security can be added to the list of challenges that software engineers face, for which there are no silver bullet.*

Today, web applications, that provide APIs for mobile and web applications, have become common in many organisations, where both complexity and cyber security can negatively affect the management of software projects.

This paper discusses a new approach to simplify the API and a network architecture that can increase security and improve developer productivity.

* The term cyber-security in this paper refers to protecting servers, data and privacy from unauthorised access.

1. Introduction

Client mobile and web applications are ubiquitous today in an ever-connected world. Such applications are driven by the clients, the so-called frontend, that makes requests to organisation's services that may be hosted in the cloud or in dedicated data centres, the so-called backend services.

Network communication is, at its very basic level, an *exchange of data between two systems*. For most frontend-backend communication, HTTP* protocol, is used to communicate information. Since the "*Architectural Styles and the Design of Network-based Software Architectures*" paper by Roy Fieldings[1], Representational State Transfer (REST) has become widely used with mobile apps and SPAs to communicate with backend services. Other methods of data transfers can also be used, such as SOAP[2], over HTTP.

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

For data communication formats, JSON is commonly used but some XML formats are still used, especially in third-part SOAP integration.

On the backend, the concept of micro-service architecture is also fairly common[3].

* Although the term HTTP is used in this paper, but it should also be thought that it includes HTTPS.

2. Complexity and Vulnerabilities

In this section, I discuss how complexity can arise and also the cyber-security vulnerabilities.

2.1 Increasing Complexity

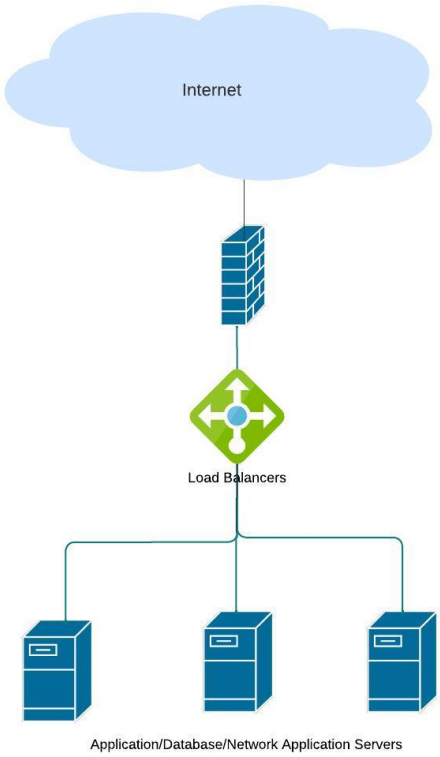
From the developer's, or software engineer's, point of view, increased complexity in an application can hinder software maintenance and enhancement. Frederick Brooks, in his 1986 paper "*No Silver Bullet Essence and Accidents of Software Engineering*" [4], describes software projects as having characteristics of werewolves. Software projects can start of as "innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products." At the time of writing of this paper, there was no "silver bullet" on the horizon.

According to [4], complexity is an "essential property, not an accident one" of any software. With additional new feature comes a "non-linear" increase in complexity. From this complexity comes "the difficulty of communication, among team members, which leads to product flaws, cost overruns, scheduled delays."

It should also be noted that it's not only the developers who face this problem. With a multitude of micro-services, messaging pipelines, etc, engineers who support such applications face the "werewolves."

Following is a list of some issues that can increase complexity for which this paper intends to be a "silver bullet":

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 exchange/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.



2.1 Internet Security

Perhaps the greatest concern is the implications to internet security when exposing APIs to clients on the Internet. The diagram on this page is a fairly typical network implementation of a web application whether on the cloud or in-house.

There may be a firewall between the Internet and the load balancer, but HTTP requests are forwarded to application services. These application services provide an API, and may either communicate with other applications through HTTP or messaging queues, may access database that can contain data that shouldn't be exposed to the Internet.

A number of exploits have over the years been revealed that has resulted in data loss. An example of this is the Apache Log4J library that was vulnerable to remote code execution[5]. When an application service is accessing the database directly, there is the issue of SQL Injection [6] that has over the years resulted in stolen data.

An application service that is accessible directly from the Internet, albeit through a firewall and load balancer, could be vulnerable.

Such an architecture does not provide isolation for applications that store sensitive data.

3. Simple Message Exchange

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 API;
3. The data contains all the information and can be passed from one service to another and across different network protocols without making changes to the structure of the message;
4. The message are constructed in such ways that vulnerabilities, such as SQL Injections, are less likely, or completely removed;
5. It is network protocol agnostic;
6. Application services that access databases are completely isolated;
7. Internet security best practices can be implemented.

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 TCP/IP based network protocol for passing Simex messages using encryption that is enhanced for Simex message exchange.

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

1. Minimal 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. Asynchronous communication - clients, such as mobile apps and SPAs, can send a SIMEX request, can 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;
4. Systems that access databases are isolated from the services that handle request/response from clients in the Internet;
5. Network architecture that has Internet security “baked-in.”

As mentioned earlier, SEP is enhanced for Simex message communication and can be used for increasing network security by using forwarders (see section 4.4).

It is described in detail in section 6.

4: Simex Oriented Asynchronous Architecture (SOAA)

4.1 Introduction

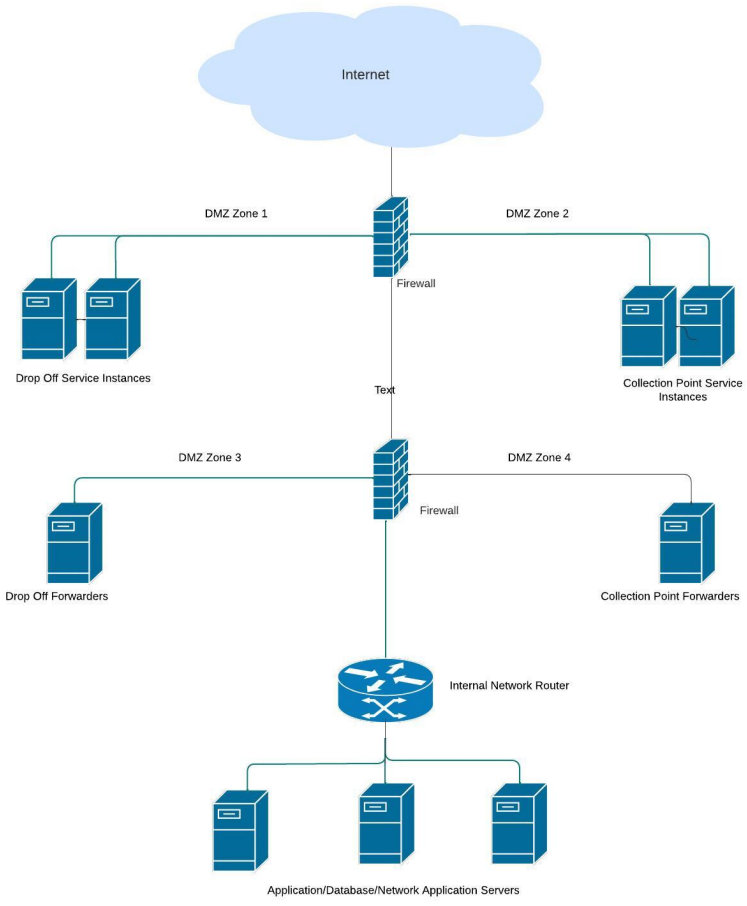
In most applications that involve HTTP, whether using REST or not, clients will send a request and then wait for a response. HTTP is a connection-oriented protocol, in which involves a network handshake with each request. One of the aims of SOAA is a complete de-coupling of request-response between clients 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.
- **DMZ** - Demilitarised Zone; a physical/logical subnetwork that contains and exposes external facing services.
- **Firewall** - a device that controls access to DMZ and protects the DMZ and internal networks from external threats.
- **Gateway** - the gateway forwards, usually HTTP, requests. In cases of multiple HTTP endpoints and instances that can handle a specific HTTP endpoint, a load balancer is used to balance requests between services.
- **Backend services** - a set of services that are behind the DMZ 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 DMZ, controlled and configured by the service owner.

The following network diagram will illustrate how SOAA can be implemented in a secure environment. What should be noted is that only two services are ever exposed to the Internet in the DMZ, the drop off and collection point services - the rest are isolated from the Internet.



In the following sub-sections, different devices and concepts are described.

4.2 Drop-Off Point

A drop-off point service is where all requests are received from clients in the external network. 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.

If HTTP is used, it will only respond with one of the HTTP 2XX status to indicate that the request was received. It can respond this way with all requests regardless of whether the destination is supported or not - this is up to the service owner's security requirement. A good candidate for response is HTTP 202 (Accepted) or 204 (No Content). The client will use HTTP POST to make this request. The client should not expect any SIMEX message returned when dropping off a request.

4.3 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.

As with the drop-off point service, if HTTP is used, the client will use HTTP POST to make this response. However, this time a SIMEX message is always returned that contains some kind of a response.

4.4 Forwarders

Forwarders are optional but provide an isolation layer between external facing services, i.e. drop off and collection point services, and the rest of the internal application services. In the diagram, they are placed in their own DMZ zone. This is not untypical in large corporations that need to have high level of firewall protection.

Forwarders are equivalent to HTTP proxy servers for the SEP protocol. However, nothing stops the service owners from using HTTP proxy servers and use HTTP.

4.5 Gateway

The gateway is not shown in the above network diagram. The reason for this is that the firewall, such as Check Point firewalls, can load-balance between instances. However, there is no reason why a load balancer or a gateway cannot be placed in the DMZ that then forwards the requests to either drop-off point or collection point instances. What should be noted here is that if some kind of gateway is used, there are only, at most, 2 URLS supported and this should never change.

4.6 Service Orchestrators

One of the key software architectural concept in SOAA is a service orchestrator. The orchestrator is the service that is responsible for preparing and sending the response, if such is required, to the collection point.

What should not happen in a SOAA is where services are chained together and as data passed through one service to another, the information is modified until eventually the response is created. This makes diagnosing faults or issues difficult in applications where a number of micro-services are involved in preparing a response.

Instead, the orchestrator involves other orchestrators by requesting data/processes and then once all the responses have been received, in preparing the response.

Service orchestrators will only receive requests from the drop-off service or other or-

chestrators. 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.

In SOAA, an applications is composed of a number of orchestrators.

4.7 Simex Exchange Protocol (SEP)

As mentioned in the introduction, SOAA proposes a network protocol, SEP, to improve SIMEX message exchange. The protocol is based on TCP/IP and is more efficient than HTTPS for the following reasons:

* There is the minimum TCP handshake to open the connection.
* There is limited set of cryptography methods used for secure communication.

As SOAA is network protocol agnostic, SEP is supported along with all the other network protocols. SEP is described in detail later in this paper.

4.8 Benefits of SOAA

So, how does SOAA help with both addressing complexity and security?

As can be seen in the network diagram, only drop-off and collection-point services are Internet facing and in a DMZ. There is no direct link between the Internet and back-end services that may be connected to sensitive data.

Drop-off services do not hold any data, it simply sends the data on to other application services.

On the other hand, collection point services do hold all the responses, but that is all. So, if data was to be stolen from these services, it would be limited to responses. It is for this reason that collection point services are in a different DMZ zone partitioned off from the drop off services.

Simex message formats are described in detail in section 5. The structure of data held in a Simex message is a string. In SOAA, Simex messages are directly deserialised into a Java/Scala class. Transformers or extractors then use the deserialised class to generate business classes that the application can use.

Although this introduces another layer, e.g. a JSON to Simex class to business logic class(es), of transformation, this does give another layer of protection from arbitrary code in the message, such as SQL Injection vectors.

Another feature of Simex message is that Simex message contains information, such as destination, what the request is, who sent it, etc. If logging is enabled, a request can be traced all the way to the response. It is also easier to understand and reason about the message then, for example, a customer record on a message queue.

Regardless of whether an application server has one or more than one service orchestrator, every service orchestrator is mapped to a unique destination. This enables the engineering team to add new features by adding new orchestrators and mapping this to destinations of a Simex message, or containing the complexity of a feature to a single orchestrator/destination.

Finally, new features (read orchestrators/destinations) can be added, removed or changed without having to make changes to services in DMZ. For example, with RESTful services, any changes to the URL will require corresponding changes in the gateway or load-balancer.

5: Simple Message Exchange (SIMEX) API

5.1 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 to show SIMEX structure 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 [7], has encoders and decoders for converting Scala classes to JSON format and vice-versa.

5.2 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.

5.3 Destination Attributes

The destination section defines who handles the request. In SOAA 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 - no changes are made to the data within the application;
- UPDATE - an update to existing data operation - this results in some changes to existing data, similar to PUT/PATCH in REST;
- INSERT - save a new piece of data - results in new record, similar to POST in REST;
- DELETE - delete an existing data - similar to DELETE method in REST;
- PROCESS - run a particular process/function on the system - this can result in changes to some information held by the application;
- 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.

SCALA

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

```
case class Endpoint(
   resource: String,
   method: String,
   entity: Option[String],
   version: String = "v1"
)
```

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.

5.4 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 SOAA 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.

5.5 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.

5.6 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). An example of datum is:

```
{
  "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 serialisation. However, as the simex-messaging library handles the Simex message/JSON serialisation/deserialisation, 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 serialised and deserialised across time zones. By using strings, contracts can be agreed upon on how the string should be interpreted.

There are some security benefits to using strings to represent values:

* The value can be encrypted string;
* To serialise a single value, the *toString* methods or bespoke methods can be used;
* To deserialise the value to a class, transformers have to be used - this can be developed to avoid vulnerabilities such as SQL Injection or other form of code injection.

It goes without saying that the data should be extracted to simple Java/Scala classes that follow the POJO/case class pattern[8].

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[9], 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.

6. Simex Exchange Protocol

The SEP protocol is a fairly simple protocol built on top of TCP/IP and involves in Simex message exchange only. It can be used to replace HTTP in frontend mobile apps, or within backend services, such as forwarders.

As at the time of writing, the Scala client and server library is in development and experimental.

The aim of SEP protocol is for a client to deliver one or more Simex messages in a batch to either a drop-off or collection point services, either directly or through forwarders.

6.1 Limitations of SEP

The development of SEP client library will be for the JVM (in Scala and Java) and IOS platforms. This makes sense as the architecture of SOAA is, by design, asynchronous and multi-threading is supported on these platforms.

However, it is not part of this paper to release a Javascript library. Javascript was not designed to be multithreaded, where processes can run concurrently, as it was running inside of a browser.

In 2009, Javascript did develop Web Worker technology for running scripts in the background thread. Web workers can perform tasks, including make network requests, without interfering with the user interface. Hence, it may be possible for web applications to fire off requests and pick up responses, whilst performing other tasks

Also, currently, there are no plans to release client or server libraries for any other platforms.

6.2 SEP Message Format

The SEP message has the following structure:

[*Session ID*][*Message Sequence*][*Message Type*][*Payload*]

Each of these sections are described below.

Session ID - 19 Bytes

When a connection is initiated by a client, it will set this to “0”s. This indicates that this is a new connection.

The server will generate a unique session ID and will be returned in the response to the client. Subsequently, this session ID will be used for all communication.

Message Sequence - 4 Bytes

An incrementing sequence set by the client. The server will respond with the same sequence in the response to that message.

Message Type - 8 Bytes

In SEP, the following message types are supported:

1. ***HELLOSVR***

Set by client when initiating a connection with the server. When initiating a connection, the client will send it’s public encrypting key in the payload.

2. ***HELLOACK***

In response to (1), the server will respond with this message type to acknowledge that the connection was established. The server will send it’s public encryption key and the symmetric encryption key in the payload.

3. ***HELLONCK***

In response to (1), the server will respond with this message when the connection could not be established. The payload will be empty.

4. ***SIMEXMSG***

Once the connection was established and the symmetric key was exchanged, the client will send the Simex message in the payload.

5. ***SIMEXACK***

The server will respond with this to (4) when it was able to accept the Simex message.

6. ***SIMEXNCK***

The server will respond with this to (4) when it was not able to accept the Simex message.

PayLoad

The payload will be Base64 encoded string. In negative acknowledgement response from

the server, the payload will not contain any data.

6.3 Payload Encryption

The payload, if present, will be encoded using Base64.

The sender of the payload will encrypt the data, whether this is the symmetric encryption key, or the serialised Simex message (i.e. in JSON format) and then encode it using Base 64.

The receiver of the payload will decode it and then decrypt it to get the original message.

Public keys are not encrypted.

The payload will itself consist of two parts: data and Base 64 encoded secure hash of the data. It will be separated by ':' character.

6.4 Cryptographic Algorithms

6.5 Post Quantum Computer Safe Algorithms

According to the paper "*On the practical cost of Grover for AES key recovery*"[10], traditional public-key algorithms such as RSA, ECDH and ECDSA are vulnerable to quantum computer attack using Shor's algorithm. However, symmetric algorithms, such as AES, are believed to be immune to Shor's algorithm.

However, according to the same paper, Grover's algorithm can cut the AES security in half. Nevertheless, by doubling the AES keys can negate this. Even AES-128 is currently deemed PQC (Post Quantum Computer) safe.

Hence, only the initial phase of symmetric key exchange, using RSA, ECDH or ECDSA, needs to be replaced.

NIST [11] already has work started on replacing vulnerable algorithms and currently FIPS-203 [12] proposes ML-KEM as a PQC safe.

There is a fairly comprehensive C library [13], supported by Post-Quantum Cryptography Alliance, that supports ML-KEM among other PQC safe algorithms. But, in order to use this within JVM, Java wrappers have to be used. Hence, there are plans to implement FIPS-203, FIPS-204 [14] and FIPS-205 [15] natively for the JVM[16]. However, this is, currently outside the scope of this work.

7. Implementation of SOAA

8. Service Orchestrator Example

9. Conclusion

Bibliography

1. https://ics.uci.edu/~fielding/pubs/dissertation/top.htm
2. https://www.w3.org/2000/xp/
3. https://martinfowler.com/articles/microservices.html
4. http://www.cs.unc.edu/techreports/86-020.pdf
5. https://nvd.nist.gov/vuln/detail/CVE-2021-44832
6. https://owasp.org/www-community/attacks/SQL_Injection
7. https://github.com/TheDiscProg/simex-messaging
8. https://en.wikipedia.org/wiki/Plain_old_Java_object
9. https://github.com/TheDiscProg/slogic
10. https://csrc.nist.gov/csrc/media/Events/2024/fifth-pqc-standardization-conference/documents/papers/on-practical-cost-of-grover.pdf
11. https://csrc.nist.gov/projects/post-quantum-cryptography
12. https://csrc.nist.gov/pubs/fips/203/final
13. https://github.com/open-quantum-safe/liboqs
14. https://csrc.nist.gov/pubs/fips/204/final
15. https://csrc.nist.gov/pubs/fips/205/final
16. https://github.com/TheDiscProg/pqc-scala