Reactive Through Services
Opinionated Framework for Developing Reactive Services

Micael Pedrosa, Jorge Miguel and Carlos Costa
Institute of Electronics and Informatics Engineering of Aveiro, Aveiro University, Campus Universitário de Santiago, Aveiro, Portugal

Keywords: Web Services, Service Specification, Reactive.

Abstract: Front-end development is inherently asynchronous, and for some time there was no correct way on how to build it in a reactive manner. Reactive Programming is a paradigm of software development centered in asynchronous data streams. Lately, this paradigm has been getting a lot of visibility due to integrations with frameworks like Angular and React. However, there is (from the electronic jargon) an impedance mismatch between the reactive UI and the required server-side services. Since common REST services do not integrate well with this new paradigm and JSON-RPC lacks some of the communication models described in this publication, we present a possible alternative for an opinionated framework which fills the gap between the reactive front-end and back-end services, maintaining the straightforward way of development that REST and JSON-RPC provides. This framework offers possibilities for new protocols and models, extending the basic REST and JSON-RPC models. It also delivers a reference implementation, certifying the viability of the proposal. Available at https://github.com/shumy/reactive-through-services.

1 INTRODUCTION

Web services have become popular in the early 90s from the DCE/RPC (Distributed Computing Environment/Remote Procedure Call), designed to distribute computation across different physical devices. This concept is now reused in the known JSON-RPC. Alongside also came up the (Representational State Transfer) REST services, introducing resources that have an associated set of standardized verbs, being these the two most used standards for Web Service Architectures. However sometimes it is difficult to map certain API concepts to the REST or RPC model. For example, REST services resolve a particularly narrow range of communication models, namely the pull based request/reply model that is fundamentally blocking and synchronous. As such, REST services have a limited use on more modern reactive applications (Mesbah and Van Deursen, 2008). It's possible to use asynchronous mode with the 202 HTTP code. Yet, standardization of asynchronous REST services in parallel with other web protocols would need to discard most of the HTTP codes and standardize some JSON structures for the requests and responses.

A complex service back-end maintains connections to other services in a chain of dependencies, propagating state changes as is defined in the Reactive Programming model (Webber, 2014). In this scenario a state change in a downstream service will be propagated to upstream services without any explicit request. In this model the application user is always aware of the status of his data by this constant feedback, even when the application is under heavy load or in a failed status. Reactive applications need to be responsive, resilient and elastic, as stated in the Reactive Manifesto (Boner et al., 2014). Message-driven architectures and push style libraries are the foundation for such responsive systems. Although there are some push style libraries available on top of HTTP, like the APE Project [1], none is quite well integrated with modern reactive libraries. The RP improves the level of abstraction of code and feedback of your applications, as such, code in RP will likely be more concise, focused in defining business logic and less on implementation details. The RP model yields higher throughput just for one thread implying lower resource usage, and most importantly, more predictable behavior under load. This programming model has become very popular due to the Node.js [2] platform.

[1] http://ape-project.org/
[2] https://nodejs.org

Pedrosa, M., Miguel, J. and Costa, C.
Reactive Through Services.
DOI: 10.5220/0006661403300337
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 330-337
ISBN: 978-989-758-295-0

The RTS (Reactive Through Services) framework was designed to build services with different communication models in a declarative way through annotations. Implementations may be provided by containers similar to JEE, aspect weavers or other similar mechanisms, or in the case of this reference implementation with compile time generators. The RTS platform was born by a necessity of building modern reactive web user interfaces that needed to connect to reactive back-end services. By defining what are the data streams, how they are linked together and what happens when their values change over time, it is possible to simplify the program flow in a declarative way. In the web environment there are many client side libraries that help in this task, one of the most known is ReactiveX [3]. Although our purpose is not to solve entirely the Reactive Manifesto challenges, we provide communication models and also ideas regarding what could be the future of reactive web services and similar frameworks.

2 TECHNOLOGY EVALUATION

It is worth mentioning that this framework is not intended to be used as an industrial system for hard or firm real time products, or even for power-constrained devices. The project goals are divided in: the architecture definition, the implementation reference and the RTS protocol. Some of the requirements that the RTS framework is trying to achieve are:

- Providing a small learning curve and development time for reactive applications.
- Improving the efficient usage of web services by providing an asynchronous API and not just an integration like in Servlets 3.0.
- Providing a good interoperability with common web technologies, like JSON, REST, WebSockets and ReactiveX.
- Providing service definitions compatible with different communication models, adding reactive models to services.
- Supporting bi-directional services where both clients and servers can act as endpoints for any communication model, avoiding additional network configurations or firewall and NAT issues.
- Providing endpoints for other protocol integrations.

[3]http://reactivex.io/

2.1 Protocols Comparison

In Table 1 there is a diversified qualitative comparison of the RTS protocol with other alternatives. A variety of scopes were selected, so to identify the exact places where the RTS fits. These scopes are classified in:

- Device-To-Device (D2D) - As defined in other publications, (Asadi et al., 2014) D2D is a direct communication between two systems without traversing the Base Station or core network.
- Device-To-Service (D2S) - Is the common client server tier architecture, using a central point for message relay.
- Service-To-Service (S2S) - Normally used for service synchronization in the underlying infrastructure and also exchanging information to maintain service availability.

Some qualitative measures are represented with: excellent, good, nice and bad. These measures consider message payload sizes, library availability and browser integration, that will serve as baselines to compare between protocols. For instance, CoAP and DDS are considered incompatible with web clients due to the underlying protocol used, not aligned with TCP/IP and thus it is not a good fit for our requirements. The RTS protocol was initially targeting the D2S scope, yet because of the framework symmetry between the client and server implementation, it can be easily extended to a S2S context. Protocols are also classified based on their communication models, with detailed descriptions in section 3.1. The RTS protocol can implement services on all the described models. In contrast, most other protocols just provide messaging and framing and are limited in describing the service semantics.

2.2 State of the Art

There are other projects that pursue some of the same RTS principles. Following is a small representation list:

- Spark Java [4] is lightweight Java web framework that despite not using annotations it follows a straight forward approach to define services. It is primarily oriented for REST and web interfaces. There is no clear separation from the service definitions and the implementations, making it difficult to integrate with other protocols and communication models. This is the norm in most REST frameworks.

[4]http://sparkjava.com/

Table 1: Qualitative comparison of some common used protocols. Purposely chosen in a wide range of contexts. Scope descriptions: S2S (Service To Service), D2S (Device To Service), D2D (Device To Device).

	Web friendly	Model	Net boundary	Network efficiency	QoS	Scope
RTS	excellent	request/reply fire-and-forget request/stream	none	nice	reliable	D2S, S2S
REST	excellent	req/reply	none	bad	reliable	D2S
JSON-RPC	excellent	request/reply pub/sub	none	nice	reliable	D2S
STOMP	good	pub/sub	none	nice	reliable	D2S
XMPP	nice	request/reply pub/sub	none	bad	reliable	D2S, S2S
MQTT	nice	pub/sub	none	good	minimal config	D2S
CoAP	bad	request/reply	NAT issues	excellent	minimal config	D2S
AMQP	nice	pub/sub	none	excellent	transaction reliable	S2S
DDS/RTPS	bad	request/reply pub/sub	LAN	excellent	high config	D2D

- Project Reactor [5] is an initiative to implement the new Reactive Streams standard (Streams, 2016) for several languages in an asynchronous foundation. It adds many additional functionalities to the standard, yet it is still just a stream library, not defining service interface semantics for the web. This can be a good project to integrate with some RTS asynchronous modules.
- Akka [6] is a serious competition, a very established framework providing reactive streams. There is a distinct list of options for endpoint interfaces with Akka-Camel (Lightbend, 2016b), cluster management, load balancing and other features. It is based in a simple message-driven architecture, not presenting any service descriptors or high level interfaces, diverging from a SOA viewpoint. When multiple actors need to be orchestrated in order to provide a specific application function, proper intertwined messaging can become demanding. There is a need for providing a higher-level abstraction. Akka is good candidate to explore in some of the RTS integrations, essentially extending endpoints.
- Lagom [7] is the most similar with RTS. Declared as a Reactive Microservice architecture (Boner, 2016) it handles service requests in a similar way and it has analogous communication models. Service descriptors are declared in Java interfaces and can be also configured for distinct transport protocols. Still there is no clear concept of communication models, and the service descriptors are bound to one kind of endpoint, e.g. when using REST identifiers (Lightbend, 2016a), not being reusable with other service interfaces. RTS is also more direct when constructing services, using a straight forward way of defining services.

[5]https://projectreactor.io/

[6]http://akka.io/

[7]https://www.lagomframework.com/

2.3 Xtend and Typescript Language Choice

For the reference implementation Xtend and Typescript languages were selected. Xtend is a language that is compiled to Java (similar to a pre-processor) specially useful when using lambda functions and asynchronous programming. Also the use of features like AA (Active Annotations) [8] aid in the implementation of declarative instructions in compile time. Avoids complex code injectors similar to the ones used in JEE containers, likewise the use of the Java reflection framework. Typescript [9] is another emerging language in the web development, introducing optional static types and other facilities that are boosting productivity for medium to large projects. Yet not very productive for small ones due to the complexities in project setup compared to JavaScript. It was also selected because it is the preferred choice when developing in Angular 2 [10]. Despite the RTS having no dependencies on the framework, the client library uses RxJS 5 from ReactiveX that is also extensively used in Angular 2.

[8]https://eclipse.org/xtend/documentation/204_activeannotations.html

[9]https://www.typescriptlang.org/

[10]https://angular.io/

3 RTS ARCHITECTURE OVERVIEW

Even thought current web architectures apply almost the same protocols as the first web servers did, internals parts have changed considerably. Notably, the rise of dynamic web contents and constant demand of feedback produced reasonable impact, pushing those architectures to a more asynchronous model. The RTS service interfaces should be considered an extension of REST services from a client point of view, it adds the push model to the existent pull network style. However from the architecture view it is quite different. The whole architecture in Fig. 1 is designed from ground to be compatible with asynchronous application servers like Vertx [11] and message-driven architectures. These are required features to fulfill the reactive manifesto (Boner et al., 2014). But there are also other points to mention in the RTS architecture:

- As presented in Fig. 1, network endpoints and protocols are detached from the service endpoints, this is a similar attempt from what was done in the SOAP architecture. In other words, the same compatible service can be accessed from a REST endpoint or from an RTS endpoint, by providing just some configurations. Endpoints are configured in designated routers for a particular protocol. It is possible to use other protocols, if they are compatible with the same communication model, mentioned in section 3.1. This improves SoC (Separation of Concerns) (Hursch and Lopes, 1995) and contribute to clean service definitions.
- Endpoint requests and replies are transformed in a Message based processing scheme. After going through the network endpoint, the created message enters the processing Pipeline where it can be intercepted by any custom user code. At this point the message can be forwarded to the next interceptor, delivered to the service or rejected, e.g by some Access Control component. Again the SoC principle is applied for these horizontal aspects of the services, by implementing common interceptors.
- Messages at the end of the Pipeline go through a decision process based on some message fields, e.g. the message type will decide if is to be delivered to a request or to a reply service handler, and if there should be a response or it's just only a notification. Some message fields reflect the used communication model avoiding the need of service descriptors to build client proxy implementations.

[11]http://vertx.io/

- There are some architecture symmetries on the client side. This means the client also has the Pipeline and the MessageBus components. It is possible to install the same services on the client Pipeline and invert the request/reply flux from server to client. However the network connection is always made in the client-server direction, avoiding network issues.

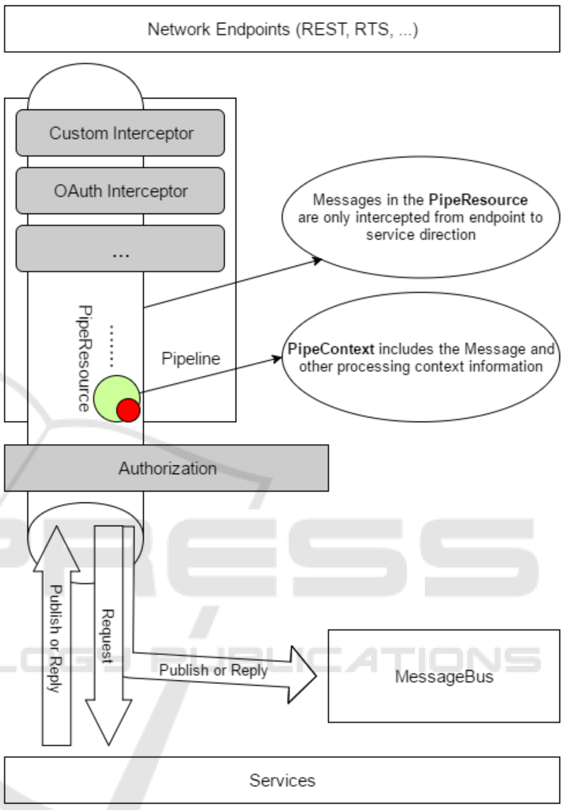
Network Endpoints (REST, RTS, ...)
Custom Interceptor
OAuth Interceptor
...
PipeResource
Pipeline
Messages in the **PipeResource** are only intercepted from endpoint to service direction
PipeContext includes the Message and other processing context information
Authorization
Publish or Reply
Request
Publish or Reply
MessageBus
Services

Figure 1: RTS Architecture essential blocks. From top to bottom: network endpoints, message pipeline, message bus and services.

3.1 RTS Communication Models

A message-driven architecture benefits from component decoupling, providing isolation through a message-bus and it's easy to add functionalities to the system over time. However messages are lazy structures when defining semantics, requiring more code to wire and construct complex service scenarios. We can verify this when trying to construct a request/reply service solely based on the pub/sub model. We need to link messages through the same identification, handle timeouts and sessions for state-full services. The RTS architecture is trying to achieve a balance between message flexibility and service semantics, by defining different feasible models in the router endpoints. RTS endpoints are build on top of WebSocket frames with JSON formatted messages. The use of JSON-RPC was initial proposed for these messages, unfortunately JSON-RPC is also bound to the request/reply and fire-and-forget models. The model we want to extend needs at least a way to send provisional responses and a correct way to identify these message types.

RTS existing models are represented in the simplified sequence diagrams on Fig. 2. Here we refer to clients and endpoints distinctly from network clients and servers. These distinctions are needed because RTS endpoints are not necessarily in the network servers, due to the already mentioned client-server symmetry. Services can be installed in the network clients or servers, however the connection is always established from client to server, avoiding network and firewall issues. We should look at the diagrams independently from who is the network client or the server.

The typical request/reply model that is used in REST services is enlarged by the JSON-RPC fire-and-forget model with the use of notifications. RTS extends this with the concept of requesting streams instead of getting all the data as a single response, each element is streamed back in order. Streams have a life cycle that can end with a complete or error signal from the server, a cancel signal from the client or with a session disconnect. Stream requests are automatically bound to the Pipeline flow that can be intercepted and controlled by any implemented Access Control mechanism without further coding.

Specifying communication models adds some advantages in terms of service compatibility. We can clearly define that REST endpoints are only compatible with the request/reply model, excluding any request/stream services to be linked to the REST endpoint. These definitions are not a common procedure when describing services. We propose that, communication models should be part of any service description and should be included in specs like RAML [12] or Swagger [13]. Although further studies should be carried out to conclude that the models described here can completely define all case scenarios.

3.2 Component Interactions

Looking at the architecture in Fig.1 we have identified some Pipeline components that will be used in the sequence diagrams Fig. 3. This is a detailed description for the interaction of the request/stream model. Two new components are presented, C-Observable and S-Observable, but actually can be considered the Stub (C)lient-Observable and Skeleton (S)ervice-Observable sides of the same component. In the Typescript client module, the C-Observable is in truth a ReactiveX Observable. Part of the initial flow is identical to a basic request/reply, but returning an Observable flag, detected by the client framework as a message reply with the field cmd="obs" instead of the typical cmd="ok". The result value is an UUID for the Observable. The client requests a response stream in step (1) and waits for the asynchronous response using the well know Promise pattern in step (2), signaling that a background process is running. A typical use case for this, is requesting a long task in the server and receiving intermediate feedbacks (3) for the ongoing process, until it ends or fails (4). When the message is delivered to the Service, it is translated to a method invocation, as also the method return to a translated reply message. The PipeContext life-cycle is also extended until the stream is completed or an error is thrown. We will not describe here all the details of all models, since the others already fall in the common knowledge. The messages are processed by the following Pipeline components:

- **PipeResource** is an abstraction for any connection or session (using TCP or any other protocol). Is also a placeholder for the connection resources, hence the chosen name. Any resources or subscriptions are state-full until the connection is released.
- **PipeContext** is an abstraction for a request context. In a REST endpoint this has the same life-cycle of a PipeResource, but in RTS with keep alive WebSockets, PipeResource's can live much longer than any PipeContext. The context is available to services via Dependency Injection to the method parameters, e.g providing the context User to the service.
- **Service** is a user class extension of an available interface IComponent (more on this later). Services expose methods as public endpoints and depending on the declared returned value it maps to a defined communication model.

3.3 Service Meta-Data and Descriptors

Service Descriptors are useful for integrations, service discover and Proxy Stub generators, effectively removing the guesswork in calling the service. In RTS, descriptors are decoupled from the service meta-data similar to how service endpoints are from the definitions. The meta-data is generated from Xtend Active Annotations and from ser-

[12]https://raml.org/

[13]https://swagger.io/

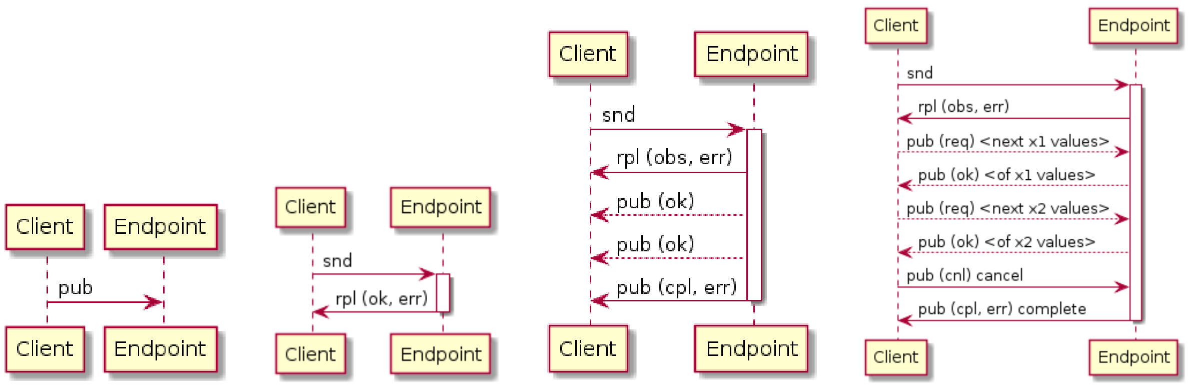

Figure 2: Available communication models in RTS endpoints. From left to right, **fire-and-forget** - no reply needed, **request/reply** - mostly used in web services, **request/stream** and **request/stream + flow control** - with or without flow control, requesting multiple asynchronous responses until completion.

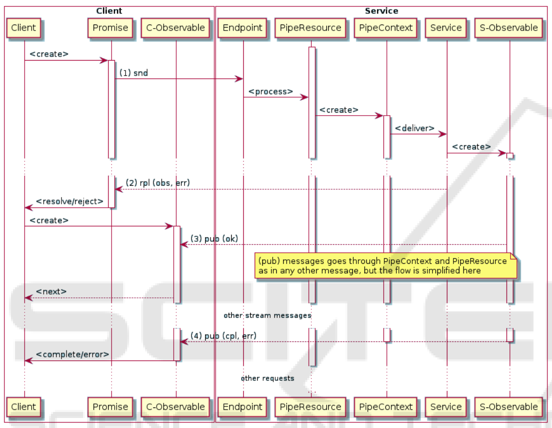

Figure 3: Request/stream model, details for message flow and component interactions.

vice DTO's (Data Transfer Objects) (Pantaleev and Rountev, 2007). It is possible to generate different descriptors for the same service e.g. both RAML and Swagger specifications. However, the problem found in most specifications is that these are frequently oriented to REST services, not describing communication models beyond the request/reply. RTS architecture is paving the way for the reactive models to be included in the service descriptors, not committing to a predefined format.

4 RTS DEMONSTRATION

Asynchronous programming is notoriously tricky, but it doesn't have to be, at least when providing service interfaces. In this section we will show some straight forward examples on how to use the framework to deliver REST and RTS services in a declarative way, using features of the Xtend language and Active Annotations. This will just demonstrate the easy of use, not covering many other optional features. In List. 1 we set a default server implementation where interceptors and services can be added. A common used interceptor is the JwtAuthInterceptor for JSON Web Token validations, and it's available in the rts-service-utils module. Also, there is a FolderManagerService service exposing a server folder for file download, upload and listing. In the service configuration, the first two parameters (service name and implementation) are mandatory. The third parameter (a similar JSON type structure) for group Access Control is optional. We will not explain the Access Control mechanisms in this document, but the code should be self explanatory.

Listing 1: Pipeline configuration example.

```
val server =
  new DefaultVertxServer(vertx, '/clt', '')

//services
val folderManagerSrv =
  FolderManagerService => [
    folder = './downloads'
  ]
...
server.pipeline => [
  addInterceptor(...)
  addService(...)
  addService(
    'folder-manager',
    folderManagerSrv,
    #{
      'list' -> 'all',
      'download' -> 'all',
      'upload' -> 'admin'
    }
  )
  ...
  failHandler = [
    println('PIPELINE-FAIL: ' + message)
  ]
]
```

In the default server implementation (DefaultVertxServer) there are 2 routers for deploying REST and RTS endpoints, available with the properties webRouter and wsRouter respectively. The provided configuration example in List. 2 for a REST endpoint and GET action at the path “/file-list/:path“ will redirect to the service name “folder-manager“ and the provided “list“ method previously defined. The REST path is defining an input variable “:path“ that, in this case is automatically mapped to the first parameter of the service method. RTS endpoints don’t need additional configurations, the wsRouter is only available to intercept some type of messages and actions e.g: on opening a connection.

Listing 2: Web Router endpoint configuration example.

```
server => [
  webRouter => [
    route(...)
    ...
    get(
      '/file-list/:path',
      'folder-manager' -> 'list'
    )
  ]
]
```

A simple service declaration example is provided in List. 3. We only need to declare the class as a Service and mark the methods as Public available for endpoints. In the hello method the return type is synchronous, but it is always processed by the internal asynchronous engine. The streamExample method returns an Observable that is recognized by the RTS as a stream response and is processed accordingly. Other options that are provided but not explained here are: service meta-data generation, interface mapping and other asynchronous options.

Listing 3: Service declaration example.

```
@Service
class TestService {
  @Public
  def hello(String firstName, String lastName)
    '''Hello <<firstName>> <<lastName>>!'''

  def streamExample() {
    val ObservableResult<String> pResult = [
      sub |
      ...
      sub.next('string-value-1')
      ...
      sub.next('string-value-2')
      ...
      sub.complete
    ]

    return pResult.observe
  }
}
```

To access the RTS endpoints there are client libraries in Xtend and Typescript. The way proxy stubs are created is by defining an interface and get the implementation from a router component, described in Listing 4. The “test“ name reference is just the service name provided in the Pipeline configuration. How to create a router is not described here, but it’s normally a step in the client initialization.

Listing 4: Service proxy stub interface definition and use example for Typescript.

```
//service proxy for TestService
interface TestProxy {
  hello(
    firstName: string,
    lastName: string
  ): Promise<string>
}

//use the proxy implementation
let testProxy =
  router.createProxy('test') as TestProxy

testProxy.hello('Johnny', 'English')
  .then(_ => { ... })
```

5 CONCLUSIONS

The myriad of emerging technologies is huge in the field that RTS is trying to innovate. To defend ourselves from redoing work from other projects, the RTS architecture was carefully constructed to achieve maximum decoupling of any implementations. It is possible to verify such claims looking at the project code, where even a simple asynchronous instruction is abstracted, like a “while“ in the rt.async.AsyncUtils class. These are not supported natively by the language and are dissociated from the reference implementation, implemented in Vertx. The RTS architecture is abstracting several concepts in a form that should be easy to plug other different technologies, to create routers and endpoints and integrate them in the same Pipeline and MessageBus. Even if the RTS protocol is proven not to be adequate for a particular use case, there are several other useful modules that can be reused to declare services and service meta-data for other implementations. We envision the use of other projects to build a micro-service oriented architecture. e.g. Eclipse Aether [14] is already integrated to build simple plugins, Apache Spark [15] is a nice target for providing processing clusters, and Akka-Camel [16] for constructing different endpoints in several protocols. Performance tests were not included since the main goal was to present the communication models, and the reference implementation that is still in an experimental stage.

6 FUTURE WORK

Although we thought that was a good idea to expose the service API's using the JavaScript Proxy, essentially reflecting the API design to the client. The main fact is that the JS Proxy is a relatively new spec, and there are no good polyfills that can be used for old browsers. This can be solved by generating the client API with the service description, instead of the dynamic construction.

The reactive-streams standard will be part of the Java 9, exposed by the Flow class. We should look at this and adapt, so that the RTS service API's could more compliant with those specs.

One of the key principles governing the Reactive Manifesto stands for quick responsiveness of the system, no matter if the response is positive or not. In a chain of asynchronous service calls, multiple failures can put the client in a long waiting due to service timeouts. There are some proposed solutions to minimize this problem, one is adding a Circuit Breaker (Fowler, 2014) between service calls for a fail fast response.

Further integration tests with stream engines are needed, e.g: Apache Spark or Akka. The implementations of meta-data and service description generations are still immature.

We recognize that breaking changes could be inserted due to advanced requirements in this future work, and that's why we are not committing to release 1.0.

7 ACKNOWLEDGMENTS

This work is financed by the ERDF European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme, and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project CMUP-ERI/TIC/0028/2014.

[14]https://projects.eclipse.org/projects/technology.aether

[15]https://spark.apache.org/

[16]http://doc.akka.io/docs/akka/snapshot/scala/camel.html

REFERENCES

Asadi, A., Wang, Q., and Mancuso, V. (2014). A survey on device-to-device communication in cellular networks. *IEEE Communications Surveys & Tutorials*, 16(4):1801–1819.

Boner, J. (2016). *Design Principles for Distributed Systems: Reactive Microservices Architecture*. O'Reilly.

Boner, J., Farley, D., Kuhn, R., and Thompson, M. (2014). The reactive manifesto. http://www.reactivemanifesto.org/. Accessed September 20, 2016.

Fowler, M. (2014). Circuit breaker pattern. http://martinfowler.com/bliki/CircuitBreaker.html. Accessed October 24, 2016.

Hursch, W. L. and Lopes, C. V. (1995). Separation of concerns.

Lightbend, I. (2016a). Lagom service descriptors: about rest identifiers. http://www.lagomframework.com/documentation/1.1.x/java/ServiceDescriptors.html#REST-identifiers. Accessed October 24, 2016.

Lightbend, I. (2016b). Untyped actors to receive and send messages over a great variety of protocols and apis.

Mesbah, A. and Van Deursen, A. (2008). A component-and push-based architectural style for ajax applications. *Journal of Systems and Software*, 81(12):2194–2209.

Pantaleev, A. and Rountev, A. (2007). Identifying data transfer objects in ejb applications. In *Proceedings of the 5th international Workshop on Dynamic Analysis*, page 5. IEEE Computer Society.

Streams, R. (2016). Standard for asynchronous stream processing with non-blocking back pressure.

Webber, K. (2014). What is reactive programming? https://medium.com/reactive-programming/what-is-reactive-programming-bc9fa7f4a7fc#.wr20gleev. Accessed October 16, 2016.