

**Deliverable D3.1**

**Document title**

|  |  |
| --- | --- |
| Editor: | Paulo Chainho, PT Inovação |
| Deliverable nature: | (R) Document, Report |
| Dissemination level: (Confidentiality) | Public (PU) |
| Contractual delivery date: | 30/09/2015 |
| Actual delivery date: | 30/09/2015 |
| Suggested readers: | Service providers’ designers and developers |
| Version: | 0.1 |
| Total number of pages: |  |
| Keywords: | Fixed: Communication networks, media, information society  Free: Hyper-linked entities, Future Internet architecture, Trustful communications, Social trustful networks, WebRTC, Independent Identity, IdP, Peer-to-peer communications,H2H, M2M, Graph ID |

***Abstract***

This Report contains a detailed specification of reTHINK Core Framework components comprised by the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. This specification is sustained by a very compreensive work in terms of state of the art research and procurement of existing open source that will be used to demonstrate the feasibility of the radical reTHINK concepts and all claimed benefits. Such aproach, will position reTHINK prototypes at the forefront, in terms of technologies and functionalities, optimising the usage of resources and complying with reTHINK ambitious calendar. This report provides the basis for the implementation tasks but it is expected to be adjusted and to be completed along the implementation phase.

[End of abstract]

Disclaimer

This document contains material, which is the copyright of certain reTHINK consortium parties, and may not be reproduced or copied without permission.

All reTHINK consortium parties have agreed to full publication of this document.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the reTHINK consortium as a whole, nor a certain part of the reTHINK consortium, warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, accepting no liability for loss or damage suffered by any person using this information.

*This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 645342. This publication reflects only the author’s view and the European Commission is not responsible for any use that may be made of the information it contains.*

Impressum

Full project title: **Trustful hyper-linked entities in dynamic networks**

Short project title: **reTHINK**

Number and title of work-package: **WP3 – Core Framework Implementation**

Number and title of task: **Task 3.1 Procurement and Specifications**

Document title: **Hyperty Runtime and Hyperty Messaging Node Specification**

Editor: **Paulo Chainho, PT Inovação**

Work-package leader: **Paulo Chainho**, company: **PT Inovação**

Copyright notice

2015 Participants in project RETHINK

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit [http://creativecommons.org/licenses/by-nc-nd/3.0](http://creativecommons.org/licenses/by-nc-nd/3.0/)

# Executive summary

reTHINK proposes a radical transformation on how real time communication services are thought and its concepts and architecture represents a significant paradigm change for the communication services domain. reTHINK approach enables the fulfilment of requirements that so far have been considered incompatible like trustful, interoperable, agility, effectiveness, quality and innovation.

Previous deliverables have already started enlightening the path to reach such ambitious objectives. A new web service paradigm, the so-called Hyperlinked Entities - Hyperties – was introduced to enable a global network of trustful services executed in web runtime environment on end-user devices or edge-network servers. Communication between Hyperties are based on the protocol-on-the-fly (ProtoFly) concept to avoid creating or modifying standard network protocols, but moving instead towards API based standards. Interoperability between Hyperties and Support Services (Registry, Catalog, Identity Management) in general are assured by a detailed but extensible data model combined with the principle of Hypermedia as the Engine of Application State.

The Current deliverable goes deeper in technical details containing the information needed by developers to start a fast prototyping of reTHINK Architecture. It takes the conceptual foundations, data models and interfaces definitions from previous work, to specify in detail the reTHINK Framework comprised by the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. Such detailed specification is sustained by a very comprehensive work done in terms of state of the art research and procurement of existing open source that will be used to demonstrate the feasibility of reTHINK architecture. Such approach, will position reTHINK prototypes at the forefront, in terms of technologies and functionalities, optimising the usage of resources and complying with reTHINK ambitious calendar.

# List of authors

|  |  |  |
| --- | --- | --- |
| Company | Author | Contribution |
| PTIN | Paulo Chainho  Micael Pedrosa  Vitor Silva  Miguel Mesquita  Luis Duarte  Joel Pinheiro |  |
| Fraunhofer | Marc Emmelmann  Andreea Ancuta Corici  Alice Cheambe |  |
| Quobis | Anton Roman Portabales  Santiago Troncoso  Yudani Riobò |  |
| Apizee | Arnaud Vallée |  |
| Deutsche Telekom | Steffen Druesedow  Frank Oberle  Ingo.Friese |  |
| Orange Labs | Ewa Janczukowicz  Simon Bécot  Kevin Corre  Eric Paillet |  |
| INESC-ID | Ricardo Lopes Pereira  Ricardo Chaves  Nuno Santos |  |
|  |  |  |
|  |  |  |

# Table of Contents

[Executive summary 3](#_Toc430300741)

[List of authors 4](#_Toc430300742)

[Table of Contents 5](#_Toc430300743)

[List of figures and/or list of tables 12](#_Toc430300744)

[Abbreviations 13](#_Toc430300745)

[Definitions 14](#_Toc430300746)

[1 Introduction 15](#_Toc430300747)

[1.1 Objectives and Overview 15](#_Toc430300748)

[1.2 Structure 17](#_Toc430300749)

[2 Requirements 18](#_Toc430300750)

[2.1 Runtime Node Requirements 18](#_Toc430300751)

[2.2 Messaging Node Requirements 19](#_Toc430300752)

[2.3 Network QoS Requirements 20](#_Toc430300753)

[2.4 Service Framework Requirements 21](#_Toc430300754)

[3 State of the Art 22](#_Toc430300755)

[3.1 Security in Runtime 22](#_Toc430300756)

[3.1.1 Web Browsers 22](#_Toc430300757)

[3.1.2 Automated Analysis of Security-Critical JavaScript APIs 27](#_Toc430300758)

[3.1.3 Secure Elements 27](#_Toc430300759)

[3.2 Standards 31](#_Toc430300760)

[3.3 Runtime 32](#_Toc430300761)

[3.4 Messaging 33](#_Toc430300762)

[3.4.1 Matrix 33](#_Toc430300763)

[3.4.2 MQ Telemetry Transport 33](#_Toc430300764)

[3.4.3 Node.js® 34](#_Toc430300765)

[3.4.4 PSYC 34](#_Toc430300766)

[3.4.5 RabbitMQ 34](#_Toc430300767)

[3.4.6 Realtime backends 34](#_Toc430300768)

[3.4.7 Redis 34](#_Toc430300769)

[3.4.8 Vert.x 34](#_Toc430300770)

[3.4.9 XMPP 34](#_Toc430300771)

[3.4.10 ZeroMQ 34](#_Toc430300772)

[3.4.11 Selected Real time Messaging Solutions 34](#_Toc430300773)

[3.5 Service Frameworks 35](#_Toc430300774)

[3.6 Projects 36](#_Toc430300775)

[3.7 Products 36](#_Toc430300776)

[3.7.1 ApiRTC 36](#_Toc430300777)

[3.7.2 State of the art of current WebRTC solutions of Quobis 39](#_Toc430300778)

[4 Hyperty Runtime Specification 47](#_Toc430300779)

[4.1 Runtime Architecture 47](#_Toc430300780)

[4.1.1 Service Provider Sandboxes 50](#_Toc430300781)

[4.1.2 Core Runtime 53](#_Toc430300782)

[4.1.3 Native Runtime 56](#_Toc430300783)

[4.2 Security analysis of the Hyperty Runtime 56](#_Toc430300784)

[4.2.1 Introduction 56](#_Toc430300785)

[4.2.2 Mitigated threats assuming an intact TCB 56](#_Toc430300786)

[4.2.3 Vulnerability assessment of the Hyperty Runtime 58](#_Toc430300787)

[4.3 Runtime APIs 65](#_Toc430300788)

[4.3.1 Runtime User Agent Interface 65](#_Toc430300789)

[4.3.2 Runtime Registry Interface 66](#_Toc430300790)

[4.3.3 Message BUS Interface 67](#_Toc430300791)

[4.3.4 Hyperty Interface 68](#_Toc430300792)

[4.3.5 Policy Enforcer Interface 68](#_Toc430300793)

[4.3.6 protoStub Interface 68](#_Toc430300794)

[4.3.7 Syncher 69](#_Toc430300795)

[4.3.8 Service Provider Sandbox interface 69](#_Toc430300796)

[4.3.9 Identity Module Interface 69](#_Toc430300797)

[4.3.10 Core Policy Decision Point (PDP) Interface 69](#_Toc430300798)

[4.3.11 Core Policy Enforcement Point (PEP) Interface 70](#_Toc430300799)

[4.3.12 QoS User Agent Interface 70](#_Toc430300800)

[4.4 Runtime Main Procedures 70](#_Toc430300801)

[4.4.1 Runtime Basic Procedures 70](#_Toc430300802)

[4.4.2 Runtime Identity Management Procedures 78](#_Toc430300803)

[4.4.3 Main Runtime Procedures for H2H Communication 83](#_Toc430300804)

[4.5 Runtime Implementation Considerations 92](#_Toc430300805)

[4.5.1 Browser Runtime Implementation 92](#_Toc430300806)

[4.5.2 Considerations about the implementation of Runtime for standalone applications 95](#_Toc430300807)

[4.5.3 Runtime implementation M2M standalone application 98](#_Toc430300808)

[5 Message Node Specification 100](#_Toc430300809)

[5.1 Messaging Node Architecture 100](#_Toc430300810)

[5.1.1 Core Functionalities 101](#_Toc430300811)

[5.1.2 Protocol Stub 101](#_Toc430300812)

[5.1.3 Connectors 101](#_Toc430300813)

[5.2 Vertx Specification 102](#_Toc430300814)

[5.2.1 Core Functionalities 102](#_Toc430300815)

[5.2.2 Protocol Stub Sandbox 103](#_Toc430300816)

[5.2.3 Connectors 103](#_Toc430300817)

[5.3 NodeJs based Messaging Node Specification 104](#_Toc430300818)

[5.3.1 Core Functionalities 104](#_Toc430300819)

[5.3.2 NodeJs implementation architecture 106](#_Toc430300820)

[5.4 Matrix.org based Messaging Node Specification 106](#_Toc430300821)

[5.4.1 Protocol Stub and Connectors 106](#_Toc430300822)

[5.4.2 Core Functionalities 107](#_Toc430300823)

[6 Conclusions 110](#_Toc430300824)

[7 References 111](#_Toc430300825)

[Annex A State of the Art 116](#_Toc430300826)

[8 Standards SOTA 117](#_Toc430300827)

[8.1.1 Overview 118](#_Toc430300828)

[8.1.2 Selected APIs with potential relevance for reTHINK 119](#_Toc430300829)

[8.1.3 OMA protocols with potential relevance for reTHINK 120](#_Toc430300830)

[8.1.4 W3C WebRTC API 120](#_Toc430300831)

[8.1.5 W3C Application Lifecycle and Events 122](#_Toc430300832)

[8.1.6 Service Workers 123](#_Toc430300833)

[8.1.7 Content Security Policy Level 2 124](#_Toc430300834)

[8.1.8 W3C Push API 125](#_Toc430300835)

[8.1.9 IETF Web Push 126](#_Toc430300836)

[8.1.10 Applicability to reTHINK 127](#_Toc430300837)

[8.1.11 References 127](#_Toc430300838)

[9 HTTP/2 128](#_Toc430300839)

[9.1 Introduction 128](#_Toc430300840)

[9.2 Main differences from HTTP1.1 128](#_Toc430300841)

[9.2.1 Binary 128](#_Toc430300842)

[9.2.2 Fully multiplexed 129](#_Toc430300843)

[9.2.3 One connection for parallelism 129](#_Toc430300844)

[9.2.4 Header compression 129](#_Toc430300845)

[9.2.5 Server push 129](#_Toc430300846)

[9.3 Features of HTTP/2 130](#_Toc430300847)

[9.3.1 HTTP/2 key concepts 130](#_Toc430300848)

[9.3.2 Framing 130](#_Toc430300849)

[9.3.3 Multiplexing 130](#_Toc430300850)

[9.3.4 Priority 131](#_Toc430300851)

[9.3.5 Flow control 131](#_Toc430300852)

[9.4 HTTP/2 encryption 131](#_Toc430300853)

[9.5 Why using for HTTP/2 in ReTHINK project. 131](#_Toc430300854)

[9.6 Websocket over HTTP/2 132](#_Toc430300855)

[10 QUIC (Quick UDP Internet Connection) 133](#_Toc430300856)

[10.1 Introduction 133](#_Toc430300857)

[10.2 Transport layer replacement 133](#_Toc430300858)

[10.2.1 Why not to use SCTP for this? 133](#_Toc430300859)

[10.3 QUIC current use 134](#_Toc430300860)

[10.4 QUIC main features 134](#_Toc430300861)

[10.4.1 Is it a good choice to use UDP for QUIC? 134](#_Toc430300862)

[10.5 Applicability to reThink project 134](#_Toc430300863)

[10.5.1 Existing QUIC implementations 135](#_Toc430300864)

[10.6 Drawbacks of using QUIC as transport protocol 135](#_Toc430300865)

[11 Projects SOTA 137](#_Toc430300866)

[11.1 WONDER Project 137](#_Toc430300867)

[11.1.1 Overview 137](#_Toc430300868)

[11.1.2 SigOfly Concept 137](#_Toc430300869)

[11.1.3 WONDER Library 139](#_Toc430300870)

[11.1.4 Input to reTHINK 140](#_Toc430300871)

[11.1.5 WONDER Messages Format 144](#_Toc430300872)

[11.2 M2M Projects 146](#_Toc430300873)

[11.2.1 CoAP/ Well-known CoRE Projects 146](#_Toc430300874)

[11.2.2 ETSI and oneM2M Projects 147](#_Toc430300875)

[11.2.3 OMA Device Management Projects 147](#_Toc430300876)

[12 Runtime SOTA 149](#_Toc430300877)

[12.1 WebRTC.org 149](#_Toc430300878)

[12.1.1 Architecture 149](#_Toc430300879)

[12.1.2 Software stack organization 150](#_Toc430300880)

[12.1.3 Code documentation 150](#_Toc430300881)

[12.1.4 Requirements Analysis 150](#_Toc430300882)

[12.2 OpenWebRTC 151](#_Toc430300883)

[12.3 V8 Javascript Engine Evaluation 152](#_Toc430300884)

[12.3.1 Overview 152](#_Toc430300885)

[12.3.2 Architecture 152](#_Toc430300886)

[12.3.3 Requirements Analysis 154](#_Toc430300887)

[12.4 Firefox OS 157](#_Toc430300888)

[12.4.1 Overview 157](#_Toc430300889)

[12.4.2 Requirements Analysis 158](#_Toc430300890)

[12.4.3 Architecture 159](#_Toc430300891)

[12.4.4 Installation Procedures 160](#_Toc430300892)

[12.4.5 Evalusation of Jitsi Meet Application 161](#_Toc430300893)

[12.5 Docker 164](#_Toc430300894)

[12.5.1 Architecture 165](#_Toc430300895)

[12.5.2 How does a Docker image works 166](#_Toc430300896)

[12.5.3 How to build a Docker image 166](#_Toc430300897)

[12.5.4 How does a Docker registry work 166](#_Toc430300898)

[12.5.5 How does a container work 167](#_Toc430300899)

[12.5.6 Dockerizing a Node.js Web App 167](#_Toc430300900)

[12.6 Janus Gateway 171](#_Toc430300901)

[12.6.1 Architecture and APIs 171](#_Toc430300902)

[12.6.2 Janus Gateway and runtime requirements 171](#_Toc430300903)

[12.7 Kurento Media Server 172](#_Toc430300904)

[12.7.1 Overview 172](#_Toc430300905)

[12.7.2 Architecture 172](#_Toc430300906)

[12.7.3 APIs 173](#_Toc430300907)

[12.7.4 Integration in Rethink 173](#_Toc430300908)

[13 Messaging SOTA 174](#_Toc430300909)

[13.1 Vert.x 2.0 174](#_Toc430300910)

[13.1.1 Overview 174](#_Toc430300911)

[13.1.2 Architecture 174](#_Toc430300912)

[13.1.3 Addressing 175](#_Toc430300913)

[13.1.4 Handlers 175](#_Toc430300914)

[13.1.5 Messaging Schemes 175](#_Toc430300915)

[13.1.6 Types of Messages 176](#_Toc430300916)

[13.1.7 Verticle 176](#_Toc430300917)

[13.1.8 Module 176](#_Toc430300918)

[13.1.9 Event Loop 177](#_Toc430300919)

[13.1.10 APIs 177](#_Toc430300920)

[13.1.11 Requirements Analysis 178](#_Toc430300921)

[13.2 Vert.x 3.0 180](#_Toc430300922)

[13.2.1 Overview 180](#_Toc430300923)

[13.2.2 Architecture 180](#_Toc430300924)

[13.2.3 Vert.x Runtime (Java 8 only) 180](#_Toc430300925)

[13.2.4 Addressing 180](#_Toc430300926)

[13.2.5 Handlers 180](#_Toc430300927)

[13.2.6 Messaging Schemes 180](#_Toc430300928)

[13.2.7 Types of Messages 181](#_Toc430300929)

[13.2.8 Verticle 181](#_Toc430300930)

[13.2.9 Module 181](#_Toc430300931)

[13.2.10 Event Loop 182](#_Toc430300932)

[13.2.11 APIs 182](#_Toc430300933)

[13.2.12 Requirements Analysis 182](#_Toc430300934)

[13.3 Realtime backends 185](#_Toc430300935)

[13.3.1 Requirements Analysis 186](#_Toc430300936)

[13.4 Matrix.org 187](#_Toc430300937)

[13.4.1 Overview 187](#_Toc430300938)

[13.4.2 Architecture 187](#_Toc430300939)

[13.4.3 APIs 187](#_Toc430300940)

[13.4.4 Requirements Analysis 188](#_Toc430300941)

[13.5 RabbitMQ Evaluation 189](#_Toc430300942)

[13.5.1 Overview 190](#_Toc430300943)

[13.5.2 Architecture 190](#_Toc430300944)

[13.5.3 APIs and Documentation 191](#_Toc430300945)

[13.5.4 Requirements Analysis 191](#_Toc430300946)

[13.6 ZeroMQ Evaluation 193](#_Toc430300947)

[13.6.1 Overview 193](#_Toc430300948)

[13.6.2 Architecture 193](#_Toc430300949)

[13.6.3 APIs and Bindings 195](#_Toc430300950)

[13.6.4 Requirements Analysis 196](#_Toc430300951)

[13.7 Redis Evaluation 197](#_Toc430300952)

[13.7.1 Overview 197](#_Toc430300953)

[13.7.2 Architecture 197](#_Toc430300954)

[13.7.3 APIs and Bindings 199](#_Toc430300955)

[13.7.4 Requirements Analysis 199](#_Toc430300956)

[13.8 XMPP 201](#_Toc430300957)

[13.8.1 Overview 201](#_Toc430300958)

[13.8.2 Architecture 201](#_Toc430300959)

[13.8.3 APIs, Bindings and Extensions 204](#_Toc430300960)

[13.8.4 Requirements Analysis 206](#_Toc430300961)

[13.9 MQTT Evaluation 207](#_Toc430300962)

[13.9.1 Overview 207](#_Toc430300963)

[13.10 Architecture 208](#_Toc430300964)

[13.11 APIs and Bindings 208](#_Toc430300965)

[13.11.2 Brokers 212](#_Toc430300966)

[13.12 Requirements Analysis 212](#_Toc430300967)

[13.13 PSYC 213](#_Toc430300968)

[13.13.1 Architecture and main functions 213](#_Toc430300969)

[13.13.2 Evolution to PSYC2 214](#_Toc430300970)

[13.13.3 Requirements Analysis 214](#_Toc430300971)

[13.14 Node.js 216](#_Toc430300972)

[13.14.1 Overview 216](#_Toc430300973)

[13.14.2 Architecture 216](#_Toc430300974)

[13.14.3 APIs 217](#_Toc430300975)

[13.14.4 Requirements Analysis 217](#_Toc430300976)

[14 QOS SOTA 220](#_Toc430300977)

[14.1 coturn 220](#_Toc430300978)

[14.1.1 Overview 220](#_Toc430300979)

[14.1.2 Requirements Analysis 220](#_Toc430300980)

[15 Service Frameworks SOTA 221](#_Toc430300981)

[15.1.1 AngularJS Framework 221](#_Toc430300982)

[15.1.2 BackboneJS Framework 223](#_Toc430300983)

[15.1.3 MeteorJS Framework 225](#_Toc430300984)

[15.1.4 StapesJS Framework 228](#_Toc430300985)

# List of figures and/or list of tables

# Abbreviations

# Definitions

# Introduction

## Objectives and Overview

This report provides a detailed specification of reTHINK Core Framework components comprised by the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. Thus, and according to reTHINK Architecture [38], the scope of this report includes the specification of the Messaging Node providing reTHINK Messaging Services and the specification of the Hyperty Runtime that will be included in User Devices and Application Servers to deliver User Hyperties and Network Side Hyperties.

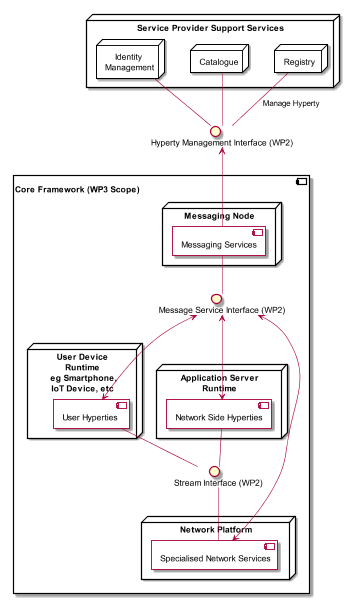


Figure 2 - WP3 Scope

It should be noted that the Network Platform specification supporting Specialised Network Services will be reported later in D3.4, as originaly planned.

These specifications are compliant with reTHINK Data Model, Hyperty Management interfaces, Stream Interface and Messaging Interface designed in [37]. It should be noted that, according to Protocol On-the-fly concept, the Messaging Interface is defined by the Message Model defined in [37].

Besides the Architecture requirements reported in D2.1 [38] additional specific requirements to Core Framework functionalities were analysed.

The specification of the Hyperty Runtime and the Messaging Node is sustained by a very compreensive work in terms of state of the art research and procurement of existing open source that will be used to demonstrate the feasibility of the radical reTHINK concepts and of all its benefits. Such aproach, will position reTHINK prototypes at the forefront, in terms of technologies and functionalities, optimising the usage of resources and complying with reTHINK ambitious calendar.

An exhastive study of relevant IETF, W3C standards and others that facilitate the fulfilment of previously analysed requirements, was conducted. Special attention was given to the research on Security in Web Runtime. In parallel, existing Open Source solutions to be used to develop Hyperty Runtime and Messaging Nodes was researched, experimented and selected.

Three solutions to implement the Messaging Node were selected in order to evaluate interoperability between different Hyperties domains that use different Message Nodes in WP6, namely Vertx, Nodejs and Matrix.

The experimentations performed on javascript engines and webrtc implementations have shown to be very difficult to extend existing runtime like V8 or Chromium to natively support Hyperties runtime. On the other hand, such approach would also not promote the adoption of Hyperty runtime by the end-users since it would demand the installation of new platforms to replace popular browsers like Chrome or Firefox. Instead, it was decided to make Hyperty runtime compliant with existing runtimes notably with existing Web Browsers like Chrome and Javascript platforms like nodejs.

The Runtime design enables the reusage of most of the core runtime components through different platforms including Browsers, Standalone Mobile Application, Network Side Application Servers and more constrained M2M/IoT standalone devices.

The specification reported in this deliverable, provides the basis for the implementation tasks but it is expected to be adjusted and to be completed along the implementation phase. In addition, during the implementation phase, an Hyperty Framework to be used by Hyperty Developers will be developed and reported in D3.2.

The final specification for messaging node and Hyperty runtime will be reported in D3.3.

## Structure

This report starts by introducing, in Chapter 2, requirements that are more specific to the domains addressed in WP3 namely Runtime Requirements, Message Node Requirements, Hyperty Framework Requirements and Quality of Service Requirements. In chapter 3 a summary of the State of the Art and Procurement work is given. The full outcome of the State of the Art work done in WP3 can be found in Annex A. The main part of this report is located in Chapter 4 - detailed specification of the Hyperty Runtime and in Chapter 5 - specification of the Message Node.

# Requirements

Besides the Architecture requirements reported in D2.1, additional specific requirements to Core Framework functionalities were analysed namely for Hyperty Runtime, Messaging Node, QoS (Specialised Network Services) and Hyperty Framework. It should be noted that Hyperty Framework will provide developer friendly tools to facilitate the development of new Hyperties and Applications.

## Runtime Node Requirements

**The effort to introduce new capabilities in the runtime should be reasonable**

The effort to introduce new capabilities in the runtime should be reasonable, for example:

* missing WebRTC features eg Identity
* Protocol on the fly mechanism
* Policy Enforcement Points
* Hyperty Registry

**The Runtime must be secured**

The runtime must support the execution of entrusted code in isolated sandboxes. It should be possible to take advantage of existing secured elements like SIM cards or embedded SIM

**The Runtime must have a good performance**

The Runtime must have a good performance

**The Runtime should support Standardised Messaging Notifications**

The Runtime should support standardised W3C [Push Messaging](http://www.w3.org/TR/push-api/) which allows applications to receive messages sent by servers regardless whether the Application is currently active. Useful to support notifications about incoming WebRTC calls.

**The Runtime should support Web Socket**

The Runtime should support [Web Socket](http://www.w3.org/TR/websockets/).

**The runtime must support standard Javascript (ECMAScript)**

The runtime must support standard at least [ECMAScript version 5](http://www.ecma-international.org/ecma-262/5.1/) (Javascript).

**The Runtime should support W3C WebRTC APIs**

The Runtime should support W3C WebRTC APIs including:

* [Media Capture and Streams API](http://www.w3.org/TR/mediacapture-streams/)
* [WebRTC](http://www.w3.org/TR/webrtc/)

**The Runtime should be deployable in the most used Devices and Operating Systems**

The Runtime should be deployable in the most used Devices and Operating Systems, including:

* Android (Smartphone and Tablet)
* iOS (Smartphone and Tablet)
* Raspberry PI
* Linux
* Windows

## Messaging Node Requirements

**Messaging Node should be resilient**

Messaging node should be Resilient when operating under overload situations or failures, in general.

**The Messaging Node MUST offer DoS and DDoS Protection**

The MN MUST provide protection against Denial of Service (DoS) and Distributed Denial of Service Attacks (DDoS)

**Messaging Node should support Protocol on-the-fly**

Messaging Node should support Protocol on-the-fly, to inter-operate with other Messaging Nodes or Back-end servers without having the need to standardize the protocol to be used.

**Messaging Node should support different Encrypted Messaging Transport Protocols**

Messaging Node should support different Encrypted Messaging Transport Protocols including:

* Encrypted WebSockets
* HTTPS Streaming
* HTTPS Long-Polling
* HTTPS REST

**Messaging Node should support logging of routed messages**

Messaging Node should support logging of routed messages and any other event (e.g. connection events) in remote log servers

**Message must support delivery reliability**

Message must support message delivery reliability. Delivery errors must be returned to clients

**Messaging Node must support worldwide scale deployments**

If required, by Messaging Node must support worldwide scale deployments

**Messaging Node should be tolerant to unstable connections**

When connections to Messaging Node are resumed from a short disconnected period of time or when client IP address changes eg due to access network handover (eg wifi to LTE), should have no impact on the client side service.

**Events must be fired when clients connect and disconnect from the Messaging Node**

It should be possible to get events with information about Messaging Node clients connection and disconnection. Such feature is useful for connection status purposes.

**Messaging Node must support very low message delivery latency**

Messaging Node must support very low message delivery latency

**Messaging Node should require minimal computing resources**

Messaging Node should require minimal computing resources in order to be deployable in constrained computing environments eg residential /enterprise gateways.

**Messaging Node must support external authentication and Authorisation**

Messaging Node must be able to use an external Authentication Service

Messaging Node must be able to use an external Authorisation Service for the following features: \* send/publish a Message \* receive a Message \* subscribe / register handlers to be notified about published messages

**Messaging Node must support multiple message oriented communication patterns**

Messaging Node must support multiple message oriented communication patterns including:

* pub/sub
* broadcast
* one to one

## Network QoS Requirements

**QoS component should assure collaboration between network and application layer actors**

There should be a collaboration between network operators (Network or Internet Service providers) and communication service providers, e.g. 3rd party communication service providers, since network operators do not have the control over the whole environment.

**QoS component should provide a single contact point for troubleshooting issues**

A single contact point should be assured for troubleshooting issues, i.e. to verify the appropriate level of the quality and make eventual improvements. A broker component will be discussed since it cas assure a single contact point and collaboration between different actors.

**QoS component should enable isolating and valorizing performance improvements**

Segments that contribute to improvement of quality should be identified. The improvements should be valorized and quantified, e.g. for monetization reasons.

**QoS component should leverage IP paths diversity**

The are different possible connection paths, i.e. paths offering best-effort services or managed paths with a certain Service Level agreement. These different possibilities should be considered.

**QoS component should impact different network types and segments**

The solution should impact mobile, wireline, wifi access networks, but also interconnection. In the discussed architecture there is a focus on offering specialized network services for fix and mobile networks and for different network segments, i.e. access and interconnection.

**QoS component should be compatible with evolution of different standards and terminals**

The proposed solution should be in line with evolution of standards and adapt to application layer technologies.

**QoS component should be compatible with deployed network technologies**

Eventual improvements should be in line with current technologies. It should not be neccessary to make big changes to existing network deployments, but current technologies should be modified or improved.

**QoS component should be compatible with application layer technologies e.g. WebRTC**

The solution should be in line with application layer technologies, so it could be easily adapted by web developers and service providers.

**QoS component should exploit partially SLA and partially best-effor network segments**

The communication service path can consist of parts provided by different actors. As not all of these actors may be willing to provide quality of service, the communication path may consist of segments with Service Level Agreement or with best effort quality. As a result there is not necessarily end-to-end path, but the quality may vary. As a result the solution should take into consideration the fact that not all network segments in the path can provide QoS.

**QoS component should take into account participating and non-participating actors**

Network QoS should take into account a fact that in the communication path there can be participating and non-participating actors, i.e. not all actors will support providing network QoS so it will not always be possible to offer end-to-end quality.

**QoS component should have global reach**

Network QoS should be provided on a global scale, i.e. it should not be limited only to a given territory or a group of clients.

## Service Framework Requirements

**Service Framework MUST be Message Node agnostic.**

Within the reTHINK project, different implementations of the Message Node will be provided. The Service Framework must be compliant across various types of Message Node used.

**The Service Framework MUST avoid any JavaScript conflicts**

The Service Framework must not pose any sort of conflict with other JavaScript Frameworks frequently used by developers. In order words, the Service Framework must be able to co-exist with other JavaScript Frameworks.

**Service Framework MUST be Modular in nature**

Development of modules on the Framework should be loosely coupled, self contained and re-usable by other libraries/modules

**The Service Framework MUST be open source**

No proprietary solution as the project itself demands all components be open source.

**Service Framework should be device agnostic**

The Service Framework should be supported on all devices and operating systems featuring the Hyperty Runtime. The Runtime is envisioned to be deployed on the most used Devices and Operating Systems, including: \* Android (Smartphone and Tablet) \* iOS (Smartphone and Tablet) \* Raspberry PI \* Linux \* Windows

**Service Framework MUST be light weight and fast**

The service framework size is important as latency plays an important role and downloading heavy weight files would add overhead thus diminishing the performance and user experience.

**Service Framework SHOULD support Model-View-Controller design pattern**

Model-View-Controller approach: enables easier maintainability and clarity. The view presents data to users through format & layout. The view is rendered by the model. The model handles data & business logic. It also allows for clear separation between the presentation (View) and application logic. Meanwhile the Controller receives user requests and calls back to the model to select a proper view via HTTP GET or POST request to manage the data. Within the Service Framework, more focus will be laid on the Model. The View and Controller will remain flexible for the developers to determine according to their requirements and preferences.

# State of the Art

In this chapter, a summary of the exhaustive work performed in terms of research of state of the art and procurement of open source solutions to be used in the implementation phase, is provided.

WP3 State of the Art, complements WP2 State of the Art and goes deeper in domains addressed by WP3, namely:

* Security in Web Runtime
* Standards that WP3 should comply with, notably W3C APIs and IETF Protocols
* Web Runtime solutions including Javascript runtime and WebRTC implementations
* Real Time Messaging solutions including nodejs and vertx
* Partners' Assets that can be leveraged by reTHINK including Quobis and APIZEE products as well as the WONDER library.

A detailed report of state of the art and procurement for Standards, Messaging, Runtime, QoS, Projects and Web Frameworks are provided in Annex A.

## Security in Runtime

In this document, we present the relevant related work on security runtime environments. We focus essentially on two areas: web browsers, and secure elements. The web browsers section present security mechanisms for JavaScript code protection in fully-featured environments (the web browsers themselves). The secure elements section provides an overview of code security runtimes for computing devices featuring less functionality and computation capabilities but requiring tighter security requirements during its operation.

### Web Browsers

#### Monolithic vs Modular Architectures

Traditionally, commercial and open-source web browsers employed a monolithic architecture. This means that both users' and web applications' data are combined into a single security domain, which brings serious performance/usability and security issues. On the performance/usability side, if a web application crashes during its execution, the whole web browser can be affected, harming the user experience. On the security side, if an attacker exploits an unpatched vulnerability in the browser while a user is using it, the attacker may gain access to the whole user space, being able to execute code on behalf of that user and access its private sensitive information, such as security credentials.

Nowadays, web browsers evolved into modular architectures, in order to achieve privilege separation and overcome monolithic architectures' limitations. This way, browser developers came up with multiple different architectures to achieve this separation between what is user's property (e.g., credentials, preferences) and what is "web’s" property (e.g., applications' code). In order to achieve this separation in these architectures, multiple techniques have been employed:

* **Sandboxing:** In computer security, a sandbox is a security mechanism which allows untrusted programs to run within a trusted environment, without affecting the environment or other co-located programs. This is usually done by restricting the resources (disk, memory, network) the untrusted software can access. An example is creating scratch memory and disk spaces where it can read/write and limiting the network capabilities it can use, in order to prevent the host environment from getting damaged. This is what Chromium browser [1] applies to separate the user and the web side in a modular architecture. It features two modules:
* A **browser kernel module** which acts on behalf of the user and is responsible for implementing the tab-based windowing system of the browser. It stores users' data as its preferences, bookmarks, credentials and cookies and also works as middleware between the native operating system window manager and every instance of the second browser module, the rendering engine.
* The **rendering engine** implements the web application behavior. It interprets and executes web content, serving calls to the DOM API. It is the unique browser part in contact with the untrusted web content. Apart from that, it is also responsible for enforcing the same-origin policy between the user and a website he's visiting.

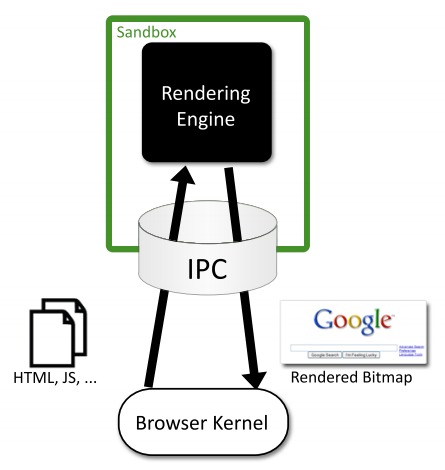


Figure 3: Chromium sandbox scheme

#### Browser Extensions Security

Browser extensions provide useful additional functionality to web browsers, such as facilitating the access to a website's content or even as almost standalone applications running on the browser environment. However, these extensions often introduce serious security issues into both user’s browser and websites. This is because oftentimes extensions are written by well-meaning developers who, however, are not security experts. Extensions can read and alter users' bookmarks and preferences, websites' content and perform requests over the network, many times on behalf of the browser user. Browser extensions are mostly written in JavaScript and HTML, and since JavaScript provides methods for converting a string to code (e.g. "eval"), an extension may be dangerous if misused.

Typically, benign extensions face two types of attackers:

* **Network attackers:** These attacks target end-users who connect to unsecure networks (i.e. public Wi-Fi hotspots), and consist in sniffing and altering HTTP traffic. These attackers search for any HTTP script - JavaScript file loaded over HTTP - loaded by the extension, and try to introduce malicious code into this script's code, in such case.
* **Web attackers:** A malicious website can launch a XSS attack on an extension if the extension treats the website as trusted, possibly stealing the browser’s userdata, like credentials. This way, it can scale up to attack multiple websites within the same entry point.

According to [2], Google Chrome and its extension platform apply three mechanisms to prevent these vulnerabilities: \* **Privilege Separation:** Every Chrome extension has two types of components which run in separate processes: zero or more content scripts and zero or one core extension. Content scripts read and modify websites as needed. The core extension implements functionality not directly involving websites, like browser UI jobs or long-running background tasks. These two types of components communicate by sending structured clones over a trusted channel. Each website that an extension communicates with, receives its own isolated instance of a content script, making content scripts highly bound to attacks. However, only the core extension is able to communicate with the Chrome extension's API, reducing the risk that a content script is able to access the user data space. The architecture scheme of a Google Chrome extension is on Fig. 4. \* **Isolated Words:** This mechanism ensures that content scripts and websites have separate JavaScript heaps and DOM objects. Consequently, content scripts never exchange pointers with websites, protecting them against web attackers. \* **Permissions:** Extension developers have to specify the desired permissions in a kind of manifest file that is packaged with the extension. For example, the bookmarks permission is needed for the extension to be able to read and alter the user's bookmarks. Only core extension can use permissions to invoke browser API methods, while content scripts are limited to interacting with the core extension and the website it is running on. This way, an extension is limited to the permissions its developer requested, so an attacker is not able to request new permissions for a compromised extension in runtime.



Figure 4: The architecture of a Google Chrome extension

#### XSS Detection Techniques

Cross-Site Scripting (XSS) attacks are getting more common on the web, since they allow an attacker to get control of a user’s browser and execute malicious code (usually JavaScript/HTML) within the trusted context of a web application. This can result in the attacker being able to access any sensitive information associated to the application (cookies, session IDs, etc.). The study of XSS attacks can be split into two distinct categories, according to [3]:

* **Persistent/Stored attacks:** Occurs when a malicious user registers itself into a web application and posts a malicious JavaScript to the application, which, by its turn, save it into the application’s data repository, persistently. After that, if another user fetches the content uploaded by the malicious one onto his browser, and since this code is coming out of the trusted context of the web application, the user’s browser will allow the script to access any possibly sensitive resource it is willing to, overcoming this way the security imposed by the same-origin policy. Apart from stealing the user’s information, XSS attacks can also be used to redirect users to a malicious website which can then perform other distinct attacks within its context. A persistent XSS attack scheme is presented on Fig. 5.

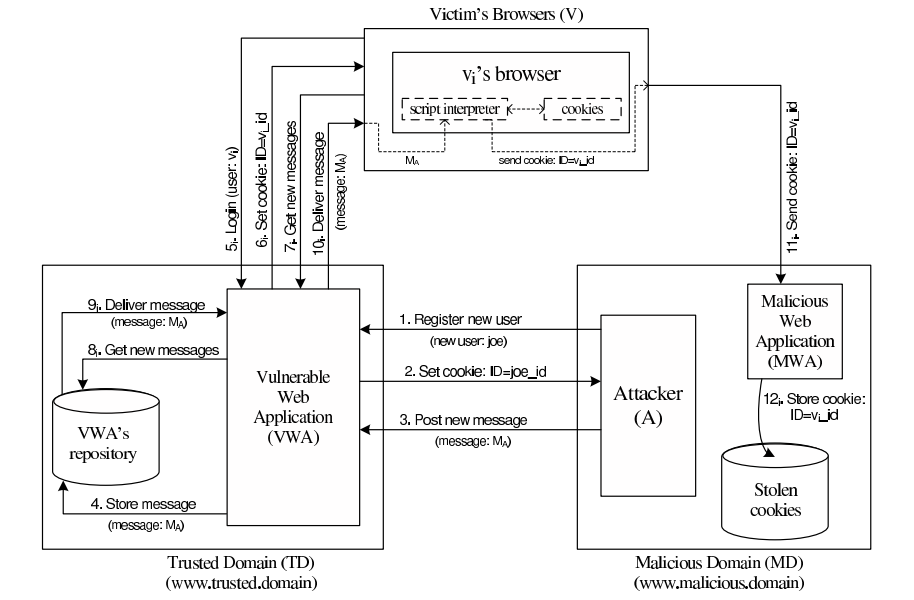


Figure 5: Scheme of a persistent XSS attack

* **Non-persistent/Reflected attacks:** Unlike the first type, reflected attacks do not persistently store malicious code in the web application data space. Instead of that, the content is automatically reflected back to the user through a third-party mechanism. For example, by using a spoofed email, an attacker can make a user click on a link containing malicious code, which will finally be interpreted by the user’s browser, but within the trusted context of the web application. This type of XSS attacks is often combined with other techniques as phishing, and is the most common type of XSS attacks in web applications. Figure 6 shows a scheme of the architecture of a non-persistent XSS attack.

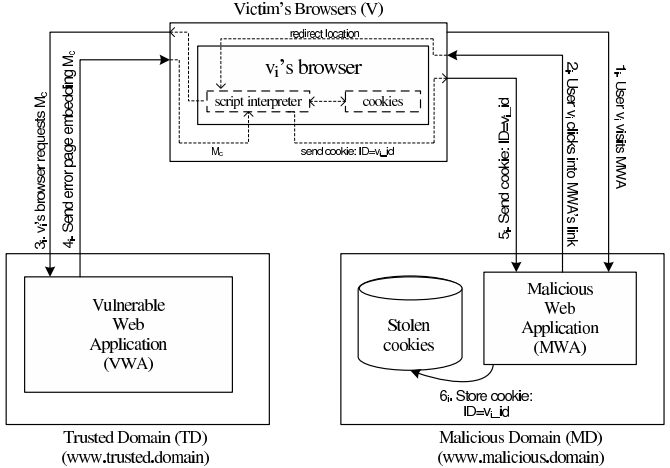


Figure 6: Scheme of a non-persistent XSS attack

#### XSS (and Other Types) Prevention Techniques

We briefly discuss two relevant XSS prevention techniques: (i) analysis and filtering of exchanged information, and (ii) security enforcement on the web browser runtime.

**Analysis and Filtering of Exchanged Information**

This technique consists in defining a list of characters or tags which users are allowed to exchange with the web application, in the form of text inputs, uploaded files, etc. Then, a filtering process simply rejects everything that is not part of the list. Other approach, reported in [4], is having a proxy-server at the web application’s site in order to filter both incoming and outgoing requests. This filtering takes into account a set of rules defined by the application developers. However, a simple use of regular expressions is able to evade both the referred methods and proxy-servers can rapidly become a performance bottleneck on the application deployment. Pietraszeck et al. [5] also suggested placing a proxy-server on the server-side of the application, but in order to differentiate trusted and untrusted traffic, driving each type to separate channels. This partitioning process uses Information Flow Control techniques to taint information and track it thenceforward.

From another point of view, some approaches [6,7] propose the content filtering to happen at the client-side. On the one hand, Kirda et al. [6] try to achieve the prevention of XSS attacks by blacklisting links embedded within the web application’s pages, making them unavailable for the client. However, the authors say this approach can only detect basic XSS attacks based on the violation of same-origin policy. On the other hand, Ismail et al. [7] present another client-proxy solution that is intended to detect malicious requests reflected from the attacker to the victim (non-persistent XSS attacks). If such a request is detected, the malicious characters are re-encoded by the proxy, trying to avoid the success of the attack.

**Security Enforcement on the Web Browser Runtime**

There are also other strategies which try to avoid the need for intermediate elements like proxy-servers by proposing startegies to enforce the runtime context of the web browser. Hallaraker et al. [8] propose an auditing system for the JavaScript interpreter of the Mozilla Firefox browser, which detects misuses on JS operations and take counter-measures to avoid violations on browser’s security. Other approach [9] presents the use of dynamic taint tracking on JavaScript code, in order to detect whether browser’s sensitive resources are going to be transferred to an untrusted third-party. In such case, the user is warned and can decide whether he allows or denies the transfer. Finally, Jim et al. [10] propose a policy-based management where a list of actions is embedded into the documents exchanged between the browser and the server. These actions help the browser to decide whether or not a script should be executed. Although, a lack of semantics in the policy-language and the restrictiveness of the approach due to the sandboxing-like mechanism are some of the drawbacks.

### Automated Analysis of Security-Critical JavaScript APIs

Current web applications usually rely on JavaScript in order to offer additional features like maps, widgets or social media content. Although, since these additions may manipulate a page Document Object Model (DOM), steal cookies or navigate on the page, untrusted third-party JavaScript may pose security threats to the hosting page.

A widely-used approach is to combine a language-based sandbox to restrict the capabilities of untrusted JavaScript with an API offered by the trusted code part to the untrusted one. This API encapsulates all security-critical resources and guarantees they are only accessed in a safe way.

Given this, Taly et al. [11] proposed ENCAP, a tool that verifies API confinement, analyzing the isolation level it can offer to the critical objects it is intended to protect. ENCAP relies on a context-insensitive and flow-insensitive static analysis method. It analyses the API implementation and generates a conservative Datalog model of all API methods. Also, they propose SESlight, an ECMA JavaScript-subset language which only allows a strict (syntactically and semantically verified) subset of the whole language to be used.

### Secure Elements

#### Java Card: Internet Computing on a Smart Card

In secure computing, a smart card is a typical card with a built-in computer chip. Until a few years ago, it was only used to produce credit and debit cards, whose information can only be accessed when in possession of the card itself and a PIN code. Due to the short information on how to communicate and program them, until a few years ago this useful technology wasn't being used on computer security in general.

**Hardware**

This single-chip computer is an off-the-shelf **8-bit microcontroller** with added tamper-safe features. While most 8-bit microcontrollers can support at least **64 KBytes** of 8-bit memory, popular smart cards contain 4 to 20 Kbytes of memory, due to size constraints. The memory space of a smart card is divided into RAM, EEPROM and ROM. RAM is used to store temporary values when a program is running, while EEPROM is used to store sensitive data as an encryption key or the account holder info on credit cards. Finally, ROM is used to store the basic programs that run on the smart card. The single-chip computer is embedded in a plastic chip carrier, and both of them hold several tamper-resistant and tamper-detection features.

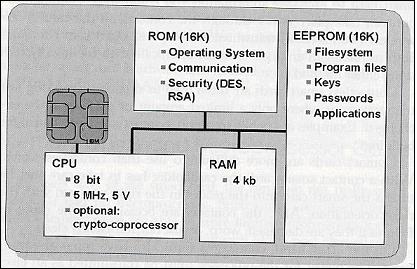


Figure 7: Java Smart Card scheme [11]

**Software**

The paucity of 8-bit assembly language courses, books and software tools led engineers to break the smart card application bottleneck by building a Java virtual machine with its runtime support into a 12-Kbyte smart card. Java was the natural answer for three reasons: \* Java brings smart card programming into the mainstream of software development \* Java “safe programming” security model based on a runtime interpreter is a nontrivial side benefit, due to its processor independence. A Java card can be deployed on multiple smart card models. \* Java interpreters were tested to the limit, holes had been found, and fixed.

With this in mind, engineers concluded that Java could preserve the required security in the smart card operation, while allowed a more friendly and well-known programming approach. However, available memory was an issue when deploying such heavy language runtime like Java. Features like garbage collection and exceptions handling were not included in Java Card because of that.

**Internet Computing with Java Smart Card**

Java Cards combine smart card’s identity-verification features with the Java “sandbox”, guaranteeing that only allowed applications run on the card and that applications are protected from each other.

#### Cloud of Secure Elements

Cloud of Secure Elements (CoSE) [12] is an emerging concept whose goal is to provide trusted computing resources to mobile and cloud applications. To achieve this, it relies on an infrastructure composed by multiple secure micro-controllers, named Secure Elements.

CoSE, in a WEB-like paradigm, are meant to support Uniform Resource Identifiers (URIs) for users to locate the different secure elements and use their embedded resources. These resources usually target two service types: Near Field Communication (NFC) facilities for mobile applications and trusted cryptographic features for cloud applications.

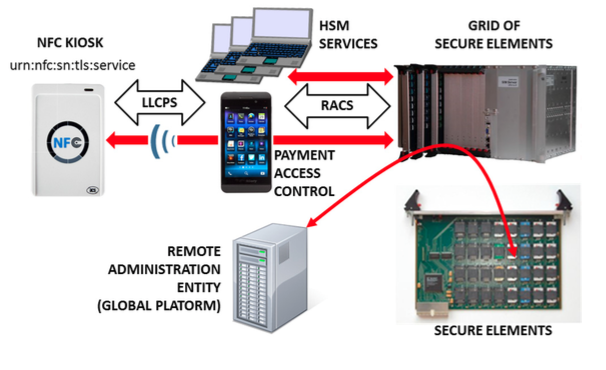


Figure 8: CoSE architecture

**Architecture**

A Cloud of Secure Elements involves the following stakeholders, as Fig. 8 shows:

* NFC kiosks, typically deliver payment facilities
* Users with NFC-enabled devices or terminals needing trusted cryptographic resources
* Grid of Secure Elements (GoSE)
* Secure elements, with resources identifiable by URI
* Remote administration entities, performing management operations over applications and secure elements

**Grid of Secure Elements (GoSE)**

A grid of secure elements is an Internet server hosting multiple secure elements. Each element may be plugged in through USB readers, hardware sockets or electronic boards. Communication may be achieved with RACS protocol (works over IP/TCP/TLS stack) and performs both the association between elements and unique identifiers and data exchange with secure elements.

**Malicious Code on Java Cards: Attacks and Countermeasures**

Despite all the advantages on using Java language in smart cards, such as the absence of low-level memory vulnerabilities, Java Cards still have an open door for attacks through malicious code. This attack entry is possible because an on-card bytecode verifier (BCV) is optional on Java Cards, and those who don't feature it, are more open to malicious code that might damage other applets running on the system or even the platform itself.

#### Defenses against Malicious Code

We present the different mechanisms for protection against malicious code actions present in Java Cards.

**Bytecode verification**

Bytecode verification of Java code guarantees type safety, and thus, memory safety. On normal Java platform, bytecode verification occurs at load time. However, since Java Cards do not support dynamic class loading, this verification must occur at the time an applet is installed to the card. Nevertheless, most Java Cards do not feature an on-card BCV and rely on a digital signature of a third party who is trusted to have performed bytecode verification off-card.

**Applet firewall**

The applet firewall is an additional defense mechanism present in Java Cards. The firewall performs runtime checks to prevent applets from accessing and/or altering data of other applets (concretely, in a different security context). For every object within an applet, the firewall records its context, and for any field or method accessed this context is checked. Only the Java Card Runtime Environment (JCRE) has unlimited permission, since it executes in root-mode, on a UNIX terminology.

#### Getting malicious code on cards

**CAP File Manipulation**

This is the easiest way of introducing ill-typed code on a Java Card. This can be achieved by editing a CAP (Converted APplet) file to introduce a type flaw in the bytecode and install it to the card. Although, this will only work for cards without an on-card BCV and with unsigned CAP files. In example, by changing a baload (byte load) opcode onto a saload (short load) one, will make the platform treat a byte array as a short array, and can potentially lead to accessing other applet's memory space.

**Abusing Shareable Interface Objects**

The shareable mechanism of Java Card can be used to create type confusion between applets without any direct editing on CAP files. Shareable interfaces allow direct communication between security contexts. Using this to create type confusion is pretty simple: Let two applets communicate through a shareable interface, but compile and generate CAP files for both applets using different definitions of the shareable interface, which is possible because the applets are compiled and loaded separately. This way we can achieve an attack like the CAP file manipulation but without ever touching the CAP file directly.

**Abusing the transaction mechanism**

The Java Card transaction mechanism is probably the trickiest aspect of the Java Card platform. It allows multiple byte-code instructions to be turned into an atomic operation, offering a roll-back mechanism in case the operation is aborted, either through card tear or calling an API method. Buggy implementations of the transaction mechanism in some cards tend to make it not behave as expected. When object references are spread around the code, by assignments to instance fields and local variables, it becomes difficult for the mechanism to keep track of all the references that should be nulled out. The root cause of the problem is that stack-allocated variables, such as short[] localArray are not subject to roll-back in the event of a programatically transaction abort (through API method call).

#### Dynamic Countermeasures

Now we enumerate some dynamic runtime checks implemented by some VMs in order to prevent ill-typed code to damage the Java Card platform. These were verified by Mostowski et al. [13], by performing tests on multiple Java Card models of multiple manufacturers against the referred vulnerabilities:

* Runtime type checking
* Object (array) bounds checking
* Physical (byte size) bounds checking
* Firewall checks
* Integrity checks in memory

## Standards

The reTHINK project describes a framework that provides solutions to manage real time communication capabilities. To implement this framework the project team tried to use the most suitable existing standards which provides compability which existing technolagies. Using consolidated and widely used standards also make the development more efficient since Open Source libraries can be used in the developments. Addtionally to well-known standards, the project team has also tried to find emerging standards which can be adapted for ReTHINK requirements. In those cases, a tradeoff analysis has been made to determine if the choice of a not consolidated standard is optimal in terms of cost of use due to the lack of existing libraries and projects which use them.

The IETF has been creating and promoting the Internet standards since 1986. The IETF is organized in a large number of Working Groups (WG) which works on specific areas. For ReTHINK project, the team has focused on standards delivered by several WG (namely Rtcweb, TRAM, HTTP/2 and Network). The Rtcweb WG has defined a set of RFCs (many of them are still drafts) which are used in WebRTC, it defines how WebRTC works on the wire. Many of the used protocols already existed but many of them were created ad-hoc to meet WebRTC requirements. Other RFCs are informational and hes been released to gather the WG knowledge in a formal way. The TRAM (TURN Revised and Modernized) working group is carrying out a modernization of the protocols used to transport real-time media over Internet which is the final function of ReTHINK framework.

HTTP/2 is the new version of HTTP/1.1 which has been used in the web for the last 16 years. It provides a new low level design to optimize current Web applications keeping the semmantic of HTTP/1.1 which is still valid. HTTP/1.1 has been historically transported over TCP, however to take advantage of all the new features of HTTP/2 a new transport protocol build over UDP has been designed: QUIC. HTTP/2 draft is based on SPDY but it includes new features and will soon become a definitive RFC. The draft belongs to the HTTP WG. QUIC was developed by Google but it has been recently become an IETF Draft taking over the last changes in the protocol until close the defintiive RFC. HTTP/2 over QUIC has been considered as an alternative for messaging in the ReTHINK framework as it is optimized to be used over wireless connection and minimizes the delay in every communication.

The IETF is in charge of standarizes all the protocols on the wire in Internet. In turn, the W3C (WWW Consortium) is the main international standards organization for the World Wide Web. It standarizes how the browser behave (e.g. WebRTC 1.0 API exposed by the browsers) and and the lenguages (e.g. HTML and Javascript) which can be executed by a standar browser. It is main role is to promote and homogenize the evolution of the Web. During the state of the Art research work we focused on the standards susceptible of being used by any element within the ReTHINK framework.

The WebRTC 1.0 API has been standarized by the W3C is the way in which a Javascript application interacts with the browser to establish real-time sessions with other WebRTC endpoints. A comprehensive knowledge of this API was necessary to make design decissions and to define the architecture and the data model of the framework.

A Community group has been created within the W3C to promote an alternative WebRTC API called ORTC (Object Real-Time Communications) which gives more control to the WebRTC developer making easier to implement some scenarios. There are still not implementations of ORTC in production-ready browser, however the features introduced by this standard which is likely to become the base of the WebRTC 2.0 API have been considered during the design phase.

Another relevant W3C API is the Push API which allows a push service to send "push messages" to a webapp regardless of whether the webapp is currently active on the user agent. This is specially usefull for webapps running on mobile devices where the webapp may need to receive a notification while the browser is not in foreground.

The use of another feature supported by browser called Service Workers has been already evaluated to be used to implement different parts of the Runtime environment. Despite the fact that this specification is still a Working Draft of the W3C it is already supported by the most important browsers. However, this is feature is not supported by server side Javascript-based runtime environment, it only can be used when the Runtime is executed by a browser.

There is another interesting W3C Draft called "Application Lifecycle and Events" which extends the Service Workers with APIs for managing the lifecycle of an application and associated events. This Draft allows web developers to create applications that manage the application lifecycle and react to system events e.g. email or VoIP application. However, this Draft has been not been adopted by many vendors so far.

In this section the standards released by the Open Mobile Alliance (OMA) were also reviewed. The OMA is a Mobile Operator driven industry forum for the definition of interoperable mobile service enablers. OMA defines APIs to offer functionalities and resources of Operator networks to developers. Amongst the API and protocols standarized by the OMA the team decided to reviewed those which are relevant for the project such as the Authorization Framework for Network APIs, the RESTful Network API for WebRTC Signaling, Quality of Service API and Notification Channel. The LWM2M/COAT protocol which was designed to be supported by constrained devices has also been considered as a suitable alternative to interact with the Registry and Discovery services.

Finally, a recent standard called Smart Device Template (SDT) and released by the HGI (Home Gateway Iniative) has been reviewed. It provides a framework to create a consistent representation of Smart Home devices. This makes easier the integration of new devices in Home Gateway or in the cloud being specially interesting to implement M2M within the ReTHINK framework.

## Runtime

A very comprehensive analysis and evaluation of existing web runtime solutions was performed.

In order to evaluate the possibility to modify native implementations of WebRTC engines, Ericsson OpenWebRTC and Google WebRTC.org solutions were considered. OpenWebRTC is a promising modular WebRTC implementation based on popular GStreamer multimedia framework open source solution. Unfortunately, OpenWebRTC is not much supported by Ericsson lacking required documentation to let it be adapted to fulfil reTHINK new requirements. Google WebRTC.org solution is the reference implementation of WebRTC specification providing all APIs defined in the standards. However, the effort required to change it to fulfill reTHINK requirements is estimated to be very high. On the other hand, having an extended version of an existing WebRTC implementation would require the user to install a new reTHINK Browser. For all the above reasons, it was decided to re-use existing native implementations of WebRTC engines without modifications.

Javascript engine solutions were evaluated to analyse the possibility to adapt them in order to fulfill reTHINK runtime requirements, notably in terms of security (sandboxing). The V8 JavaScript Engine is an open source JavaScript engine developed by Google for the Google Chrome web browser. It has since seen use in many other solutions and it is considered the most powerful Javascript engine in terms of features and performance. It has mechanisms to facilitate its extension with new features but lacks required mechanisms for sandbox creation. One evaluated alternative, is to use nodejs that runs on top of V8 as well as having nodejs inside Docker taking advantage of its management and security features. Both solutions fulfill reTHINK security requirements and will be considered for reTHINK runtime implementations that are not based on browsers.

Firefox OS is a good candidate to implement reTHINK runtime in mobile devices supporting this Operating Systems. It natively suports JavaScript and HTML APIs 5 (including WebRTC) as programming language, and a robust privilege model to communicate directly with cellphone hardware, and application marketplace.

Three WebRTC based Media Server solutions were evaluated. Jitsi Videobridge supports Selective Forwarding Unit (SFU) for multiuser video communication and it is based on XMPP architecture. Kurento, supports MCU/SFU Star topologies and a modular architecture to implement media processing services. Janus Gateway is a flexible and modular WebRTC gateway that can be used to deploy a full-fledged WebRTC gateway on a cloud provider or just a small nettop/box to handle a specific use case, looking at applications as pluggable modules that a client can connect to through this gateway. These solutions, are good candidates to support server side Hyperties providing media related services.

## Messaging

The Messaging Services, as it appears in the architecture, is the server side platform that will support several functions provided by the Service provider. In order to evaluate the options to implement the messaging service, different existing solutions have been considered: Matrix, MQTT, Nodejs, Psyc, RabbitMQ, realtime backends (also knwon as noBackends or Backend-as-a-Service), Redis, Vertx, XMPP and ZeroMQ.

The following criteria are seen as particularly important for the choice of a solution for messaging node implementation: - it should support Protocol on-the-fly, to inter-operate with other Messaging Nodes or Back-end servers without having the need to standardize the protocol to be used. - it should support different Encrypted Messaging Transport Protocols including: Encrypted WebSockets, HTTPS Streaming, HTTPS Long-Polling and HTTPS REST - It must be possible to cache submitted messages - it has to support logging of routed messages and any other event (e.g. connection events) in remote log servers - it must support message delivery reliability. Delivery errors must be returned to clients - If required, Messaging Node must support worldwide scale deployments - it should be tolerant to unstable connections (e.g. short disconnections) - It should be possible to get events with information about Messaging node client’s connection and disconnection. Such feature is useful for connection status purposes. - Messaging Node must support very low message delivery latency - Messaging Node must be deployable in the most used Virtual Machines - it should require minimal computing resources in order to be deployable in constrained computing environments - Messaging Node must support external authentication and Authorisation - Messaging Node must support multiple message oriented communication patterns including: pub/sub, broadcast, one to one.

### Matrix

The end goal of Matrix is to be a ubiquitous messaging layer for synchronising arbitrary data between sets of people, devices and services. Matrix doesn’t support external authentication and authorisation. It also needs to adapt support of messaging transport protocols by wrapping Event/messages in REST messages.

### MQ Telemetry Transport

MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. As it fulfils all the criteria defined above MQTT is a potential candidate for messaging node.

### Node.js®

Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Nodejs doesn’t support pub/sub by itself, but it can if it is associated with another Pub/Sub mechanism (e.g. Redis).

### PSYC

PSYC is a mostly text-based protocol, aiming at providing a decentralized global messaging infrastructure for unicast/multicast chatting and social media exchanging. Its goal is to replace the popular IRC protocol. There is no evidence in the documentation that PSYC is able to accept external authentication/authorisation methods other than its own one. Moreover, a certain degree of latency is inevitable, due to the use of TLS and DoS techniques.

### RabbitMQ

RabbitMQ is defined as a robust and easy to use messaging platform that can work synchronously an asynchronously. It partially supports Messaging transport protocols but should not be tolerant to unstable connections.

### Realtime backends

Realtime backends (aka noBackend or BackendAsAService(BaaS)) is a concept related to real time databases. The backend and its remote framework is taking into account all low level mechanism of client-server dialogue, allowing developer to concentrate in service logic, in its local runtime. The realtime backend concept would allow defining and managing interworking with other services, in a way that entirely belongs to each application.

### Redis

Redis is an open source advanced key-value cache and store. Its weaknesses are that Redis has no logging features; they need to be implemented externally. Moreover, there is no indication that Redis is tolerant to unstable connections.

### Vert.x

Vert.x is an application framework providing possibilities to develop loosely coupled network service applications.

### XMPP

The Extensible Messaging and Presence Protocol (XMPP) is an open technology for real-time communication, which powers a wide range of applications including instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.

### ZeroMQ

ZeroMQ is a high-performance, low level, asynchronous messaging library originally written in C++, that now has multiple native Implementations. It is used as a thin layer between the application and transport layers.

### Selected Real time Messaging Solutions

In the scope of reThink framework, Matrix, Nodejs and vert.X have been selected to implement the messaging node.

## Service Frameworks

Objective of the Service Framework is to develop a JavaScript Framework of libraries that can be used to facilitate the development of Hyperties. This framework will compliment the features provided by the Hyperty Runtime (T3.4) and Network QoS Policy Enforcement (T3.3). The end results of the Service Framework should support Hyperty Development (T5.2) which further assist in the implmentation of T5.3 Conversational Services (audio, video, chat, screen sharing) and T5.4 Context Enabled Services (Conversational Services, IoT, context delivery, location).

An analysis of existing JavaScript frameworks based on the reTHINK Service Framework requirements was carried out on some of the popularly used frameworks today. These frameworks all endeavor to facilitate the development of web applications utilizing the Model-View-Control design pattern. For the reTHINK project however, focus was on the data model management and routing capabilities of these frameworks.

Even though AngularJS is the most popular framework, which provides great two way data-binding allowing for synchronization of data, and has a large community base, it was not considered a suitable applicant for this project due to its complex directives API and inflexibility on configuring (i.e. it offers no configuration possibilities after the Bootstrap procedure). Another reason is that Angular is more suitable for Single Page Apps (SPA) unlike a dynamic environment like the reTHINK runtime where multiple applications can be downloaded and executed concurrently.

Another framework analyzed was BackboneJS which also did not fit into the reTHINK service framework requirements due to the lack of a modular structure. Backbone lacks a controller concept and views and Models are relatively tightly coupled, resulting to tightly coupled modules which are not desirable for the reTHINK project.

StapesJS another framework analyzed offered a lightweight less complex framework especially suitable of mobile platforms. However it in itself offers very little APIs and demands combination with other libraries such as JQuery, React and Rivets.

MeteorJS on the other hand is a good applicant. MeteorJS offers rapid prototyping and produces cross-platform code for mobile and fixed platforms. It offers the distinct advantage to be used on all devices and operating systems featuring the Hyperty Runtime. However it has very strong dependency to the back end server being NodeJS. What this means for the reTHINK project is it will fit in perfectly, if the tool of choice for the Messaging Node where NodeJS. This is compatible with the other components as NodeJS is one of the tools considered for the reTHINK Messaging Node.

From the above anaöyzed frameworks, there is no strong conclusive statement which one is best to be used to develope the Service Framework as they all have advantages and disadvantages. To fulfill the objectives of these task, the approach has to be more specific as to what the above frameworks have to offer. We will identify the main requirements from the selected use cases specified in WP1, identify the data objects and interfaces defined in WP2 and provide JavaScript libraries in form of SDK to the developers. This SDK will include utility functions, Message factories for creating common data objects, templates for defined Hyperty types and other high level APIs from the underlying runtime and policy enforcement APIs.

The executable Hyperty Runtime will be the basis of all application development. With a middle layer of the Service Framework offering bulding blocks to choose from, a new ecosystem is formed on top of which other frameworks and applications can exist.

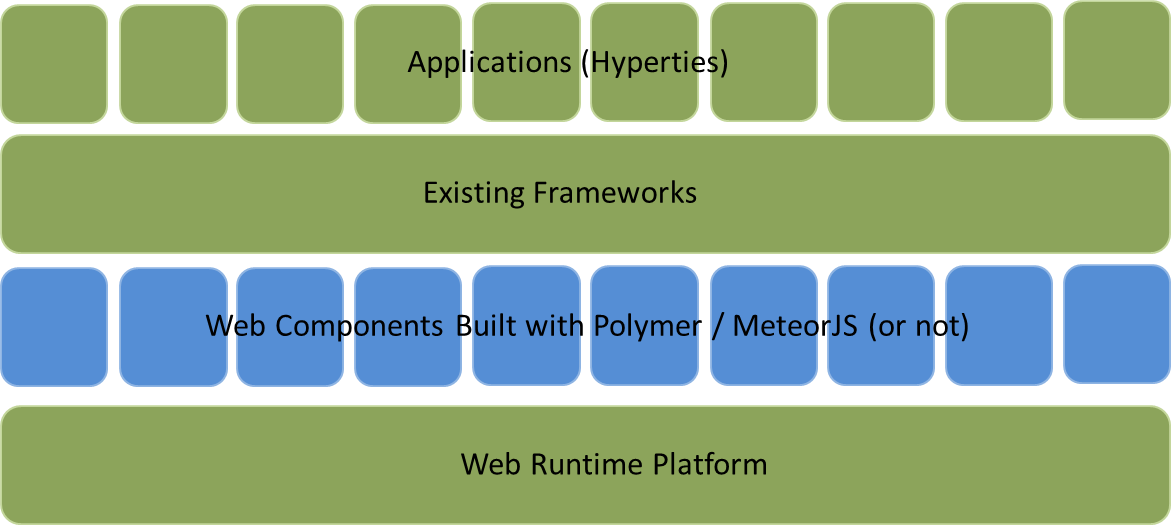


Figure 9: Service framework middle layer

## Projects

The WONDER project has enlightened some foundations paths to be followed in a post-IMS era dominated by Web technologies that reTHINK is currently exploiting. Notably, the novel Signalling On-the-fly (SigOfly) concept was conceived and successfully demonstrated to enable seamless interoperability between different WebRTC service domains. reTHINK Protocol On-the-fly concept extends WONDER, the Signalling On-the-fly concept to any other service domain where needed protocol stacks can be executed in a Web Runtime.

The WONDER Library used to validate SigOfly concept can be used in reTHINK as a good starting point to design and implement reTHINK runtime APIs and reTHINK Javascript framework.

## Products

### ApiRTC

#### What is ApiRTC?

ApiRTC is the communication platform developped by Apizee. This includes a communication platform and a client JavaScript library that can be used by developpers to developped their own applications without having to consider the technical aspects of communication. Complete version of ApiRTC with tutorials is described on www.apirtc.com

#### Features Overview

ApiRTC Entreprise edition includes following features :

**Session :**

Connexion : long polling , webSocket; HTTP, HTTPS; Presence : group connection and subscription; Custom User Data sharing ; Browsers type and version detection

**IMClient :**

Instant Messaging : 1 to 1, Group

**WebRTC Client :**

* Voice Calls, Voice and Video Calls
* Audio, video mute
* ScreenSharing
* TakeSnapshot
* Support of IE and Safari for audio and video calls through a plugin
* Network disconnection detection
* Network traversal management for media flows
* DataChannel
* Calls recording
* Connection to IMS, RCS, SIP Architecture
* Conference calls

**Data Client :**

Custom data sending and reception

**Compatibility :**

Window, linux, OSx, Android devices through WebRTC compatible browsers Plugin for Android and iOS application development

#### Architecture Overview

ApiRTC solution use different components on server and client side.

**Messaging Node :**

On server side, main used components are NodeJs and Redis :

NodeJs : https://nodejs.org/ - Description is available : http://en.wikipedia.org/wiki/Node.js

NodeJs is a Javascript engine that can be enhanced through diffrent existing modules for connections, log, ...

Redis : http://redis.io/ - Description is available : http://en.wikipedia.org/wiki/Redis

Redis is a NoSQL database that is really interesting for real time data and that provide a publish/subscribe that can be used to establish communication between several nodeJs process.

**Runtime / Framework :**

ApiRTC use a javascript library on client side to provide teh developers APIs that enables teh developpesr to use platform feature.

#### Architecture

ApiRTC actual architecture is presented on following diagram :

Components such as NodeJs, Redis or socket.io are used. ApiRTC uses JSON over WebSocket to manage signalling between clients and server.

#### APIs

ApiRTC provides API for developers : complete set of APIs is describe on http://apirtc.com/api-docs/

APIS are decomposed with main following classes : \* ApiRTCSession : manage user connection to the platform (presence) \* ApiRTCWebRTCClient : manage WebRTC feature : call, dataChannel ... \* ApiRTCIMClient : manage Instant messaging feature \* ApiRTCDataClient: : manage data sending feature \* ApiRTCWhiteBoardClient : manage Whiteboard feature

#### Requirements Analysis

Analysis regarding WP3 Messaging node requirements :

**Messaging Node with carrier grade deployment features :** NodeJs and Redis enables to buld a resiliante and scalable architecture

**The Messaging Node MUST offer DoS and DDoS Protection :** User authentication, message rate limitation are example of feature taht may be implemented to fulfill this requirement

**It should be possible to support Protocol on-the-fly :**

ProtOFly connector can be developped. JS connector can be develop on top of NodeJs to enable protofly on server side. This connector will be for example reusable to connect an external CSP, Kurento Media Server, or the Identity manager

**Messaging Transport Protocols:**

Socket.io enables the usage of different transport protocol to establish connection between user and server. (Long polling, WebSocket ...)

**Messaging Node logging :**

Several logging modules are available : log4js, winston, bunyan ... Logs can be dispalyed in console, store in file with log rotate, send to a network entity ...

**Message delivery reliability :** Socket.io enables message acknowledgement

**Messaging Node deployments with carrier grade scalability :**

Using Redis cluster mode : it is possible to use Redis Cluster with PUB/SUB mechanism : several NodeJs entities can be connected through the redis cluster : this can enable load balancing, redundancy

**Messaging Node should be tolerant to unstable connections :**

Socket.io can manage reconnection with different configurable parameters (timeout, retries ...) reconnection whether to reconnect automatically (true)

reconnectionDelay how long to wait before attempting a new reconnection (1000) reconnectionDelayMax maximum amount of time to wait between reconnections (5000). Each attempt increases the reconnection by the amount specified by reconnectionDelay. timeout connection timeout before a connect\_error and connect\_timeout events are emitted (20000)

**Events about clients connection / disconnection from Messaging Node :**

Using socket.io different events are fired on connection status : connect. Fired upon connecting. error. Fired upon a connection error disconnect. Fired upon a disconnection. reconnect. Fired upon a successful reconnection. reconnect\_attempt. Fired upon an attempt to reconnect. reconnecting. Fired upon an attempt to reconnect. reconnect\_error. Fired upon a reconnection attempt error. reconnect\_failed. Fired when couldn’t reconnect within reconnectionAttempts

**Messaging Node must support very low message delivery latency :**

**Messaging Node must be deployable in the most used Virtual Machines :** NodeJs is available on Linux, windows, mac and can be deployed on small virtual machine or devices

**Messaging Node should require minimal computing resources :** Messaging nodes components can be isntalled in only one VM

**Messaging Node must support external authentication and Authorisation :** Module like Passport : http://passportjs.org/ enables to use external authentication like facebook, twitter, google .. (We will have to check if passport can be used as it seems to require Express which may not be relevant in rethink case)

**Messaging Node must support multiple messaging functionalities :** Several routing can be performed with socket.io. Send message to only one dest, broadcast message to several users

#### Integration in Rethink

ApiRTC can be used in a nodejs based Messaging Node.

Integration of ApiRTC in Rethink can be done by adding differents connectors depending of needs : - Identity Management : connector to Identity server - QoS Management : connector to QoS server - Other Web communication platform : connector to communication platform using ProtOFly - VoIP Platform : Connector to WebRTC GW - Connector to Media Servers

A Redis Cluster with Pub/Sub mechanism can be used to manage communications between connectors

For Rethink, Apizee propose the usage of apiRTC, for instance to simulate an external CSP connection.

### State of the art of current WebRTC solutions of Quobis

#### What is Sippo?

Sippo is the name of a WebRTC product family authored by Quobis which includes the following products: - Sippo WebRTC Application Controller: the server which provides the services. - Sippo WebRTC Apps: reference web applications which leverage the main features provided why the WAC. Two examples: - Sippo WebCollaborator: Reference enterprise WebRTC softphone - Sippo Click To Call: Reference customer contact WebRTC softphone

#### What is a “WebRTC Application Controller”?

Sippo WebRTC Application Controller (WAC, in short) is a solution that allows to deploy WebRTC applications fully-interconnected with existing services (AAA, OSS, BSS, etc.) and legacy VoIP or UC systems.

Sippo WAC supports a number of business cases, through its APIs, ranging from a simple click-to-dial button to advanced scenarios like RCS-based services, integration with existing Web Portals (including Facebook, Twitter or GMail), Banking, Health, Logistics, call centers/CRMs, UC, etc.

Sippo is standards compliant and has been designed and developed by engineers who participate in WebRTC standardization forums like W3C, IETF, 3GPP, SIPForum and GSMA. Thanks to its abstraction layer, Sippo can include new signaling modules rapidly or even use different signaling protocols within the same application (e.g. one signaling protocol for audio/video, another for IM/presence, etc.).

Sippo WAC is the right tool to develop, adapt or deploy any WebRTC tool in a SDN, in the case of telcos, or corporate architecture, with the security that it is going to be interoperable with the existing services and WebRTC gateways. In addition it provides features to manage user provisioning, store call detail records and provides contextual information

Sippo has been developed by Quobis and it's distributed worldwide through a network of first-class partners and UC vendors.

#### Reference architecture

Sippo WAC is a network component which sits on the edge of the network, in close collaboration with the WebRTC gateway. The following picture describes how the WAC fits into a service provider or enterprise voice network:

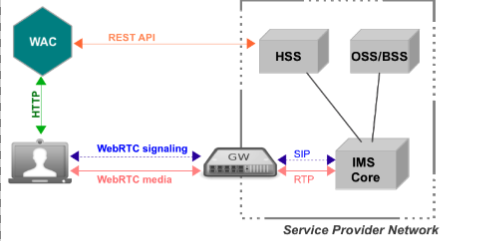


Figure 10: Sippo WAC reference architecture

The following network elements are the basic ones to understand the reference architecture (from right to left):

* Service Provider Network: this block represents the existing UC platform owned by the enterprise (where we might find a corporate PBX) or service provider (where we might find an IMS core or a Class 4/5 softswitch). In the latter case we will also find OSS/BSS systems and other identity management platforms that interact with Sippo in some way.
* Third-party WebRTC gateway: in some cases where the UC core does not support WebRTC traffic, there is a need for a WebRTC gateway which takes care of the translation of both the signaling and media plane. Signaling can be standard based (like SIPoWS) or a vendor-specific signaling protocol. The WebRTC gateway can be a standalone network element or it can be a functionality embedded into an existing network element like a SBC or an application server. Sippo excels in interoperability with leading gateway vendors thanks to its award-winning abstraction layer, please consult your sales manager for a complete list of supported vendors.
* WebRTC Application Controller (WAC): this is the network element where the WebRTC applications are deployed and managed. Applications are downloaded to the browser from the WAC vía HTTP, while the actual media and signaling traffic goes to the customer network through the WebRTC gateway. Sippo runs on a dedicated server which can be installed at the customer premises or in the cloud.
* Web browsers: the WebRTC applications are downloaded into the web browser after the user has been authenticated. From the point of view of the end-user, this is the only application that he/she will need to use. Sippo applications needs to have HTTP connectivity with the WAC and with the WebRTC gateway.

In a real deployment there are a number of additional network elements involved such a Session Border Controller, firewalls, STUN/TURN servers, SIP routers, etc… which will interact in some way with the WebRTC services and applications.

#### Understanding the role of a WebRTC Application Controller

The term “WebRTC Application Controller” has been coined by Quobis after our experience deploying WebRTC projects in large service providers all around the world. In a real setup, there are a number of features that are not meant to be provided by the service provider network, the WebRTC gateway or the browser (for example, authentication, identity management or security).

Sippo brings to the market a rich set of features which speeds up the deployment of WebRTC into existing networks, as for example:

* Multi signaling mechanisms
* SIP over WebSockets (RFC 7118)
* JSON-based APIs
* REST-based APIs
* Identity Management
* User provisioning
* Security Control
* Policy Control
* Statistics and logging
* Address book synchronization
* Browser abstraction layer

Besides those features, Sippo provides sippo.js, a ORCA.js (http://www.orcajs.org) compatible API for application developers hiding all the complexities of WebRTC signaling and media, hence enabling applications to be developed once and run in different devices, browsers and network environments.

Along with Sippo, Quobis has developed a number of WebRTC applications for specific verticals such as the Sippo Web Collaborator, Sippo Click to Call or Sippo GMail Toolbar.

#### Sippo interfaces and API’s

Sippo offers a set of different API’s and service interfaces that are summarised in the picture below:

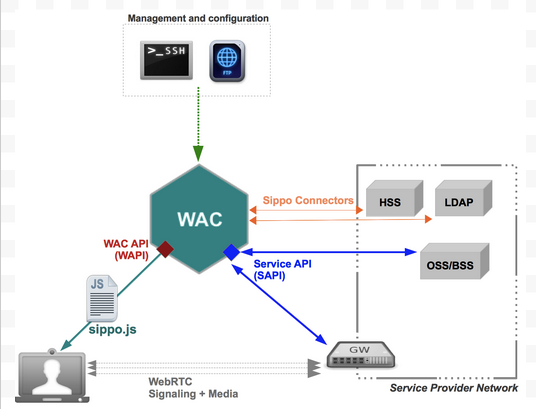


Figure 11: Sippo interfaces and APIs

##### Sippo.js API

Sippo.js is a Javascript API that is downloaded to the user’s browsers, thus containing all the signaling stacks and WebRTC media API calls. Sippo applications are built on top of this sippo.js API and it can also be used by third-party developers to code their own client applications.

Sippo.js API supports a complete set of signaling stacks, including both standards-based (like SIPoWS, authored by Quobis at RFC7118) and vendor-specific ones. That means that the applications built on top of the Sippo.js API are capable of communicating with different gateways from different vendors without changing the code. That’s one of the benefit of using Sippo.js API as it hides the complexity of the underlying signaling plane and provides a single and simple-to-use javascript API to the applications.

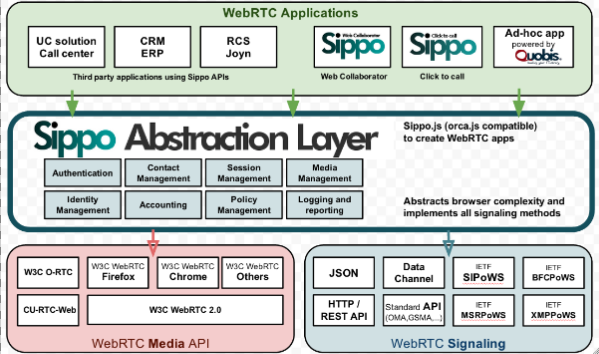


Figure 12: Sippo.js abstraction layer

##### Sippo Service API (SAPI)

Sippo Service API (SAPI) is a REST API which allows to connect Sippo WAC to different elements from the operator’s core and access network. This API can play both client-role and server-role to integrate the Sippo WAC and the WebRTC applications into the core.

SAPI is used in server-role between the WebRTC gateway and the WAC. It can be used for Identity Management (IdM) checks as part of the authentication process and check the permission set of the subscriber. When a some requests reach the WebRTC Gateway from a WebRTC Application, the gateway in turn verify the identity of the subscriber using the WebRTC application by sending an IdM request to the WAC through the SAPI.

##### Sippo connectors

Some of the Sippo features requires to connect to external services or to behave as a server to third party platforms. Some of those features are exposed to the sippo.js API while others are internal to the Sippo core.

Sippo connectors available so far in this version are:

* LDAP connector: Sippo can synchronize with an external LDAP server to retrieve contact lists, phone numbers and related information.
* Vendor-specific connectors: Sippo provides specific connectors for some features provided by the gateway vendors. The details of each connector is described in the joint application notes issued by Quobis and each vendor, please contact your sales representative for more information. The configuration of these connector is described in annex documents to this guide.

##### Sippo WebRTC API (WAPI)

This is an internal API offered by the Sippo WAC to the client applications, and it’s not intended to be used by third parties. This API basically interchanges messages between the application and the WAC using WebSockets (JSONoWS) or HTTP.

#### Sippo internals: services and backends

This explains some basic concepts of the Sippo architecture, in order to understand how to properly configure the controller and all the services provided and also how the different sippo.js API calls are interpreted and managed from the WAC depending on the configuration.

There are two key concepts to understand the internal architecture of Sippo: services and backends. A service is a functionality provided by the WAC, whereas a backend is a implementation of a specific service. In other words, we can say that a services is “what” to do and the backend is “how” to do it.

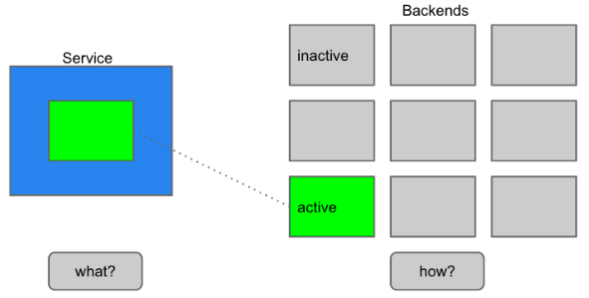


Figure 13: Sippo services and backends

There are thirty-three available services at Sippo WAC that are listed alphabetically in the table below. Some of those services have a 1-to-1 implementation at sippo.js API calls while others are internals and not exposed to the end user, but are explained here for completeness.

#### 1.7. Sippo WebRTC applications

Services providers and enterprises can deploy their own WebRTC applications using Sippo WAC, developed by using the existing Sippo Javascript API sippo.js (which includes the Sippo Abstraction Layer) and also making use of all the Sippo services like authentication, contacts, etc…

Every Sippo application needs to run connected to a Sippo WAC, as some of the features are not implemented on the browser but on the WAC.At the current Sippo version, both the applications and the sippo.js libreries must be hosted and donwloaded from the WAC. This is mandatory on this current version. Please note that, in this scenario, some cross-domain issues may arise. Please contact Quobis system engineering department for more information on this topic.

Communication between the WebRTC applications running on the browser and the Sippo WAC is done by using the WAPI interface, which dispatches the incoming messages to the corresponding services, as shown in the picture below:



Figure 14: Sippo WebRTC applications stack

#### Potential integration with Wonder proposal

##### About signaling-on-the-fly

The WONDER Javascript Framework was designed and implemented to address the lack of a standard WebRTC signalling protocol by implementing the novel Signalling On-the-Fly concept, enabling seamless interoperability between different WebRTC Service Provider domains.

The WONDER library assumes there won’t be a standard WebRTC signaling protocol to give developers the freedom to select (or invent) the protocol that better suits WebRTC Application needs and, at the same time, standardization tasks effort are minimized, shortening innovation to market timing. This means, the message server and associated protocol stack can be selected, loaded and instantiated during runtime. Such characteristic enables signaling protocols selected per WebRTC Conversation to ensure full Signaling interoperability among peers using Triangle based Network topologies. Such mechanism we call Signalling on-the-fly.

##### Signaling-on-the-fly versus multi-signaling support

The Sippo WebRTC Application Controller tries to hide the complexity on vendors thanks to the support of different signaling stacks. This means that while a web client is making a request to the WAC to have access to a WebRTC application, the WAC adapts the JS code of the application to the type of gateway to use the signaling protocol that the gateway is supporting.

The Sippo WAC has a mechanism to deal with different gateways (including those from different vendors) in an active way, so high availability and scalability can be achieved with no need to use a load balancer for the gateways. It’s important to mention that he Sippo WAC does not manage real time traffic as this goes from the browser to the other browser (or to the gateway in case of interconnection with legacy networks).

In order to leverage the result and proposals of Wonder around signaling on the fly we can explore the possibility to move to the application (and browser) the complexity of selection the signaling for the call (now the abstraction layer is part of the WAC, as described in section 1.5) or try to adapt the Sippo WAC to manage the rehydration of signaling of the clients during a call or session.

The WAPI, as the API that interchanges messages between the application and the WAC using WebSockets (JSONoWS) or HTTP, can play an active role in both options to manage this approach.

#### Requirements Analysis

Sippo.js provides a high level abstraction layer which allow to build WebRTC applications in an easy and quick way. Sippo.js supports many signaling protocols for WebRTC and can be used with WebRTC gateways from many vendors. This is possible thanks to it implements a static-flavor of the protocol-of-the-fly approach used in reTHINK project. This was identitified in the early stages of WebRTC as a need to deal with the signaling diversity in the WebRTC arena. Sippo.js can be adapted to be an intermediate layer between the hyperty and the web application hidding all the innecesary complexity to te developer. This will also allow that all the applications already build over Sippo.js can be used in reTHINK reducing considerably the integration costs.

# Hyperty Runtime Specification

This Chapter contains the detailed specification of the Runtime, where Hyperties are executed. It describes in detail the Runtime Architecture and the Core Runtime components required to support the execution of Hyperties. The Hyperty Runtime architecture followed a Security by Design approach since it was highly influenced by a carefull security analysis also included in this chapter.

The APIs to be implemented by the Runtime components are specified in detail and they provide functionalities that were identified in an iterative approach. In such iterative approach, the design of the static view of the runtime APIs progressed along the design of the main procedures to be performed by the runtime.

The Runtime Main procedures are also described in detail in this chapter using UML Message Sequence Charts. They correspond to the dynamic view of the Hyperty Runtime and they validate the static design for the most important use cases that were already used in WP2 and originaly described in WP1.

Four main types of Runtime procedures are described:

1. Basic Runtime procedures are in general performed independently of the Hyperty or protocol stub executed in the runtime including procedures for the deployment of protocol stubs and hyperties, and procedures performed to route messages among Hyperties.
2. Identity Management Runtime procedures are the procedures performed to register and log in users in the domain, as well as procedures performed to associate identities to Hyperties and assert user identities.
3. Runtime Procedures to support Human to Human Communication with special focus on the validation of the Reporter-Observer communication pattern to WebRTC.
4. Runtime Procedures to support Machine to Machine Communication

It should be noted that the description of the main procedures also include the detailed definition of messages exchanged among Hyperties and protocol stubs, as defined in D2.2 Message Model, when appropriate.

At the end, some implementation considerations are presented for the different types of runtime platforms that are the target of this specification namely the browser runtime, standalone runtime applications and M2M devices with more constrained capabilities. These considerations are mainly about the implementation of the runtime sandboxing solution since all core runtime components will be shared among all platforms.

## Runtime Architecture

The main Hyperty runtime architecture is presented in fig. 15. It is comprised by different types of components that, for security reasons, are executed in isolated sandboxes. Thus, components downloaded from a specific Service Provider (e.g. Service Provider 1 from fig. 15) are executed in sandboxes that are different from the sandboxes used to execute components downloaded from another service provider (e.g. Service Provider 2 from fig. 15). In addition, for the same Service Provider, and also for security reasons, protocol stubs and Hyperties are isolated from each other and executed in different sandboxes. Communication between components running in different sandboxes are only possible through messages exchanged through a message bus functionality provided by the Core Sandbox. On the other hand, the protocol stub provides the bridge for the Hperty runtime to communicate with associated Service Provider. For example, in fig. 15, protostub1 is the only way that Hyperty instances have to communicate with Service Provider 1. In general, in the Core Sandbox, all required functionalities to support the deployment, execution and maintenance of components downloaded from service providers, are executed. Core components are, ideally, natively part of the device runtime. However, to support existing platforms including Browsers and Mobile Operating Systems, to minimise the need to install new applications, the existing device native runtime functionalities (e.g. Javascript engine) are distinguished from the hyperty core runtime functionalities. In such situations, the Hyperty Core Runtime components are downloaded from the Hyperty Runtime Service Provider and are executed in an isolated core sandbox.

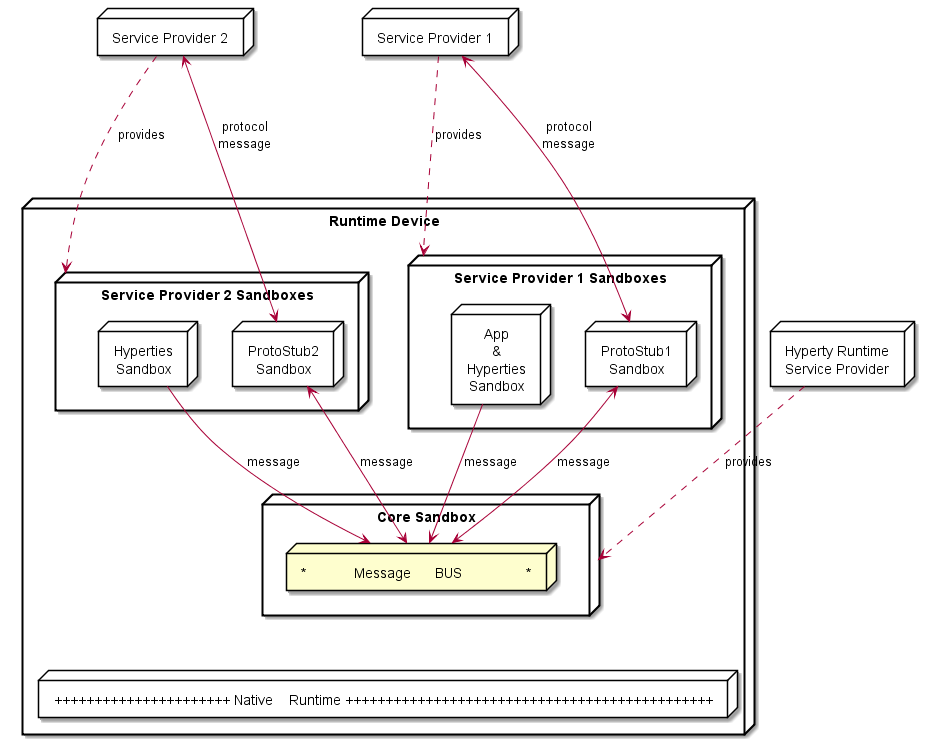


Figure 15 High Level Runtime Architecture with trusted Hyperties

In figure 15, the Application and the Hyperty Instances it consumes, are downloaded from the same Service Provider, and they trust each other, i.e. they are executed in the same sandbox. In figure 16, it is depicted the Runtime Architecture where the Application and the Hyperty Instances it consumes, don't trust each other, for example, they are downloaded from different service providers. In such situation, Hyperties and the Application are isolated from each other and they are executed in different sandboxes.

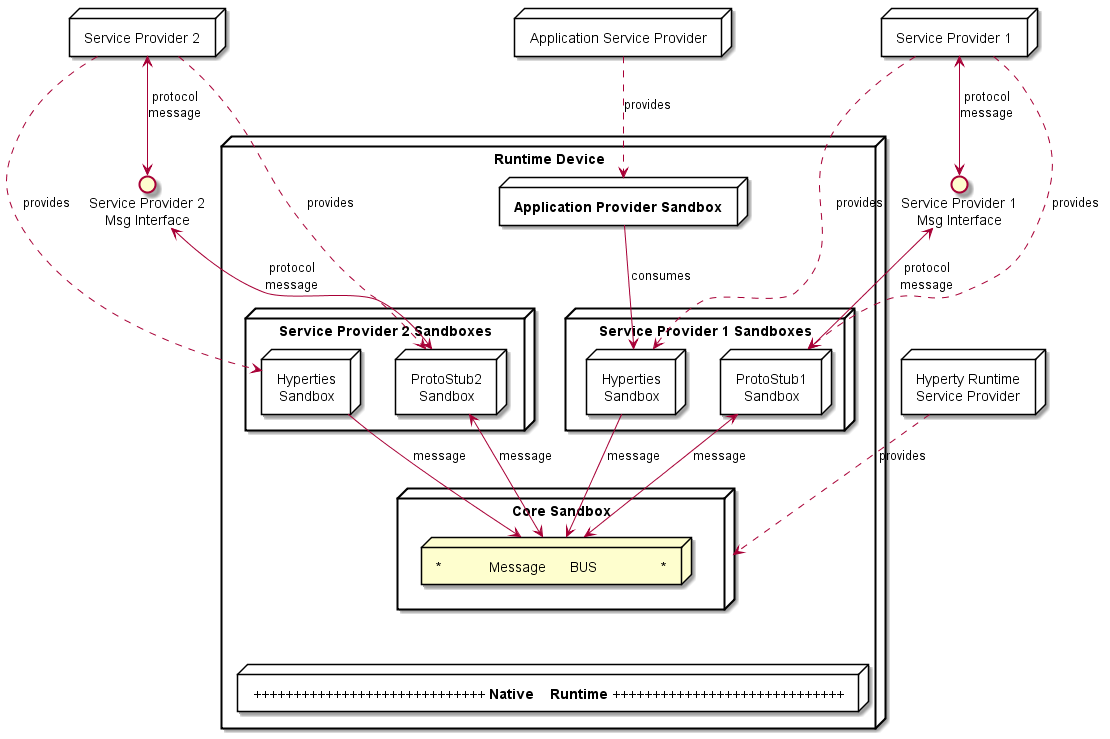


Figure 16 High Level Runtime Architecture with untrusted Hyperties

As described below, to prevent cross origin attacks / spy, access to Core Runtime Message BUS is subject to authorisation, by using standardised policies downloaded from each involved Service Provider. In addition, the Hyperty Runtime Architecture also supports the enforcement of Service Provider policies, with its own Policy Enforcer component executed in a dedicated sandbox (see fig. 17) enabling the enforcement of proprietary policies.

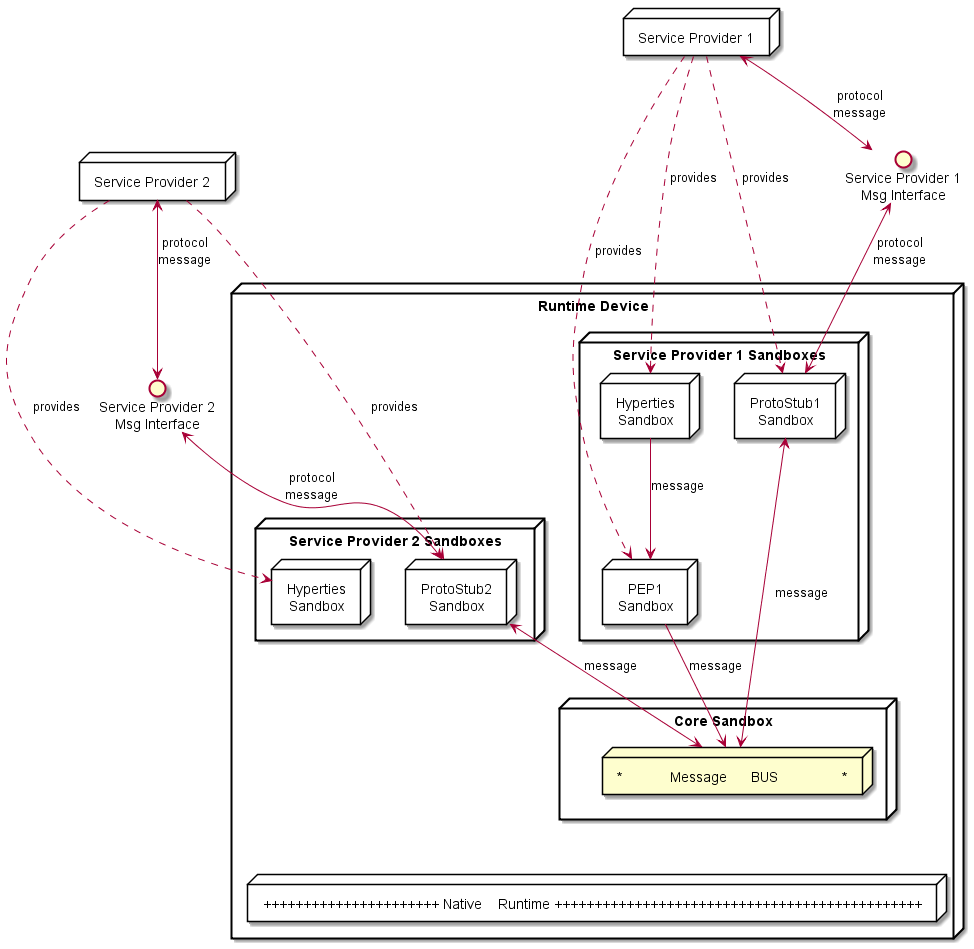


Figure 17 High Level Runtime Architecture with domain specific Policy Enforcer

The different types of policies to be applied on these different points, namely in the Message BUS, requires further research to avoid performance overhead and potential conflicts. In principle, if for a specific domain there is Policy Enforcer, it will not be needed to enforce policies from that domain in the Mesg BUS PEP.

In addition, Message BUS PEP should enforce general access control policies that are agnostic of sender and target domains, or specific to the domain managing the device runtime (Core Runtime Provider). The policies used to control the access to synchronised Data Objects used in Hyperty Communication (see below) , are a good example of such policies.

Some more details are provided in the following sections.

### Service Provider Sandboxes

#### Hyperty

As defined in [D2.2] Hyperties communicate through [data object synchronisation](https://github.com/reTHINK-project/architecture/blob/master/docs/datamodel/data-synch/readme.md) where different access control policies can be used. The Reporter-Observer pattern introduced in D2.2 will be evaluated in order to simplify the management of inconsistencies in such distributed data synchronisation communication model.

The main Reporter-Observer pattern principle is to only grant writing permissions to Object owner (creator). Such policy to control the access to synchronised object has to be enforced by the Message BUS Policy Enforcer the Hyperty Core Runtime to be able to enforce .

The following Terminology is used:

Observer hyperty is not allowed to change objects

Reporter hyperty creator of the object is allowed to change the object. Only one hyperty instance reporter per Synched object instance.

Such Model is depicted in figure 18. The Reporter-Observer pattern is supported by the exchange of messages between Reporter Syncher and Observer Syncher as defined in the reTHINK Message Model [D2.2].

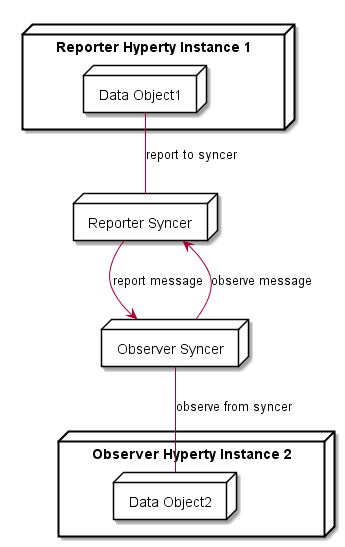


Figure 18 Reporter-Observer Communication Pattern

Additional, and more sophisticated and proprietary data synchronisation alghorithms can be used, by deploying a Policy Enforcer in the Runtime.

Hyperty Communication through data object synchronisation are provided by the Syncer component running in the Hyperty Sandbox. Data object synchronisation should take advantage on emerging [javascript Object.observer API](http://www.html5rocks.com/en/tutorials/es7/observe/).

#### Policy Enforcer

Policy Enforcer complements the Message BUS Policy Enforcer functionality enabling the enforcement of proprietary or closed Policies in the runtime for a specific Hyperty instance including access control policies to synchronised object.

#### Protocol Stub

The protocol stub implements a Protocol Stack to be used to communicate with Service Provider Backend Servers (including Messaging Server or other functionalities like IdM) according to Protocol on the Fly and codec on the fly concept as introduced in D2.2.

Protocol stubs are only reachable through the Message BUS. In this way it is ensured that all messages received and sent goes through the message bus where policies can be enforced and additional data can be added or changed including message addresses and identity tokens.

### Core Runtime

The Core Runtime components are depicted in fig. 19.

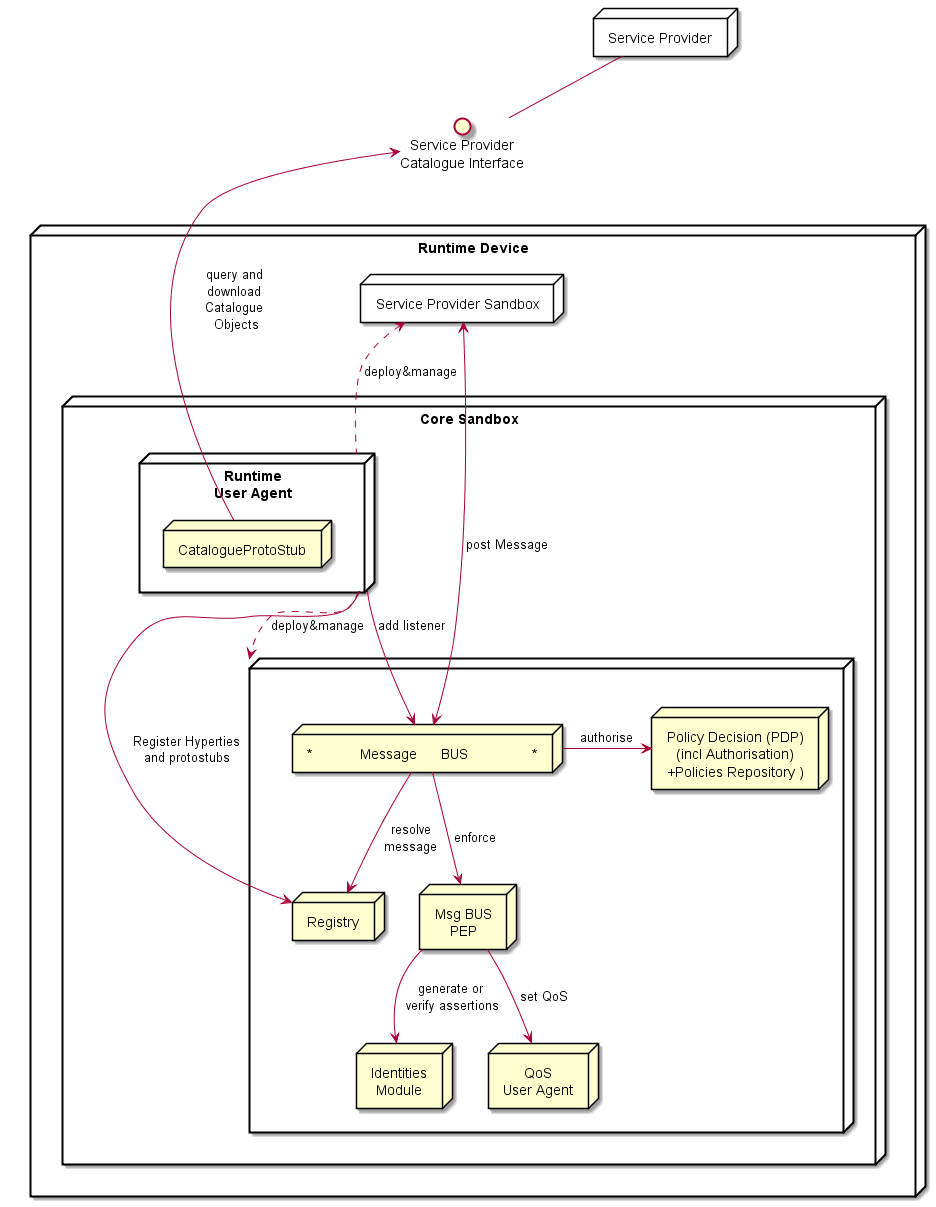


Figure 19 Runtime Core Architecture

Runtime Core components should be as much as possible independent on the Runtime type. They should be deployed once and executed at the background. The next time the runtime is started there should be no need to download the core runtime again unless there is a new version. Runtime core components instances should be shared by different Apps and Hyperty instances.

The Core Runtime is provided by a specific Service Provider (the Core Runtime Service Provider) that handles a Catalogue service to with Runtime Descriptors and a Registry service to handle the registration of Runtime instances.

#### Message BUS

Supports local message communication in a loosely coupled manner between Service Provider sandboxes including Hyperty Instances, Protocol Stubs and Policy Enforcers. Messages are routed to listeners previsouly added by the Runtime User Agent, to valid Runtime URL addresses handled by the Runtime Registry functionality.

Access to message BUS is subject to authorisation to prevent cross origin attacks / spy from malicious downloaded code including Hyperties, Protocol Stubs or Policy Enforcers.

#### Policy Decision Point and Message BUS authorisation

It provides Policy decision functionalities for incoming and outgoing messages from / to Service Provider sandboxes, according to Policies downloaded and stored locally when associated Hyperties are deployed by the Runtime User Agent. The possibility to consult Policies stored remotely should also be investigated. It also provides authorisation / access control to the Message BUS.

#### Message BUS Policy Enforcement Point

The Message BUS Policy Enforcement Point, is used by the Message BUS to enforce policy actions requested by the Message BUS Policy Decision Point e.g. to verify or generate identity assertions, to get valid Access tokens.

#### Runtime Registry

The Runtime Registry handles the registration of all available runtime components including Core components, Service Provider Sandboxes and each component executing in each sandbox like Hyperty Instances, Protocol Stubs, Policy Enforcers and Applications.

The Runtime Registry handles the allocation of Runtime URL addresses for all these components and manages its status.

In addition, the Runtime Registry should ensure synchronisation with Back-end Service Provider Registry.

#### Identity Module

The Runtime Identity Module manages ID and Access Tokens required to trustfuly manage Hyperty Instances communication including trustful association between Hyperty Instances with Users. In addition, it also supports the generation and validation of Identity assertions. Identity module is compliant with [WebRTC IdP Proxy](http://w3c.github.io/webrtc-pc/#identity) but not limited to WebRTC.

Messages routed by Message BUS should be signed with a token according to the Identity associated to it and managed by the Identity Module.

#### Runtime User Agent

The Runtime User Agent, manages Core Sandbox components including its download, deployment and update from Core Runtime Provider. It also handles Device bootstrap and the download, deployment and update of Service Provider sandboxes including Hyperties, Protocol Stubs and Policy Enforcers. It manages the descriptors of deployed components that are downloaded from the Service Provider Catalogue via the [Catalogue Service interface](https://github.com/reTHINK-project/architecture/blob/master/docs/interface-design/Interface-Design.md#73-catalogue-interface)[D2.2].

#### QoS User Agent

The QoS User Agent Manages network QoS in the runtime. This component requires further investigations which will be reported in D3.3.

### Native Runtime

The Native Runtime provides Functionalities that are natively provided by the runtime, e.g. Javascript engine or WebRTC Media Engine to support for Stream communication between Hyperties according to WebRTC Standards when available.

## Security analysis of the Hyperty Runtime

### Introduction

This document presents the security analysis of the Hyperty Runtime architecture [[1]](https://github.com/reTHINK-project/core-framework/blob/master/docs/specs/runtime/runtime-architecture.md).

The Hyperty Runtime depends on a trusted computing base (TCB) that consists of several components: the Core Sandbox, the Native Runtime, and underlying Operating System and hardware. Subverting the Core Sandbox components may result in (1) incorrect decision and enforcement of policies by the PDP, (2) failure in routing messages through the Message Bus, (3) flawed registration and discovery of Hyperty and protoStubs by the Registry, and (4) incorrect maintenance of identities by the Identities Container. If the Native Runtime is compromised, so it will be the support for WebRTC stream communication between hyperties. Since the Native Runtime implements the JavaScript engine (e.g., V8), tampering with the Native Runtime will undermine the execution of components implemented in JavaScript code, namely the components of the Core Sandbox (i.e., PDP, Message Bus, Registry, Identities Container, and WebRTC engine) and client code instances (i.e., Hyperty Instances, ProtoStubs, Service Provider Policy Enforcers (SPPEs), and Applications). Lastly, compromising the Operating System or the hardware may result in incorrect behavior of any of their overlying components, in particular the Native Runtime.

Next, we analyze the security properties of our system assuming that the trusted computing base is intact. Then, we assess the security vulnerabilities of the Hyperty Runtime when deployed on platforms featuring specific software and hardware configuration. In particular, we explore three platform configurations: *browser*, *standalone*, and *M2M standalone application*. We analyze each target platform under its specific threat model.

### Mitigated threats assuming an intact TCB

When the TCB is intact, our architecture ensures the correct isolation of client JavaScript code (i.e., Hyperties, ProtoStubs, SPPEs, and Applications). Isolation is enforced both between different client code instances and between client code instances and the environment (e.g., external applications, or OS resources). The Hyperty Runtime enforces access control decisions based on policy rules attached to Hyperty code. Such policies can regulate different aspects of the behavior of a Hyperty: access to local resources (e.g., cookies, files, network, etc), routing, charging, and privacy restrictions. The system also ensures the authenticity of client code and the identity of the involved entities.

In the basic threat model, we assume that an attacker can serve arbitrary client code to the Hyperty Runtime. The attacker can impersonate a legitimate service provider and deliver malicious ProtoStub, Hyperty, or SPPE code. When instantiated on the Hyperty Runtime, this code may attempt to execute JavaScript instructions in order to access private data held (1) by other client code (including applications’), (2) by the Hyperty Runtime TCB, or (3) by the surrounding environment (e.g., the JavaScript Engine, or the Operating System). Malicious code may also aim to tamper with security-critical components, such as Hyperty policies or the policy enforcement engine, in order to escalate privileges. Finally, malicious code may launch denial of service attacks (e.g., by executing CPU intensive code). Below in this document, we expand on this threat model to consider potential vulnerabilities of our system when deployed on different environments. Next, we describe how our system defends against several classes of potential attacks.

#### T1: Unauthorized access by client code

The basic mechanism of our architecture to prevent unauthorized access by client code is sandboxing. Each Hyperty instance running in the system runs in its own sandbox. A sandbox defines a security perimeter for the Hyperty instance, preventing it from reading or writing the memory (or other resources) allocated to other Hyperty instances or by other components in the surrounding environment. An independent sandbox hosts the ProtoStub instance required by local Hyperty instances to communicate with external services. This sandbox will prevent potentially malicious ProtoSub code from unauthorized access to resources. To communicate outside the sandboxes, the runtime provides well defined interfaces: the Syncer, which is used by the Hyperty instance to communicate with the SPPE, and an API to communicate with the Message Bus. The SPPE and the PEE are responsible for enforcing the policy associated with the Hyperty instance.

The origin of the client code is validated. An origin is a combination of URI scheme, hostname, and port number. The origin can be asserted using certificates (e.g. using TLS) thus we only allow client code from secure origin.

Client code is subject to Same Origin Policy for direct interactions between client code instances. However, this can be relaxed using Cross Origin Resource Sharing (CORS) policy declarations. Pieces of client code from different origins can still communicate without CORS using the Message Bus API. Message exchange must be identified by the origin of senders and recipients. Subscription to messaging channels (where multiple client codes could publish messages) must be subject to authorization.

Note that, in our architecture, sandboxing is also used to secure the components of the Hyperty Runtime that are implemented in JavaScript, namely the components allocated in the Core Sandbox. The JavaScript engine implements both the client code sandboxes and the Core Sandbox.

#### T2: Policy subversion

Every Hyperty instance is constrained by a policy. A policy defines a set of rules, which can be of several types: access control rules, routing rules, charging usage rules, and privacy rules. Altogether, a policy's rules are responsible for regulating, supervising, or restricting the operations that a Hyperty can perform, e.g., prevent access to a local file, enforce a predefined network route, or define the usage costs of a service. To prevent a malicious Hyperty instance (or ProtoSub) from subverting policy rules and escalate its privileges, the policy decision components and the policy repository are protected from the Hyperty instance by the Core Sandbox. As a result, policy integrity and enforcement are safe from malicious client code.

#### T3: Threats to client code authenticity

The authenticity of client code -- Application, Hyperty, ProtoStub, or SPPE -- can be compromised if at least one of two events has occurred without being detected before the code is loaded and instantiated into a sandbox: an attacker has modified the original code bytes (e.g., by embedding malware into a Hyperty code), or (ii) has modified the code identity. To prevent such attacks, client code's origin must be digitally signed and transmitted over a secure channel. Additionally the client code may be signed by its manufacturer. By checking these signatures before instantiating the Hyperty, ProtoStub, or SPPE code on the sandboxes and assuming that the cryptographic primitives are correctly implemented, the Hyperty Runtime can guarantee the integrity and identity of the code.

#### T4: Denial of service attacks

A malicious Hyperty instance, ProtoStub, or SPEE implementation can launch denial of service attacks by holding to specific resources, e.g., hogging the CPU by sitting on an infinite loop, or flooding the network with bogus messages. The JavaScript engine featuring the Hyperty Runtime prevents such attacks by placing a limit to the maximum utilization of a given service by a client code instance, for example by bounding the CPU cycles that a Hyperty instance is allowed to execute uninterrupted.

### Vulnerability assessment of the Hyperty Runtime

The Hyperty Runtime can effectively thwart the threats described in the previous section so long as the system's TCB remains intact. However, when deployed on a specific platform, the Hyperty Runtime may become vulnerable to some environment-specific security risks. In this section, we study the potential vulnerabilities of the TCB when deployed on three different platforms. But first, we describe our methodology to ensure a uniform vulnerability assessment of our system across platforms.

#### Methodology

Our basic methodology is based on a *vulnerability matrix*. A vulnerability matrix indicates representative practical attacks that can be carried out against the TCB on a given platform as a mean to compromising the security of the Hyperty Runtime. An attack is successful by achieving one or more goals described in the section above: permit unauthorized access by client code (T1), subvert Hyperty policies (T2), compromise the authenticity of client code (T3), and launch denial of service attacks (T4). We classify the attacks to the TCB along two dimensions: (1) the layer of the computer stack where the attack is directed to (e.g., the operating system), and (2) the difficulty level of the attack based on the technical skills and resources required by the adversary.

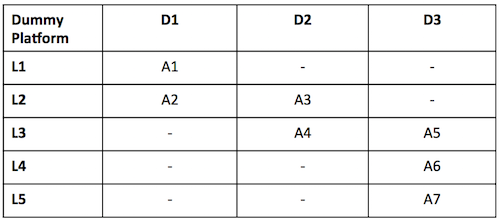


Figure 20: Vulnerability matrix for a dummy platform

The figure above provides an example of a vulnerability matrix for a dummy platform. The content of each cell describes examples of attacks that can be launched to the TCB, e.g., "A1: inspection of JavaScript code through the browser", "A7: probing the system bus". Columns represent the difficulty level and rows the attack layer (both of them will be explained below). Intuitively, the vulnerability matrix allow us to grasp how exposed the TCB is to attacks: the lower the difficulty degree of the attacks is the more vulnerable the Hyperty Runtime will be when deployed on a particular target platform. Next, we describe the classification for attack layers and difficulty levels:

**Attack layers.** Attack layers can be classified in five types, ordered top-down, from the highest to the lowest layer of the computer stack, as shown in the figure below:

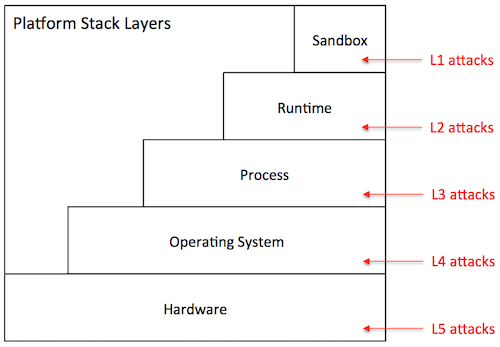


Figure 21: Stack

* *Sandbox level (L1)*: The attacker has direct access to the sandbox environment, hence to the code and execution state of Hyperty instances. For example, on a browser platform, users typically have access to the JavaScript of a given page. This means that a malicious user can leverage that mechanism to tamper with the JavaScript code of local Hyperty instances.
* *Runtime level (L2)*: The attacker has direct access to the code or execution state of the Hyperty Runtime. Depending on the specific exploit, he can mount attacks that disable defenses against the attacks described in the previous section. On the browser, for example, a L2 attack can be achieved by installing a malicious browser extension that bypasses the policy enforcement mechanism of the Hyperty Runtime.
* *Process level (L3)*: The attacker has access to the execution state of the process where the Hyperty Runtime is hosted. Just like the L2 attacks, this type of attack can result in catastrophic consequences. Examples of attacks performed at the process level include attaching a debugger to the Hyperty Runtime process and inspect its internal data structures, or dumping its memory state to disk by reading from /dev/mem.
* *Operating system level (L4)*: The adversary has access to the execution state of the operating system, and therefore to the execution state of the Hyperty Runtime. Similarly to L2 and L3, L4 attacks can be catastrophic. An attack performed at this layer, for example, installs a rootkit to mainain a log of all operations performed by local Hyperty instances.
* *Hardware level (L5)*: The adversary has physical access to the hardware of the platform and can launch simple attacks that do not involve tampering with the circuitry. Attacks in this category include, removal or inspection of the hard disk, probing the system bus in order to extract secrets from volatile memory, etc. An attack at this level may also include tampering with the silicon chips, perform side-channel attacks, etc. Such attacks require a high-level of expertise and committed resources. In theory, attacks performed at this level can reveal the entirety of the system state, including the operating system's. In practice, however, such attacks are more directed towards the extractions of specific secrets when L3 attacks or above are not possible.

**Difficulty level.** The difficulty level of an attack depends on several factors: the privileges owned by the adversary (e.g., user or superuser), the skills that are required (e.g., know how to run a debugger or tamper with silicon), and the necessary resources to carry out the attack (e.g., specific software exploits, memory probes, etc.). Based on these factors, we define three difficulty levels for a given attack:

* *Easy (D1)*: The attack is easy to perform. The tools that are necessary to launch the attack are accessible, well documented, and simple to handle. Some examples of D0 attacks include: (i) on a browser platform, a malicious user leverages the browser interface to modify Hyperty code, (ii) on a constrained device, the device owner abuses superuser privileges to disable the policy enforcement mechanisms of the Hyperty Runtime.
* *Medium (D2)*: The attack requires considerable skills and / or resources. It can be launched by mastering the tools presently available in the system (e.g., tools provided by the operating system, debuggers) or by installing new ones that can be found on the Internet (including malware or exploits). The attacker has limited skills or resources to discover new vulnerabilities or to develop exploits autonomously. Examples of such attacks include, attaching debuggers to extract in-memory secrets from the Hyperty Runtime, patch the Hyperty Runtime using exploit code published on the Web, etc.
* *Hard (D3)*: The attack is very sophisticated. To mount the attack, the attacker must be able to develop its own exploit code, find new vulnerabilities in the system, and / or launch software hardware attacks. For example, finding bugs in a device driver’s code and write software exploits. The attacks performed at the deep hardware level are also considered hard to execute.

When drawing a vulnerability matrix, we define *attacker profiles*, which define sets of possible attacks that characterize possible attack agents in that particular platform. For example, for the browser platform, we define three profiles: regular user, advanced user, and power user. The regular user captures an average web user, which is able to launch attacks like "inspection of JavaScript code through the browser", but not "probing the system bus". We now present our vulnerability assessment for each of the target platforms.

#### Browser platform

The primary platform targeted by reTHINK is the browser. Browsers can be highly heterogeneous; we may be talking about desktops, laptops, or mobile devices featuring many different configurations with respect to: hardware architecture, operating system in use, installed software, and specific browser version and extensions. In spite of this diversity, a Hyperty-enabled browser will tend to follow the general architecture represented in the figure below.

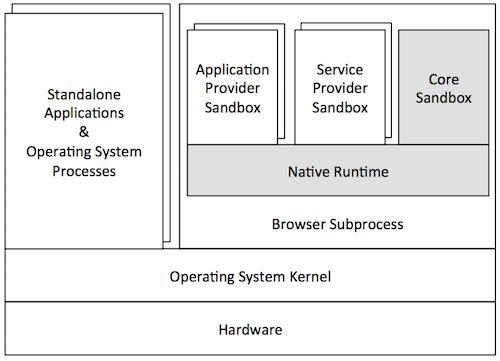


Figure 22: Browser

In this architecture, the Hyperty Runtime (represented by the shaded components of the Figure) is deployed on an independent browser process. This process is in fact a "subprocess" of the browser that implements a sandboxing mechanism of its own (as in the Chrome browser). This mechanism is responsible for isolating the Hyperty Runtime from the browser's rendering engine. The JavaScript engine is responsible for the secure execution of JavaScript code inside individual sandboxes: (1) the Core Sandbox of the Hyperty Runtime, (2) service provider sandboxes for hosting Hyperty instances, ProtoStubs and SPPEs, and (3) application provider sandboxes for executing guest applications. As expected, the Hyperty Runtime process depends on the operating system, which in turn depends on the underlying hardware setup. Browser processes run side-by-side with other standalone application processes and operating system services.

From the security point of view, the threats to the TCB of the Hyperty Runtime are mainly caused by an adversarial user. To better characterize these threats, we define three attacker profiles and draw the vulnerability matrix as follows:

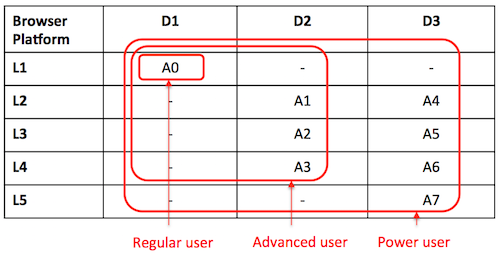


Figure 23: Security Browser

* *Regular user*: This attacker profile captures the class of users with an average proficiency level in computing, but are willing to subvert the security properties of the system's TCB. The user's privileges allow for limited operations, such as: launch the browser, and run Hyperty-based applications. A regular user is expected to mount the following attacks:
* *A0*: Access and modify client JavaScript code through the browser interface.
* *Advanced user*: This profile captures users with superuser privileges and some degree of skills and knowledge of the system. The user is aware of existing tools and techniques that can be leveraged to hack into the system's components, has access to exploits available on the Internet, and can handle auxiliary tools (e.g., debuggers, Unix advanced commands, etc.). The user can assemble and disassemble the basic hardware components of the system (e.g., plugging in / out the hard disk). For mobile devices, the user can root or jailbreak the platform by following instructions. Thus, considering this set of skills, in addition to A0, an advanced user can perform several other attacks at different stack layers such as these:
* *A1*: Compromise the runtime by installing a malicious browser extension.
* *A2*: Dump the memory contents of the process to disk.
* *A3*: Install a rootkit on the operating system that keeps track of Hyperty instances' communication.
* *Power user*: This profile corresponds to highly skilled users, who have deep knowledge of the system and can launch sophisticated attacks. A user is able to investigate unknown vulnerabilities in the software (including in the Hyperty Runtime or in the OS) and develop specific software exploits. Moreover, the user has enough resources and tools to launch hardware attacks that involve tampering with silicon. A power user is able to mount not only the attacks described previously, but more sophisticated attacks on various layers of the stack:
* *A4*: Find and exploit a bug in the Hyperty Runtime.
* *A5*: Attach a debugger to the browser’s subprocess and inspect / modify its memory.
* *A6*: Build a device driver to continuously monitor the execution of Hyperty Instances.
* *A7*: Probe the system bus and extract private key material of Hyperty Instances.

**Vulnerability assessment:** As illustrated by the vulnerability matrix, the browser platform is vulnerable to a range of attacks. Some of these attacks can be mounted by regular users with relative ease. In addition, there are several ways for advanced users to successfully compromise the TCB by exploiting the system at different stack layers. As a result, we recommend that browser platforms are avoided for hosting client code which the local user has incentives to subvert. Examples of such code include: Hyperty instances restricted by specific usage charging policies, ProtoStubs that encode proprietary communication protocols, or Applications that access copyrighted digital data.

#### Standalone platform

A variant of the browser platform is to deploy the Hyperty Runtime as a standalone application, for example to be executed as a mobile app on mobile devices such as smartphones or tablets. The Hyperty Runtime can also be packaged as a classical standalone application for desktop platforms running Linux or Windows. To allow for the development and maintenance of such applications, reTHINK will provide an SDK that will include APIs and platform specific libraries for adapting the Hyperty Runtime to the underlying operating system platform.

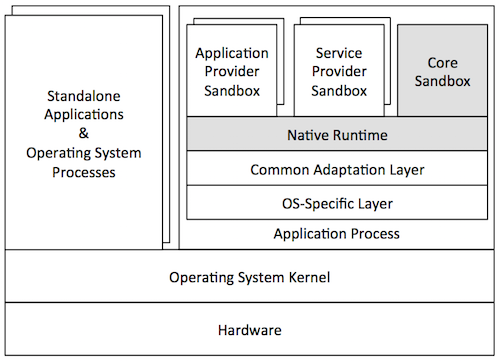


Figure 24: Application platform

The figure above illustrates a general standalone platform tailored for Android mobile devices. Just like in the browser platform, the Hyperty Runtime is wrapped around a host process. The host process is responsible for (1) mediating the system calls issued by the Hyperty Runtime to the operating system and (2) providing a user interface to the Hyperty Runtime and client JavaScript applications (and Hyperties). In addition to the Hyperty Runtime, the host process application consists of: a platform-independent adaptation layer, and platform-specific libraries, e.g., for IO, storage, and memory management. In the example of the figure, the platform-specific libraries correspond to the Android API framework.

From the security point of view, standalone and browser platforms are quite similar; for that reason we adopt the same attacker profiles (regular user, advanced user, and power user). The main difference between architectures is twofold. First, the host application will prevent direct introspection of the JavaScript code running inside Hyperty Runtime sandboxes. As a result, the application architecture is able to mitigate simple attacks to the browser (A0 in the browser’s vulnerability matrix), raising the bar for regular users. Second, the host application will not support software extensions. This restriction prevents some advanced attacks to the runtime based on installation of malicious extension code, and to the browser process (see attacks A1 and A2, respectively, in the browser's vulnerability matrix). Next, we present the vulnerability matrix of the standalone platform and provide alternative attack examples.

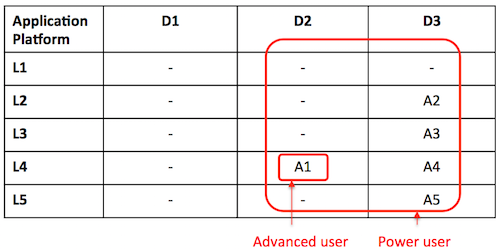


Figure 25: Security Application platform

* *Advanced user*: An advanced user can compromise the entire system by launching attacks at the OS level:
* *A1*: Root the device and instrument the operating system in order to introspect Hyperty instances' sandboxes.
* *Power user*: A power user can mount more sophisticated attacks on various layers of the stack:
* *A2*: Find and exploit a bug in the Hyperty Runtime.
* *A3*: Find a bug in the host application code and exploit it.
* *A4*: Monitor the execution of Hyperty Instances by rooting the device.
* *A5*: Hack the device hardware to extract sensitive Hyperty data from memory.

**Vulnerability assessment:** As highlighted by the vulnerability matrix, an Android-based standalone platform is more robust to attacks than the browser platform. This is mainly due to the fact the application architecture allows us to close security holes in the browser architecture that can hardly be thwarted without modifying the browser. Nevertheless, it is still possible to for an advanced user to compromise the system by rooting the device; the need to root the device will likely deter the regular users. Nevertheless, we recommend prudence in deploying client code that the local user has high incentives to subvert.

#### M2M standalone platform

reTHINK also targets M2M communication use cases. For this reason, a standalone platform is necessary to run the Hyperty Runtime and guest client code. The targeted devices consist of Raspberry Pi and Beagle Boards. Such devices adopt an internal architecture very similar to the standalone platform: they can run Linux or even Android operating systems. We envision that these devices will run Linux-based operating systems. Essentially, the main difference between M2M and vanilla standalone application platform take place at the implementation level. Therefore, our security analysis of the standalone platform is applicable to both instances. As NodeJs was chosen as Native Runtime for the reThink M2M standalone application platform, attacks like server side injection caused by eval function are well known and there are best practices to avoid and protect the software components against such attacks. A valuable source of information that will be taken into account during the implementation is located at: https://nodesecurity.io/resources.

## Runtime APIs

This section describes the programmable interfaces to be implemented by each Hyperty Runtime Component. These interfaces will evolve according to input received from the implementation tasks. Data types defined in [D2.2] are used as much as possible to describe input and output parameters of interface functions.

### Runtime User Agent Interface

#### registerHyperty

Register Hyperty deployed by the App that is passed as input parameter. To be used when App and Hyperties are from the same domain otherwise the RuntimeUA will raise an exception and the App has to use the loadHyperty(..) function.

registerHyperty( Object hypertyInstance, URL.HypertyCatalogueURL descriptor )

#### loadHyperty

Deploy Hiperty from Catalogue URL

loadHyperty( URL.URL hyperty)

#### loadStub

Deploy Stub from Catalogue URL or domain url

loadStub( URL.URL stub)

#### checkForUpdate

Used to check for updates about components handled in the Catalogue including protocol stubs and Hyperties. *check relationship with lifecycle management provided by Service Workers*

checkForUpdate(CatalogueURL url)

#### discoverHiperty

accomodate interoperability in H2H and proto on the fly for newly discovered devices in M2M

discoverHiperty( CatalogueDataObject.HypertyDescriptor descriptor)

### Runtime Registry Interface

#### init

To initialise the Runtime Registry with the RuntimeURL that will be the basis to derive the internal runtime addresses when allocating addresses to internal runtime component. In addition, the Registry domain back-end to be used to remotely register Runtime components, is also passed as input parameter.

init( HypertyRuntimeURL runtimeURL, DomainURL remoteRegistry )

#### registerHyperty

To register a new Hyperty in the runtime which returns the HypertyURL allocated to the new Hyperty.

HypertyURL registerHyperty( postMessage, HypertyCatalogueURL descriptor)

#### unregisterHyperty

To unregister a previously registered Hyperty

unregisterHyperty( HypertyURL url )

#### registerStub

To register a new Protocol Stub in the runtime including as input parameters the function to postMessage, the DomainURL that is connected with the stub, which returns the RuntimeURL allocated to the new ProtocolStub.

HypertyRuntimeURL registerStub( postMessage, DomainURL )

#### unregisterStub

To unregister a previously registered protocol stub

unregisterStub( HypertyRuntimeURL )

To register a new Policy Enforcer in the runtime including as input parameters the function to postMessage, the HypertyURL associated with the PEP, which returns the RuntimeURL allocated to the new Policy Enforcer component.

#### registerPEP

HypertyRuntimeURL registerPEP( postMessage, HypertyURL hyperty )

#### unregisterPEP

To unregister a previously registered protocol stub

unregisterPEP( HypertyRuntimeURL )

#### onEvent

To receive status events from components registered in the Registry

onEvent( Message.Message event )

#### discoverProtostub

To discover protocol stubs available in the runtime for a certain domain. If available, it returns the runtime url for the protocol stub that connects to the requested domain. Required by the runtime BUS to route messages to remote servers or peers (*do we need something similar for Hyperties?*).

RuntimeURL discoverProtostub( DomainURL url)

#### getSandbox

To discover sandboxes available in the runtime for a certain domain. Required by the runtime UA to avoid more than one sandbox for the same domain.

RuntimeSandbox getSandbox( DomainURL url )

#### resolve

To verify if source is valid and to resolve target runtime url address if needed (eg protostub runtime url in case the message is to be dispatched to a remote endpoint ).

Message.Message resolve( Message.Message message )

### Message BUS Interface

To send messages with optional call back. This function is accessible outside the Core runtime.

#### postMessage

postMessage( Message.Message message , callback)

#### addListener

To add "listener" functions to be called when routing messages published on a certain "resource" or send to a certain url. This function is only accessible by internal Core Components.

addListener( listener, URL.URL url )

#### removeListener

To remove previously added listeners. This function is only accessible by internal Core Components.

removeListener( listener, URL.URL url )

#### addPEP

To add an interceptor Policy Enforcer which "listener" function is called when routing messages published on "interceptedURL" or send to the "interceptedURL". To avoid infinite cycles messages originated with from "pepURL" are not intercepted. This function is only accessible by internal Core Components.

addPEP( listener, URL.URL pepURL, URL.URL interceptedURL)

#### removePEP

To remove a previously added interceptor Policy Enforcer. This function is only accessible by internal Core Components.

removePEP( listener, URL.URL pepURL)

### Hyperty Interface

#### init

To initialise the Hyperty instance including as input parameters its allocated Hyperty url, the runtime BUS postMessage function to be invoked to send messages and required configuration retrieved from Hyperty descriptor.

init( HypertyURL url, postMessage, ProtoStubDescriptor.ConfigurationDataList configuration )

#### postMessage

To post messages to be received by the Hyperty instance

postMessage(Message.Message message)

### Policy Enforcer Interface

#### init

To initialise the Policy Enforcer including as input parameters its allocated component runtime url, the runtime BUS postMessage function to be invoked to send messages and the url of the Hyperty associated to the Policy Enforcer (it will forward received and processed messages to this address).

init( URL.RuntimeURL pepURL, bus.postMessage , HypertyURL hyperty)

#### postMessage

To receive messages from the message BUS

postMessage(Message.Message message)

### protoStub Interface

#### init

To initialise the protocol stub including as input parameters its allocated component runtime url, the runtime BUS postMessage function to be invoked on messages received by the protocol stub and required configuration retrieved from protocolStub descriptor.

init( URL.RuntimeURL runtimeProtoSubURL, bus.postMessage, ProtoStubDescriptor.ConfigurationDataList configuration )

#### connect

To connect the protocol stub to the back-end server

connect( identity )

#### disconnect

To disconnect the protocol stub.

disconnect( )

#### postMessage

To post messages to be dispatched by the protocol stub to connected back-end server.

postMessage(Message.Message message)

### Syncher

#### setSender

To set postMessage() function to be used by the Syncher to send messages, i.e. "Policy Enforcer" or the MessageBUS.

setSender( postMessage )

#### createAsObserver

Hyperty instance uses this function to provide the object to be changed by the (observer) syncher according to messages received. The Hyperty instance has previsouly used the *Object.observe* javascript api to set as an observer of this object

Promise <SyncObject> createAsObserver( receivedMessage )

#### createAsReporter

To create a new object and ask another Hyperty instance to observe it. A Create Message will be generated and sent by the Syncher. Promise is used to handle Response messages to this object.

Promise <SyncObject> createAsReporter( resourceURL, schemaURL, toURL, dataObject?)

#### postMessage

To receive messages from other Hyperties that will be reported to the Hyperty.

postMessage(Message.Message message)

### Service Provider Sandbox interface

#### postMessage

To send messages to components running in the sandbox

postMessage(Message.Message message)

### Identity Module Interface

Functions to deal with assertions compliant with [WebRTC RTCIdentityProvider](http://w3c.github.io/webrtc-pc/#identity-provider-interaction)

#### generateAssertion

Generates an Identity Assertion

IdAssertion generateAssertion( contents, origin, usernameHint )

#### validateAssertion

Validates an Identity Assertion

validateAssertion( assertion, origin )

### Core Policy Decision Point (PDP) Interface

#### addPolicies

To add policies to be enforced for a certain deployed Hyperty Instance.

addPolicies( URL.HypertyURL hyperty, HypertyPolicyList policies)

#### removePolicies

To remove previously added policies for a certain deployed Hyperty Instance.

removePolicies( URL.HypertyURL hyperty)

#### authorise

Authorisation request to send a Message. Returns PolicyAction in the AuthorisationResponse, to be performed in case authorisation is not granted.

AuthorisationResponse authorise( Message.Message message)

### Core Policy Enforcement Point (PEP) Interface

#### enforce

Enforcement request to perform a PolicyAction in a Message. Returns the Message resulted from the action performed.

EnforceResponse enforce( PolicyAction action, Message.Message message)

### QoS User Agent Interface

#### getCurrentConnectivityStatistics

Get Connectivity Statistics data

getCurrentConnectivityStatistics( .. )

#### sendConnectivityStatisticsToBroker

Sends Connectivity Statistics data to QoS Broker.

sendConnectivityStatisticsToBroker( ... )

## Runtime Main Procedures

This section describes in detail the Runtime Main procedures by using UML Message Sequence Charts. They correspond to the dynamic view of the Hyperty Runtime and they validate the static design for the most important use cases that were already used in WP2 and originaly described in WP1.

The previsouly defined APIs are used as much as possible in the messages signature in order to validate the Runtime design. The detailed definition of messages exchanged among Hyperties and protocol stubs, as defined in D2.2 Message Model, are also used when appropriate.

### Runtime Basic Procedures

This section, describes in detail the Basic Runtime procedures that are required to support the deploy and operation of Hyperties in the runtime. It includes:

* Deployment of the Core Runtime components when they are not natively supported by the device
* Deployment of protocol stubs
* Deployment of Hyperties
* Generic procedure to route messages
* Four different Hyperty communication situations namely:
  + local communication between Hyperties from the same domain (running in the same Sandbox)
  + local communication between Hyperties from different domains (running in different sandboxes but in the same Runtime Instance)
  + Remote communication between Hyperties from the same domain
  + Remote communication between Hyperties from different domains

#### Deploy Hyperty Runtime

In case the device does not support the Hyperty Core Runtime components e.g. an existing browser like Chrome or a Network Node.js Server, they have to be deployed in the Device or in the Server.

The main data flows to support the deployment of the Hyperty Core Runtime is depicted in the diagram below.

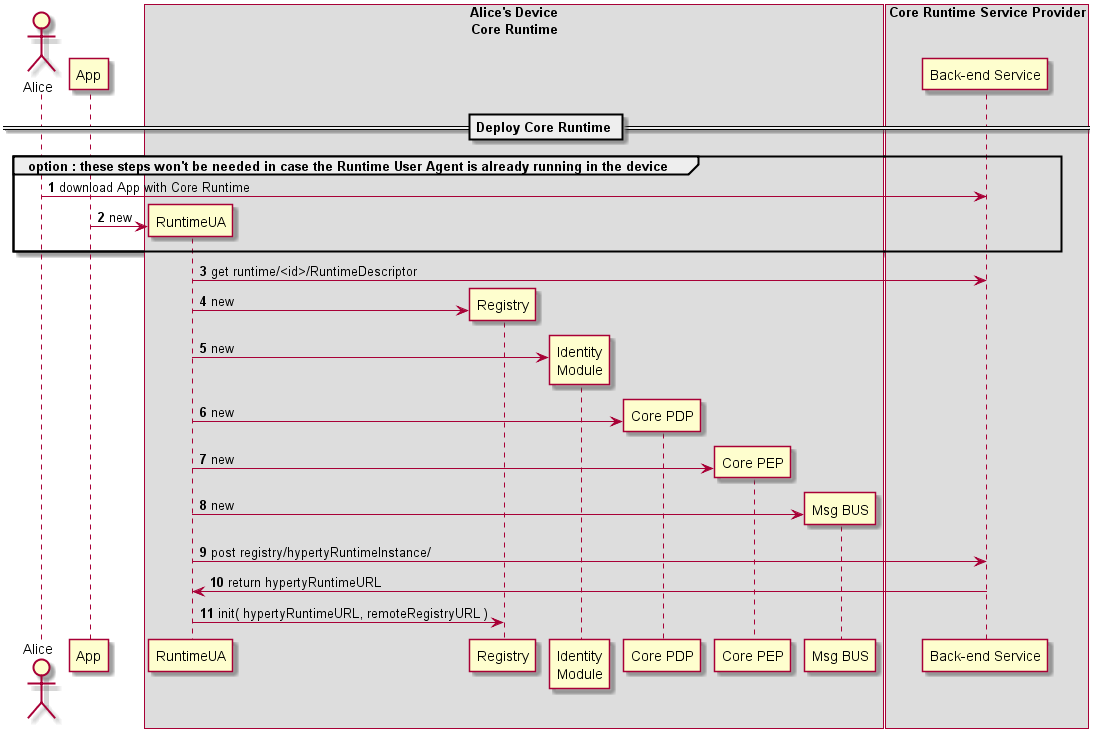


Figure 26: Deploy Core Runtime Components in the Native Runtime

Steps 1 - 2 : the Runtime can be explicitly deployed by a specific Application or can be implicitly deployed when an Hyperty or Protocol Stub is required. The usage of existing libraries like require.js will be evaluated.

Steps 3 - 8 : the Runtime User Agent handles the download, instantiation and initialisation of required Runtime Core components including the Runtime Registry, Identity Module, Runtime PDP/PEP and the Message BUS.

Steps 9 - 11 : the Runtime User Agent registers the Runtime Instance into the remote Registry Service of the Hyperty Runtime Service Provider which returns the RuntimeURL allocated to the new Runtime. Then, the Registry is initialised with the previously returned RuntimeURL that will be used to derive the internal runtime addresses to be allocated to runtime components.

#### Deploy Protocol Stub

The main data flows to support the deployment of protocol stubs required to connect the Hyperty Runtime to a specific back-end server, is presented in the figure below and described in this section.

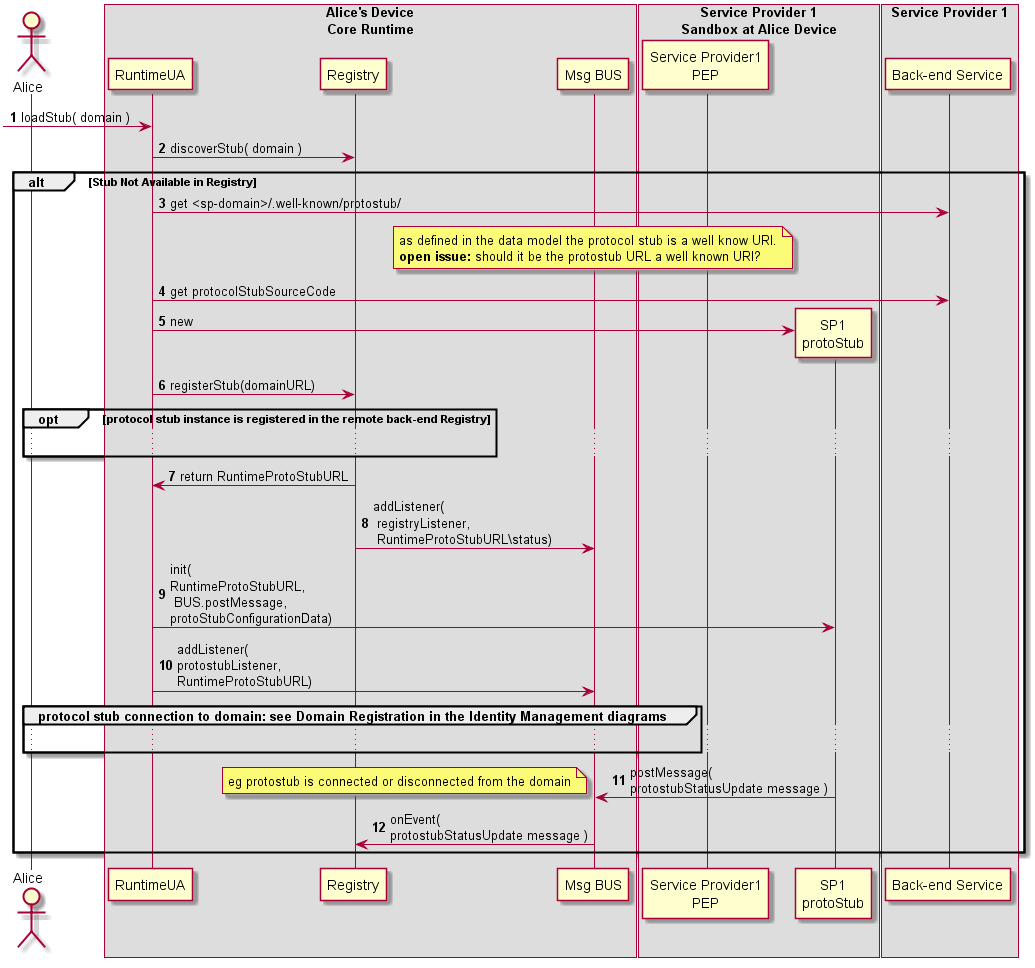


Figure 27: Deploy Protocol Stub

Steps 1-2 : The protocol stub deployment may be triggered by the deployment of an Hyperty or by some attempt from a local Hyperty to communicate with a remote Hyperty running in the domain served by the protocol Stub. In this case the Runtime Registry would take the initiative to start the protocol stub deploy (FFS). Such trigger may take advantage of some existing libraries like require.js (to be validated with experimentations). The Runtime UA only downloads and deploys requested protocol stub after checking in the Registry that there is no protocol stub available in the runtime.

Steps 3 - 5 : the Runtime UA is able to derive the URL to download the protocol stub descriptor from the domain url, since it is a well known URI defined in the reTHINK Architecture Interfaces [15]. The protocol stub descriptor contains the url that the Runtime UA uses to download and instantiate the protocol stub in the runtime. Depending on the Runtime Sandbox implementation, the download and instantiation may have to be performed inside the Sandbox.

Steps 6 - 8 : the new protocol stub is registered in the Runtime Registry, which allocates and return the runtime address (RuntimeURL) for the new runtime component. In addition, the runtime Registry requests the runtime BUS to add its listener to receive events about the protocol stub status.

Steps 9 : The Runtime UA initialises the new protocol stub with configuration data contained in its descriptor. Depending on the sandbox implementation, the initialisation may have to be remotely executed by a Execution message type routed by the Message BUS.

Steps 10 : The Runtime UA adds in the runtime BUS the protostub listener to receive messages from the runtime. Protocol stubs are connected by using credentials handled by the Core Runtime Identity Module which are detailed in the [Domain Login](#domain-login).

Steps 11 - 12 : protocol stub publishes its status (including events about when it is connected or disconnected) in its resource status. Components registered on the protocol stub status resources, like the Registry, are notified about the new protocol status.

Message to publish Protocol Stub Status

"id" : "1"  
"type" : "UPDATE",  
"from" : "hyperty-runtime://sp1/protostub/123",  
"resource" : "hyperty-runtime://sp1/protostub/123/status",  
  
"body" : { "value" : "LIVE" }

#### Deploy Hyperty

The Runtime procedures to deploy a new Hyperty are described in this section.

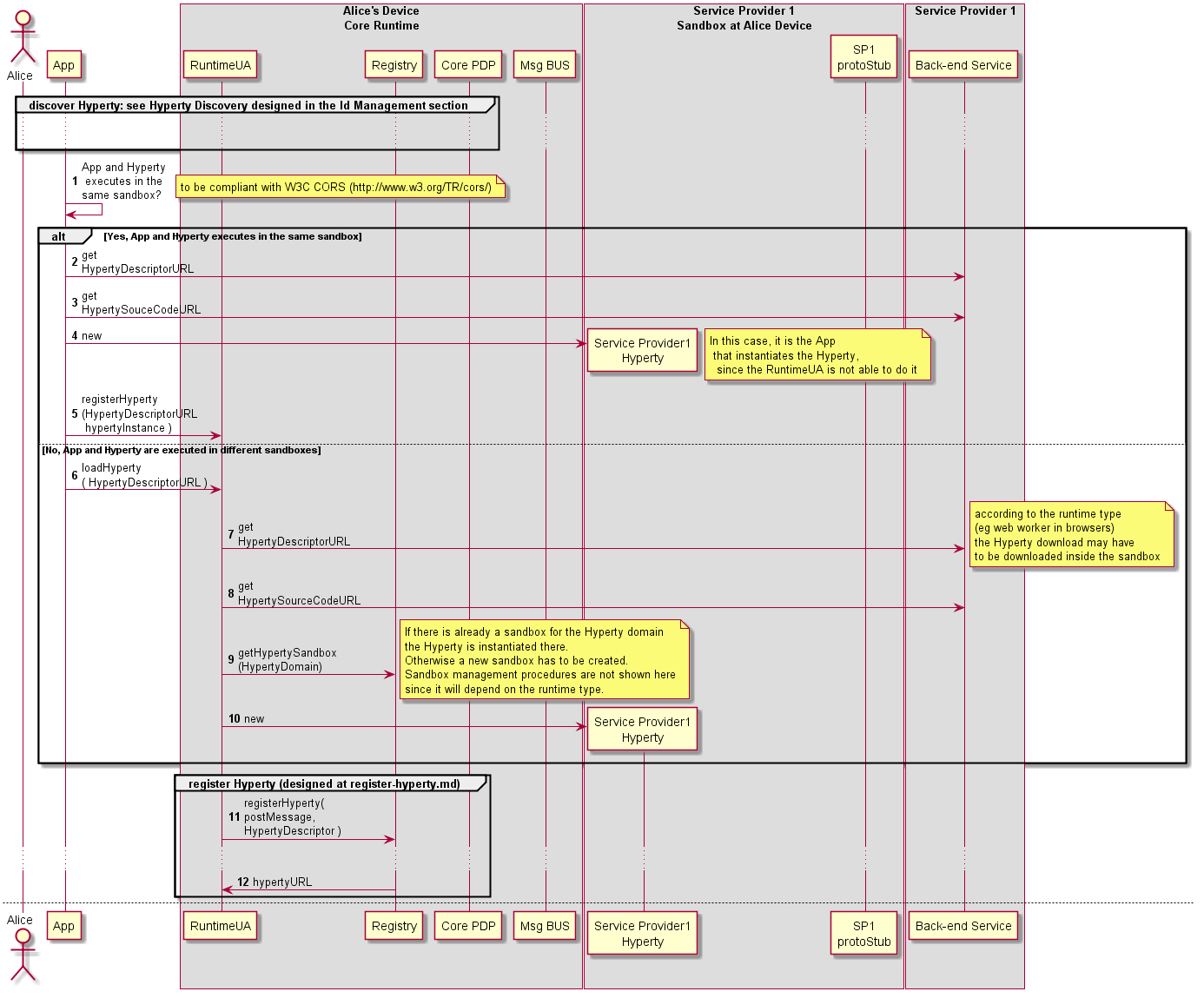


Figure 28: Deploy Hyperty (part1)

Note: The trigger of Hyperty deployment may take advantage of some existing libraries like require.js.

Step 1: As discussed in the Runtime Architecture, and according to security policies, Hyperties and the Application can be deployed in the same sandbox or in separated domains.

**Hyperty and App deployed in the same sandbox**

Steps 2 - 5: In this situation, the App and the Hyperty are running in the same isolated sandbox which is different from the Hyperty Core Runtime Sandbox. This means the download and instantiation of the Hyperty has first to be performed by the Application. Then the App asks the Runtime UA to register and activate the new Hyperty in the runtime.

**Hyperty and App deployed in different sandboxes**

Steps 6 - 10: In this situation, the App and the Hyperty must run in different isolated sandboxes. In this case the Hyperty sandbox is managed by the runtime UA which means the runtime UA can download and instante the Hyperty. The runtime UA should avoid the creation of new sandboxes in case there is already a sandbox for the same domain

Steps 11 - 12 : the new [Hyperty instance is registered](register-hyperty.md) by the Runtime Registry. See section ? for more details.

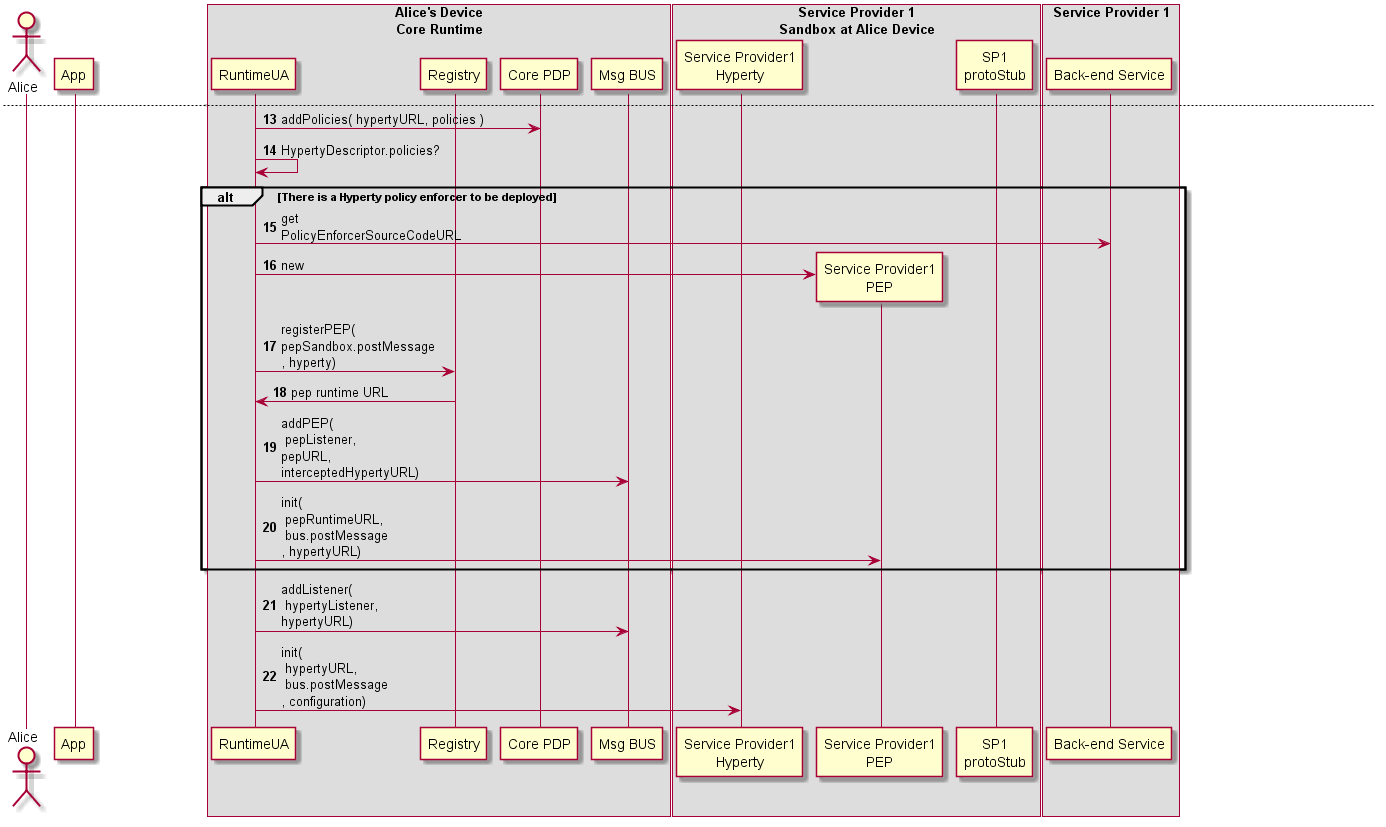


Figure 29: Deploy Hyperty (part2)

Steps 13: policies contained in the Hyperty Descriptor, are deployed in the BUS Authorisation component

Steps 14: the runtime UA checks in the Hyperty Descriptor if a Policy Enforcer is required

**Hyperty PEP deployment is required**

Steps 15 - 16 : the runtime UA downloads and instantiates the Hyperty PEP in a isolated sandbox.

Steps 17 - 18 : the Runtime UA register in the runtime Registry the new PEP for the new deployed Hyperty and the Registry returns PEP Runtime component URL

Steps 19 : the runtime UA adds PEP intercepting listener to the runtime BUS to receive messages targeting the Hyperty URL.

Step 20 : The Runtime UA activates the Hyperty PEP with its RuntimeURL, the postMessage function to be called to send messages to BUS and the Hyperty instance URL the PEP is intercepting. Depending on the sandbox implementation, the initialisation may have to be remotely executed by a Execution message type routed by the Message BUS.

Steps 21 : the runtime UA adds Hyperty listener to the runtime BUS to receive messages targeting the Hyperty URL. It should be noted in case there is an intercepting PEP, the Hyperty listener will only be called for Messages forwarded by PEP.

Steps 22 : the runtime UA activates the Hyperty instance with its Hyperty URL instance, the postMessage function to be called to send messages to BUS and configuration data contained in its descriptor. Depending on the sandbox implementation, the initialisation may have to be remotely executed by a Execution message type routed by the Message BUS. .

#### Message Routing in Message BUS

The Runtime procedures to route a message by the Runtime BUS are described in this section.

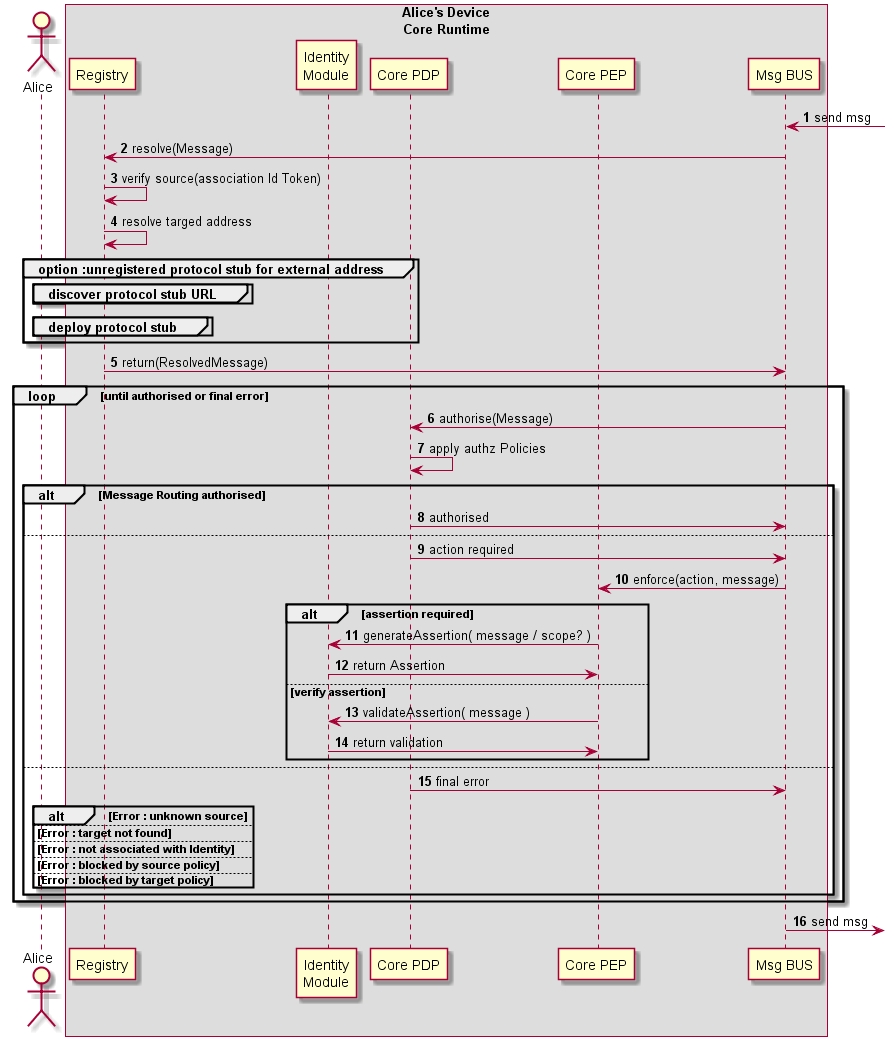


Figure 31: Message Routing in Message BUS

Steps 1 - 5 : on receiving a message, the Runtime BUS requests the Registry to verify if the originator is valid (3) (i.e. its Runtime URL has been previously registered) and checks if the target address is external to the Runtime. If yes, it looks for the protostub Runtime URL to be used. The process to [deploy the protocol Stub in the runtime](deploy-protostub.md) (section ?) is triggered, in case it is not available yet.

Steps 6 - 8: in case the message requires authorisation, the Core PDP applies applicable policies to authorise its routing.

Steps 9 - 14 : The Core Policy Enforcer enforces auhtorisation policies (including generation of Assertions or verification of assertions) in case the Runtime PDP requests it. In case policy enforcement is performed successfuly, routing authorisation is requested again (step 6).

Step 15 : the application of authorisation policies by the PDP can result in different types of final errors including:

* target does not exist
* Hyperty instance that is sending the message is not associated with an appropriate Identity
* the message is blocked by a source or target policy

#### Intra-domain Local Communication

Communication between two Hyperties running in the same Runtime instance can be performed locally by using some non-standard function or through the Runtime BUS using postMessage standard function.

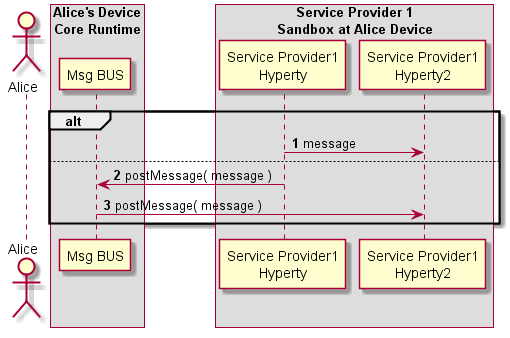


Figure 32: Intra-domain Local Communication

#### Intra-domain Remote Communication

The routing of messages between two Hyperties running in different Runtime instance but from the same domain, is described below.

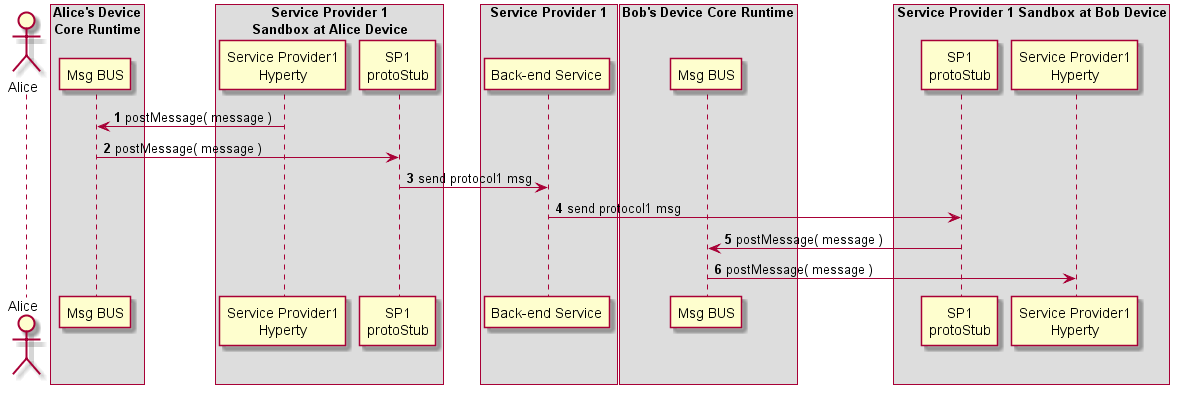


Figure 33: Intra-domain Remote Communication

#### Inter-domain Local Communication

The routing of messages between two Hyperties running in the same Runtime instance but in different sandboxes (e.g. they are from different domains) is described below.

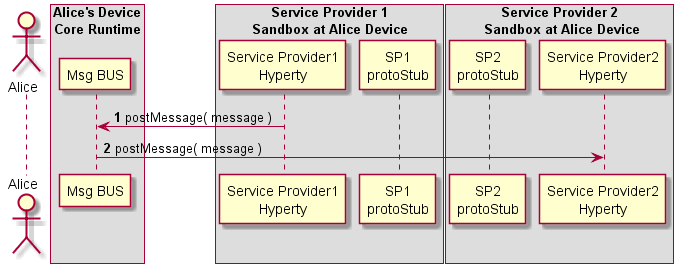


Figure 34: Inter-domain Local Communication

#### Inter-domain Remote Communication

The routing of messages between two Hyperties running in different Runtime instance and from different domains, is described below.

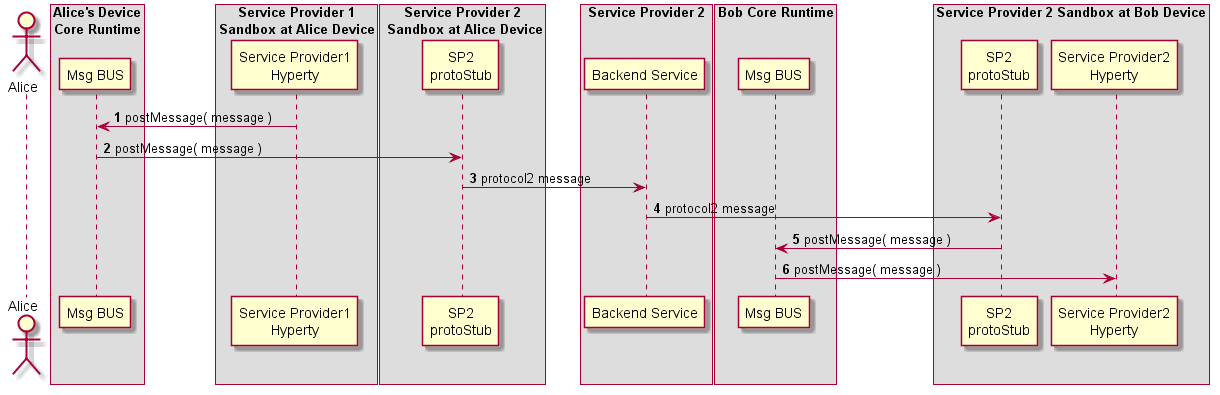


Figure 35: Inter-domain Remote Communication

### Runtime Identity Management Procedures

This section, describes in detail the Runtime procedures that are required to manage Identities used by Hyperties. It includes:

* User Registration in the Domain
* Domain Login
* Association between Identities and Hyperty Instances
* and Assertion of User Identities

#### User Registration

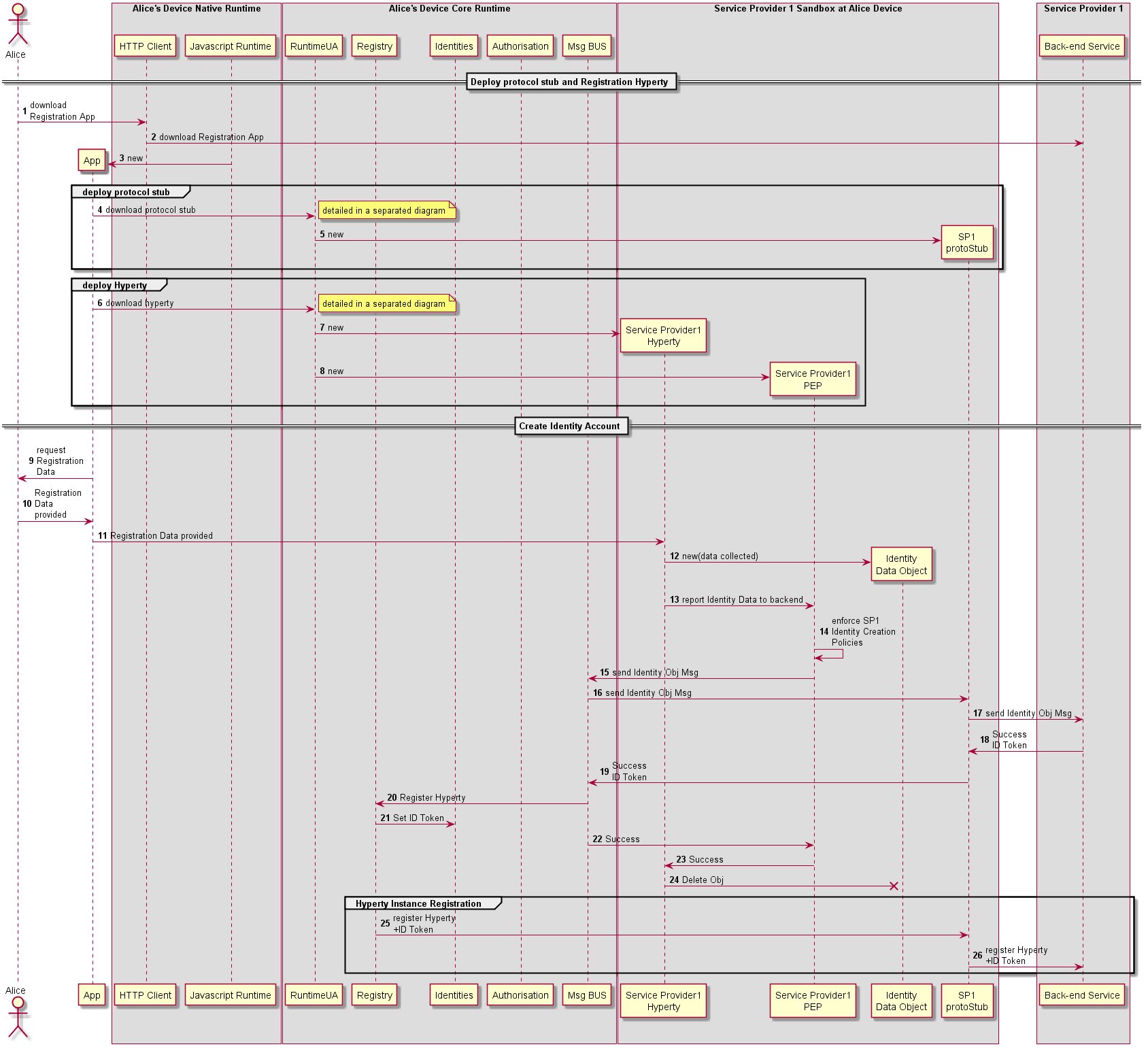


Figure 36: User registration

In this use case, it is considered there is a single protocol stub to interact with all back-end services including Identity Management. Another option is to have different protocol stubs to interact with different back-end services including authentication, authorisation and messaging services.

#### Domain Login

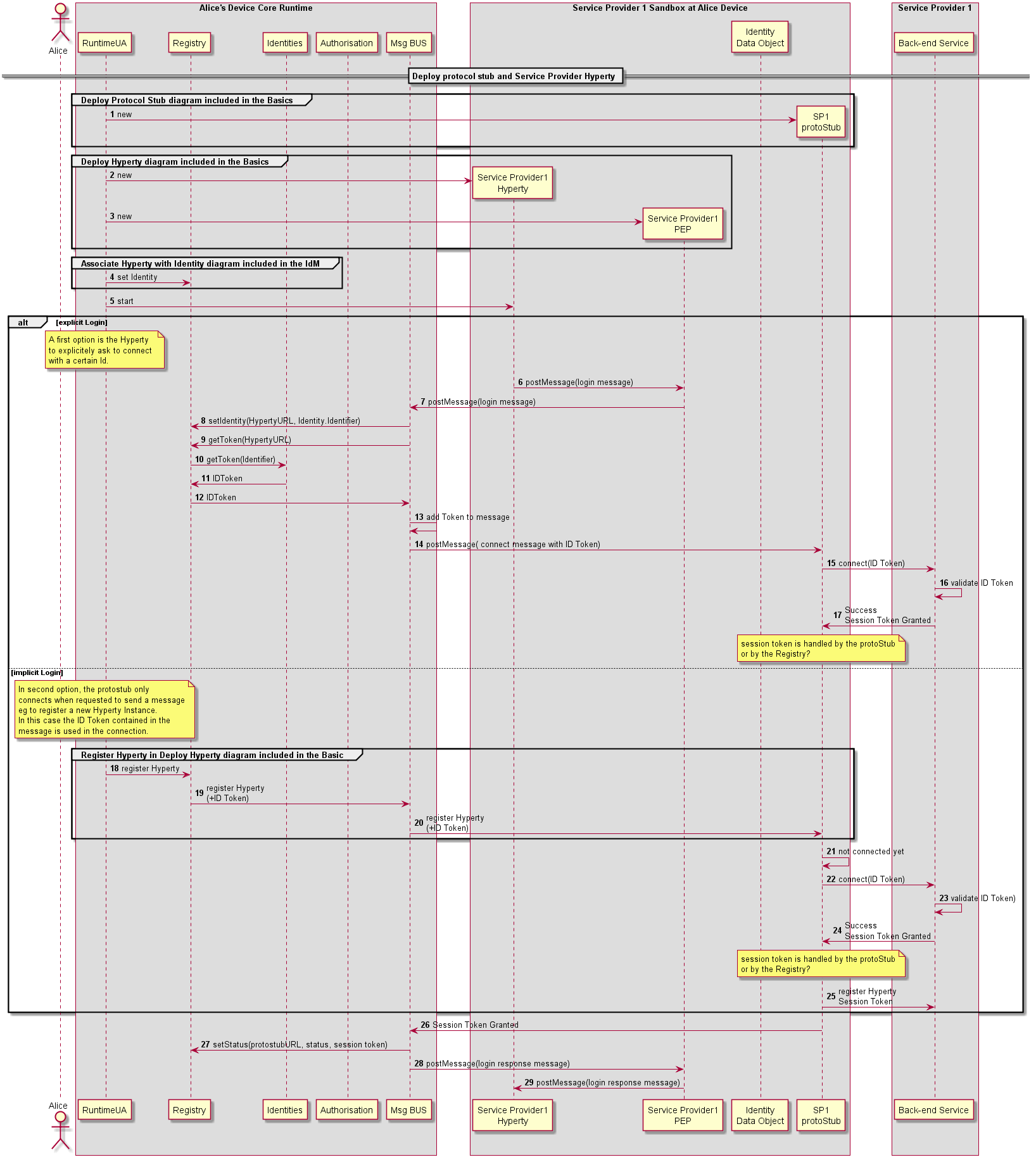


Figure 39: Domain Login

In this use case, it is considered there is a single protocol stub to interact with all back-end services including Identity Management. Another option is to have different protocol stubs to interact with different back-end services including authentication, authorisation and messaging services.

#### Associate User Identity to Hyperty Instance

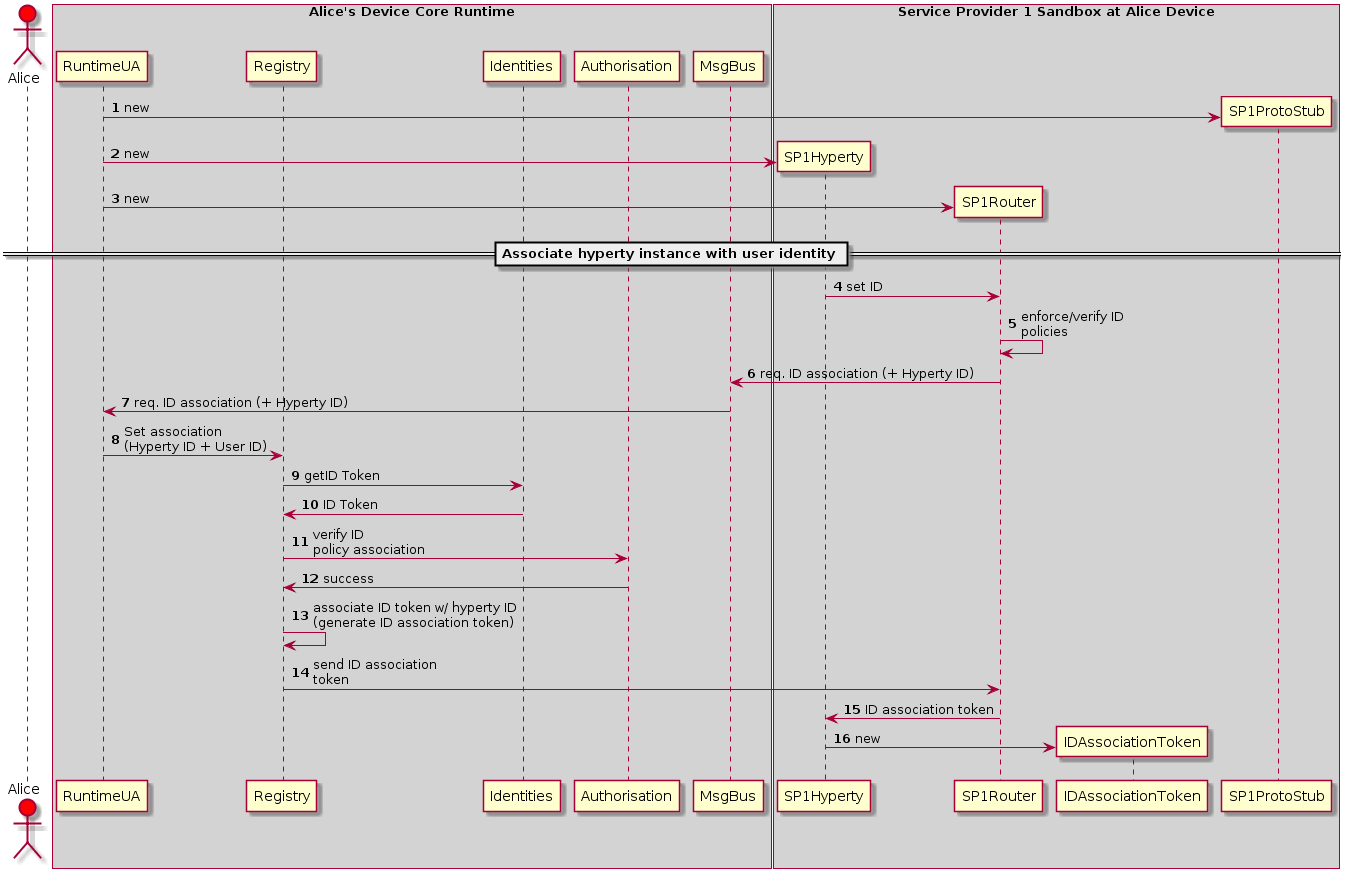


Figure 40: Associate User Identity to Hyperty Instance

This sequence details the steps needed to associate the user identity to a given Hyperty instance.

*1*- Create ProtoSutb1 sandbox.

*2*- Create Hyperty 1 instance for Service Provider 1.

*3*- Create SP1 router and the respective PEP connector.

*4*- The application using Hyperty 1, triggers a request to set the Identity to be associated to this Hyperty instance. This request is sent to the SP1 router to be touted to the RunTime UA

*5*- Optimally the SP1 router checks the policies of the application itself in regard to the internal identity rule/policies. note that, this verification is internal and not related with the verification performed by the Core Runtime.

*6*- SP1Router send the request (if authorized by the Application internal rules) to associate a identity to the Hyperty 1 instance. This request is sent to the Core Runtime Message Bus. This request includes the Identification Token of Hyperty 1.

*7*- The MsgBus sends the Hyperty-user association to the RunTime UserAgent.

*8*- The RunTime UserAgent 'selects' the user identity to be used (eventually by asking Alice which used ID to use) and sends it to the Registry.

*9*- The registry sends a request to the Identities Engine.

*10*- The Identities Engine replies with the identity token (ID Token) for the selected user. This step assumes that a identity Token has already exists for the requested user. If it does not, a [Domain Login](domain-login.md) must be performed.

*11*- The Registry sends a request to the Authorization/Policy engine to verify if the User Identity association request by the Hyperty Instance is authorized by the existing Policies.

*12*- If the association is allowed a success message is replied to the registry. If not a reject message is replied (not depicted in the figure).

*13*- The Register Engine generates an Association Token. This Association Token will allow the Hyperty instance to use the requested ID Token.

*14*- The created ID Association Token is sent to the SP1 router.

*15*- The router forwards the ID Association Token to the Hyperty instance (how requested it).

*16*- Hyperty 1 created a new ID Association Token object.

Note: This association protocol is assuming that the request for the ID association is triggered by the Application/Hyperty instance. The Second option is for the association action to be triggered by the User Agent (RuntimeUA). In this case steps 4 to 7 need to be changed.

Question: Which option should be provided? If both, which should be the default one?

#### User identity assertion

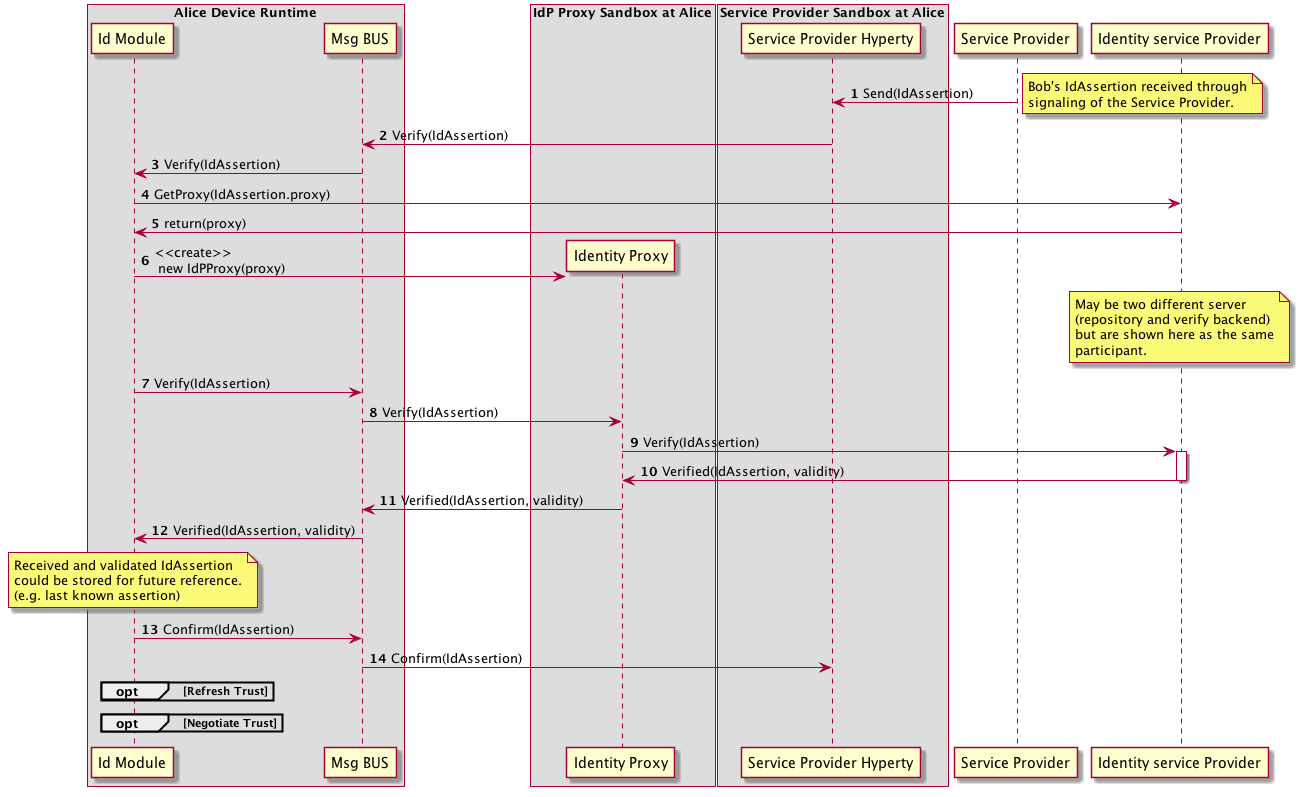


Figure 41: User identity assertion sequence diagram

In this sequence, Alice Hyperty receives an Identity Assertion from its signaling/backend service. Before prompting the user with the asserted identity (e.g. an incoming call notification) it must be verified. The hyperty thus uses the Verification API to ask the runtime to verify the received assertion. Communication with the Runtime is done through the Msg bus.

Upon receiving the verification request, the runtimeUA instantiate an IdP Proxy from a URL. This URL is given in the IdToken. Alternatively the IdP Proxy may already have been instantiated. Once instantiated, the RuntimeUA uses it to verify the IdToken. Communication with the IdP Proxy is also done through the Msg bus.

If the IdToken is validated, the RuntimeUA confirms the validity to the Hyperty.

### Main Runtime Procedures for H2H Communication

This section, describes in detail the Runtime procedures that are required to support Human to Human communication in the runtime. The description are focused on the validation of the Reporter-Observer communication pattern with WebRTC communications. Two main use cases are considered:

1. Intra-domain communication where both parties are logged in the same domain
2. Inter-domain communication where involved parties are logged in different domains and interoperability is achieved thanks to the protocol-on-the-fly concept.

For each these Use Cases, six procedures are performed:

1. Alice invites Bob
2. Bob receives Invitation from Alice
3. Alice is aknowledged Bob received Invitation
4. Bob's App interaction and Alice's connection update
5. Bob gathers WebRTC resources
6. Synchronization of Alice's Data Object

#### Main Runtime Procedures for Intra-domain H2H Communications

##### H2H Intradomain Communication - Alice invites Bob

This MSC diagrams shows the most relevant steps to support the initial invitation of Alice to Bob.

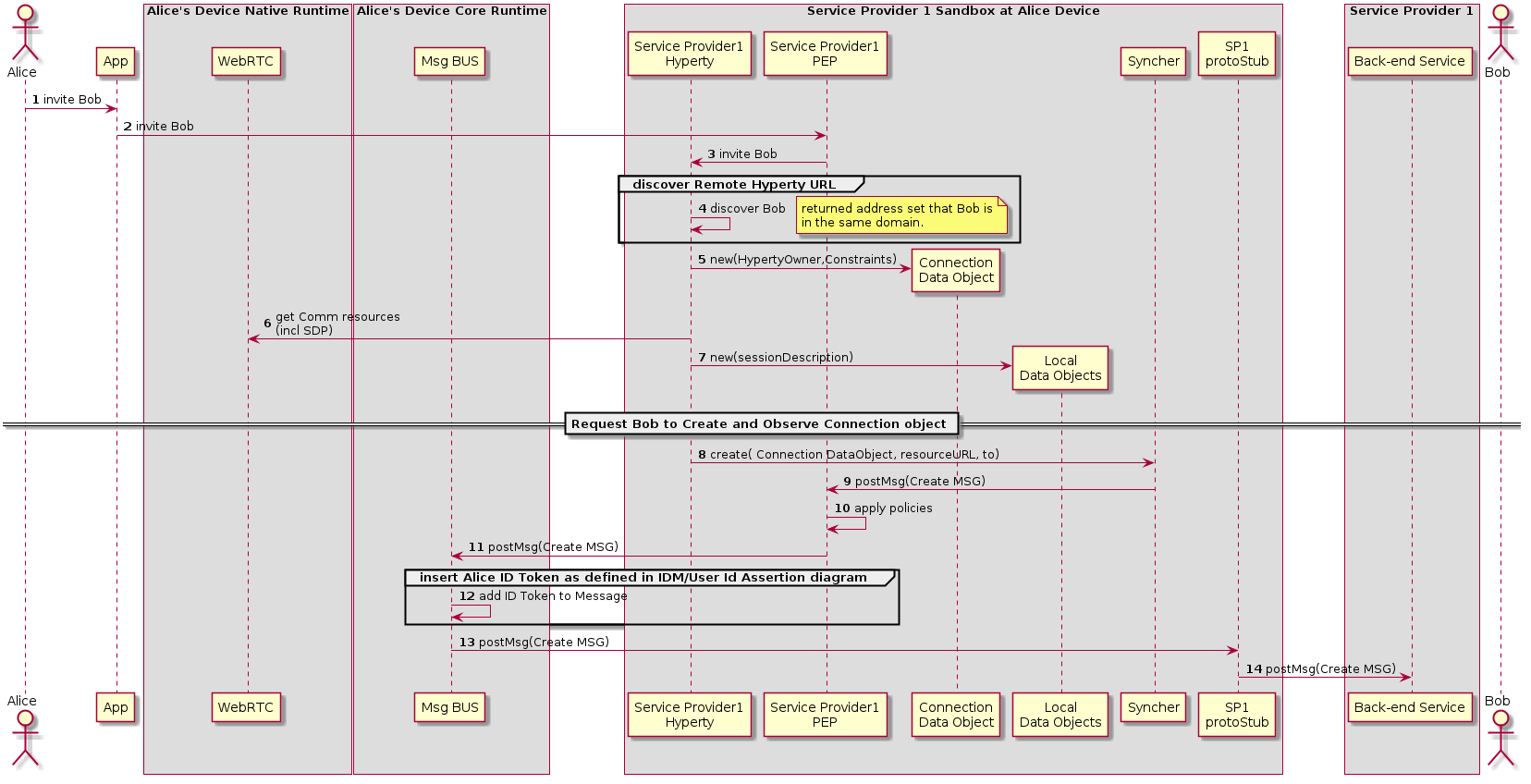


Figure 42: Alice invites Bob for a communication

(Steps 1 - 4) : Alice decides to invite Bob for a communication. The discovery of Bob's Hyperty Instance URL is described here(../identity-management/discovery.md).

(Steps 5 - 7) : the Hyperty Instance creates the Connection, the LocalConnectionDescription and the LocalIceCandidates data objects as defined in [15].

(Steps 8 - 9) : the Hyperty Instance requests the Syncher to ask Bob to create and observe these objects. Syncher generates CREATE messages for each object and puts it in the Body in JSON format. For simplification purposes we assume the CREATE msg contains the Connection object plus local SDP and local IceCandidates:

[**Create Message**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#createmessagebody)

"id" : "1"  
"type" : "CREATE",  
"from" : "hyperty-instance://sp1/alicehy123",  
"to" : "hyperty-instance://sp1/bobhy123",  
"contextId" : "qwertyuiopasdfghjkl",  
"body" : { "resource" : "comm://sp1/alice/123456", "value" : "<json object with connection, sdp and ice candidates>"}

(Steps 10) : Alice's PEP applies local policies if required including outgoing communication request access control

(Steps 11) : Alice ID Token assertion is added to the message (see chapter "User identity assertion" for more details).

(Steps 12 - 14) : the message is routed through Alice Message BUS reaching Service Provider Back-end Messaginge Service.

##### H2H Intradomain Communication - Bob receives invitation

This MSC diagrams shows how Bob receives invitation from Bob.

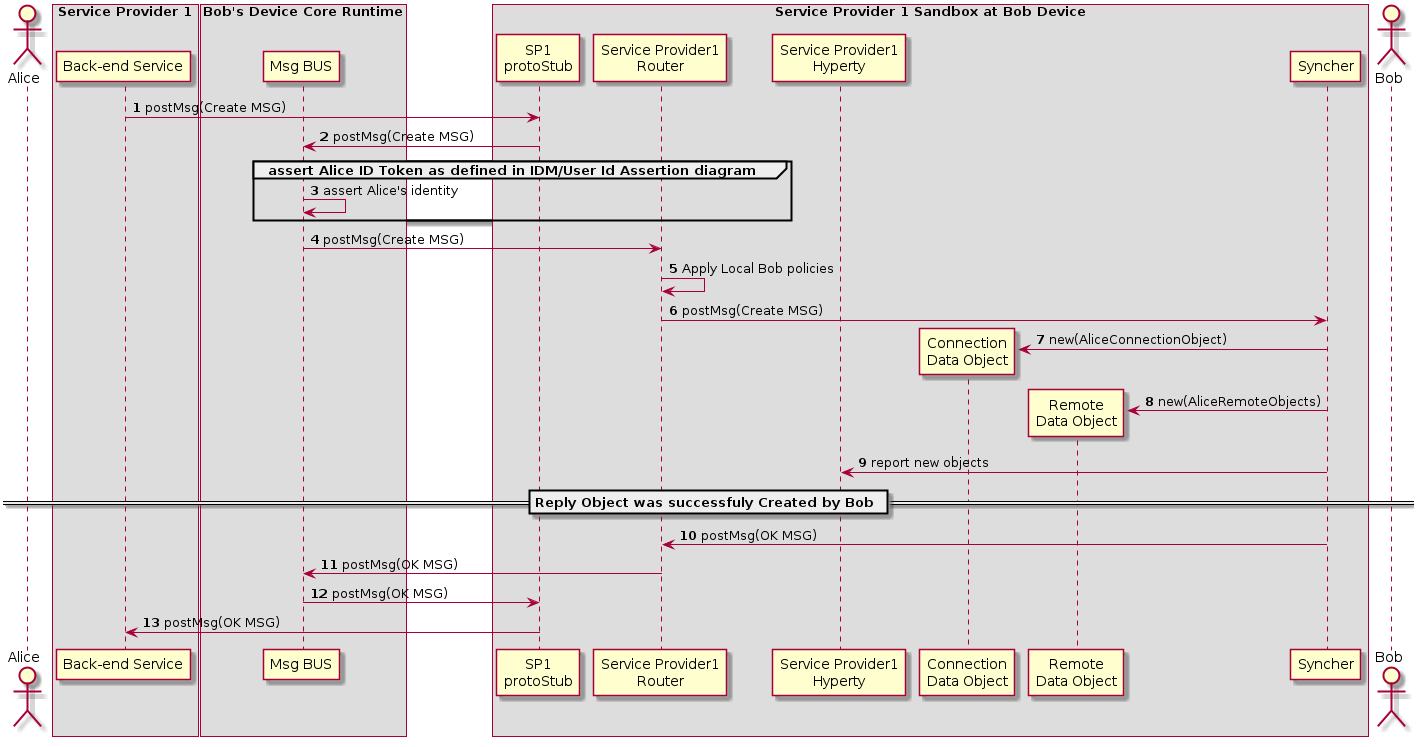


Figure 43: Bob receives invitation

(Steps 1 - 4) : Service Provider Back-end Messaginge Service routes the message to Bob's Message BUS, asserts Alice's identity and forwards the message to Bobs Router reaching Bob's PEP component

(Step 4) : Bob's PEP applies local policies if required including incoming communication request access control

(Steps 5 - 8) : the message is forwarded to Bob's Syncher which creates the requested new objects and reports to Bob's Hyperty Instance the new created objects.

(Steps 9 - 13) : As soon as the new Objects were created by Bob's syncher, it responds back to Alice to confirm the objects were created with a [Response Message](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#responsemessagebody).

"id" : "1"  
"type" : "RESPONSE",  
"from" : "hyperty-instance://sp1/bobhy123",  
"to" : "hyperty-instance://sp1/alicehy123",  
"contextId" : "qwertyuiopasdfghjkl",  
"body" : { "code" : "200" , "description" : "ok"}

##### H2H Intradomain Communication - Invitation Acknowledgement

This MSC diagrams shows how Alice is aknowledged that Bob received the invitation

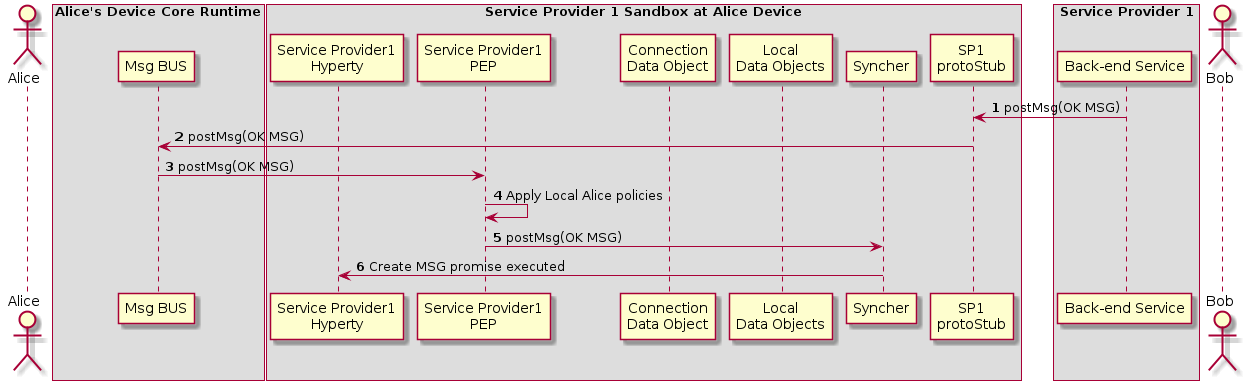


Figure 44: Aknowledged that Bob received the invitation

(Step 1 - 3) : Service Provider Back-end Messaginge Service routes the OK Message to Bob's Message BUS which forwards it to its PEP

(Step 4) : Bob's PEP applies local policies if required

(Steps 5 - 6) : the message is forwarded to Alice's Syncher which updates the Data Object and reports the change to Alice's Hyperty Instance

##### Incoming call is notified to Bob's application and Alice is updated

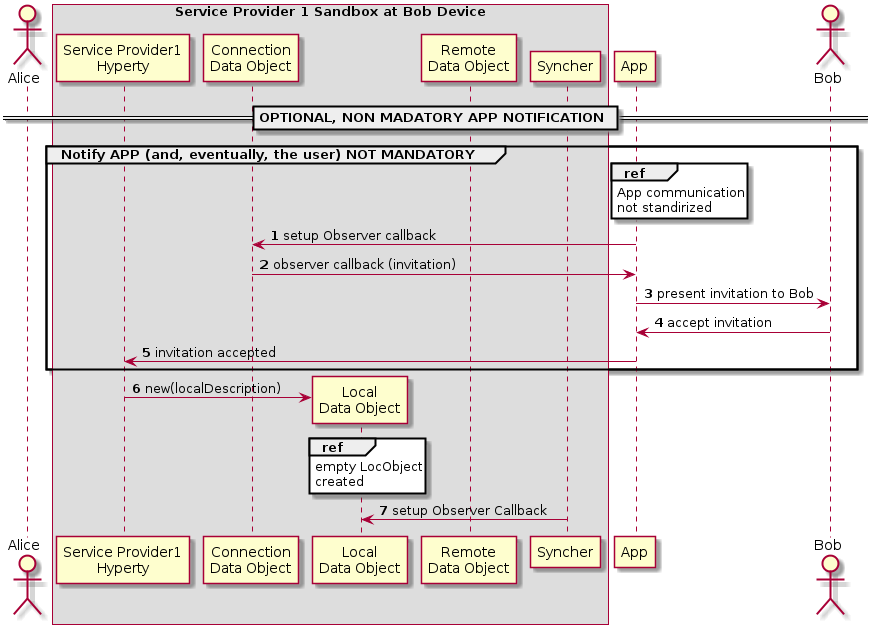


Figure 45: notification update

(step 1) : The Application which interacts with the human user setups a callback in to be notified when the Connection data Object is modified.

(step 2) : When a Data Connection Object receives any modification request from another Hyperty, the callback setup in the step before is called. The App is aware of the incoming invitation to establish a media session.

(step 3) : The App can show this invitation to the human user in some way through a human interface. (step 4) In such a case the human typically will accept the communication. (step 5) The App accepts the invitation through the API exposed by the the Service Provider Hyperty. In order to start the media session a Local Data Object is created (step 10) where the data related to the local parameters of the media session is going to be established.

(step 6) : The Syncher element from the Hyperty setups an Observer callback in the Local Data Object which will be called when the Local Data Object changes. (step 7) The observer reports that there is a communication in progress to the Syncher.

##### Bob starts WebRTC API

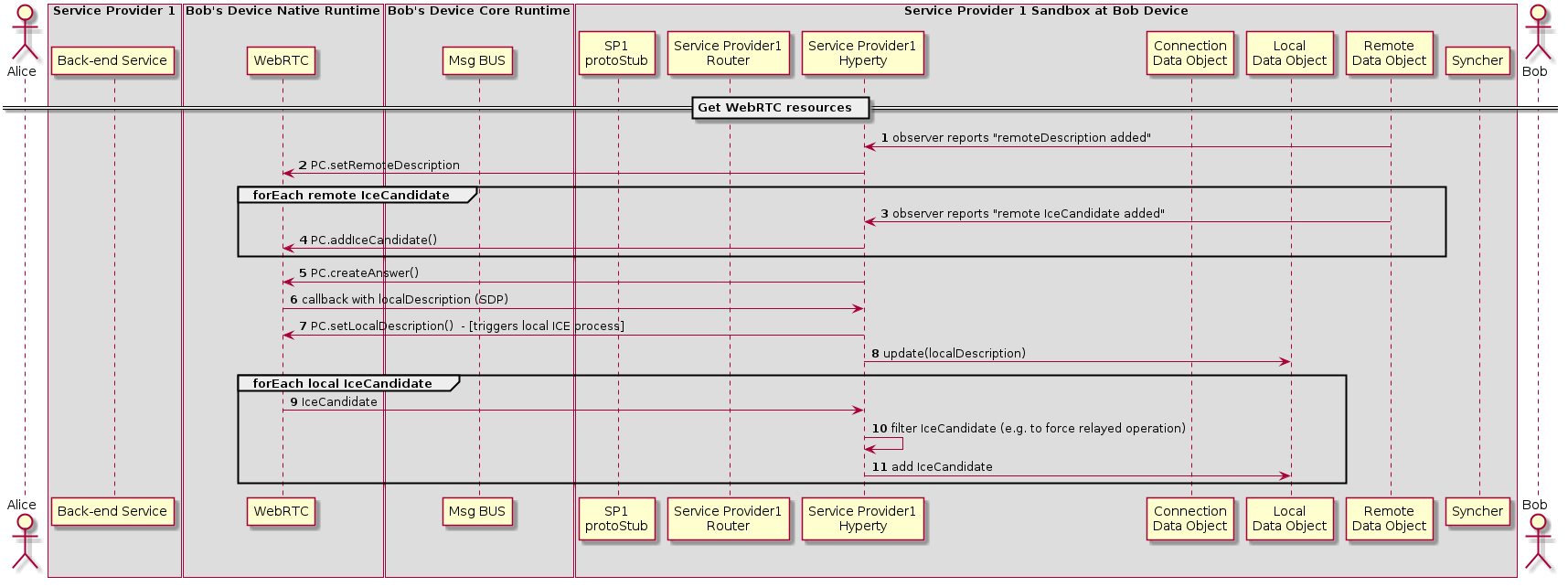


Figure 46: Bob gatheres WebRTC resources

(Step 1) : The Hyperty is notified about the added remoteDescription object.

(Step 2) : The Hyperty calls the WebRTC API from the browser including the remote parameters from the Remote Data Object. The same happens when a new Ice Candidate is updated in the Remote Data Object (step 3 and Step 4).

While remote Ice Candidate are added (step 3 and Step 4 may take place several times as Trickle Ice is supported) the Hyperty calls the Peer Connection method to create an SDP answer (step 5) to be sent to it with all the parameters used to establish the media session between Alice and Bob but the Ice Candidates which will be received asynchronously later. When the SDP with the local description is ready a callback is called and the SDP is sent to the Hyperty (step 6).

(Step 7) : The Hyperty calls the Peer setLocalDesciption API method from the WebRTC API exposed by the browser so that the browser is aware of the media parameters which are going to be used to establish the media session with Alice. At this point the gathering process of local Ice Candidates starts.

(Step 8) : The Hyperty updates the Local Data Object with the parameters from the localDescription.

(Step 9) : As a result of the started ICE process local connectivity candidate will be reported from the WebRTC engine to the Hyperty. For each reported localCandidate the Hyperty can optionally perform a filter operation (Step 10), e.g. to filter out non-relay candidates to force TURN based operation, and reports the remaining candidates to the Local Data Object (Step 11)

##### Synchronization of Alice's Data Object

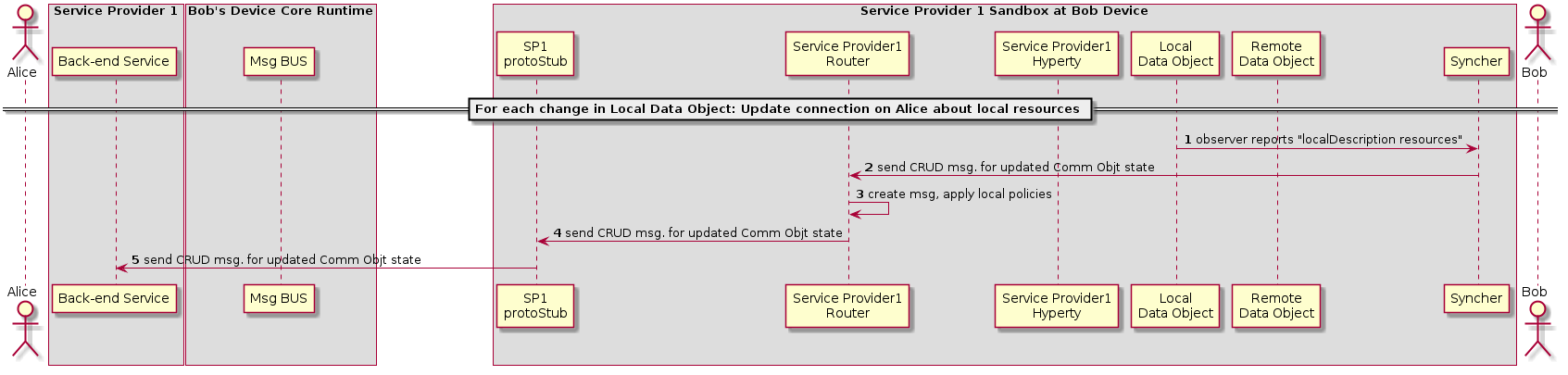


Figure 47: Synchronization of Alice's Data object

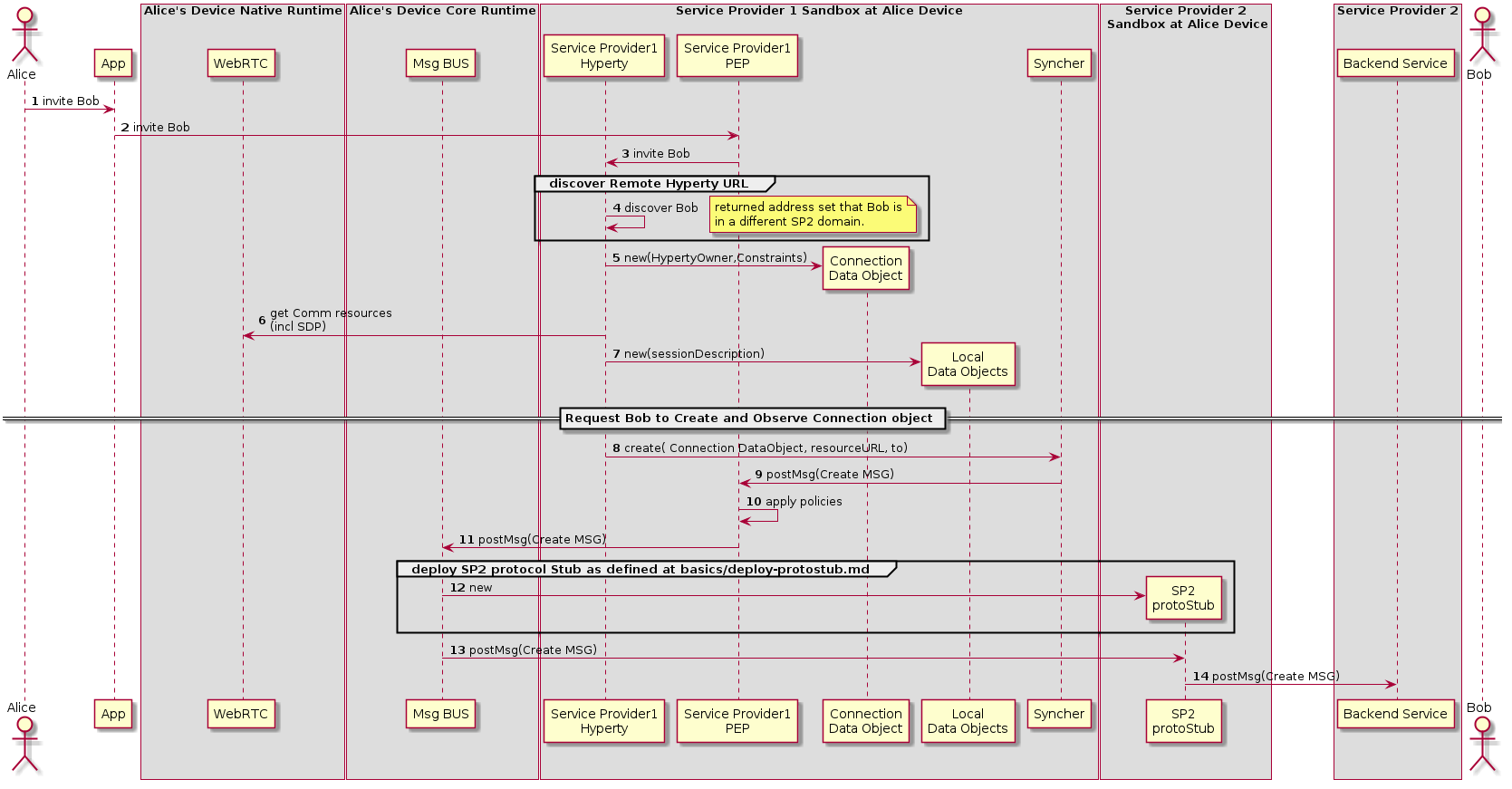
1. Synchronization of Alice's Data object (Step 1) The local Data object reports that there have been changes in the connection parameters and the Syncher sends a CRUD message through the Policy Enforcer to Update the Remote Data Object at Alice's Hyperty (Step 2).

(Step 3) the Policy Enforcer checks if the message is compliant with the local policies and the message is sent to the ProtoStub (Step 4) to be in turn sent to the Service Provider 1 Back-End (Step 5)

#### Main Runtime Procedures for Inter-domain H2H Communications

##### H2H Interdomain Communication - create communication

This MSC diagrams shows the most relevant steps to support the initial invitation of Alice to Bob, where Alice and Bob are in different domains.



H2H Intradomain Communication : create communication

(Steps 1 - 4) : Alice decides to invite Bob for a communication. The discovery of Bob's Hyperty Instance URL is described here(../identity-management/discovery.md).

(Steps 5 - 7) : the Hyperty Instance creates the Connection, the LocalConnectionDescription and the LocalIceCandidates data objects as defined in [15].

(Steps 8 - 9) : the Hyperty Instance requests the Syncher to ask Bob to create and observe these objects. Syncher generates CREATE messages for each object and puts it in the Body in JSON format. For simplification purposes we assume the CREATE msg contains the Connection object plus local SDP and local IceCandidates:

[**Create Message**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#createmessagebody)

"id" : "1"  
"type" : "CREATE",  
"from" : "hyperty-instance://sp1/alicehy123",  
"to" : "hyperty-instance://sp2/bobhy123",  
"contextId" : "qwertyuiopasdfghjkl",  
"body" : { "resource" : "comm://sp1/alice/123456", "value" : "<json object with connection, sdp and ice candidates>"}

(Steps 10) : Alice's PEP applies local policies if required including outgoing communication request access control

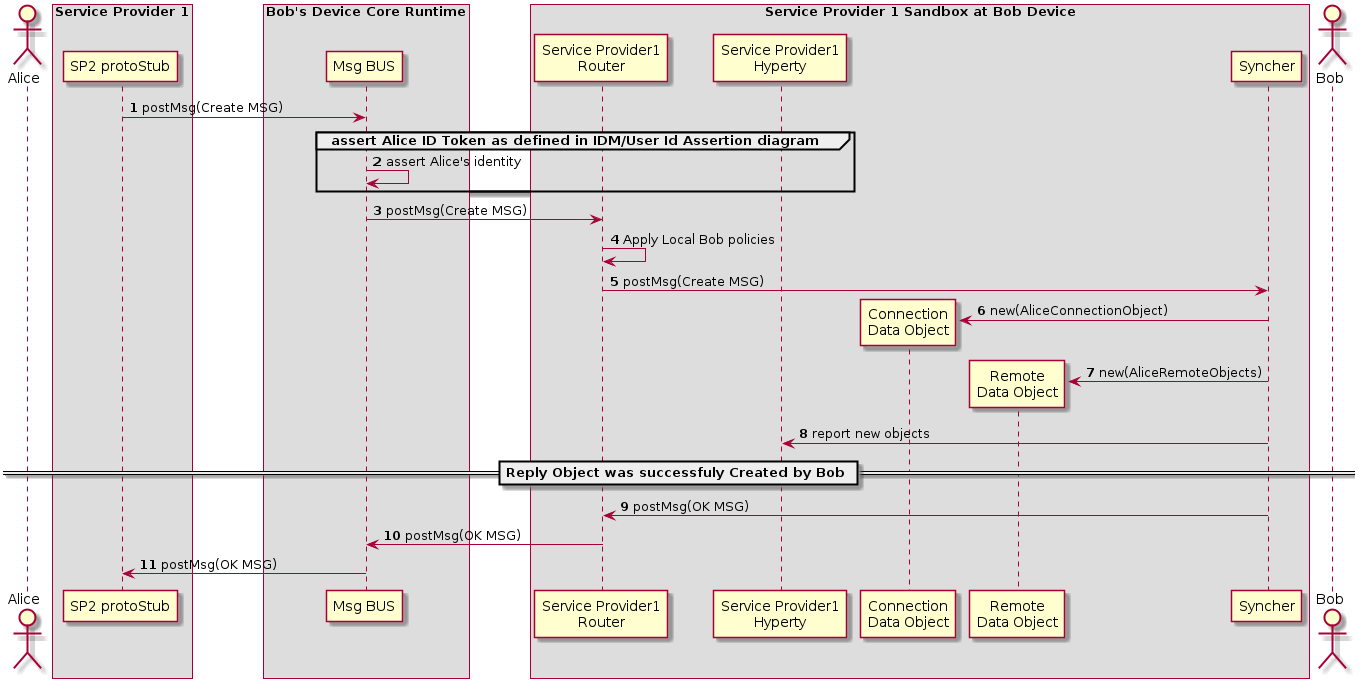
(Step 11) : The message is routed towards Alice Message BUS.

(Step 12) : SP2 protostub is deployed in the runtime if not deployed yet as defined in chapter "Deploy Protocol Stub"

(Steps 13 - 14) : The Message BUS routes the message to the SP2 protocol stub which processes it to send it to Service Provider 2 Back-end Messaging Service.

##### H2H Interdomain Communication - Bob receives invitation

This MSC diagrams shows how Bob receives invitation from Bob.



H2H Interdomain Communication : bob receives invitation

(Steps 1 - 3) : The Service Provider 2 Stub that has been deployed in Alice's Runtime sends the message to Bob's Message BUS, asserts Alice's identity and forwards the message to Bobs Router reaching Bob's PEP component

(Step 4) : Bob's PEP applies local policies if required including incoming communication request access control

(Steps 5 - 8) : the message is forwarded to Bob's Syncher which creates the requested new objects and reports to Bob's Hyperty Instance the new created objects.

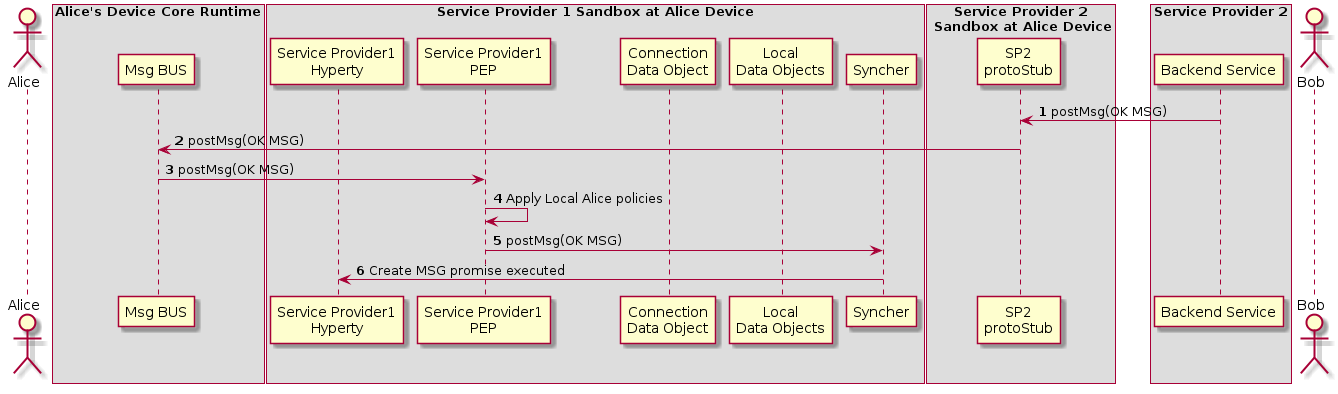
(Steps 9 - 10) : As soon as the new Objects were created by Bob's syncher, it responds back to Alice to confirm the objects were created with a [Response Message](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#responsemessagebody).

"id" : "1"  
"type" : "RESPONSE",  
"from" : "hyperty-instance://sp1/bobhy123",  
"to" : "hyperty-instance://sp1/alicehy123",  
"contextId" : "qwertyuiopasdfghjkl",  
"body" : { "code" : "200" , "description" : "ok"}

(Step 11) : The message Bus sends the message to Alice via the SP2 stub, deployed in Alice's runtime

##### H2H Interdomain Communication - Invitation Acknowledgement

This MSC diagrams shows how Alice is aknowledged that Bob received the invitation



H2H Interdomain Communication : Alice is Aknowledged

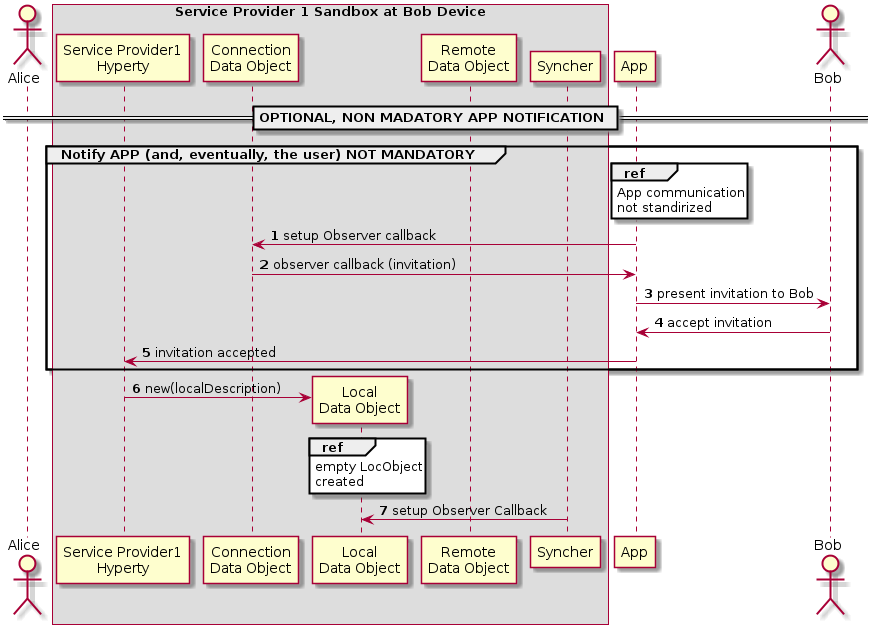
(Step 1 - 3) : Service Provider Back-end Messaginge Service sends the OK Message to via the SP2 protocol Stub to Bob's Message BUS which forwards it to its PEP

(Step 4) : Bob's PEP applies local policies if required

(Steps 5 - 6) : the message is forwarded to Alice's Syncher which updates the Data Object and reports the change to Alice's Hyperty Instance

##### Incoming call is notified to Bob's application and Alice is updated

The sequence for presentation of the call notification to the user is the same as the corresponding sequence for an Intradomain communication.



H2H Interdomain Communication : notification update

(Step 1) : The Application which interacts with the human user setups a callback in to be notified when the Connection data Object is modified.

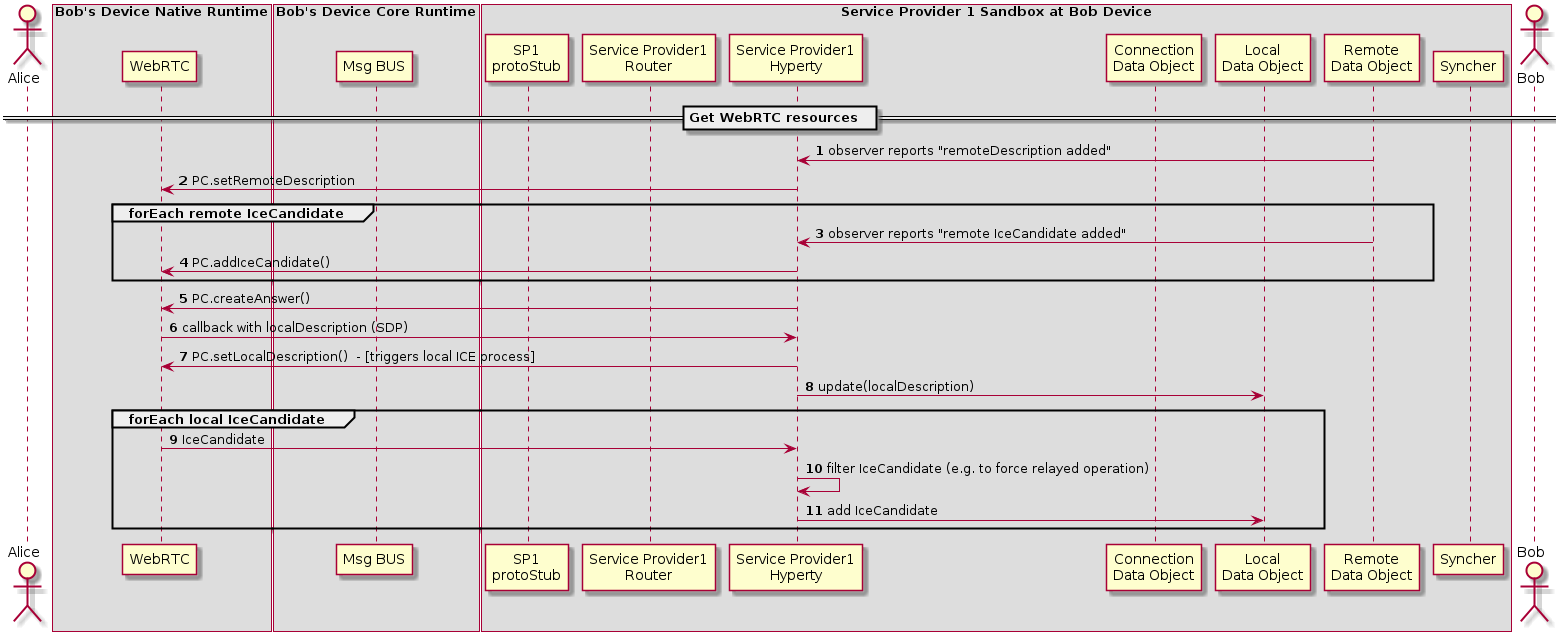
(Step 2) : When a Data Connection Object receives any modification request from another Hyperty, the callback setup in the step before is called. The App is aware of the incoming invitation to establish a media session.

(Step 3) : The App can show this invitation to the human user in some way through a human interface. (Step 4) In such a case the human typically will accept the communication. (Step 5) The App accepts the invitation through the API exposed by the the Service Provider Hyperty. In order to start the media session a Local Data Object is created (Step 10) where the data related to the local parameters of the media session is going to be established.

(Steps 6 - 7) : The Syncher element from the Hyperty setups an Observer callback in the Local Data Object which will be called when the Local Data Object changes. (Step 7) The observer reports that there is a communication in progress to the Syncher.

##### Bob starts WebRTC API

The sequence for the gathering of the WebRTC resources is the same as the corresponding sequence for an Intradomain communication.



H2H Interdomain Communication : Bob gatheres WebRTC resources

(Step 1) : The Hyperty is notified about the added remoteDescription object.

(Step 2) : The Hyperty calls the WebRTC API from the browser including the remote parameters from the Remote Data Object. The same happens when a new Ice Candidate is updated in the Remote Data Object (step 3 and Step 4).

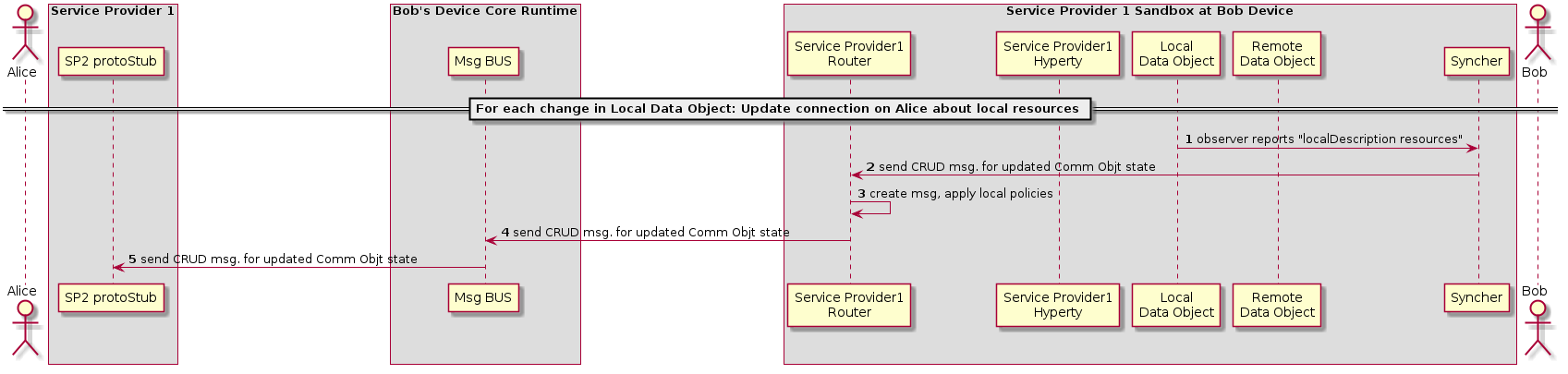
While remote Ice Candidate are added (step 3 and Step 4 may take place several times as Trickle Ice is supported) the Hyperty calls the Peer Connection method to create an SDP answer (step 5) to be sent to it with all the parameters used to establish the media session between Alice and Bob but the Ice Candidates which will be received asynchronously later. When the SDP with the local description is ready a callback is called and the SDP is sent to the Hyperty (step 6).

(Step 7) : The Hyperty calls the Peer setLocalDesciption API method from the WebRTC API exposed by the browser so that the browser is aware of the media parameters which are going to be used to establish the media session with Alice. At this point the gathering process of local Ice Candidates starts.

(Step 8) : The Hyperty updates the Local Data Object with the parameters from the localDescription.

(Step 9) : As a result of the started ICE process local connectivity candidate will be reported from the WebRTC engine to the Hyperty. For each reported localCandidate the Hyperty can optionally perform a filter operation (Step 10), e.g. to filter out non-relay candidates to force TURN based operation, and reports the remaining candidates to the Local Data Object (Step 11)

##### Synchronization of Alice's Data Object



H2H Interdomain Communication : Synchronization of Alice's Data object

(Steps 1 - 2) : The local Data object reports that there have been changes in the connection parameters and the Syncher sends a CRUD message through the Policy Enforcer to Update the Remote Data Object at Alice's Hyperty (Step 2).

(Steps 3 - 4) : The Policy Enforcer checks if the message is compliant with the local policies and forwards the message the Msg Bus (Step 4)

(Step 5) : The message Bus sends the message to Alice via the SP2 stub, deployed in Alice's runtime

## Runtime Implementation Considerations

In this section, some implementation considerations are presented for the different types of runtime platforms that are the target of the Hyperty Runtime specification namely the browser runtime, standalone runtime applications and M2M standalone runtime to be installed in devices with more constrained capabilities. These considerations are mainly about the implementation of the runtime sandboxing solution since all other core runtime components will be shared and common in all target platforms.

### Browser Runtime Implementation

#### General design considerations

The Runtime implementation at browsers plays a central role in reTHINK project. Browsers are almost always present in devices aimed to be used by human beings so using its runtime to execute any application will ensure that it will be correctly interpreted and executed. However, browser's runtime has many security constraints the developer must deal with in order to get a functional web application.

The design of the browser runtime implementation for reTHINK project has been directed by security and functional requirements along as well as the security limitations forced by the browser. Some of the design decissions are expected to be modified during the implementation phase, however all the proposed design has been tested with real code which implemented prototypes of the different parts.

#### Description of the proposed implementation design.

The diagram below shows all the elements presents in the runtime environment in a browser executing we web application which uses hyperties.

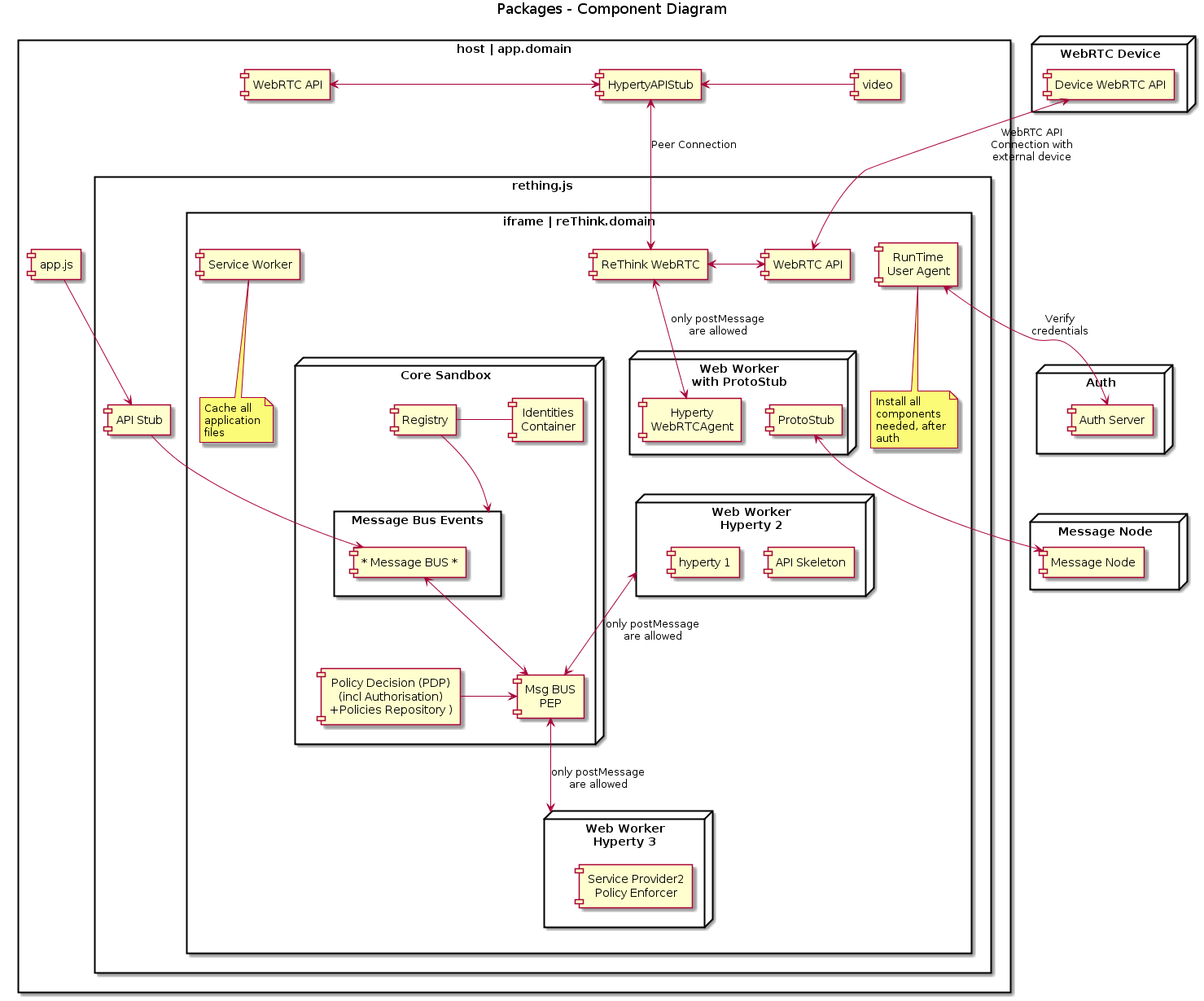


Figure 54: Runtime browser implementation

The web application labeled as *app.domain* represents the html file which is downloaded from the server (hosted by domain which can be an entity different from teh CSP which provides de hyperties).

app.js represents a Javascript file used by app.domain which allows to interact from app.domain with the *rethink.js* library.

The *rethink.js* library contains the Javascript code necessary to setup all the runtime used by reTHINK in the browser. In the next setion all the elements instantiated by rethink.js will be covered. iFrames and Service workers will be used to implement the necessary runtimes.

##### Service workers.

A service worker is a script that is run by your browser in the background, separate from a web page, allowing to execute features which do not need a web page or user interaction. They are used to manage the cache of Runtime Core Components. Web Workers are only able to interact to each other by sending messages with self.postMessage(..) which can be caught by en event listener implemented by the Runtime MsgBUS Core Component.

###### Hyperties and Protocol Stubs

As described in the diagram both Hyperties and Protocol Stubs will be implemented inside Web Workers so they can be executed as separated threads which run independent from the Core runtime. The same Service Worker may also be used to manage the cache of Hyperties and protostubs.

Since it is not possible to use webrtc APIs inside a web worker, there will be a "reTHINK WebRTC" component inside the iFrame but outside the web worker, that is in charge of interacting with the WebRTC API on behalf of Hyperties running inside Web Workers, through messages exchanged between Hyperties and the "reTHINK WebRTC". There will be a "HypertyWebRTCAgent" that will expose standard WebRTC APIs to be used by the Hyperty. In this way the Hyperty is not aware that it is not interacting directly with the native WebRTC API. It should be analysed whether communication between "reTHINK WebRTC" and "HypertyWebRTCAgent" will be supported by the Message BUS or by something else.

The Hyperty API to be consumed by the Application can not be directly used by the App (because it is inside a Web Worker) there will a kind of RPC communication through messages exchanged between the HypertyAPIStub component running on the App side and an API Skeleton running on Hyperty side. It should be analysed whether communication between these components will be supported by the Message BUS or by something else.

#### Runtime Message Bus Core Component.

The Message Bus Core component which will be in charge of listening to messagies comming from the different elements and sending them to the right destionation based on the information included in the message headers. For example, it will capture the events coming from the service workers which implement the hyperties and the protocol stubs by instantiating and event listener: *window.addEventListener('message', handleSizingResponse, false)*.

Attached to the Message Bus there will be a Policy Enforcer which will implement a set of policies to apply to the messages being transported by the bus. It will also determine whether a message is allowed to be sent or not.

#### iFrames

As depicted in the diagram all Runtime Core components, Hyperties and Protocol Stub are executed inside an iFrame loaded from reTHINK runtime provider domain. This the mechanism allows to have a different runtimes for each of them which has been identified as a good security practice as the runtime are isolated. These iFrames are not inteneded to show any content in the Webapp so they will be hidden iFrames.

##### How to send media stream from the reTHINK iFrame to the Web App.

Due to the runtime constraints it is not possible to pass WebRTC Media and Data Streams handled inside the iFrame towards the Application that is outside the iFrame, a local loop peerconnection is established between the "reTHINK WebRTC" and the "HypertyAPIStub" running on Application side.

After some investigation it was found away to send stream from app client to iframe with our domain.An internal loop between peer connection objetcs is used to send to send the media stream between the iFrame where is received from the remote peer and the App which consumes the media coming from the hyperty (it is displayed in and elements).

The performance impact of this technique has not been considered very relevant in the preliminary tests however other alternatives will be considered in case a performance penalty is observed in more complex applications.

###### Practical implementation

The peer getUserMedia from app client and make a call to peer inside the rethink iframe, and this answer with null stream (we send stream one way), after this, peer can send the stream through peer connection to another client.

### Considerations about the implementation of Runtime for standalone applications

A couple of tools have emerged to build native apps using standard web technologies. Among them: - crosswalk - cordova / phonegap / ionic

#### Crosswalk

Crosswalk is a runtime for mobile and desktop web applications. It enables to deploy standard web application for various devices (Android/IOS/Linux). It is based on Chrome and Blink for rendering.

By using the Crosswalk Project, an application developer can:

* Use all the features available in modern web browsers: HTML5, CSS3, JavaScript.
* Access the latest recommended and emerging web standards.
* Use experimental APIs not available in mainstream web browsers.
* Control the upgrade cycle of an application by distributing it with its own runtime.
* Add custom extensions to an application, to leverage platform features not exposed by Crosswalk or the standardized web platform.
* Crosswalk supports WebRTC applications so it makes possible to send and receive real-time flows from Android and iOS devices.

#### Crosswalk Architecture

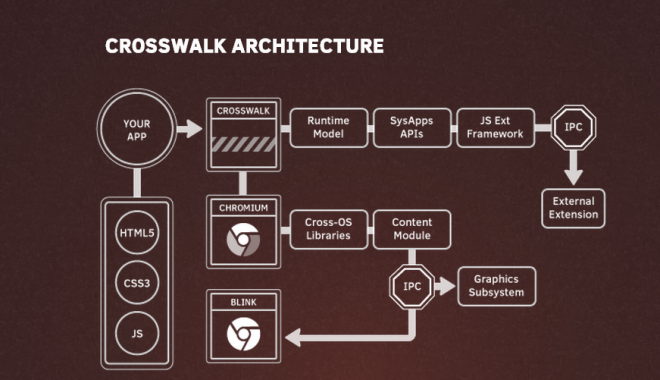


Figure 55: Crosswalk Architecture

Crosswalk supports an efficient way of creating your own Web APIs as extensions by writing native Java code. This way the user can expose new platform and device APIs as they need them. New Api could be available in crosswalk before they get standardized at the W3C level.

#### cordova /Ionic / phonegap

Apache Cordova is a library used to create native mobile applications using Web technologies. The application is created using HTML, CSS and JavaScript and compiled for each specific platform using the platform native tools. Cordova provides a standard set of JavaScript APIs to access device features on all supported platforms. Additional features can be provided through the development of plugins

#### Cordova functionnal schema

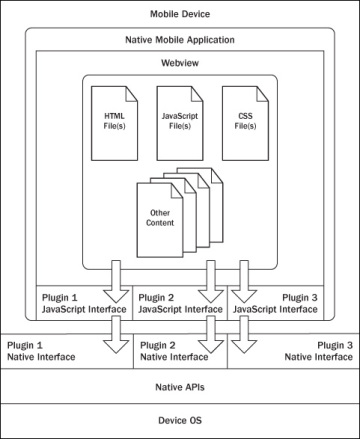


Figure 56: Cordova functionnal schema

The application itself is implemented as a web page, by default a local file named index.html, that references whatever CSS, JavaScript, images, media files, or other resources are necessary for it to run. The app executes as a WebView within the native application wrapper, which you distribute to app stores.

At its core, Cordova offers a simple but powerful API to call Javascript functions that map to native code or plugins. This means you can transfer any kind of data from native land into web land. Cordova can do almost a native app can do, it just needs the right plugins that send the right data to your web code

#### Cordova plugins

A Cordova plugin bridges a bit of functionality between the WebView powering a Cordova application and the native platform the Cordova application is running on. Plugins are composed of a single JavaScript interface used across all platforms, and native implementations following platform-specific Plugin interfaces that the JavaScript will call into. It should be noted that all of the core Cordova APIs are implemented using this exact architecture. Cordova has a high quality plugin API, we just need more great plugins that expose data from the native layer, not just hard coded features or UIs. While the default plugins are very simple and easy to use, they don’t scale well when you want to build something really custom

#### Some plugin examples

##### iosRTC

iosrtc is a wrapper around Google’s WebRTC library and simply provides PeerConnection, getMediaDevices and getUserMedia APIs , without any limitations or artificial constraints.

##### phoneRTC

phoneRTC : https://github.com/alongubkin/phonertc

##### Crosswalk-based Cordova Android

Crosswalk-based Cordova Android is derived from Cordova Android and uses Crosswalk as the HTML5 runtime. It is an Android application library that allows for Cordova-based projects to be built for the Android Platform. It is aimed at replacing default Android Webview with Crosswalk Webview, bringing all new functionalities of Chrome.

This solution has been succesfully used by companies part of the reTHINK project to develop WebRTC hybrid applications so it is a suitable candidate to be used to implement standalone reTHINK applications for Android.

#### Cordova vs PhoneGap

Cordova is the community powered version of PhoneGap, which is Adobe’s productized version and ecosystem on top of Cordova.

#### Cordova vs Ionic

Ionic uses and extends Cordova

#### Webview

The WebView class is an extension of Android's View class that allows you to display web pages as a part of your activity layout. It does not include any features of a fully developed web browser, such as navigation controls or an address bar. All that WebView does, by default, is show a web page. This allows to leverage features provided by the browser engine in any App without adding extra libraries.

Since Android 4.4 (KitKat), the WebView component is based on the Chromium open source project. WebViews now include an updated version of the V8 JavaScript engine and support for modern web standards previously missing in old WebViews. New Webviews also share the same rendering engine as Chrome for Android, so rendering should be much more consistent between the WebView and Chrome.

In Android 5.0 (Lollipop), the WebView has moved to an APK so it can be updated seperately to the Android platform.

##### Webview WebRTC support

From WebView v36 WebRTC is supported so it makes easier to add WebRTC capabilities to any native. Webview 36.0.0.0 is still a developer preview version so it can not be used in official Apps currently but it is expected to become soon the stable release..

##### Crosswalk vs Webview

The size of the apps is lower compared to Crosswalk applications which must include all the libraries to implement the browser functionality. The WebView can be updated separately from the rest of the application. This can be an advantage as it will allows to fix any kind of issue and support new features, but it may cause issues if the App using it is not updated to fix any possible incompability.

One of the obvious drawbacks is that Webview is not available in iOS.

#### OpenWebRTC

OpenWebRTC is an open sourced project from Ericsson Research : https://github.com/EricssonResearch/openwebrtc

A flexible, mobile-first, cross-platform WebRTC client framework based on GStreamer. OpenWebRTC currently supports iOS, Android, Mac OS X and Linux.

#### Selected solutions for the implementation:

##### Solutions that have already been tested :

###### Android :

Crosswalk : integrate chromium in the application with different possible integration : - Crosswalk embedded in the application - Crosswalk cordova plugin

Crosswalk usage should ensure us a compatibility with what is done for browser runtime as it embed Chromium

###### iOS :

iOSRTC, cordova plugin : https://github.com/eface2face/cordova-plugin-iosrtc

Usage of Cordova will enables us to reuse the components that will be developped on the browser runtime.

###### Android & iOS :

Crosswalk and iosRTC can be embeded in the same application code to support both platform.

Hybrid solution will be selected for the project as it enable to use JavaScript for the runtime

##### Solution to be tested during the implementation :

* Usage of Webviews will be interesting as it should facilitate the integration of WebRTC API.
* openWebRTC can also be insteresting as it should enable the possibility to build complete native and hybrid application.

### Runtime implementation M2M standalone application

NodeJs is considered one of the options for implementing the Runtime API for platforms like Raspberry PI and [Beagle Board](http://beagleboard.org/bone):

#### NodeJs Installation

For installing NodeJs on Raspberry Pi, 2 steps are required: download the debian package and then install it (http://weworkweplay.com/play/raspberry-pi-nodejs/)

wget http://node-arm.herokuapp.com/node\_latest\_armhf.deb   
sudo dpkg -i node\_latest\_armhf.deb

For installing NodeJs on BeagleBoard (http://beagleboard.org/Support/BoneScript) one can compile it from scratch (http://www.armhf.com/node-js-for-the-beaglebone-black/) or install it in a similar way as for Raspberry using one of the versions from the download page: http://www.armhf.com/download/

An important package based on NodeJs is Cylon (http://cylonjs.com/) supporting 36 hardware platforms and providing APIs to interact with sensors or actuators of the platforms.

#### Design

The goal of the design is to use stable NodeJs open-source or business friendly modules that provide functionality for the components that are part of the architecture of the Runtime.

One of the key functional requirements is security of the Runtime. Thus multiple sandboxes to separate code is present in the Runtime architecture as a security by design feature. There are 3 types of sandboxes to be used: Core Sandbox, Service Provider Sandbox and Hyperty Sandbox (http://gf3.github.io/sandbox/).

For the Runtime UA a module implementing the protocol LWM2M is already available for NodeJs (https://github.com/telefonicaid/lwm2m-node-lib). Special care has to be taken into consideration for having one implementation of the Runtime UA that can also run on the other platforms: browser and H-2-E (Human to Everything) standalone.

#### Code Snippets

For creating several sandboxes the following code can be used:

var s = new Sandbox()  
s.run( '1 + 1 + " apples"', function( output ) {  
 // output.result == "2 apples"  
})

with the basic syntax: sandbox\_instance.run( code, hollaback ), where

code is the string of Javascript to be executed.

hollaback is a function, and it's called with a single argument, output.

output is an object with two properties: result and console. The result property is an inspected string of the return value of the code. The console property is an array of all console output.

#### Other evaluated runtimes

Another platform that was evaluated was IotJs (http://samsung.github.io/iotjs/). It currently supports Raspberry Pie2 and STM32F4-Discovery + BB as hardware platforms and Linux and Nuttx(http://nuttx.org/) Real-Time Operating System using C++ to build the runtime Javascript Environment. Although supported by an important device manufacturer, it is still in its infancy and cannot be used in the development of reThink in which fast-prototyping of new paradigms is intended. During the development allignment with the iotJs is considered and tests of the components to validate the support of ioJs is envisioned.

# Message Node Specification

This Chapter contains the functional design of the Message Node Architecture which enables messaging communication among Hyperty instances running in different Runtime devices.

Since the protocol-on-the fly concept is used together with the message model defined in D2.2, it is not required to specify in detail the Messaging Node APIs to guarantee interoperability between different domains. Instead, a more detailed specification is provided for each messaging solution selected during the procurement activity namely for Vertx.io, Nodejs and Matrix.

## Messaging Node Architecture

The Messaging Node functional architecture is presented in the figure below and it comprises three main types of functionalities including the Core Functionalities, Connectors and Protocol Stubs.



Figure 57: Messaging Node Architecture

### Core Functionalities

#### Message BUS

The Message BUS routes messages to internal Messaging Node components and external elements by using Connectors or Protocol Stubs. It supports different communication patterns including publish/subscribe communication.

#### Access Control

Message Routing including pub/sub Subscriptions are subject to Access Control in cooperation with authentication and authorisation provided by Identity Management functionalities.

#### Session Management

Session Management functionalities are used to control messaging connections to service provider back-end services. For example, when user turns-on the device and connects to its domain, providing credentials as required by Identity Management functionalities. In general, each message should contain a valid token that is generated when the client connects to the Messaging Node. It also manages the registry of protocol stubs and connectors supported by the Messaging Nodes to support the routing of messages to these components.

#### Address Allocation Management

The Address Allocation Management functionality handles the allocation of messaging addresses to Hyperty Instances in cooperation with Session Management when users connect to the domain.

It also manages the allocation of messaging addresses to foreign Hyperty Instances i.e. Hyperty Instances that are provided from external domains but that use the protofly concept to interact with Hyperty Instances served by this Messaging Node. For example, if the Messaging Node is implemented by core IMS or a simple SIP Proxy/SIP Registry, it is required the management of a pool of SIP addresses to be allocated to clients that have no account in the IMS HSS or in the SIP registry.

### Protocol Stub

In special situations e.g. when the download of external software (protocol stubs) into end-user devices is not allowed, it should be possible to have interoperability between Messaging Nodes from different domains, by using the protofly concept.

Thus, a Protocol Stack to be used to communicate with another Messaging Node can be deployed.

It should be noted that protocol stubs can also be used to implement a Messaging Node connector, in case it does not exist.

### Connectors

Connectors implements protocol stacks used to interoperate with external elements from the domains, including:

* IdM Connector to interact with remote Identity Management functionalities
* Registry Connector to interact with remote Registry functionalities
* End-User Device Connector to interact with Hyperty Instances running in the end-user device
* Network Server Connector to interact with Hyperty Instances running in a Network Server

As mentioned above, Connectors can be supported by using protocol on-the-fly concept, giving more flexibility for the integration of the Messaging Node in the Service Provider infra-structure.

## Vertx Specification

### Core Functionalities

* Main objective of core functions are to **connect**, **intercept**, **process**, **filter** and **deliver** messages. Messages are JSON objects that should have 2 blocks, HEADER and BODY, and are processed from different components of core. Inbound messages should be intercepted and processed in the Pipeline before deliver in to the Message Bus.
* Pipeline components will implement a simple interface that we can reuse from io.vertx.core.Handler<E> replacing E with a PipelineContext object. Using the vertx Handler has the advantage to be compatible with io.vertx.ext.web.Router, that can be a replacement for the Pipeline.
* Outbound messages should be processed in a Pub/Sub system. If message BODY block are for CRUD operations, there should be a Pub/Sub protocol for object/model subscriptions, where should this be processed? The address scheme of the vertx EventBus is not enough for this functionality. We need to control the Pub/Sub functionality better than what vertx provides with the address scheme! Hyperties need to subscribe to objects/collections not just addresses.

#### Pipeline

Pipeline functionality is to **intercept**, **process** and **filter**. The Pipeline configuration can reflect the concept of activity diagrams, controlling the path flow of the message that is dependent of the message type. This concept is generic enough to contemplate different message flows in the future. This is a new component to be developed which is similar to vertx Router but without the URL addressing scheme. The io.vertx.ext.web.Router class could be a possible candidate for Pipeline functionalities, however the Router is hard coded to work with HTTP protocols, and there is no need for static configurations of routing schemes. The alternative is to implement a simple Pipeline system instead of using the Router, less dependencies and better decoupled from the protocol.

#### Session Management

Session Management is one of the Pipeline handlers that will intercept messages and verify the sessionID. A session instance is linked to a connection resource (WebSocket, SockJS) if authorized. Every message header is intercepted, session token is verified and if exist, a "user" or other identification URL is replaced in HEADER. The JSON object is forwarded to "Access Control" handler.

#### Address Allocation Management

This is not a Pipeline handler (it doesn't process messages), but it's used by the "Session Management" to allocate Hyperty identification URL's that will be linked to a Session when the Hyperty is connected. This will be used to translate Hyperty an URL address into the correspondent Connector Resource.

#### Access Control

This handler is able to analyze the HEADER (identification URL from "Session Management") and BODY blocks and decide if the message should be forwarded to the "Message Bus" or denied. There is a possibility to add a rule engine in this step, but it's not specified for now, what kind of rule engine.

#### Message BUS

Main objective of the MB is to **deliver** the message, being independent of the cluster node that has the connection to the destination. Vertx EventBus can be used directly for the Message Bus component. Important headers of the original JSON (like the identification URL) must be forwarded to io.vertx.core.eventbus.Message.headers() map.

### Protocol Stub Sandbox

The protocol Stub sandbox will be managed by a ProtocolStubManager class that loads, registers and removes protocol stubs on request. If ProtoStubs are in JavaScript, the sandbox model could be implemented using the java NashornScriptEngineFactory and controlling the available API's with ClassFilter.

### Connectors

#### End User Device Connector

The aim of this Connector is to enable interaction with Hyperty instances running in the end-user device. This component will need to interact somehow with the Protocol Stub sandbox to achieve this, since the communication protocol will not be standardized. It will need to implement a simple protocol for sending and receiving requests. In itself it is not responsible for processing communication requests, that is left to the protocol stack. It merely forwards messages to and from the Hyperty instance.

#### Network Server Connector

The aim of this Connector is to enable interaction with Hyperty instances running in a network server. This component will need to interact somehow with the Protocol Stub sandbox to achieve this, since the communication protocol will not be standardized. It will need to implement a simple protocol for sending and receiving requests. In itself it is not responsible for processing communication requests, that is left to the protocol stack. It merely forwards messages to and from the Network server.

#### Registry Connector

The Registry provides an interface for registration and deregistration of Hyperty instances, as well as for keeping the published information up to date. For each Hyperty instance, the Registry stores data (hyperty location, type, description, start-time, presence information of user) that enables other applications to contact it. The implementation of the Registry service is thought to be basically a distributed database. It will provide service interfaces for CRUD operations to allow users to retrieve data for a given GraphID, publish (i.e. create, update, and delete) their own information on the ring. To verify authenticity and integrity of the published data, digital signatures will be applied. The Connector will exposed the available interfaces of the Registry Services to users of managing Hyperty instances. This will have to be implemented as a standalone application with an adapter interface to the Event Bus for encoding and decoding messages and deployed as a fat executable jar which contain all the dependencies it needs to run on vertx.

#### IdM Connector

This Connector is to provide functionalities for interacting with the remote Identity Management Functionalities. As hyperties need to be linked to an end-user identity when downloaded and instantiated on a device, an Identity Module should be present on the device. This module at minimum should act as an identity selector for the user and as a secure local repository for identity tokens provided by IdPs

If the connector is thought to provide authentication and authorisation, Vert.x offers Auth APIs (Common, JDBC, JWT and Shiro).

There is also a library for authentication and discorvery, [vertx-pac4j] (https://github.com/pac4j/vertx-pac4j). This vertx module provides multiple authentication mechanisms (OAuh, CAS, HTTP, OpenID, SAML2.0 and OpenIDConnect) for different IdPs.

## NodeJs based Messaging Node Specification

For each [functional block](msg-node-architecture.md) the WP3 team has identified existing nodeJs modules which can be either reused or extended.

### Core Functionalities

This section attempts to match the functional blocks of the Message Node architecture to features and functional blocks of the nodeJs and Redis architecture.

#### Message BUS

The message bus can be implemented with Redis. http://redis.io

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.

##### Usage of Redis with NodeJs

Redis integrate a PUB/SUB mechanism : http://redis.io/topics/pubsub

SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm where (citing Wikipedia) senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

Redis can be used to add scalability/redundancy to the messaging node as the different components of the architecture can easily be splitted on different servers. This Pub/Sub mechanism is simple to use and It can also facilitate the development and the integration of new connectors

Communication between NodeJs and Redis can be managed by a NodesJs Redis client module : https://github.com/NodeRedis/node\_redis

Redis instance can be a single instance or a Redis cluster.

#### Access Control

User connection to NodeJs connectors can be authentified on the NodeJs module. Socket.io integrate a way to authenticate incoming request, authenication component will have to be develop on NodeJs connectors.

This component is able to analyze HEADER (identification URL from "Session Management") and DATA blocks and decide if the message should be forwarded to the "Message Bus" or denied.

PassportJs, which is an intesreting middleware, that could enable us to add third party authentication should be used : http://passportjs.org/

An authentication can also be done between NodeJs and Redis.

#### Session Management

For a complete session management on NodeJs, it will be interesting to use express which is a Web framework for NodeJs : http://expressjs.com/

#### Address Allocation Management

This component will have to be developped on a NodeJs server

#### Protocol Stub & Connectors

Connectors will be NodeJs process to be developped.

Goal will be to mutualize connectors by using the protoStub/protoFly mechanism : this will add flexibility to connect other GWs, CSP ...

##### IdM Connector

This Connector is to provide functionalities for interacting with the remote Identity Management Functionailities. Node.js can easily interact with OAuth servers in order to authenticate and authorize users.

It this is for authentication purpose the authentication agqinst the IdP has to be done at the begining. If the CRUD operations have to be authorized on a per identity basis (e.g. user A, correctly authenticated, is only allowed to do 'RU' over a Data Objet) we should get

##### Registry Connector

The Registry provides an interface for registration and deregistration of Hyperty instances, as well as for keeping the published information up to date. For each Hyperty instance, the Registry stores data (hyperty location, type, description, start-time, presence information of user) that enables other applications to contact it. The implementation of the Registry service is thought to be basically a distributed database. It will provide service interfaces for CRUD operations to allow users to retrieve data for a given GraphID, publish (i.e. create, update, and delete) their own information on the ring. To verify authenticity and integrity of the published data, digital signatures will be applied. The Connector will exposed the available interfaces of the Registry Services to users of managing Hyperty instances.

##### End-User Device Connector

Communication between Users and NodeJs can be managed by socket.io Socket.io is a popular Node.js library to handle connections at application level. It can use Websocket and it falls back to HTTP automatically if WS connectivity is not possible.

##### Network Server Connector

The aim of this Connector is to enable interaction with Hyperty instances running in a network server. This component will need to interact somehow with the Protocol Stub sandbox to achieve this, since the communication protocol will not be standardized. It will need to implement a simple protocol for sending and receiving requests. In itself it is not responsible for processing communication requests, that is left to the protocol stack. It merely forwards messages to and from the Network server.

##### Node Sandbox framework

[Node-sandbox](https://www.npmjs.com/package/node-sandbox) allows to run untrusted code outside of the main node process. The code can be interfaced with code running in the sandbox via RPC (or any library that works over the node Stream API).

### NodeJs implementation architecture

**Architecture : NodeJs and Redis :**

Here is decription of the architecure with Redis :

**Architecture : Integration in ReThink :**

Following architecture shows the target integration with the different components of the ReThink projet :

**Architecture : Integration in ReThink with Actors:**

Following architecture shows the actors in the architecture to unsderstand the decomposition of work to be done and the interaction with other partners :

## Matrix.org based Messaging Node Specification

This section matches the requirements for the functional blocks of the Message Node architecture to features and functional blocks of the matrix.org architecture. Functional gaps are identified and proposals for extensions to the standard Matrix.org Homeserver are made in order to fill these gaps.

### Protocol Stub and Connectors

Protocol Stubs and Connectors are means to make a Messaging Node interoperable with foreign signalling protocols.

A protocol stub is the core entity of the Protocol-on-the-fly concept. It is a downloadable piece of JavaScript code that is executed in the client's runtime and performs the required adaptations on the messaging protocol. In a Protocol-on-the-fly based communication relation there is always one side in the client role (i.e. the side that downloads the stub) and the other side in the server role (the side that the stub connects to).

A Messaging node has to support both operation modes in order to provide full bi-directional interoperability.

#### Matrix as Protocol-on-the-fly client

The most appropriate feature that Matrix provides for this purpose is the concept of "Application Services". An Application Service is an implementation of a special service function that can be attached to a Homeserver (HS). Based on certain patterns, messages are filtered and forwarded to the Application Service that performs application specific tasks. This concept is quite comparable to Application Servers in the IP Multimedia Subsystem (IMS) framework. It can, for example, be used for aggregation and accounting purposes, but also for the implementation of "breakout" communication to other types of messaging infrastructures it is well suited.

A special dedicated Application Service is proposed that will implement a Protocol-on-the-fly client engine to allow the "breakout" to different signalling domains that provide a Protocol-on-the-fly stub. Such an Application Service will be a very flexible mechanism for interdomain collaboration.

#### Matrix as Protocol-on-the-fly server

In order to support the server role in the Protocol-on-the-fly architecture, a specialized Matrix protocol stub needs to be implemented that connects to a Homeserver. Since the Matrix Homeserver has a well-documented API and the Matrix message format allows the transport of arbitrary payload, this implementation should be straight forward. The implementation can make use of the SDK's that are available for Matrix client developers. These SDK's encapsulate a lot of the internal complexity for REST based communication.

#### Connectors in Matrix

Connectors also play the role of protocol adapters, which makes them comparable to protocol stubs. The difference is that they are not downloaded to the Messaging Node clients. Instead they are executed in the scope of the Messaging Node itself. Such Connectors are intended to connect with different "legacy" clients that don't support the Protocol-on-the-fly concept.

Also for the implementation of such connectors the concept of Application Services seems well suited. The matrix.org developer community has implemented this as a proof of concept that connects the Matrix ecosystem with the Internet Relay Chat (IRC) world. Messages that contain a specially prefixed address are filtered out, converted to IRC messages, forwarded to the corresponding IRC client and vice versa. This can be used as pattern for the implementation of additional adapters.

### Core Functionalities

#### Message Bus

The Message Bus is responsible for the routing of messages to internal Messaging Node components and external elements by using Connectors or Protocol Stubs. This routing shall support different communication patterns including publish/subscribe communication.

These main routing requirements are fulfilled out-of-the-box by standard matrix features. In order to route messages to internal Messaging Node components it will be required to provide such components with virtual identifiers that can be used internally to address them.

#### Access Control

The main task of the access control is to enforce manageable policies to the forwarding of individual messages. For example, a single type of message shall be blocked if a special combination of sender and/or receiver matches.

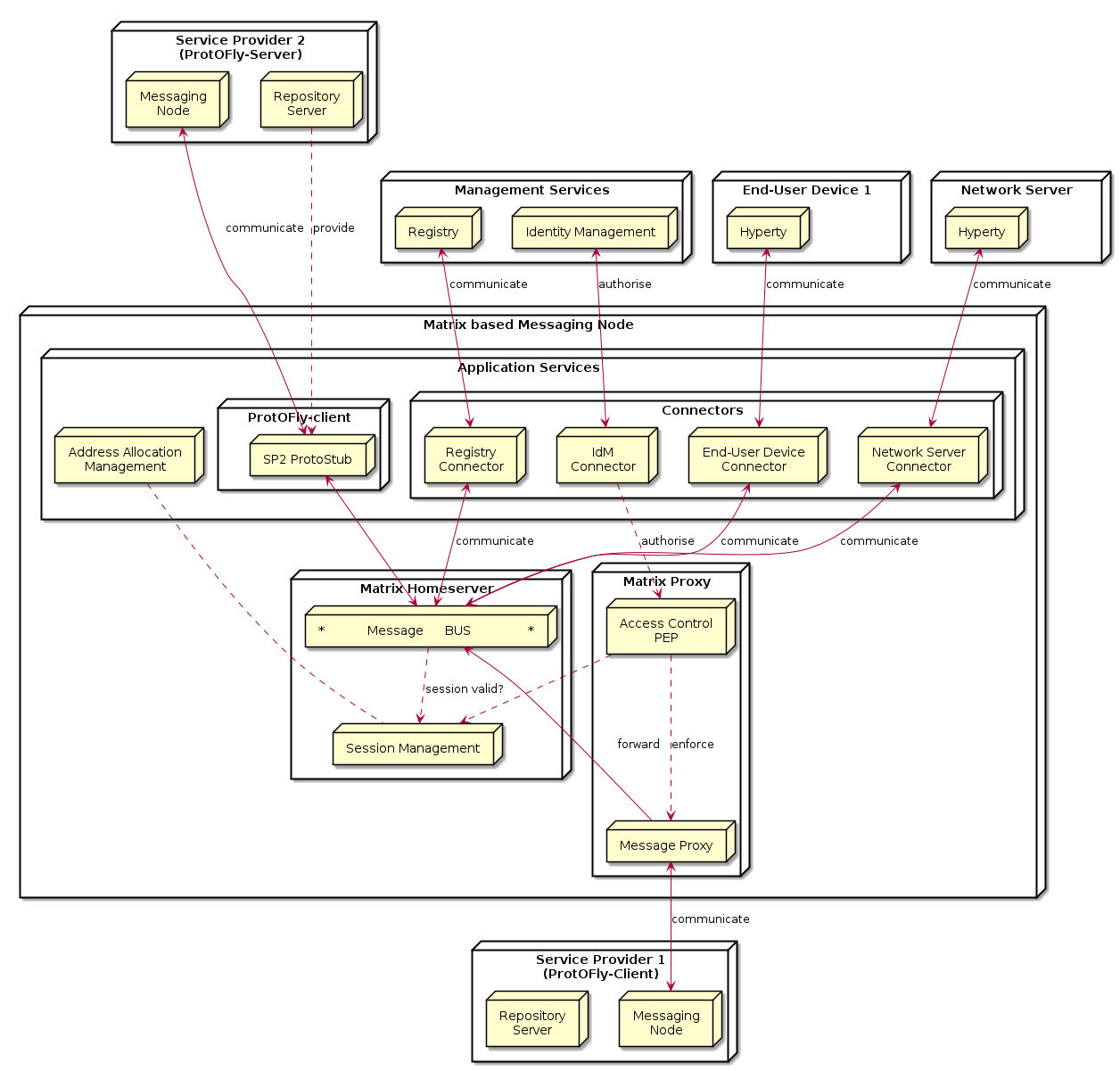
Matrix.org requires registration/subscription and login of users in order to exchange any messages with other users. These authentication and authorisation methods however always apply to a complete user- and communication session, that means to ALL messages that are exchanged in a session scope. This concept does not provide an access control on a "per message" base.

The matrix developer community already discusses the integration of a "policy service", but so far this integration in not yet specified.

In order to achieve a "per message"-policy enforcement without deeper changes in the matrix core, we propose the introduction of a message proxy as first step of the message flow. This proxy has the task to check the messages and to apply the policies. It would forward messages according to the policies and should reject the rest. A potential bypassing of this proxy must be avoided by appropriate network configurations.

The design of this message proxy component should be closely coordinated with the MessagingStub that is used to connect to this Matrix based Messaging Node, because the proxy will be the first contact point for the stub.

The following figure gives an overview of the intended architecture of the Matrix based Messaging Node.



Matrix Messaging Node Architecture

#### Session Management

The requirements regarding session management as described in the Messaging Node architecture can be separated in three aspects which are handled in the following sub-chapters: \* User session control, \* Communication session control, and \* Stub and connector management.

##### User session control

In order to use matrix based messaging users have to be registered/subscribed with a matrix HomeServer. Matrix provides an API for the subscription of new users with their HomeServers. This API can be used to provision accounts also programmatically, when required.

In order to establish a communication session with other peers, users have to pass a login sequence. During this sequence an access token is generated which is valid for this login session. This access token must be present in all sub-sequent requests during this user session. No mandatory authentication methods are specified. This is left as implementation specific for the particular HomeServers. The specification lists following standard methods:

* m.login.password,
* m.login.recaptcha,
* m.login.oauth2,
* m.login.email.identity, and
* m.login.dummy.

The HomeServer Client API provides means to request the supported methods before login.

##### Communication session control

Communication sessions between two or more users require a valid user session. Communication sessions are always based on "rooms". Each room is identified by a unique room-id. Messages are sent to room-ids and not to individual users. Users must explicitly create or join rooms in order to send and receive messages. Some rooms might be open - others may require an invitation by the creator of the room. Rooms are persistent, i.e. they exist also if not all room members are currently logged in. The message history is maintained by the Matrix HomeServers and can be requested by clients.

##### Stub and connector management

Matrix.org provides powerful means to connect, federate, and synchronise Matrix HomeServers from different domains. The resolution of the peer HomeServers connectivity is done via DNS. The message exchange between them is secured by encryption mechanisms.

However - for the interoperability with non-Matrix infrastructures there is no "golden" way. The selected and most appropriate approach is via Application Services, as described before.

The "Stub and connector management" function is responsible for the management of the Application Services that implement the Protocol-on-the-fly clients and the connectors.

#### Address Allocation Management

In order to be addressable each hyperty instance should be treated as an individual client of the Messaging Node that registers with an own identity and needs a login before it can exchange messages. The Messaging Node allocates the identity of a hyperty during the registration/subscription process. The allocated identity serves then as a messaging address for domain internal communication.

External Hyperties from foreign domains (that might use different communication protocols and identifiers) will need an address representation in the Matrix domain that is compatible with the local addressing scheme. The Messaging Node is responsible for the creation and assignment of such transient addresses for domain external entities.

Since we have identified Application Services as the most appropriate way of connecting to other signalling domains, also the management of such virtual transient addresses is in the responsibility of the corresponding Application Service. Each Application Service itself has to maintain an own namespace of virtual users and must be able to operate (send/receive) "on behalf" of such a virtual user.

# Conclusions

This report provided a detailed specification of reTHINK Core Framework that comprises the Hyperty Runtime, where Hyperties are executed and the Message Node, which supports the messaging communication among Hyperty instances running in different devices.

The current specification aims to promote a rapid and iterative prototyping of reTHINK Core Framework with optimised usage of resources, in order to provide in time, the required components to start the implementation of scenarios in WP5.

# References

[1] - [Barth, A.; Jackson, C.; Reis, C. and Team, Google Chrome. 2008. The Security Architecture of the Chromium Browser.](http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf)

[2] - [Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the Google Chrome extension security architecture. In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA.](http://nicholas.carlini.com/papers/2012_usenix_chromeextensions.pdf)

[3] - [Garcia-Alfaro, J. and Navarro-Arribas, G. 2007. A Survey on Detection Techniques to Prevent Cross-Site Scripting Attacks on Current Web Applications., in Javier Lopez & Bernhard M. Hämmerli, ed., 'CRITIS' , Springer, , pp. 287-298 .](http://eprints.uoc.edu/research/bitstream/10363/605/1/JGA01.pdf)

[4] - [Scott, D. and Sharp, R. Abstracting application-level web security. 11th Internation Conference on the World Wide Web, pp. 396–407, 2002.](http://rich.recoil.org/publications/websec.pdf)

[5] - [Pietraszeck, T. and Vanden-Berghe, C. Defending against injection attacks through context-sensitive string evaluation. Recent Advances in Intrusion Detection (RAID 2005), pp.124– 145, 2005.](http://tadek.pietraszek.org/publications/pietraszek05_defending.pdf)

[6] - [Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. 21st ACM Symposium on Applied Computing, 2006.](https://iseclab.org/papers/noxes.pdf)

[7] - [Ismail, O., Etoh, M., Kadobayashi, Y., and Yamaguchi, S. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. 18th Int. Conf. on Advanced Information Networking and Applications (AINA 2004), 2004.](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1283902&abstractAccess=no&userType=instima)

[8] - [Hallaraker, O. and Vigna, G. Detecting Malicious JavaScript Code in Mozilla. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’05), pp.85–94, 2005.](http://www.cs.ucsb.edu/~vigna/publications/2005_hallaraker_vigna_ICECCS05.pdf)

[9] - [Jovanovic, N., Kruegel, C., and Kirda, E. Precise alias analysis for static detection of web application vulnerabilities. 2006 Workshop on Programming Languages and Analysis for Security, pp. 27–36, USA, 2006.](https://iseclab.org/papers/pixy2.pdf)

[10] - [Jim, T., Swamy, N., Hicks M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. International World Wide Web Conferencem, WWW 2007, May 2007.](http://www2007.org/papers/paper595.pdf)

[11] - [Uwe Hansmann, Martin S. Nicklous, Frank Seliger, and Thomas Schaeck. 1999. Smart Card Application Development Using Java (1st ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA.](http://dl.acm.org/citation.cfm?id=555354)

[12] - [Pascal Urien. Cloud of Secure Elements Perspectives for Mobile and Cloud Applications Security. IEEE Conference on Communications and Network Security 2013 - Poster Session](http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6682733)

[13] - [Wojciech Mostowski and Erik Poll. 2008. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications (CARDIS '08), Gilles Grimaud and François-Xavier Standaert (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16. DOI=10.1007/978-3-540-85893-5\_1 http://dx.doi.org/10.1007/978-3-540-85893-5\_1](http://www.cs.ru.nl/E.Poll/papers/cardis08.pdf)

[14] - [Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11). IEEE Computer Society, Washington, DC, USA, 363-378. DOI=10.1109/SP.2011.39 http://dx.doi.org/10.1109/SP.2011.39](http://www-cs-students.stanford.edu/~ataly/Papers/sp11.pdf)

[15] - ReTHINK Deliverable D2.2 “Data Models and Interface Specification of the Framework ”, 30-09-2015

[16] - http://w3c.github.io/webrtc-pc/

[17] - http://w3c.github.io/mediacapture-main/

[18] - http://www.webrtc.org/

[19] - http://www.openwebrtc.org/

[20] - http://gstreamer.freedesktop.org/

[21] - https://developers.google.com/v8/

[22] - https://nodejs.org/en/

[23] - https://www.docker.com/

[24] - https://www.mozilla.org/en-US/firefox/os/2.0/

[25] - https://wiki.mozilla.org/WebAPI - Firefox Web-API status. (Last Update March 2015)

[26] - https://jitsi.org/Projects/JitsiVideobridge

[27] - http://xmpp.org/

[28] - http://www.kurento.org/

[29] - https://janus.conf.meetecho.com/

[30] - [Alessandro Amirante, Tobia Castaldi, Lorenzo Miniero, Simon Pietro Romano. 2015. Performance analysis of the Janus WebRTC gateway. In Proceedings of the 1st Workshop on All-Web Real-Time Systems](http://dl.acm.org/citation.cfm?id=2749223)

[31] - [Janus: a general purpose WebRTC gateway](http://www.rtc-conference.com/wp-content/uploads/gravity_forms/2-2f7a537445fa703985ab4d2372ac42ca/2014/09/Romano_Janus.pdf)

[32] - P. Chainho, et Al, FP7 Open Lab Deliverable D4.15, WONDER Assessment Report, April 2014

[33] - Paulo Chainho, Kay Haensge, Steffen Druesedow, Michael Maruscheke, “Signalling-On-the-fly: SigOfly, WebRTC Interoperability testbed in contradictive Deployement Scenarios”, Proc. 18th Int’l Conf. Intelligence in Next Generation Networks (ICIN), 2015.

[34] - https://github.com/hypercomm/wonder/wiki/Signalling-on-the-fly

[35] - https://raw.githack.com/hypercomm/wonder/master/docs/api/index.html

[36] - https://github.com/hypercomm/wonder/tree/master/src/libs

[37] - https://raw.githack.com/hypercomm/wonder/master/docs/api/symbols/MessagingStub.html

[38] ReTHINK Deliverable D2.1 “Framework Architecture Definition”, 31-07-2015.

[39] - [Meteor](http://docs.meteor.com/#/full/quickstart)

[40] - [Cookbook MVC](https://github.com/awatson1978/meteor-cookbook/blob/master/cookbook/model-view-controller.md)

[41] - [Meteorpedia](http://www.meteorpedia.com/read/Why_Meteor)

[42] - [AngularJS vs. Backbone.js vs. Ember.js](https://www.airpair.com/js/javascript-framework-comparison)

[43] - [Why Meteor] (http://www.meteorpedia.com/read/Why\_Meteor)

[44] - [Most Popular JavaScript Frameworks 2015](http://www.improgrammer.net/most-popular-javascript-frameworks-2015/)

[45] - [Peering through WebRTC with SocketPeer](https://hacks.mozilla.org/2015/04/peering-through-the-webrtc-fog-with-socketpeer/)

[46] - [Web Components] (http://www.w3.org/wiki/WebComponents/)

[47] - TURN rfc, https://tools.ietf.org/html/rfc5766

[48] - STUN rfc, https://tools.ietf.org/html/rfc5389

[49] - IETF TRAM, https://datatracker.ietf.org/wg/tram/documents/

[50] - coturn, https://github.com/coturn/coturn

[51] - [AngularJS](https://angularjs.org/)

[52] - [BackboneJS](http://backbonejs.org)

[53] - [StapesJS](https://hay.github.io/stapes/)

[54] - http://en.wikipedia.org/wiki/Real-time\_database

[55] - http://www.leggetter.co.uk/real-time-web-technologies-guide/

[56] - http://www.matrix.org/

[57] - http://vertx.io/vertx2/

[58] - http://vertx.io/

[59] - https://www.rabbitmq.com/

[60] - http://www.amqp.org/

[61] - http://mqtt.org/

[62] - http://www.psyced.org/

[63] - http://redis.io/

[64] - http://xmpp.org/

[65] - http://zeromq.org/

[66] http://www.w3.org/2012/sysapps/app-lifecycle/

[67] https://lists.w3.org/Archives/Public/public-sysapps/2015Apr/0001.html

[68] https://www.w3.org/community/trustperms/

[69] https://whatwg.org/

[70] http://www.w3.org/TR/CSP2/

[71] https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing\_Content\_Security\_Policy

[72] http://en.wikipedia.org/wiki/Content\_Security\_Policy

[73] http://w3c.github.io/push-api/

[74] http://thenewdialtone.com/webrtc-browser-push-notification/

[75] http://datatracker.ietf.org/doc/draft-thomson-webpush-protocol/?include\_text=1

[76] http://www.w3.org/TR/workers/

[77] https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker\_API

[78] https://github.com/slightlyoff/ServiceWorker/blob/master/explainer.md

[79] http://www.w3.org/TR/service-workers/

[80] https://jakearchibald.github.io/isserviceworkerready/

1. WP3 Scope
2. Chromium sandbox scheme
3. The architecture of a Google Chrome extension
4. Scheme of a persistent XSS attack
5. Scheme of a non-persistent XSS attack
6. Java Smart Card scheme
7. CoSE architecture
8. Service framework middle layer
9. Sippo WAC reference architecture
10. Sippo interfaces and APIs
11. Sippo.js abstraction layer
12. Sippo services and backends
13. Sippo WebRTC applications stack
14. Runtime High Level Architecture
15. Runtime High Level Architecture with Unstrusted Hyperties
16. Runtime High Level Architecture with Policy Enforcer
17. Reporter-Observer Communication Pattern
18. Core Runtime Architecture
19. Vulnerability matrix for a dummy platform
20. Stack
21. Browser
22. Security Browser
23. Application platform
24. Security Application platform
25. Deploy Core Runtime Components in the Native Runtime
26. Deploy Protocol Stub
27. Deploy Hyperty (part1)
28. Deploy Hyperty (part2)
29. Register Hyperty
30. Message Routing in Message BUS
31. Intra-domain Local Communication
32. Intra-domain Remote Communication
33. Inter-domain Local Communication
34. Inter-domain Remote Communication
35. User registration
36. Prepare Discovery
37. Use Discovery
38. Domain Login
39. Associate User Identity to Hyperty Instance
40. User identity assertion sequence diagram
41. Alice invites Bob for a communication
42. Bob receives invitation
43. Aknowledged that Bob received the invitation
44. notification update
45. Bob gatheres WebRTC resources
46. Synchronization of Alice's Data object
47. Runtime Main Procedures for M2M Communication
48. M2M Device Bootstrap
49. Context Discovery in M2M Intradomain Communication
50. Communication 4 pub sub 1
51. Communication 4 pub sub 2
52. Communication 4 pub sub 3
53. Runtime browser implementation
54. Crosswalk Architecture
55. Cordova functionnal schema
56. Messaging Node Architecture
57. WebRTC.org architecture scheme
58. OpenWebRTC Architecture
59. V8 Architecture
60. V8 Multiple Contexts
61. Docker Architecture
62. Firefox OS Architecture
63. Jitsi Videobridge Architecture
64. Kurento Architecture
65. Janus Gateway architecture
66. OMNA Network
67. WebRTC API evolution
68. Web Push Architecture
69. Main flows of events for subscription, push message delivery, and unsubscription
70. HTTP/2 Push
71. HTTP/2 Framing
72. HTTP/2 Streams
73. signalling on-the-fly concept
74. Wonder Library Main Classes
75. Video Bridge Component
76. OpenFire VideoBridge and Jicofo Components
77. Meteor Plataform Overview
78. Main data flow in a matrix architecture
79. RabbitMQ Architecture2
80. RabbitMQ Architecture
81. ZeroMQ types of communication patterns
82. ZeroMQ Clusters
83. Redis architecture
84. PubSub model
85. XMPP Architecture
86. XMPP Protocols
87. XMPP Jingle
88. MQTT Architecture
89. PSYC Message
90. NodeJS Architecture
91. Vertx Architecture Diagram
92. AngularJS Framework
93. BackboneJS Framework
94. Backbone Collections
95. State of the Art

# Standards SOTA

## Open Mobile Alliance (OMA)

### Overview

The Open Mobile Alliance (OMA) is a Mobile Operator driven industry forum for the definition of interoperable mobile service enablers. OMA defines APIs to offer functionalities and resources of Operator networks to developers.

#### OMA API Program

The OMA API program provides standardized interfaces to the service infrastructure residing within communication networks and on devices. Focused primarily between the service access layer and generic network capabilities, OMA API specifications allow operators and other service providers to expose device capabilities and network resources in an open and programmable way—to any developer community independent of the development platform.

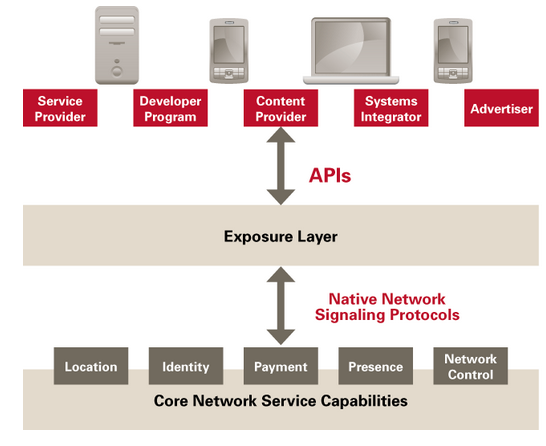


Figure 2: OMNA Network

OMA APIs exposes the network assets that developers need—no matter what protocols, platforms or other APIs they use. The Core network assets must be made available in order to deploy the wide variety of new applications and services that enter the market every day. These APIs are the tool that Operator offers to developer to make its services accessible to massive markets in a standard way. The OMA set of APIs increases the portability of applications and services in order to reach the subscriber base of operators and service providers that deploy OMA APIs.

As the number of APIs that perform the same functionality proliferate, fragmentation occurs. This limits developer access to subscribers, and operator and service providers’ choices of development platforms and communities. The OMA API Program, through standardization, solves this problem. A full OMA API inventory can be found at: <http://technical.openmobilealliance.org/Technical/technical-information/oma-api-program/oma-api-inventory>

### Selected APIs with potential relevance for reTHINK

There are several OMA APIs which can be potentially adopted by reTHINK project:

1. [RESTful Network API for WebRTC Signaling V1.0](http://technical.openmobilealliance.org/Technical/technical-information/oma-api-program/oma-api-inventory/api-details?API_ID=141). This API was released in February 2014 and it is a comprehensive REST-API for the WebRTC offer/answer signaling model. The payload that is transferred in the requests and responses is defined here as XML.
2. [Authorization Framework for Network APIs](http://technical.openmobilealliance.org//Technical/Release_Program/docs/Autho4API/V1_0-20131120-C/OMA-ER-Autho4API-V1_0-20131120-C.pdf). It was released in November 2013.The Authorization Framework for Network APIs enables a Resource Owner owning network resources exposed by Network APIs and RESTful APIs in particular, to authorize third-party Applications (desktop, mobile and web Applications)to access these resources via that API on the Resource Owner’s behalf.

There are also APIs for: Converged Address Book, Customer Profile, Network Message Store, Notification Channel, OneAPI, Payment, Presence and Quality of Service.

### OMA protocols with potential relevance for reTHINK

#### The Lightweight M2M

OMA Lightweight M2M is a protocol from the Open Mobile Alliance for M2M or IoT device management. Lightweight M2M enabler defines the application layer communication protocol between a LWM2M Server and a LWM2M Client, which is located in a LWM2M Device. The OMA Lightweight M2M enabler includes device management and service enablement for LWM2M Devices. It is normally used with CoAP. This protocol can be used in reTHINK for several components such as the Catalogue as proposed in [15].

### W3C WebRTC API

The Web Real-Time Communications Working Group was created in May 2011 within the W3C to define client-side APIs to enable Real-Time Communications in Web browsers.

It has defined a functional WebRTC 1.0 API which is implemented by major browser vendors to build real-time media applications in the browser without the need of installing any additional plugin. Additionaly to real-time media, WebRTC also supports the exchange of generic peer-to-peer data thanks to the Datachannel feature. This API is currently supported and production-ready in Firefox, Chrome and Opera.

Together with WebRTC 1.0 API the W3C is working in a series of drafts for which can be used toghether with the WebRTC API to create real-time media Web applications:

* Media Capture and Streams (getUserMedia): set of JavaScript APIs that allow local media, including audio and video, to be requested from a platform. This API allows to capture real time audio and video from the device which is running the web browsers. It is used by all the WebRTC applications which require capturing audio or video.
* MediaStream Recording: a recording API for use with MediaStreams as defined in [GETUSERMEDIA]
* MediaStream Image Capture: specific the takePhoto() and grabFrame() methods, and corresponding camera settings for use with MediaStreams as defined in Media Capture and Streams [GETUSERMEDIA]
* Media Capture Depth Stream Extensions: extends the Media Capture and Streams specification [GETUSERMEDIA] to allow a depth stream to be requested from the web platform using APIs familiar to web authors.
* Media Capture from DOM Elements: defines how a stream of media can be captured from a DOM element, such as a , , or
* element, in the form of a MediaStream [GETUSERMEDIA].
* Audio output devices API: defines a set of JavaScript APIs that let a Web application manage how audio is rendered on the user audio output devices.
* Identifiers for WebRTC's Statistics API: defines a set of Javascript APIs that allow access to the statistical information about a PeerConnection
* Screen Capture: defines how a user's display, or parts thereof, can be used as the source of a media stream using getOutputMedia, an extension to the Media Capture API [GETUSERMEDIA].

#### Applicability in reTHINK

The WebRTC is going to be intensively used in reTHINK to implement hyperties for Human-to-Human scenarios and also M2M scenarios where WebRTC Datachannel is used.

## ORTC

### Introduction

ORTC is an alternative to the current WebRTC API 1.0 to write WebRTC Applications to be executed in Web browser.

The protocols on the wire are exactly the same so it is compatible with aaplications written using the current API.

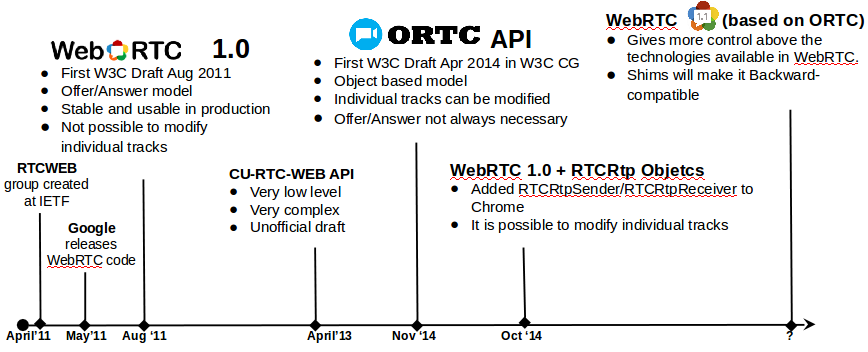


Figure 3: WebRTC API evolution

### Differences between ORTC and WebRTC 1.0.

Those are the main differences between ORTC and WebRTC 1.0:

ORTC specifically tailored to provide the direct control needed to enable advanced multimedia and conferencing features.

Limitations of the WebRTC API 1.0 have been addressed in the new version.

many of the parameters which are automatically handled by the browser in WebRTC 1.0 can be now modified by using native methods.

SDP is not the mandatory format to exchange information between browsers.

### Advantages of ORTC over WebRTC 1.0

An example of WebRTC 1.0 limitation it is that is not possible to modify parameters on individual media tracks since the SDPs must contain all the mediatracks of a mediastream. ORTC includes the RTCRtpSender object which associated to a sending MediaStreamTrack which provides methods to tweak its parameters.

An example of application of this new element is the ability to change the bitrate used for a video in-progess session which is being sent over a bad-quality connection keeping the parameters of the audio mediatrack. In WebRTC 1.0 this changes requires a SDP re-negotiation but with ORTC this is not nececesary.

Although RTCRtpSender was not included in the WebRTC 1.0 definition, this element was planned to be supported by Chrome from version 39.

#### Compatibility between ORTC and WebRTC 1.0.

Although SDP is not mandatory but WebRTC 1.0 applications would make it compatible thanks to Javasrcipts shims. All the use cases that ORTC would enable are already possible but they are more complex to implement as they require manual SDP manipulations which are error-prone and may require several Offer/Answer SDP exchanges to update the multimedia sessions.

### ORTC implementations

ORTC is the WebRTC API which Microsoft will implement in Internet Explorer and Skype will be supported in the Web Browser supporting ORTC along as other browsers which supports WebRTC 1.0.

## W3C SysApps

The W3C SysApps working group was originally chartered to provide a runtime and security model however, it has not been possible to reach such objective. In the meanwhile we may find proprietary web runtimes, FFOS and Chrome, which provide a security model for installed packaged web runtimes.

The WHATWG was formed by individuals of Apple, the Mozilla Foundation, and Opera Software in 2004, in response to the slow development of World Wide Web Consortium (W3C) Web standards and W3C's decision to abandon HTML in favor of XML-based technologies.

On 10 April 2007, the Mozilla Foundation, Apple, and Opera Software proposed that the new HTML working group of the W3C adopt the WHATWG’s HTML5 as the starting point of its work and name its future deliverable as "HTML5". On 9 May 2007, the new HTML working group resolved to do that

### W3C Application Lifecycle and Events

The W3C Application Lifecycle and Events draft (last version from 16 May 2014) extends the [Service Worker](w3c-service-workers.md) global execution context, to allow web developers to author applications that manage the application lifecycle and react to system events e.g. email or voip application. These capabilities allow application developers to create applications that integrate closely with the underlying system.

The following functionalities are provided: \* A background App or Service can run without a visible user interfaces \* An application is able to decide when to show the user interface \* The Application can be terminated without user’s consent, and that is able to restore to its previous state. \* The application is able to show a different user interface given how the app was launched. For example, if launched as a photo picker, the application will not show the default application window, but instead creates a special purpose user interface. \* The Application is only launched for a specific set of events eg the runtime uses somekind of events pre-filtering mechanism. For example, if an application listens to a "USB plugged" event, it can additionally ask to only listen to a specific device connected or a specific port. \* The application is able to enumerate windows associated with it, and create new windows.

This API extends the ServiceWorkerGlobalScope interfaces in the following way:

partial interface ServiceWorkerGlobalScope {  
 attribute EventHandler onlaunch;  
 attribute EventHandler onterminate;  
 attribute EventHandler onterminatecanceled;  
 readonly attribute TaskScheduler taskScheduler;  
};

#### Applicability in reTHINK

Similar to Service Workers, this extension can facilitate the development of some Runtime features notably to govern the runtime life-cycle of Hyperty instances. However, it seems this draft has not much support by the industry. However, [Chrome Packaged App lifecycle](https://developer.chrome.com/apps/app_lifecycle) looks similar. [Firefox Add-ons](https://developer.mozilla.org/en-US/Add-ons) should also support some kind of App life-cycle.

### Service Workers

Service workers are based on previous [Web Worker](http://www.w3.org/TR/workers/) W3C work and they essentially act as proxy servers that sit between web applications, and the browser and network (when available.) They are intended to (amongst other things) enable the creation of effective offline experiences, intercepting network requests and taking appropriate action based on whether the network is available and updated assets reside on the server.

A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web page/site it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available.)

A service worker is run in its context: it therefore has no DOM access, and runs on a different thread to the main JavaScript that supports the web app, so it is not blocking. It is designed to be fully async;

A service worker is first registered and, if successful, it will be downloaded to the client and attempt installation/activation for URLs accessed by the user inside the whole origin, or inside a subset specified by you.

Service Worker registration Example:

if ('serviceWorker' in navigator) {  
 navigator.serviceWorker.register('/my-app/sw.js', {  
 scope: '/my-app/'  
 }).then(function(reg) {  
 console.log('Yey!', reg);  
 }).catch(function(err) {  
 console.log('Boo!', err);  
 });  
}

Where /my-app/sw.js is the location of the ServiceWorker script, and it controls pages whose URL begins /my-app/.

At this point, your service worker will observe the following lifecycle: \* Download \* Install \* Activate

The service worker is immediately downloaded when a user first accesses a server worker–controlled site/page. Installation is attempted when the downloaded file is found to be new — either different to an existing service worker (byte-wise compared), or the first service worker encountered for this page/site.

The user agent may terminate service workers at any time it has no event to handle or detects abnormal operation such as infinite loops and tasks exceeding imposed time limits, if any, while handling the events.

Service Workers can be used to intercept network messages. Example:

self.addEventListener('fetch', function(event) {  
 console.log(event.request);  
});

Service Workers provides the basis for other features including: \* [Push](http://w3c.github.io/push-api/) \* [Background sync](https://github.com/slightlyoff/BackgroundSync) \* [Geofencing](https://github.com/slightlyoff/Geofencing)

Service Workers are still an experimental technology only supported in Desktop Chrome and Firefox.

#### Applicability in reTHINK

Service Workers provides features that can facilitate the development of some Runtime features including Event BUS, ProtOfly engine, Policy Engine. Its usage to support the Hyperty instance itself should also be evaluated. However it seems this technology is only available in Browsers and not in server side javascript runtime like node.js.

### Content Security Policy Level 2

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware.

Defines a policy language used to declare a set of content restrictions for a web resource, and a mechanism for transmitting the policy from a server to a client where the policy is enforced.

CSP provides a standard HTTP header that allows website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.

The following header names are in use as part of an experimental CSP implementations:

Content-Security-Policy — standard header name proposed by the W3C document. Google Chrome supports this as of version 25. Firefox supports this as of version 23, released on 6 August 2013. X-WebKit-CSP — experimental header introduced into Google Chrome and other WebKit-based browsers (Safari) in 2011. X-Content-Security-Policy — experimental header introduced in Gecko 2 based browsers (Firefox 4 to Firefox 22, Thunderbird 3.3, SeaMonkey 2.1). Support for the sandbox directive is also available in Internet Explorer 10 and Internet Explorer 11 using the experimental X-Content-Security-Policy header.

There's initial support for CSP in some web frameworks such as AngularJS and Django.

Example: script-src 'self'; object-src 'none'

Security policies contain a set of security policy directives (script-src and object-src in the example above), each responsible for declaring the restrictions for a particular resource type, or manipulating a specific aspect of the policy’s restrictions.

The server delivers a policy to the user agent via an HTTP response header or an HTML meta element. The Content-Security-Policy header field is the preferred mechanism for delivering a policy. The grammar is as follows:

"Content-Security-Policy:" 1#policy-token

For example, a response might include the following header field: Content-Security-Policy: script-src 'self'

A Content Security Policy consists of a U+003B SEMICOLON (;) delimited list of directives.

#### Applicability in reTHINK

In a preliminary analysis CSP seems too limited to be applied for the runtime policy engine but it may be useful to improve security in the protOfly engine.

### W3C Push API

W3C Push API work began in the W3C Web Applications (WEBAPPS) Working Group to provide very simple JavaScript APIs for push client registering and message receipt. It was in large part, driven by the urgent WebRTC use case of waking up an application for a real-time incoming call. The API has two logical pieces:

* a pushRegistrationManager that consults the user agent about which push service to use and returns information (a registration id and a URI) that can be sent to the application server for it to know how to reach the push service. In general, the URI is where the application server can send push messages and the registration id is to be provided to the push service to indicate the delivery target for the messages.
* a Service Worker that is used to catch, store if necessary, and ultimately deliver push messages to the application.

The following general entities and example operation for delivery of push messages are the following (see figure below):

* Application servers request delivery of a push message to a webapp via a RESTful API exposed by a push service
* The push service delivers the message to a specific user agent
* The user agent delivers the push message to the specific webapp intended to receive it.

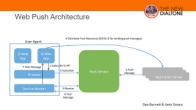


Figure 4: Web Push Architecture

The main data flows involved in the PUSH API usage are described below:

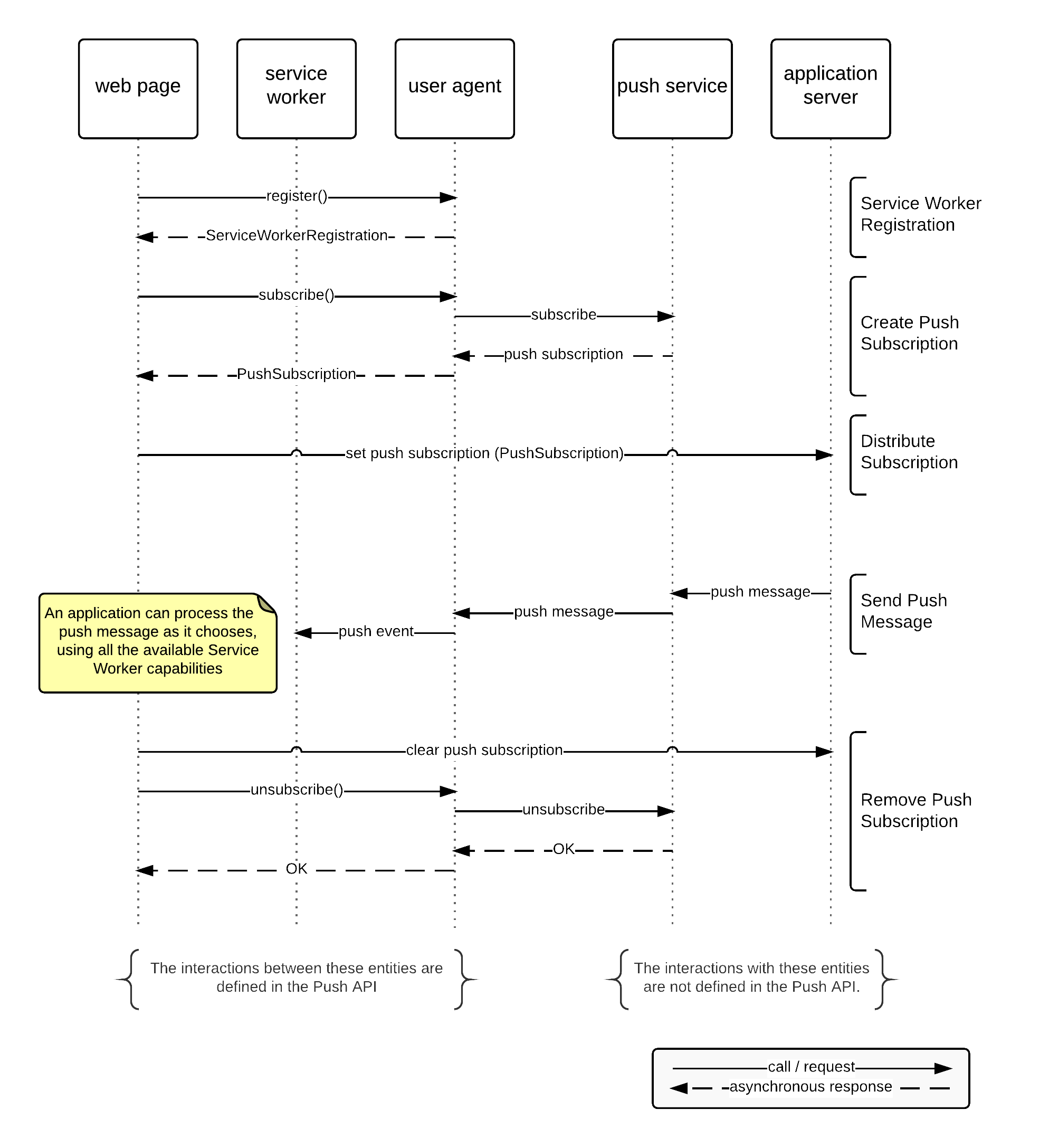


Figure 5: Main flows of events for subscription, push message delivery, and unsubscription

The W3C Push API is a client API that does not provide any standard for how the application server sends messages to the push server. This part is covered by the IETF WEBPUSH protocol.

### IETF Web Push

In order to provide a standard way for push notification to be used on the Internet, the IETF WEBPUSH working group was created in late 2014. The WEBPUSH WG is developing an HTTP2-based mechanism for applications “to request the delivery of data to a device using a consolidated push notification service. This protocol will include the ability to push the same message to multiple subscribed devices.

Expected clients are both web applications and field gateways that consolidate and forward messages to embedded devices. Several models have been discussed, and there currently seems to be most interest in a publication-subscription model where each device subscribes individually, but with no requirement for a separate registration in advance.

This working group will create an HTTP based protocol that will allow applications to request delivery of data to applications through a consolidated service. The working group will work in cooperation with W3C’s Web Push API.

### Applicability to reTHINK

The ability to push notifications towards Hyperty runtime is an essential feature that must be supported according to these standards.

### References

* http://w3c.github.io/push-api/
* http://thenewdialtone.com/webrtc-browser-push-notification/
* http://datatracker.ietf.org/doc/draft-thomson-webpush-protocol/?include\_text=1

## HTTP/2

### Introduction

HTTP/1.1 has served the Web for more than 15 years, but its age is starting to show. Web application has evolved a lot from the beginning but the protocol which transport it has not evolved at the same pace.

Loading a Web page is more resource intensive than ever because HTTP practically only allows one outstanding request per TCP connection: \* Browsers have used multiple TCP connections to issue parallel requests. This is counter-productive (TCP congestion control is effectively negated leading to congestion events), and unfair (browsers take more network resources). \* At the same time, the large number of requests means a lot of duplicated data “on the wire”. \* These factors mean that HTTP/1.1 requests have a lot of overhead associated with them; the more requests are made, the worse performance we get.

This problems leaded the industry to consider “Best Practices” things like: spriting, data inlining, domain sharding and concatenation which are just workarounds which improves the user experience.These hacks are indications of underlying problems in the protocol itself, and cause a number of problems on their own when used. On the other side, the way in which the Web is accessed has also change a lot, mobile devices has become the main point of entry to the web. Characteristics of wireless connections (high latency, jitter and packet loss) may prevent Web applications served with HTTP over TCP from being responsive and even usable.

[HTTP/2](https://tools.ietf.org/html/rfc7540), which is already a definitive RFC, was designed to be adapted to the new conditions of the WWW and its main goal is to improve the user experience. HTTP/2 is an evolution of SPDY, experimental protocol mainly developed by Google which is currenlty being used in production in many Google applications. To take advantage of HTTP/2 new features a new transport protocol, QUIC, has been designed. Both protocols combined will be extensily used in Internet in the next years.

### Main differences from HTTP1.1

|  |  |
| --- | --- |
| HTTP/1.1 | HTTP/2 |
| textual | binary |
| ordered and blocking | fully multiplexed |
| several connections for parallelism | one connection for parallelism |
| only content compression | header compression |
| not proactive push | allows servers to “push” responses proactively into client caches |

#### Binary

HTTP/2 is a binary protocol, it means that no human-understable ASCII chars are sent on the wire. The main advantages of being binary are: \* Binary protocols are more efficient to parse by applications which does not have to handle with issues related to text-protocol parsing. \* It is more compact “on the wire” since no extra information needs to be sent. \* It is much less error-prone, compared to textual protocols like HTTP/1.x with whitespace handling, capitalization, line endings, blank links... HTTP/1.1 defines five different ways to parse a message; in HTTP/2, there’s just one code path.

#### Fully multiplexed

HTTP/1.x suffers “head-of-line blocking” only one request can be outstanding on a connection at a time. Pipelining of request is not a solution since a large or slow response will block others behind it. Additionally, it has been found very difficult to deploy, because many intermediaries and servers do not process it correctly.

#### One connection for parallelism

With HTTP/1, browsers open between 4 and 8 connections per origin. Since many sites use multiple origins, this could mean that a single page load opens more than 30 connections. One application opening so many connections simultaneously breaks a lot of the assumptions that TCP was built upon. Since each connection will start a flood of data in the response, there is a real risk that buffers in the intervening network will overflow, causing a congestion event and retransmits. Additionally, using so many connections unfairly monopolizes network resources, “stealing” them from other, better-behaved applications (e.g., VoIP).

#### Header compression

HTTP/1 supported compression for content but not for headers. Assuming an average of 80 assets per page and each request has 1400 bytes of headers, it takes at least 7-8 round trips just to get the headers out “on the wire.” That is not counting response time - that is just to get them out of the client. Headers add a lot of overhead traffic and increase latency.

#### Server push

In HTTP/2 the server can push resources to the client before receiving a request to server that resource. This reduce the load time as the browser does not have to send GET request to ask for all the resources avoiding RTT delays.

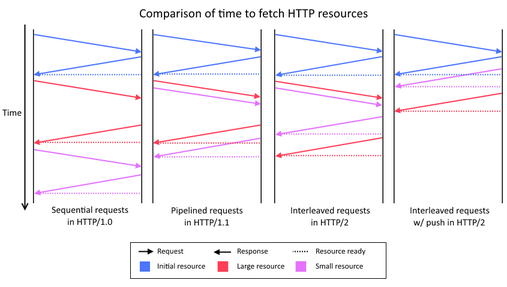


Figure 6: HTTP/2 Push

### Features of HTTP/2

#### HTTP/2 key concepts

1. All communication is performed over a single TCP connection that can carry any number of bidirectional streams.
2. Each stream has a unique identifier and optional priority information that is used to carry bidirectional messages.
3. Each message is a logical HTTP message, such as a request, or response, which consists of one or more frames.
4. The frame is the smallest unit of communication that carries a specific type of data—e.g., HTTP headers, message payload, and so on.
5. Frames from different streams may be interleaved and then reassembled via the embedded stream identifier in the header of each frame.

#### Framing

The frame is the smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs.

The most relevant frames are the used to transport Headers and Data, but there are also frames for other functions like stream control, push content and set the priority of the streams. DATA frames are used to transport HTTP message bodies and HEADERS frames are used to communicate header fields for a stream. The rest of frames are used for control tasks and they are called Control Frames.

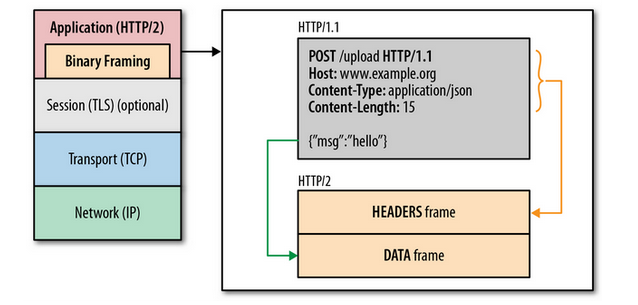


Figure 7: HTTP/2 Framing

#### Multiplexing

HTTP/2 interleave multiple requests and responses in parallel without blocking any one using a single connection to deliver and receive them. This allow to remove unnecessary HTTP/1.x workarounds such as concatenated files, image sprites, and domain sharding. It also resultes in lower page load times by eliminating unnecessary latency and improving utilization of available network capacity.

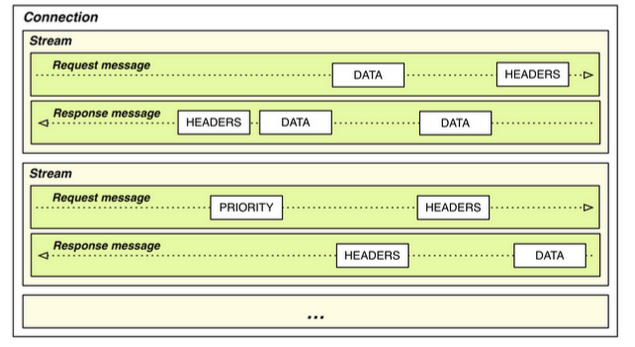


Figure 8: HTTP/2 Streams

#### Priority

Not all the elements of a Web page are equally important for the functionality and visualization. An user can start consuming the content before loading all the elements. HTTP/2 allows to set different priorities to each stream and priority can be specified by the client side and it can be changed in runtime.

#### Flow control

HTTP/2 applies flow control mechanism to the different streams. The flow control is directional with overall control provided by the receiver and the frame type determines whether flow control applies to a frame. Only DATA frames are subject to flow control (the most important part part of the traffic will transported in DATA frames) so control frames are not blocked by flow control. On the other side, flow control cannot be disabled according to the HTTP/2 RFC draft.

### HTTP/2 encryption

The RFC allows HTTP/2 to be sent over unencrypted transport protocols like TCP. However Google and Mozilla have officially said that HTTP/2 is not going to be allowd to be sent over plain TCP but over TLS connection to avoid issues with intermediates element between the browser and the server. This has an impact in ReTHINK project as if a web site is served over TLS or a secure protocol, the rest of elements of that site must be served using an encrypted protocol.

### Why using for HTTP/2 in ReTHINK project.

HTTP/2 is aimed to make the Web more efficient and responsive and it advantages are more notable for web sites. However the following points should be considered when discussing the use of HTTP/2 in the ReTHINK project: \* HTTP/2 is going to be massively adopted by Internet companies following Google and Twitter movements. \* The most relevant projects (Apache, NGINX, ) already supports or plan to support HTTP/2. ReTHINK Javscript libraries are expected to be used also in web applications so supporting HTTP/2 would avoid mixing HTTP/1.1 and HTTP/2. \* ReTHINK will take advantage of the benefits of HTTP/2. \* HTTP/2 is normally going to be used over QUIC so we will take advantage of using QUIC as the transport protocol. \* Despite the fact of not being used to transport media, its characteristics would make it possible.

All the available implementations of HTTP/2 are gathered in the official HTTP/2 WG Github repository: https://github.com/http2/http2-spec/wiki/Implementations

### Websocket over HTTP/2

The WebSocket protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code. Since it requires one TCP connection for every WebSocket connection, having multiple WebSocket connections between the same client and the same server is inefficient. On the other hand, HTTP/2 specifies a fast, secure, multiplexed framing protocol. The [draft!](http://tools.ietf.org/html/draft-hirano-httpbis-websocket-over-http2) document provides bi-directional multiplexed communication by layering WebSocket on top of HTTP/2. The document is still a draft but this could be used in ReTHINK project as it allows to get a bidirectional communication over an existing HTTP connection without the need of creating additional socket connections.

## QUIC (Quick UDP Internet Connection)

### Introduction

Internet has evolved a lot since the first version of TCP was designed 40 years ago. Many RFCs have added functionality to TCP since then and it played a vital role being the transport of the WWW for decades. UDP was designed to be used in real-time scenarios where the realiability is not so important as getting a very low delay and being tolerant to packet losses.

More flexible protocols which combine feature from TCP and UDP were also designed. The most remarkable effort was SCTP which was first designed to transport by the Signaling Transport Working Group to transport Signaling System 7 (SS7) and ISDN communications protocols over IP networks. Amongst other features, SCTP creates a tunnel over UDP and to multiplex a connection into multiple streams. It also provides notification of duplicated or missing data chunks, creating a reliable transport over UDP.  
The fact of allowing multiple streams over a single connections prevents Head-of-Line blocking from hapenning. It means that a packet loss which affect to a stream will not imply a delay in the rest of streams as it would happen if several streams are multiplexed over a TCP connection.

### Transport layer replacement

Google started the definition, implementation and real filed testing of a new transport protocol called QUIC in 2013. QUIC stands for "Quick UDP Internet Connection" and it was designed as a replacement for TCP as response of the new needs and the evolutions of Internet: streaming-based services are more and more demanded, many connections are established over wireless connections and they are established from mobile devices.

The features the new protocol must support to meet the today's Internet requirements: \* it is necessary to make the web faster and the low layer protocols must adapt to the evolution of the web applications. \* connection establishment latency must be reduced to improve the user experience and to make the web and Internet more usable. \* the web must be secyre by default, TCP does not secure the traffic and an optional upper layer must be added to provide this security.

QUIC introduces improvement in many aspects of the TCP protocol: \* TCP needs a three-handshake to create a connection. If the connection is established over TLS more RTT are necessary. \* TCP provides reliable, ordered and error-checked delivery based re-tx, acknowledgments and checksum. The cost of having this features is that any packet loss will trigger retransmissions which will normaly delay the delivery of the rest of packets. \* TCP allows to stablish a single full duplex byte stream and all the data over that stream will be treated processed indistinctly. \*TCP requires an extra protocol on top of it (TLS) to provide encryptin and authentication. It adds an overhead and delays in the connection setup.

#### Why not to use SCTP for this?

SCTP can be considered as an alternative for TCP. It has two main features which could make it an atarctive choice:  
\* SCTP also provides stream multiplexing over a single connection. \* DTLS provides SSL quality encryption and authentication over UDP in the same way as TLS over TCP.

These features made that SCTP had been chosen to be used internally by WebRTC Datachannels however it presents some problems which forced the QUIC developer to design a new alternative:

* SCTP requires 4 roundtrips are necessary to establish a SCTP over DTLS connection. This means an unacceptable delay in many applications and degrades the user experience.
* It was not designed to reduce latency, SCTP connections were designed to be persistent between two peers.

In contrast the goal of QUIC is to have to perform a connection establishment with zero RTT overhead.

### QUIC current use

QUIC is currently being used by Chrome to interact with Google Apps and it is going to be used by more services in short-term. HTTP/2 is going to become a definitive RFC in a few months and it is being used in production by many relevant Internet companies such as Twitter and Facebook. HTTP/2 performs much better over QUIC since it allows to leverage it stream-based designed so QUIC isvery likely to be adopted by the IETF in short-term.

### QUIC main features

QUIC can be deployed in today's internet, actually it is been used in many real deployments without having to apply any modification in any intermediate node. This is a key point which makes its adoption much easier than IPv6.

It provides a very low latency in connections and responses. That is a critical point as many services are cloud -based and must be accessed from mobile and transoceanic connections which high RTTs.

It has a reliable-stream support and the packet losses which affect one strem does not affect the rest of streams. This reduces the Head Of Line (HOL) blocking due to packet loss.

It provides a better congestion avoidance than TCP which implcitly considered that any packet loss was due to a network congestion which is not normally true for wireless connections.

The privacy and Security it provides is comparable to TLS but with a much lower delay in the connection setup which can from 0 in the most optmistic cas to 2 RTTs in the worst case. This is a key feature of QUIC as it allows to improve the user experience.

Connection are identified by a Connection Identifier not by layer 3 and layer 4 elements (IPs and ports). This fits very well in mobile scenarios where the IP of the clients may change due to handover mechanism.

#### Is it a good choice to use UDP for QUIC?

QUIC can be considered as an intelligent layer over UDP which provides enhanced features. UDP is intensively used by VoIP and gamers for years in very latency-sensitive applications. On the other side, 91-94% of the users which had TCP connectivity with Google can make outbound UDP connections so it is possible to build a transport in today's internet over UDP. It is also necesary to consider NAT unbinding which does not happen to TCP. This problem has been addressed internally by QUIC designers through the use of keepalives packets.

### Applicability to reThink project

Including the use of QUIC as a requirement or a recommendation could help to support more reliably mobility scenarios where the End-User IP may be changed during a connection with the Signaling service. Additionaly thee transport layer connectivity provided by QUIC is more suitable for wireless connections (longer RTTs, packet lost and changes at IP level) than TCP. QUIC has been designed bearing HTTP/2 in mind as it improves its performance a lot, however the use of QUIC by other protocols can also be very advantageous.

During a media session, the change of an IP requires an SDP re-negotiation when a media sessions is ongoing, so we can't leverage QUIC features for media. However QUIC would be helpful in all the scenarios at signaling level.

#### Existing QUIC implementations

The QUIC reference library is libquic (https://github.com/devsisters/libquic). It has been mainly developed by Google. This repository and its sources and dependencies were extracted from Chromium's QUIC Implementation with a few modifications and patches to minimize dependencies needed to build QUIC library. This code can be used to be integrated with HTTP Server like Apache and nginx but this has not been done so far. This library implementes

In the Chromium repository it is available a standalone client and server which can be used as a reference for ReThink project implementations. http://src.chromium.org/viewvc/chrome/trunk/src/net/tools/quic/

In all the imeplementations QUIC is used with SPDY and HTTP/2 so its use separated from those protocols has to be investigated.

### Drawbacks of using QUIC as transport protocol

QUIC is a very new protocol so it is still not widely used. It means that many existing systems and projects does still not support it so testing and implementation which requires an additional effrot compared to TCP.

On the other side, despite the fact that is a protocol likely to become an RFC darft in short-term it has not been formally specified by the IETF. This is the official definition document mantained by Google: https://docs.google.com/document/d/1RNHkx\_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit Any implementation made today may not be completely compliant with the final protocol.

# Projects SOTA

## WONDER Project

### Overview

The main motivation of the [OpenLab WONDER experimentation project](http://hypercomm.github.io/wonder/) [32] was to experiment and evaluate some of the P2252 Eurescom Study recommendations. WONDER evaluated whether to use IMS to deliver services to WebRTC endpoints or to use a more disruptive pure Web based approach to deliver services to WebRTC endpoints.

For the second option WONDER has enlightened some paths to be followed in a post-IMS era dominated by Web technologies and large eclectic cooperative eco-systems. The novel [Signalling On-the-fly (SigOfly)](http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7073799&filter%3DAND(p_IS_Number%3A7073795)) concept was conceived and successfully demonstrated to address the lack of a standard WebRTC signalling protocol. The SigOfly concept enables seamless interoperability between different WebRTC service domains avoiding NNI interfaces by using peer side APIs and restful push notification services.

WONDER experimentation involved Portugal Telecom and Deutsche Telekom.

### SigOfly Concept

The SigOfly concept leverages the use of scripts (JavaScript) by WebRTC Applications to implement signalling protocol stacks. This means, the signalling protocol stack can be selected, loaded and instantiated during runtime. Such characteristic enables signalling protocols to be selected per WebRTC Conversation to ensure full signalling interoperability among peers using Triangle based Network topologies. The SigOfly procedures should be applied at the end-user client to benefit from WebRTC model. However, the concept is also feasible between Messaging Servers supporting Javascript execution engines (e.g. nodejs or vertx.io).

Before the SigOfly concept is described in detail, some terms require a definition:

**Messaging Server:** the server that supports the exchange of signalling messages required for the establishment of WebRTC sessions. Each Messaging Server belongs to a domain;

**Domain Channel:** the signalling channel that is established with domain’s messaging server as soon as a domain‘s user is registered and is online;

**Transient Channel:** the signalling channel that is established, typically with a foreign messaging server (i.e. from another domain) in scope of a inter-domain conversation;

**Messaging Stub:** the script containing the protocol stack and all the logic needed to establish a channel to a certain Messaging Server;

**Conversation Hosting Messaging Server:** is the Messaging Server that is used to support the exchange of all signalling messages among peers belonging to different domains. The Hosting peer uses the Domain Channel to exchange signalling messages, while other peers use Transient Channels that connect to the Hosting Messaging Server.

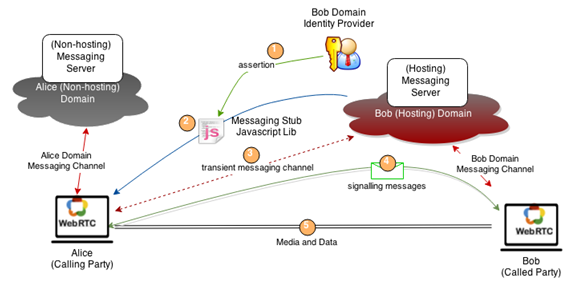


Figure 9 signalling on-the-fly concept

The classic Alice and Bob example is used to explain the SigOfly concept. We assume that Alice and Bob are registered in different Service Provider domains having each one a Domain Channel established with their own Messaging Server (see Fig. below). In case Alice wants to talk to Bob by using Bob’s WebRTC identity e.g. bob@domain.com, the following steps will be performed:

Step 1: Information about the Identity of Bob, including Bob’s Messaging Stub provider, is provided and asserted by Bob’s Identity Provider (IdP).

Step 2: Alice downloads and instantiates Bob’s Messaging Stub in her browser to setup a Transient Channel with Bob’s domain Messaging Server.

Step 3: As soon as the Transient Channel is established, Alice can send an Invitation message to Bob containing her Session Description Protocol (SDP) based communication offer.

Step 4: Since Bob is connected in the same Message Server via his Domain Channel, he will receive Alice’s invitation in his Browser. If Bob accepts the invitation, an Accepted message containing Bob SDP response will be send to Alice.

Step 5: As soon as Alice’s browser receives Bob’s SDP, the media and/or data streams can be directly connected between the two browse

It should be noted that SigOfly does not directly address identity management aspects but aims to be compliant with ongoing WebRTC Identity Management work from W3C and IETF, mainly by extending RTCIdentityAssertion to also include the assertion of MessagingStubs. This means, Alice and Bob authentication is done outside SigOfly procedures, described above, which are agnostic of the IdP and authentication protocols used.

The SigOfly concept is also applicable in use cases where conversations are hosted by calling parties, in multiparty conversations or to support interoperability with legacy networks (e.g. IMS and PSTN).

#### Data Codec On-the-fly

In addition, the SigOfly principle is also applied to address Data Channel Services Interoperability, to conceive another novel concept called “Data Codec On-the-fly”. the “Data Codec On-the-fly” concept ensure all peers are using the same protocol on top of the Data Channel. A Data Codec is a JavaScript library that implements a data communication processing algorithm to code and send data to the Data Channel and to receive and decode data from the Data Channel. As with Messaging Stubs in the SigOfly concept, Data Codecs are downloaded and instantiated by peers according to URLs identified in Conversation setup or update signalling messages

Further information about SigOfly and Data Codec On-the-fly is provided in [34](https://github.com/hypercomm/wonder/wiki/Signalling-on-the-fly).

### WONDER Library

A JavaScript framework, the WONDER lib, was designed and implemented to validate the SigOfly and Data Codec On-the-fly concepts.

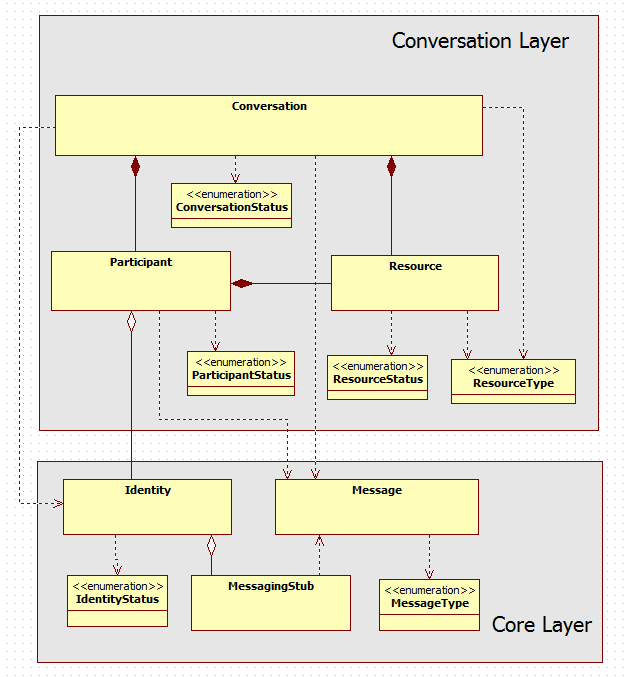


Figure 10 Wonder Library Main Classes

Main WONDER library classes are:

* The Identity, representing a user and containing all information needed to support Conversation services. This also includes the service endpoint to retrieve the protocol stack (Messaging Stub) which will be used to establish a signalling channel with the Messaging Server of the Identity domain. The Identity entity extends the current Identity concept defined by W3C to support seamless interoperability by using the SigOfly mechanism.
* The MessagingStub implementing the protocol Stack used to communicate with a certain Messaging server.
* The Conversation class managing all participants including the setup, update or close of media and data connections.
* The Participant class, handling all operations needed to manage the participation of an Identity (User) in a conversation including the WebRTC PeerConnection functionalities. The Local Participant is associated with the Identity that is using the Browser while the Remote Participant is associated to remote Identities (users) involved in the conversation.
* The Resource class representing the digital assets that are shared among participants in the conversation including participants’ voice, video, screens, photos, video Clips, music clips, documents, etc. These assets are usually managed by the Participant that owns it. For local participants assets are sent (e.g. WebRTC outgoing stream tracks) while for remote participants assets are received (e.g. WebRTC incoming stream tracks). Some Resource types like Chat are not managed by a Participant but by the Conversation.
* The Data Codec, which is used by Resources that are shared on top of the Data Channel, like file sharing and Textual Chat, to decode and encode the data in a consistent way by all the peers. The Data Codec may also be downloaded on-the-fly by the peers.
* The Message, which is used to exchange all data needed to setup, update and close media and data connection between peers via the Messaging Server. It may also be used for other purposes e.g. presence information management. Each message is comprised by a Header and a Body.

The [API documentation](https://raw.githack.com/hypercomm/wonder/master/docs/api/index.html) and the [source code](https://github.com/hypercomm/wonder/tree/master/src/libs) were published in a public github repository [35][36] .

### Input to reTHINK

WONDER Library can be used in reTHINK in different ways that are detailed in the following sections.

#### Runtime Messaging API

The MessagingStub API is a good starting point to design the Javascript API to support Hyperty Message communication which is comprised by the following functions:

connect(ownRtcIdentity, credentials, callbackFunction) {  
// connects to Msg Server  
};  
  
addListener(StubEvtHandler, URI, contextId){  
//Adds a listener for a certain context.  
};  
  
removeListener(StubEvtHandler, URI, contextId){  
//Removes a listener from a certain context.  
};  
  
getListeners(){  
//Gets the list of listeners.  
};  
  
sendMessage = function(message) {  
 // send Message  
};  
  
disconnect = function() {  
 // disconnects from server  
};

Check WONDER MessagingStub API documentation in [37](https://raw.githack.com/hypercomm/wonder/master/docs/api/symbols/MessagingStub.html).

#### Messages Format

The WONDER Message class provides good input for the design of Hyperty Messages. Wonder Message is a JSON structure and it is comprised by a Header and a Body. The following Message Header attributes are defined:

type Type of the Message   
from Sender of the message  
to Recipients of the message  
context identifies a certain context for the message eg the Id of the conversation

The following Message Types are defined:

* INVITATION - Message to invite a peer to a conversation.
* ACCEPTED - Answer for conversation accepted or Context subscription accepted
* CONNECTIVITY\_CANDIDATE - Message contains ICE connectivity candidate
* NOT\_ACCEPTED - Answer for conversation not accepted or Context subscription not accepted
* BYE - Message to finish the communication with a peer
* UPDATE - Message to Update conversation by adding or removing a Resource
* UPDATED - Answer to Message UPDATE
* CONTEXT - Message used to publish the context and status of an Identity.
* SUBSCRIBE - Message to request to receive CONTEXT notifications from a certain Identity
* MESSAGE - Mainly used to support Pager Mode Chat. But it can be used for other use cases instead of Data Channel eg small files.
* CRUD\_OPERATION - Messages to handle data persistence in a resource tree

The Message body will depend on the Message Type. Detailed description of WONDER Messages are provided [here](wonder-messages.md).

#### Runtime Identity API

The Identity and IDP classes could also provide good input for the support of Identity Management functionalities by reTHINK Runtime.

The WONDER IDP is a singleton object which handles WONDER Identities creation and retrieval from IDP Servers. IDP is agnostic of the protocol used by IDP Server. IDP main function is:

/\*This method takes either a single rtcIdentity or an array of rtcIdentities and creates Identity objects from them. The successfully created Identities are then returned in an Array in the success callback. If one or more rtcIdentities can't be created then the returned array is shorter than the given array.\*/  
  
createIdentities(rtcIdentities, onSuccessCallback, onErrorCallback)

The WONDER Identity class represents a user and contains all information needed to support a Conversation service including the service endpoint to retrieve the protocol stack (Messaging Stub) that will be used to establish a signalling channel with the Identity domain messaging server. Identities are only created by using the corresponding create-methods of the IDP class.

Identity is comprised by the following attributes:

context; // including Identity presence status;  
sessionId // identification of the session established with the domain through the Messaging Server.  
username;  
messagingStubLibUrl; // the service URL from where Identity domain messagingStub can be downloaded  
notificationAddress; // to support notifications when the user is not connected  
credentials; // to be used to connect to the domain  
avatar; //

Some of these attributes that are more sensitive in terms of security should be handled by the Runtime itself (e.g. credentials / tokens, sessionId, username, messagingStubLibUrl) while the others could be handled by an Hyperty representing the Identity eg avatar, context.

The main WONDER Identity functions are:

/\*\*  
 \* This method downloads a messaging stub and keeps a reference to it in a local  
 \* attribute, if not already done before. That means the download will only be performed once.  
 \* After download it invokes the given callback with a reference to the downloaded MessagingStub.  
 \*   
 \* @param callback {callback(MessagingStub)} callback that is invoked with messagingStub as param; if download failed then the stub param is empty  
 \*/  
Identity.prototype.resolve = function( callback ) {  
};  
  
/\*\*   
 \* This method subscribes to add a listener to receive status information (CONTEXT message type) from the user associated to this Identity.   
 \* The Signalling on the fly concept is also used to ensure cross domain Presence management interoperability  
 \* by calling the Identiy.resolve() function  
 \* @param subscriber :  
 \* Identity ... The identity of the subscriber  
 \* @param type :  
 \* SubscriptionType ... The subscription type  
 \*  
 \*/  
  
Identity.prototype.subscribe = function(subscriber) {  
};  
  
/\*\*   
 \*   
 \* This method removes a listener previously added with "subscribe()" function to receive status information   
 \* (CONTEXT message type) from the user associated to this Identity  
 \* @param subscriber :  
 \* Identity ... The identity of the subscriber  
 \* @param type :  
 \* SubscriptionType ... The subscription type  
 \*  
 \*/  
Identity.prototype.unsubscribe = function(subscriber, type) {  
};  
  
/\*\*  
 \* To set Identity context and to publish it by sending a CONTEXT message to address "rtcIdentity.context"  
 \*   
 \* @param context :  
 \* String ... The context to set  
 \*   
 \*/  
   
Identity.prototype.setContext = function(context) {  
};  
  
/\*\*  
 \* getContext  
 \*   
 \* @returns ContextData ... gets the context attribute for this Identity  
 \*   
 \*/  
Identity.prototype.getContext = function() {  
};  
  
/\*Handler to receive incoming messages eg context update\*/  
  
Identity.prototype.onMessage = function(message){  
}  
  
/\*\*  
 \* addListener function usually implemented by the App  
 \* @param.. listener  
 \*/  
  
Identity.prototype.addListener = function( listener, rtcIdentity ){  
}  
  
/\*\*  
 \* removeListener function  
 \* @param.. listener  
 \*/  
  
Identity.prototype.removeListener = function( listener, rtcIdentity ){  
}

The "resolve(..)" function should be handled by the Runtime itself but all the others could be implemented by an Hyperty representing the Identity. For further discussion.

#### Javascript Framework

WONDER library can provide some input for the design and implementation of reTHINK Javascript framework that should facilitate the development of Hyperties, namely: \* Conversation \* Participant \* Resource \* Identity \* MessageFactory

#### Javascript Shim Layer for non-compliant reTHINK Runtime

WONDER library can provide some input for the design and implementation of reTHINK Javascript Shim Layer to be used in non-compliant reTHINK Runtime, namely:

* MessagingStub
* Idp
* Identity
* DataBroker
* DataCodec

### WONDER Messages Format

The WONDER Message class provides good input for the design of Hyperty Messages. Wonder Message is a JSON structure and it is comprised by a Header and a Body. The following Message Header attributes are defined:

type Type of the Message   
from Sender of the message  
to Recipients of the message  
context identifies a certain context for the message eg the Id of the conversation

The following Message Types are defined:

* INVITATION - Message to invite a peer to a conversation.
* ACCEPTED - Answer for conversation accepted or Context subscription accepted
* CONNECTIVITY\_CANDIDATE - Message contains ICE connectivity candidate
* NOT\_ACCEPTED - Answer for conversation not accepted or Context subscription not accepted
* BYE - Message to finish the communication with a peer
* UPDATE - Message to Update conversation by adding or removing a Resource
* UPDATED - Answer to Message UPDATE
* CONTEXT - Message used to publish the context and status of an Identity.
* SUBSCRIBE - Message to request to receive CONTEXT notifications from a certain Identity
* MESSAGE - Mainly used to support Pager Mode Chat. But it can be used for other use cases instead of Data Channel eg small files.
* CRUD\_OPERATION - Messages to handle data persistence in a resource tree

The Message body will depend on the Message Type. Some of these messages and associated bodies are more detailed below.

##### Invitation Message Type

Invitation for a new conversation to be hosted by the inviting identity ie to use Messaging Server of the inviting identity which is provided in the message body as well as the connection description of the inviting identity.

**Invitation Message Body**

conversationURL;  
 connectionDescription; // SDP  
 subject;  
 hosting; // Identity of who is hosting the conversation  
 agenda;  
 peers;  
 constraints; // To describe media and data constraints for each resource including Audio, Video constraints and direction (in,out,inout)

##### Accepted Message Type

To accept eg Invitations, Conversation updates or Context Subscription. Similar to SIP 200 OK

**Accepted Message Body**

connectionDescription; // SDP  
 hosting; // Identity of who is hosting the conversation  
 constraints; // To describe media and data constraints for each resource including Audio, Video constraints and direction (in,out,inout)

##### Not Accepted

Eg Busy, Reject, No\_answer to: - Invitation requests - Update requests - Subscription requests

This information will go in the message body as a String

##### CONNECTIVITY\_CANDIDATE Message Type

Messages used to exchange ICE connectivity candidates between peers

**Message Body**

label - The label of the candidate.  
id - The id of the candidate.  
candidate - The ICE candidate string.  
lastCandidate - Boolean indicating if the candidate is the last one. If true, include the full SDP in the candidate parameter for compatibility with domains that don't support trickling.

##### CRUD\_OPERATION

These Messages are used to handle data persistence in a resource tree by using the four basic functions create, read, update and delete.

**Message Body**

operation // create, read, update or delete.  
syntax // syntax used for CRUD operation field "criteria" examples: mongoDB, SQL  
criteria // some filtering expression used in read and update operations  
doc // Contains data for CREATE and UPDATE operations  
resource; // Resource URI where the operation is applied

## M2M Projects

Several (open source) projects implement key functionality that may be potentially essential to realize the Catalogue Service as well as the Registry Service simultaneously in the machine-to-machine and human-to-human domain. The following sections provide an overview of implementations for

1. CoAP/Well-known CoRE,
2. ETSI and oneM2M, and
3. OMA Device Management

### CoAP/ Well-known CoRE Projects

If Hyperties, Codecs, Protostub and other artifacts to be provisioned on the end devices are to be regarded as resources with attributes describing: capabilities (audio, video, text), running platform (OS), configuration (DNS name of the messaging node, DNS server), implementation (code/script, codecs), these artifacts can be organized as resources in the Repository/Catalogue component.

#### LibCoap Project

##### Overview

The implementation includes features for receiving and sending CoAP requests. It also supports the [CoRE-link format RFC 6690] (https://tools.ietf.org/html/rfc6690) to organize the CoAP resources as a well-known CORE. It has support for Linux but also Contiki Operating Systems.

The implementation is using C as programming language.

Link: https://gitlab.informatik.uni-bremen.de/bergmann/libcoap/tree/master

The library is published as open-source software without any warranty of any kind. Use is permitted under the terms of the GNU General Public License (GPL), Version 2 or higher, OR the revised BSD license.

##### How to use

For starting a CoAP server based on a configuration file, a main program has to be written. Handlers for CRUD operations triggered by CoAP requests: Post, Get, Put and Delete can be registered to the main information named coap\_context. Callbacks will be generated to the registered handlers when the requests or replies are received. When creating resources, attributes can be associated. The attributes are then XML encoded when Get messages are received. A command line application, example of code and ETSI tests are included.

#### Copper (Cu) CoAP user-agent Project

##### Overview

The CoAP User Agent is a JavaScript implemention of [Constrained Application Protocol (CoAP) RFC 7252] (http://tools.ietf.org/html/rfc7252) with support for DTLS, Observe and blockwise transfers. A plugin for Mozilla is also included. The project is available on github at: https://github.com/mkovatsc/Copper

The license is 3-Clause BSD with the text available at: http://opensource.org/licenses/BSD-3-Clause, and permits redistribution.

##### How to use

The JavaScript code can be used directly in other JavaScript components.

#### Californium Project

##### Overview

The project implements CoAP RFC 7152 with DTLS, a CoAP-HTTP translator. The implementation is using Java as programming language and is designed for IoT Cloud services with the focus on scalability and usability instead of resource-efficiency like for embedded devices.

The project is available on github at: https://github.com/eclipse/californium The license is business-friendly and of type Eclipse Distribution License, available at http://www.eclipse.org/org/documents/edl-v10.html.

#### How to use

It can be used as a CoAP server that supports all CRUD operations and Observe/Notification mechanism. For example for the Catalogue it would make sense to use it.

### ETSI and oneM2M Projects

#### OM2M Project

##### Overview

The project is developed under the Eclipse umbrella and is described at: http://www.eclipse.org/proposals/technology.om2m/ and available at: http://www.eclipse.org/om2m/

The project is a Java implementation of the ETSI M2M standard in version 0.8.x, available at the moment. It aims to implement also oneM2M in version 1.x.x, the compatibility with ETSI M2M will not be included.

The current version has support for both CoAP and HTTP. For the OMA-DM support it uses SyncML files and can interoperate with [Funambol](http://sourceforge.net/projects/funambol/files/) as server and [Koneki](http://www.eclipse.org/koneki/omadm-simulator/), the OMA-DM simulator for firmware update operation.

The code is licensed using [Eclipse Public License](https://www.eclipse.org/legal/epl-v10.html)

##### How to use

The project could be used to store data coming from sensors or smart devices and expose it to applications.

### OMA Device Management Projects

For exchanging information on the device properties and also monitor/manage connectivity of the device, [OMA LWM2M standard] (http://member.openmobilealliance.org/ftp/Public\_documents/DM/LightweightM2M/) can be used, as an energy efficient and scalable evolution from OMA DM standard.

Several projects have been analyzed in terms of features, flexibility and license in order to be able to choose the most suitable for the Rethink project.

#### Leshan Project

The project is supported by Sierra Wireless and hosted by the Eclipse foundation.

##### Overview

The implementation supports all the interfaces: Bootstrap, Registration, Device Management and Service Enablement, Information Reporting.

The project uses Java as programming language.

The project is hosted on github at: https://github.com/eclipse/leshan

This program and the accompanying materials are made available under the terms of the [Eclipse Public License v1.0] (http://www.eclipse.org/legal/epl-v10.html) and [Eclipse Distribution License v1.0](http://www.eclipse.org/org/documents/edl-v10.html) which accompany the distribution. The license is business-friendly. "Neither the name of the Eclipse Foundation, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission."

##### How to use

The project can be further extended with some new Management Objects, if necessary. The server is to be compiled and run according to a configuration file.

#### OMA LWM2M Dev Kit Project

The project can act as multiple virtual OMA LWM2M clients by connecting to a remote OMA LWM2M server.

##### Overview

The supported features include interfaces: Registration, Device Management and Service Enablement Interface, Information Reporting Interface.

The programming language is Javascript.

The homepage can be found on: https://github.com/OpenMobileAlliance/OMA-LWM2M-DevKit

The license is [BSD-like](https://github.com/OpenMobileAlliance/OMA-LWM2M-DevKit/blob/master/LICENSE), the name of the project has to be mentioned in the redistribution.

##### How to use

The project provides a Web GUI as an addon for Firefox to get the user/developer familiarize. The core functionality can be included in any software package like a Javascript library and also, the Core Framework of the client side for the Rethink project.

# Runtime SOTA

## WebRTC.org

[WebRTC.org](http://www.webrtc.org/)[18] is an open-source project aiming at allowing developers to write applications bringing real-time communication capabilities to browsers, mobile platforms and Internet of Things (IoT) devices, without installing proprietary plugins or extensions. These challenge of integrating these different systems is leveraged by the definition of simple cross-platform APIs.

WebRTC comes with a native code package for developers to work over. This package features audio, video and network transport components. The audio component comes with a complete software stack for voice communications that includes not only codecs, but also software to help in communications' noise reduction, echo cancellation, automatic gain control, between others. The video component is built over the VP8 codec and comes with software for cleaning up noisy images, leveraging packet loss in transmissions and also record/playback functionality. Finally, the network package features components to establish P2P connections using ICE/Turn/STUN/RTP-over-TCP, and also software for error stashing on audio and video communications. Also, WebRTC provides browser developers the ability to choose their own audio, video and network protocols, to work with the packaged software.

### Architecture

WebRTC architecture offers two different layers, one for browser developers and other for third-party application developers. The first one is a C++ API intended to enable the proposed Web API for video/audio capture and render, making it possible for application developers to make use of it. The second one is the Web API for developers to produce applications to interact with WebRTC-powered browsers. Currently, several JavaScript APIs are in process of standardization, like [WebRTC 1.0](http://w3c.github.io/webrtc-pc/)[16] and [Media Capture and Streams](http://w3c.github.io/mediacapture-main/)[17]. In fact, there is another abstract layer responsible for session management and signalling, leaving the signalling protocol implementation up to the application developer, who has to choose between currently existing alternatives.

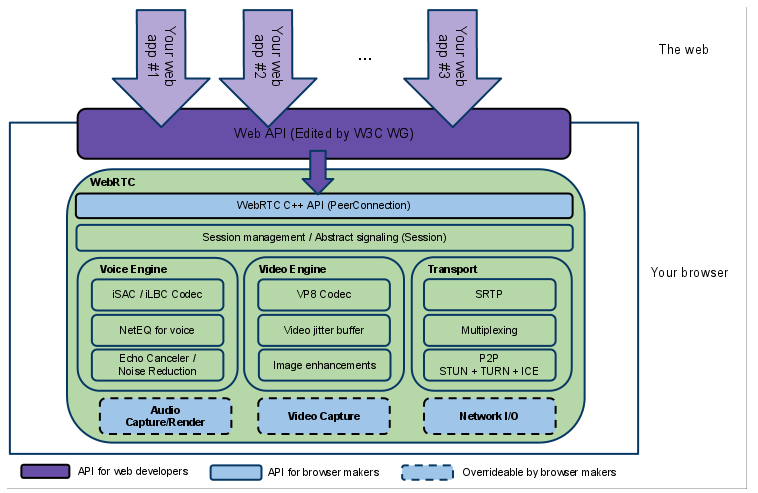


Figure 11: WebRTC.org architecture scheme

### Software stack organization

As explained before, WebRTC.org comes with a software stack that splits itself into a network package, an audio package and a video package.

#### Packages identification

The different packages can be easily identifiable on the WebRTC source code tree. The network package is under src/net and both the audio and video packages are under src/webrtc/, with some mixed up classes. Also, there is not a class diagram which helps developers to get the big picture on this code's organization.

### Code documentation

In the audio/video package almost every file is well-documented. However, the network package doesn't, and it even comes with a README file whose content just states that code documentation is a TODO task on the network package.

### Requirements Analysis

Analysis against **Runtime** requirements

* [The Runtime should be deployable in the most used Devices and Operating Systems](https://github.com/reTHINK-project/core-framework/issues/1)
* WebRTC is intended to be used on latest browser like Google Chrome, Mozilla Firefox, mobile platforms like Android and iOS and also IoT devices like Raspberry Pi.
* [The Runtime should support W3C WebRTC APIs](https://github.com/reTHINK-project/core-framework/issues/2)
* WebRTC.org implements the W3C WebRTC APIs.
* [The runtime must support standard Javascript (ECMAScript)](https://github.com/reTHINK-project/core-framework/issues/3)
* Yes, both the WebRTC 1.0 and Media Capture and Streams APIs use ECMAScript.
* WebRTC.org is meant to be used within a runtime, providing the WebRTC functionality. The runtime will provide Javascript functionality.
* [The Runtime should support Web Socket](https://github.com/reTHINK-project/core-framework/issues/4)
* The WebRTC 1.0 API, and concretely its Peer-to-peer Data API for sending and receiving data models the behaviour of WebSockets
* WebRTC.org is meant to be used within a runtime, providing the WebRTC functionality. The runtime will provide WebSockets functionality.
* [The Runtime should support Standardised Messaging Notifications](https://github.com/reTHINK-project/core-framework/issues/5)
* Yes, WebRTC 1.0 supports Web Messaging Notifications.
* [The Runtime must have a good performance](https://github.com/reTHINK-project/core-framework/issues/6)
* The WebRTC runtime provided as open-source is used in chromium and google chrome. It's performance is state of the art.
* [The Runtime must be secured](https://github.com/reTHINK-project/core-framework/issues/7)
* WebRTC provides encrypted communications betweens peers.
* [The effort to introduce new capabilities in the runtime should be reasonable](https://github.com/reTHINK-project/core-framework/issues/8)
* The effort to perform changes in the runtime like protocols for network I/O, signalling, session management, video capture and audio capture/render depends on the package these changes are meant to be inserted. The audio and video package is well-documented, despite not having a class diagram. The network package, by its turn, is not documented, increasing the effort to understand the functionality and to perform changes in the runtime.

## OpenWebRTC

[OpenWebRTC](http://www.openwebrtc.org/)[19] in another open source reference implementation of WebRTC standard [16][17] that can be used to build native WebRTC apps that communicate with browsers that supports the WebRTC standard, such as Chrome, Firefox and Bowser. OpenWebRTC is especially focused on mobile platforms, with powerful features such as hardware accelerated video coding and OpenGL-based video rendering.

OpenWebRTC architecture is highly modular, enabling easy modifications and possible extensions. Since the WebRTC standard is still evolving, that flexibility is a very desirable trait.

OpenWebRTC is built to support cross-platform operation, supporting IOS, Android, MAC OS and Linux. It is expected that Windows will be supported soon.

It is built on the expectation that several browsers will be able to support its operation. The bulk of the API layer is implemented in JavaScript, making it super fast to modify and extend with new functionality. This is expecially important on mobile platforms with its rapid developping application environment.

With support for both H.264 (OpenH264) and VP8 (libvpx) video codecs, OpenWebRTC is compatible with most video communication services.

The OpenWebRTC project is free and Open Source with a permissive BSD-2 license.

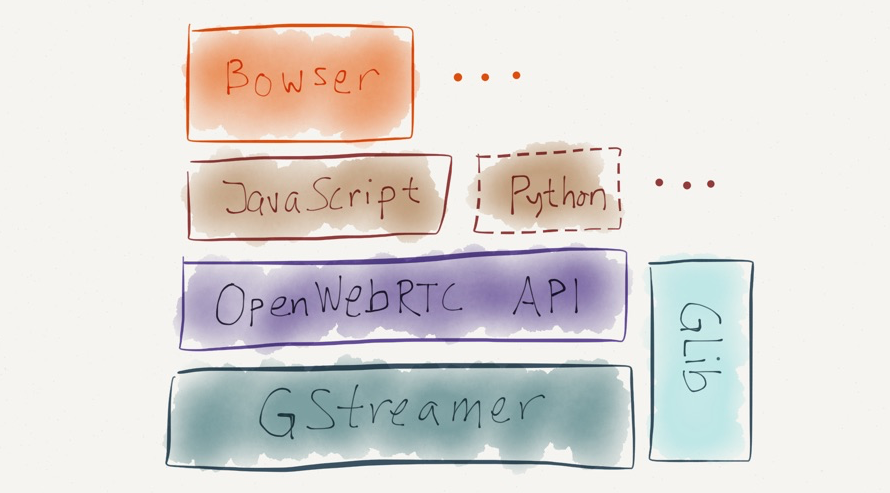


Figure 12 OpenWebRTC Architecture

OpenWebRTC is built on top of widely used and powerful [GStreamer multimedia framework](http://gstreamer.freedesktop.org/)[20]. GStreamer is a set of libraries and plugins that can be used to implement various multimedia applications ranging from desktop players, audio/video recorders, multimedia servers, transcoders, etc. Applications are built by constructing a pipeline composed of elements. An element is an object that performs some action on a multimedia stream such as:

* read a file
* decode or encode between formats
* capture from a hardware device
* render to a hardware device
* mix or multiplex multiple streams

Unfortunately, OpenWebRTC has very few available information making it hard to use and extend it.

## V8 Javascript Engine Evaluation

### Overview

The [V8 JavaScript Engine](https://developers.google.com/v8/) [21] is an open source JavaScript engine developed by Google for the Google Chrome web browser.

V8 compiles JavaScript to native machine code (IA-32, x86-64, ARM, or MIPS ISAs)before executing it, instead of more traditional techniques such as interpreting bytecode or compiling the whole program to machine code and executing it from a filesystem. The compiled code is additionally optimized (and re-optimized) dynamically at runtime, based on heuristics of the code's execution profile. Optimization techniques used include inlining, elision of expensive runtime properties, and inline caching, among many others

### Architecture

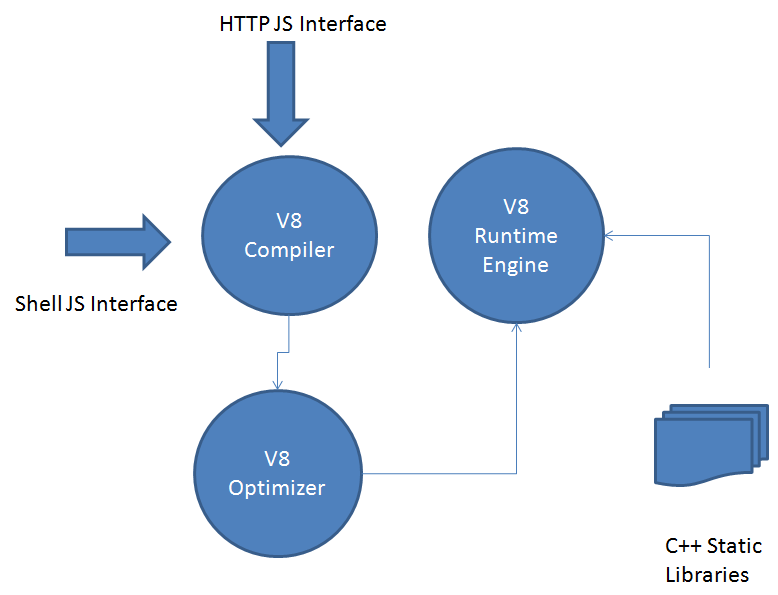


Figure 13 V8 Architecture

**Handles & Garbage Collection**

Handles represent a reference for a Javascript object location on the process heap. The Garbage collector deletes any object on the heap with no valid reference on the process. The Garbage collector besides deleting objectws on the heap frequently moves objects and updates all references to those objects. Obviously the Garbage Collector does not operate often, but from time to time it deletes all obsolete objects. Handles may come in different flavours inside v8, ranging from local handles, which have limited scope and terminate when the scope finishes, therefore susceptible to garbage collection, to persistent and even eternal scopes. In fact we can look on scopes as handle containers. Each time a scope terminates the objects refered by the handlers, in it residing, are flagged for collection. We always have to be in mind that an handle cannot survive its default scope, unless we predetermine its scope to be a special one (EscapableHandleScope ).

**Contexts**

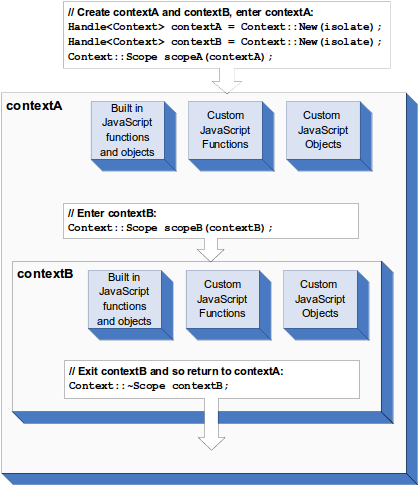


Figure 14 V8 Multiple Contexts

Contexts are different execution environments that allow separate even unrelated Javascript applications to run concurrently on v8. In fact, the context in which a Javascript code is run must be explicitly specified. This happens because Javascript provides functions and objects that may be changed globally and that may turn into unexpected results. One of the advantages of V8 is that it gives you an extensive cache, so in the first time a context may be expensive in time and resources, subsequente times will be substantialy less. Additionally v8 has a snapshot feature that by default has pre-compiled Javascript code on the heap, diminishing time procedures on first context initialization.

**Templates**

Templates are blueprints for Javascript functions and objects in a context. Templates may be used to wrap c++ code onto Javascript objects permiting its manipulation. One can only have one instance of a template on any given context. There are two types of templates:

* function templates - blueprint for a function;
* object templates - each template has associated an object

**Accessors**

Accessors are c++ callbacks that obtain and return a value when an object property is accessed by Javascript. Obviously then can be used to set or read these values. The complexity of them depends on the data being manipulated (Static Global Variables or Dynamic Variables).

**Interceptors**

Interceptors are callbacks used to permit access to an object property. They can be: named property interceptors - when accessing by string names; indexed property interceptors - when the access is made by index.

**Exceptions**

v8 throws exceptions when an error occurs. In fact v8 returns an empty handle on an unsuccessfull call.

**Inheritance**

While Javascript is a class free language, c++ has classes and instances. It is important to take this in consideration because Javascript only has objects, it is a prototype based language. To adapt both we have to refer to templates in v8.

**V8 Code provided for Javascript processing**

process.cc - this code provides the capability to extend the proccess of an HTTP request. The Javascript argument must provide a method named Process() for the execution to succed. This provides an interface for HTTP Javascript introduction on V8 and runtime execution.

shell.cc - this code takes as argument a filename with a Javascript code inside and executes it. It extendes several functionalities to Javascript including a shell capability to run Javascript snipets and their disponibilization to other Javascript code in runtime.

### Requirements Analysis

Analysis against [Hyperty Runtime Requirements](https://github.com/reTHINK-project/core-framework/labels/Runtime%20Requirement) (section ?)

#### [Runtime Performance](https://github.com/reTHINK-project/core-framework/issues/6)

Its aparently clear that V8 provides a significant improvement over previously adopted JavaScript interpretation engines like:

* JScript from IExplorer;
* SpiderMonkey (in Firefox);
* JavaScriptCore (in Safari).

The amount of the improvements will depend on the multiplicity of the calls made to implemented methods. If the methods are made to be run only once the gains would be minimal, otherwise the gains will improve exponentially.

The reasons for these obtained improvements are:

* Fast Property Access - unlike strong type languages like C# and Java, Javascript like Python is a dynamic programing language. This means that properties can be added to and deleted from objects on the fly, so likelly to change over time. Most Javascript engines use a dictionary-like data structure as storage for the object properties. The fetching of each property, on access case, involves a dynamic lookup of the property memory location. This approach turns these accesses much slower than accesses in strong type languages. In these languages, the instance variables are located at fixed offsets determined at compile time due to the fixed layout of objects defined by the object's class. In fact, objects are obtained and stored frequently with only a single instruction. V8 does not use dynamic lookup of properties. It creates hidden classes behind the scenes. Each time a change of property occurs in an object a new hidden class is created and the object changes its representative class for the new hidden class. The hierarchy of hidden classes is mantained and shared each time a new object of the refered type is used again.This type of behaviour promotes reuse by sharing off the hierarchy of hidden classes therefore avoiding dictionary lookups and eficiency by the inline caching of the classes in use.
* Dynamic Machine Code Generation - V8 generates machine code directly from source code the first time the script is executed. A current Javascript engine usually creates intermediate byte code and interpreter. The consequence thus is an object property access is handled with inline cache code in execution that may be patched with further instructions on execution. It may be explained by the execution of an access to an object property, V8 retrieves its associated hidden class and optimizes all future property accesses using this template, providing they share the same scope. This information is used in code patching of the inline cach code. If the V8 has gessed right, the property value is fetched in one operation, otherwise V8 patches the code to remove the optimization. This kind of optimization mirrors the beneficts of static languages and achieves most benefits the more accesses to properties from an object in an wider scope.
* Efficient Garbage Collection - V8, like most garbage collecting languages, reclaims memory used by objects that are no longer used in a process. Obviously garbage collection has known problems like memory fragmentation, pauses for garbage collection and fast object allocation. To avoid those problems as much as possible:
* stops the program execution when in a cycle of garbage collection;
* slices the object heap and only operates on part of it during a collecting cycle - lesses the time the application is stopped;
* correct identification of objects and pointers in memory, avoiding memory leaks by wromg identification.

The V8 separates the heap in two distinct parts. The new space is where new objects are created and the old space where objects surviving a garbage collection cycle are promoted. V8 actualizes references when each cycle finishes.

#### [How to extend and to introduce new Features](https://github.com/reTHINK-project/core-framework/issues/8)

It is possible to extend the functionalities of V8 by adding new modules in c++. These new functionalities would be available to any programer in Javascript where this particular v8 engine resides. V8 provides functions that permit accessing c++ methods and classes, handling errors and enabling security checks. It provides full duality, in which it permits access from javascript scripts to c++ structures an vice-versa.

Code to add a new Javascript code to V8

Handle<Value> Include(const Arguments& args) {  
 for (int i = 0; i < args.Length(); i++) {  
 String::Utf8Value str(args[i]);  
  
 std::string js\_file = load\_file(\*str);  
  
 if(js\_file.length() > 0) {  
 Handle<String> source = String::New(js\_file.c\_str());  
 Handle<Script> script = Script::Compile(source);  
 return script->Run();  
 }  
 }  
 return Undefined();  
}  
  
Handle<ObjectTemplate> global = ObjectTemplate::New();  
  
global->Set(String::New("include"), FunctionTemplate::New(Include));

Obviouly we also have to implement load\_file(). It obtains in string format the content of a file.

#### [Runtime Security](https://github.com/reTHINK-project/core-framework/labels/Runtime%20Requirement)

The "Same Origin Policy" is applied and in fact prevents one document from changing the properties of another. This means one document has the same origin when protocol, domain name and port are the same. This provides a usefull protection against malicious alterations. In v8 origin is defined as its context. To access other context it is necessary to use security tokens and callbacks. The security token are generated by v8 for each context created. when security tokens are not equal a callback must be made to challenge acceptable access.

#### Using Sandboxes with Node.js

[Node.js](https://nodejs.org/en/) [22] is a platform built on Chrome's JavaScript runtime V8 for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

It is open source and specialized for server-side networking applications. Node.js operates on a single thread, using non-blocking I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context-switching. The design of sharing a single thread between all the requests means it can be used to build highly concurrent applications. The design goal of a Node.js application is that any function performing I/O must use a callback.

Used to execute untrusted code. Has support for timeouts preventing infinite loops. Handles errors gracefully. Provides limited access to node.js methods, Supports print and console.log.

**Using Docker to run unsafe code on Node.js**

[Docker](https://www.docker.com/) [23] has a following on its own. Obviously security over docker is a well addressed issue.

#### [Web Messaging Notifications](https://github.com/reTHINK-project/core-framework/issues/5)

Implemented on WebKit on Chromium, so not applied directly on v8.

#### [Web Sockets](https://github.com/reTHINK-project/core-framework/issues/4)

V8 does not have an implementation of web sockets per si. There is an implementation of Web sockets in Chromium, and node.js seems to have an implementation also. It is not considered an issue for v8.

## Firefox OS

### Overview

[Firefox OS](https://www.mozilla.org/en-US/firefox/os/2.0/) (project name: Boot to Gecko, also known as B2G) [24] is a Linux kernel-based open-source operating system for smartphones, tablet computers and smart TVs, developed by Mozilla.

Firefox OS is designed to provide a complete, community-based alternative system for mobile devices, using open standards and approaches such as HTML5 applications, JavaScript, a robust privilege model, open web APIs to communicate directly with cellphone hardware, and application marketplace.

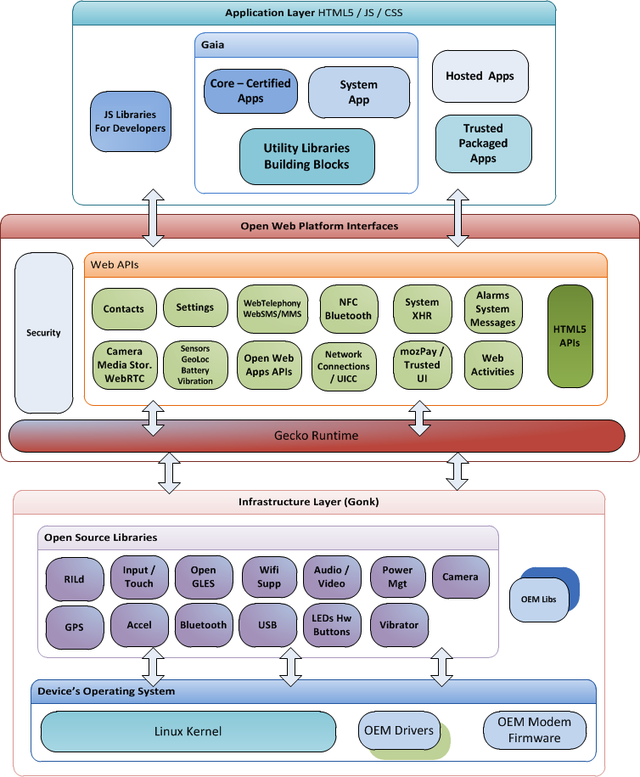


Figure 15 Firefox OS Architecture

The initial development work involves three major software layers:

* Gonk – platform denomination for a combination of the Linux kernel and the HAL from Android
* Gecko – the web browser engine and application run-time services layer;
* Gaia – an HTML5 layer and user-interface system.

[FXOS Web-API status page](https://wiki.mozilla.org/WebAPI) [25].

### Requirements Analysis

Analysis against [Hyperty Runtime Requirements](https://github.com/reTHINK-project/core-framework/labels/Runtime%20Requirement) (section ?)

* [The Runtime should be deployable in the most used Devices and Operating Systems](https://github.com/reTHINK-project/core-framework/issues/1)
* NO
* Firefox OS is made for a certain set of FXOS devices (phones, tables, smart TVs)
* [The Runtime should support W3C WebRTC APIs](https://github.com/reTHINK-project/core-framework/issues/2)
* YES
* Since FXOS version 2.1 this is officially stated as done.
* Tests with 2.0, showed that it basically worked there already
* tested basic A/V calls + separate apps that used the DataChannel to transport arbitrary files
* [The runtime must support standard Javascript (ECMAScript)](https://github.com/reTHINK-project/core-framework/issues/3)
* YES
* "Gecko" is the Javascript interpreter
* provides Javascript access to a lot of Web APIs (even non-standardized)
* Whole UI (Gaia) is based on HTML, Javascript, CSS
* [The Runtime should support Web Socket](https://github.com/reTHINK-project/core-framework/issues/4)
* YES, client side
* Websockets clients are supported
* Websocket servers are not supported
* --> same situation as in a browser runtime
* [The Runtime should support Web Messaging Notifications](https://github.com/reTHINK-project/core-framework/issues/5)
* NO
* according to Web-API status, no indication of planned support
* must be double-checked with practical tests
* what they have is a "Simple Push" API
* [The Runtime must have a good performance](https://github.com/reTHINK-project/core-framework/issues/6)
* very subjective, depends on device hardware it is running on
* tested 3 different devices with rather different experience in terms of performance
* [The Runtime must be secured](https://github.com/reTHINK-project/core-framework/issues/7)
* this would require much more analysis and expertise in attacking the device or the running applications
* general assumptions is that the security is comparable to a browser
* but because the browser IS the middle layer of the OS a potential breakout of the sandbox might have stronger consequences
* [The effort to introduce new capabilities in the runtime should be reasonable](https://github.com/reTHINK-project/core-framework/issues/8)
* YES
* extension with Javascript libraries is possible very easy
* due to the open source nature of the Gecko and Gonk layers it is also possible to add low- and medium-level capabilities there
* The effort for low-level extensions will be relatively high.

o## Jitsi Videobridge [Jitsi Videobridge](https://jitsi.org/Projects/JitsiVideobridge) [26] is a WebRTC compatible Selective Forwarding Unit (SFU) that allows for multiuser video communication.

Jitsi Video bridge supports RTP Relay, audio mixing, Call encryption with DTLS/SRTP and ICE.

### Architecture

JItsi Video bridge is a [XMPP component](http://xmpp.org/) [27] and can be integrated with any compliant XMPP Server like eJabberd or Openfire. In addition there is another XMPP component, the Jicofo, that uses an XMPP extension protocol called COLIBRI (COnferences with LIghtweight BRIdging) to provide conferencing focus functionalities including channels allocation and add / remove participants from each call. Finally, SIP interoperability is provided by a third XMPP component called Jigasi. There is an OpenSource WebRTC JavaScript application, called Jitsi Meet, that uses Jitsi Videobridge to provide high quality, scalable video conferences.

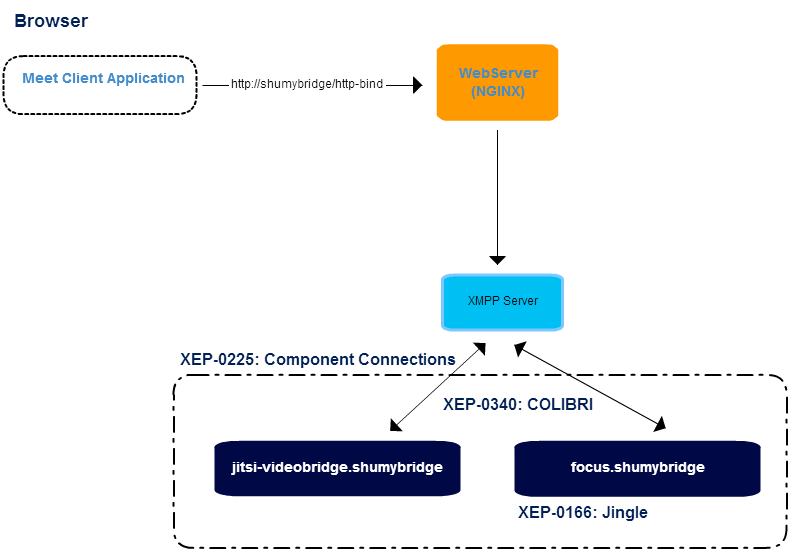


Figure 16 Jitsi Videobridge Architecture

### Installation Procedures

**Required software** \* JVM (select the latest version) \* XMPP Server (openfire, prosody.im, Tigase ...) \* Jitsi VideoBridge (stream XMPP component) \* Jicofo (Session handler XMPP component) \* NGINX (web server and proxy) \* Jitsi Meet App

**Domain selection** \* Select a name for a domain, "shumybridge" will be use for this example. \* Add an entry in DNS hosts file "127.0.0.1 shumybridge".

**XMPP Server (using Openfire)** \* Download and install openfire from http://www.igniterealtime.org/downloads/index.jsp \* Access admin console at http://localhost:9090 \* For the machine name and XMPP domain is important that you use "shumybridge", server certificates will be generated for the domain. \* Select embedded SQLLite database, and an admin user account. Just enough for testing. \* On config "Server -> Server Settings -> HTTP Binding", enable "Script Syntax -> BOSH" and "Provides support for XFF (X-Forwarded-For) headers" \* On config "Server -> Server Settings -> External Components" enable and set the password, ex: xpassword

**Jitsi VideoBridge** \* Download and install Jitsi Videobridge from https://jitsi.org/Projects/JitsiVideobridge \* Run videobridge with: jvb --host=shumybridge --secret=xpassword \* You should see an entry in XMPP components like:

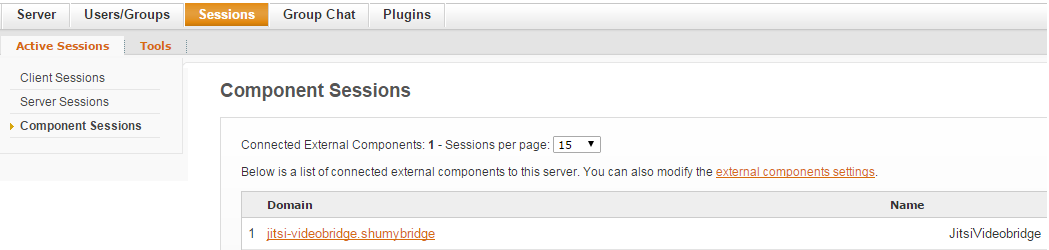


Figure 17: Video Bridge Component

**Jicofo Session** \* Clone from "git clone https://github.com/jitsi/jicofo.git" \* Ant build with "ant dist.{os-name}" \* Add lines "org.jitsi.impl.neomedia.transform.srtp.SRTPCryptoContext.checkReplay=false" and "org.jitsi.jicofo.auth.URL=XMPP:shumybridge" to the file sip-communicator.properties. In Windows this is located at "C:"user"\.sip-communicator-communicator.properties" or in linux "/usr/share/jicofo/.sip-communicator/sip-communicator.properties" \* Run videobridge with: jicofo --host=shumybridge --port=5275 --secret=xpassword \* You should see an entry in XMPP components like:

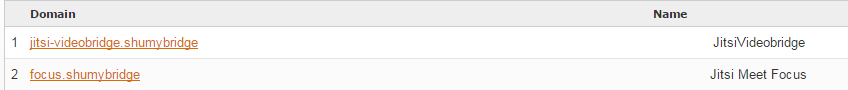


Figure 18: OpenFire VideoBridge and Jicofo Components

1. OpenFire VideoBridge and Jicofo Components

**NGINX** \* Download and install from http://nginx.org/en/download.html \* Change nginx.conf file with:

server {  
 listen 80;  
 server\_name shumybridge;  
  
 location ~ ^/([a-zA-Z0-9]+)$ {  
 rewrite ^/(.\*)$ / break;  
 }  
   
 location / {  
 root srv/jitsi.example.com;  
 index index.html;  
 }  
  
 # BOSH  
 location /http-bind {  
 proxy\_pass http://shumybridge:7070/http-bind/;  
 proxy\_set\_header X-Forwarded-For $remote\_addr;  
 proxy\_set\_header Host $http\_host;  
 }  
   
 # redirect server error pages to the static page /50x.html  
 error\_page 500 502 503 504 /50x.html;  
 location = /50x.html {  
 root html;  
 }  
}

**Jitsi Meet App** \* Clone or download Meet App from https://github.com/jitsi/jitsi-meet.git \* Copy Meet App to NGINX folder ./srv/jitsi.example.com \* Change config.js file with:

var config = {  
 hosts: {  
 domain: 'shumybridge',  
 muc: 'conference.shumybridge',  
 bridge: 'jitsi-videobridge.shumybridge',  
 focus: 'focus.shumybridge'  
 },  
  
 ...  
 bosh: '//shumybridge/http-bind',  
 clientNode: 'http://shumybridge/jitsimeet',  
 ...  
};

### Evalusation of Jitsi Meet Application

Jitsi Meet uses strophe.js internally, but it's clustered with UI dependencies and other non wanted stuff. **Strophe.js** is an XMPP library for JavaScript. Its primary purpose is to enable web-based, real-time XMPP applications that run in any browser. There are Jingle plugins for strophe.js. You need to include the following files in your application from projects [jingle](https://github.com/estos/strophe.jingle) and [strophe](https://github.com/strophe/strophejs):

<!--add jQuery lib-->  
 <script src='strophe/strophe.js'></script><!-- strophe-->  
 <script src='strophe/strophe.disco.js'></script><!-- strophe.disco, optional -->  
 <script src='strophe/strophe.jingle.js' charset='utf-8'></script><!-- strophe jingle connection plugin -->  
 <script src='strophe/strophe.jingle.session.js' charset='utf-8'></script><!-- strophe jingle connection plugin -->  
 <script src='strophe/strophe.jingle.sdp.js' charset='utf-8'></script><!-- sdp library -->  
 <script src='strophe/strophe.jingle.adapter.js' charset='utf-8'></script><!-- getusermedia cross browser compat layer -->

Starting the XMMP session is normaly made with:

var BOSH\_SERVICE = '/http-bind';  
var ICE\_CONFIG = {iceServers: [{url: 'stun:stun.l.google.com:19302'}]};  
  
var DOMAIN = window.location.hostname;  
var CONFERENCEDOMAIN = 'conference.' + DOMAIN;  
  
var connection = null;  
var rtc = null;  
var localStream = null;  
  
var myroomjid = null;  
var roomjid = null;  
var listMembers = [];  
  
$(document).ready(function () {  
 rtc = setupRTC();  
 connection = new Strophe.Connection(BOSH\_SERVICE);  
 connection.jingle.ice\_config = ICE\_CONFIG;  
 connection.jingle.pc\_constraints = rtc.pc\_constraints;  
   
 //nice for debug purposes...  
 connection.xmlInput = function (data) { console.log('RECV: ', data); };  
 connection.xmlOutput = function (data) { console.log('SEND: ', data); };  
});  
  
//call this on a click button (connect)  
getUserMediaWithConstraints(['audio', 'video']);

**getUserMediaWithConstraints** will fire an event configured with jQuery.

$(document).bind('mediaready.jingle', function (event, stream) {  
 localStream = stream;  
 connection.jingle.localStream = stream;  
 RTC.attachMediaStream($(<video-tag>), localStream);  
   
 //connect to videobridge  
 connection.connect(<user>, <pasword>, function (event) {  
 //TODO: handle other connection states Strophe.Status  
 if (status == Strophe.Status.DISCONNECTED) {  
 if (localStream) {  
 localStream.stop();  
 localStream = null;  
 }  
 } else if (status == Strophe.Status.CONNECTED) {  
 connection.jingle.getStunAndTurnCredentials();  
   
 // disco stuff  
 if (connection.disco) {  
 connection.disco.addIdentity('client', 'web');  
 connection.disco.addFeature(Strophe.NS.DISCO\_INFO);  
 }  
   
 //CONNECTED:  
 roomjid = <hash> + '@' + CONFERENCEDOMAIN; //select room id  
 myroomjid = roomjid + '/' + Strophe.getNodeFromJid(connection.jid);  
  
 //config XMPP presence event handlers...  
 connection.addHandler(onPresence, null, 'presence', null, null, roomjid, {matchBare: true});  
 connection.addHandler(onPresenceUnavailable, null, 'presence', 'unavailable', null, roomjid, {matchBare: true});  
 connection.addHandler(onPresenceError, null, 'presence', 'error', null, roomjid, {matchBare: true});  
  
 var pres = $pres({to: myroomjid }).c('x', {xmlns: 'http://jabber.org/protocol/muc'});  
 connection.send(pres);  
 }  
 });  
});

and define presence handlers:

function onPresence(pres) {  
 var from = pres.getAttribute('from');  
 var type = pres.getAttribute('type');  
   
 if (type !== null) {  
 return true;  
 }  
   
 if ($(pres).find('>x[xmlns="http://jabber.org/protocol/muc#user"]>status[code="201"]').length) {  
 // http://xmpp.org/extensions/xep-0045.html#createroom-instant  
 var create = $iq({type: 'set', to: roomjid})  
 .c('query', {xmlns: 'http://jabber.org/protocol/muc#owner'})  
 .c('x', {xmlns: 'jabber:x:data', type: 'submit'});  
 connection.send(create); // fire away  
 }  
   
 //manage list members  
 if (from == myroomjid) {  
 for (i = 0; i < listMembers.length; i++) {  
 connection.jingle.initiate(listMembers[i], myroomjid);  
 }  
 } else {  
 listMembers.push(from);  
 }  
   
 return true;  
}  
  
function onPresenceUnavailable(pres) {  
 connection.jingle.terminateByJid($(pres).attr('from'));  
  
 //manage list members  
 for (var i = 0; i < listMembers.length; i++) {  
 if (listMembers[i] == $(pres).attr('from')) {  
 listMembers.splice(i, 1);  
 break;  
 }  
 }  
   
 return true;  
}  
  
function onPresenceError(pres) {  
 //TODO: process error  
 return true;  
}

Handle add/remove video/audio streams:

$(document).bind('remotestreamadded.jingle', function (event, data, sid) {  
 var el = $("<video autoplay='autoplay' style='display:none'/>").attr('id', 'largevideo\_' + sid);  
 RTC.attachMediaStream(el, data.stream);  
 });  
   
 $(document).bind('remotestreamremoved.jingle', function (event, data, sid) {  
 //TODO: remove video element  
 });

## Docker

[Docker](https://www.docker.com/) [23] is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components.

Docker containers are lightweight and fast. Containers have sub-second launch times, reducing the cycle time of development, testing, and deployment.

Docker consists of:

* The Docker Engine - lightweight and powerful open source container virtualization technology combined with a work flow for building and containerizing your applications.
* Docker Hub - SaaS service for sharing and managing your application stacks.

Docker is a standard container format that lets developers care about their applications inside containers while sysadmins and operators can work on running the container in the deployment environment. This separation of duties streamlines and simplifies the management and deployment of code.

Docker containers run (almost) everywhere including desktops, physical servers, virtual machines, into data centers, and up to public and private clouds. Since Docker runs on so many platforms, it's easy to move applications around including moving an application from a testing environment into the cloud and back. Docker's lightweight containers also make scaling up and down fast and easy. More containers can be launched when needed and then shut them down easily when they're no longer needed.

### Architecture

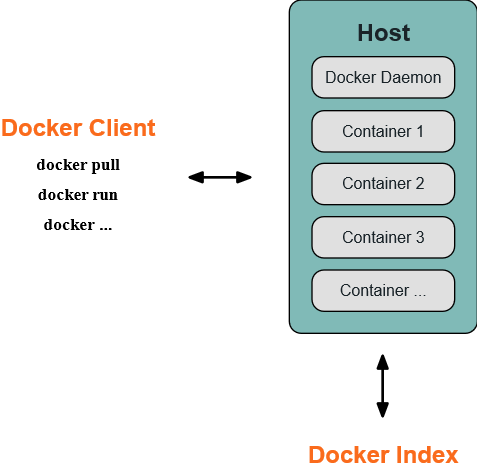


Figure 19 Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. Both the Docker client and the daemon can run on the same system, or can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate via sockets or through a RESTful API.

#### How to obtain security on standalone components using Docker.

To understand how to obtain security using Docker we have to look at its architecture:

The Docker daemon - the Docker daemon runs on a host machine. The user does not directly interact with the daemon, but instead through the Docker client.

The Docker client - The Docker client, in the form of the docker binary, is the primary user interface to Docker. It accepts commands from the user and communicates back and forth with a Docker daemon.

To understand Docker, we need to understand its three components:

Docker images - A Docker image is a read-only template. For example, an image could contain an Android minimal operating system with a minimal HTTP demon and a web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or we can download Docker images that other people have created

Docker registries - Docker registries hold images. These are public or private stores from which we upload or download images. These can be images we create ourselves or we can use images that others have previously created.

Docker containers - Docker containers are similar to a directory. A Docker container holds everything that is needed for an application to run. Each container is created from a Docker image. Docker containers can be run, started, stopped, moved, and deleted. Each container is an isolated and secure application platform.

Obviously the containers are the base for our security architecture. Each container is absolutely independent and each interface for comunication must be stricly observed. There is no other way to access the container and its content in operation.

### How does a Docker image works

We've already seen that Docker images are read-only templates from which Docker containers are launched. Each image consists of a series of layers. Docker makes use of union file systems to combine these layers into a single image. Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

One of the reasons Docker is so lightweight is because of these layers. When we change a Docker image—for example, update an application to a new version— a new layer gets built. Thus, rather than replacing the whole image or entirely rebuilding, as we may do with a virtual machine, only that layer is added or updated. Now we don't need to distribute a whole new image, just the update, making distributing Docker images faster and simpler.

Every image starts from a base image, for example an Android base image. Or we can also use images of our own as the basis for a new image, for example if we have a base HTTPD image we could use this as the base of all our web application images.

### How to build a Docker image

Docker images are then built from base images using a set of steps we call instructions. Each instruction creates a new layer in our image. Instructions include actions like:

* Run a command.
* Add a file or directory.
* Create an environment variable.
* What process to run when launching a container from this image.

These instructions are stored in a file called a Dockerfile. Docker reads this Dockerfile when we request a build of an image, executes the instructions, and returns a final image.

### How does a Docker registry work

The Docker registry is the store for our Docker images. Once we build a Docker image we can push it to a public registry Docker Hub or to our own registry running behind our firewall.

Using the Docker client, we can search for already published images and then pull them down to our Docker host to build containers from them.

Docker Hub provides both public and private storage for images. Public storage is searchable and can be downloaded by anyone. Private storage is excluded from search results and only we and our users can pull images down and use them to build containers.

### How does a container work

A container consists of an operating system, user-added files, and meta-data. As we've seen, each container is built from an image. That image tells Docker what the container holds, what process to run when the container is launched, and a variety of other configuration data. The Docker image is read-only. When Docker runs a container from an image, it adds a read-write layer on top of the image in which our application can then run.

Either by using the docker binary or via the API, the Docker client tells the Docker daemon to run a container.

$ sudo docker run -i -t ubuntu /bin/bash

Let's break down this command. The Docker client is launched using the docker binary with the run option telling it to launch a new container. The bare minimum the Docker client needs to tell the Docker daemon to run the container is:

What Docker image to build the container from, here ubuntu, a base Ubuntu image; The command we want to run inside the container when it is launched, here /bin/bash, to start the Bash shell inside the new container.

when we run this command, the following actions are performed:

1 - Pulls the ubuntu image. Docker checks for the presence of the ubuntu image and, if it doesn't exist locally on the host, then Docker downloads it from Docker Hub. If the image already exists, then Docker uses it for the new container.

2 - Creates a new container. Once Docker has the image, it uses it to create a container.

3 - Allocates a filesystem and mounts a read-write layer. The container is created in the file system and a read-write layer is added to the image.

4 - Allocates a network / bridge interface. Creates a network interface that allows the Docker container to talk to the local host.

5 - Sets up an IP address. Finds and attaches an available IP address from a pool.

6 - Executes a process that we specify. Runs our application.

7 - Captures and provides application output. Connects and logs standard input, outputs and errors for we to see how our application is running.

### Dockerizing a Node.js Web App

The goal of this example is to show how to build your Docker images from a parent image using a Dockerfile. We will do that by making a simple Node.js hello world web application running on CentOS.

Create Node.js app

First, create a directory src where all the files would live. Then create a package.json file that describes your app and its dependencies:

{  
 "name": "docker-centos-hello",  
 "private": true,  
 "version": "0.0.1",  
 "description": "Node.js Hello world app on CentOS using docker",  
 "author": "Miguel Mesquita <mesquita@av.it.pt>",  
 "dependencies": {  
 "express": "3.2.4"  
 }  
}

Then we need to create an index.js file that defines a web app using the Express.js framework:

var express = require('express');  
  
// Constants  
  
var PORT = 8080;  
  
// App  
  
var app = express();  
  
app.get('/', function (req, res) {  
 res.send('Hello world\n');  
});  
  
app.listen(PORT);  
  
console.log('Running on http://localhost:' + PORT);

We'll look at how to run an application inside a CentOS container using Docker. First we need to build a Docker image of our app.

**Creating a Dockerfile**

Create an empty file called Dockerfile:

$ touch Dockerfile

Open the Dockerfile in your favorite text editor (I'm an old fashioned guy, I use vi)

Define the parent image we want to use to build your own image on top of. Here, we'll use CentOS (tag: centos6) available on the Docker Hub:

FROM centos:centos6

Since we're building a Node.js app, we have to install Node.js as well as npm on your CentOS image. Node.js is required to run our app and npm to install our app's dependencies defined in package.json. To install the right package for CentOS, we'll use the instructions from the Node.js wiki:

# Enable EPEL for Node.js  
  
RUN rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm  
  
# Install Node.js and npm  
  
RUN yum install -y npm

To bundle our app's source code inside the Docker image, we use the COPY command:

# Bundle app source  
  
COPY . /src

Install our app dependencies using the npm command:

# Install app dependencies  
  
RUN cd /src; npm install

Our app binds to port 8080 so we use the EXPOSE command to have it mapped by the docker daemon:

EXPOSE 8080

Define the command to run our app using CMD which defines our runtime, i.e. node, and the path to our app src/index.js (see the step where we added the source to the container):

CMD ["node", "/src/index.js"]  
  
Our Dockerfile should now look like this:  
  
  
FROM centos:centos6  
  
# Enable EPEL for Node.js  
  
RUN rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm  
# Install Node.js and npm  
  
RUN yum install -y npm  
  
# Bundle app source  
  
COPY . /src  
# Install app dependencies  
  
RUN cd /src; npm install  
  
EXPOSE 8080  
CMD ["node", "/src/index.js"]

**Building our image**

Go to the directory that has our Dockerfile and run the following command to build a Docker image. The -t flag adds a tag to our image so it's easier to find later using the docker images command:

$ sudo docker build -t <your username>/centos-node-hello .

Our image will now be listed by Docker:

$ sudo docker images  
  
# Example  
  
REPOSITORY TAG ID CREATED  
centos centos6 539c0211cd76 8 weeks ago  
<your username>/centos-node-hello latest d64d3505b0d2 2 hours ago

**Run the image**

Running our image with -d runs the container in detached mode, leaving the container running in the background. The -p flag redirects a public port to a private port in the container. Run the image we previously built:

$ sudo docker run -p 49160:8080 -d <your username>/centos-node-hello

To print the output of our app:

# Get container ID  
  
$ sudo docker ps  
  
# Print app output  
  
$ sudo docker logs <container id>  
  
# Output

Running on http://localhost:8080

**Test**

To test our app, get the port of our app that Docker mapped:

$ sudo docker ps  
  
# Example  
  
ID IMAGE COMMAND ... PORTS  
ecce33b30ebf <your username>/centos-node-hello:latest node /src/index.js 49160->8080

In the example above, Docker mapped the 8080 port of the container to 49160.

Is our application working? Lets test it with curl (ok install it with sudo apt-get install curl)

$ curl -i localhost:49160  
  
HTTP/1.1 200 OK  
  
X-Powered-By: Express  
  
Content-Type: text/html; charset=utf-8  
  
Content-Length: 12  
  
Date: Sun, 02 Jun 2013 03:53:22 GMT  
  
Connection: keep-alive   
  
Hello world

Yes!!! It's working.

Every application must connect through the port. The code is absolutely isolated from misuse. We implicitly have created an internal virtual net using docker from a internal pool of IPs docker has assumed on instalation. Every new application has a brand new on initiation which is relented on finishing the app. For the external user, well, it is invisible.

## Janus Gateway

[Janus Gateway](https://janus.conf.meetecho.com/) [29] is a WebRTC gateway implemented in C, which means it doesn't provide any functionality for itself, apart from establishing a WebRTC media connection between browsers and server-side applications they might be using (e.g. echo tests, conference bridges, media recorders). This communication consists in exchanging JSON messages and relaying RTP/RTCP packets between a browser and the application logic. Janus is intended to be a light component to cover a big range of use-cases. It can be used to deploy a full-fledged WebRTC gateway on a cloud provider or just a small nettop/box to handle a specific use case, looking at applications as pluggable modules that a client can connect to through this gateway.

### Architecture and APIs

Janus Gateway splits itself into three software modules. The Core module implements the WebRTC protocols (SDP, ICE, DTLS-SRTP, RTP/RTCP), which are themselves listed in the Protocols module, in a web server that interacts with browsers. This Core module takes care of aspects like session handling, media signalling, negotiation and plugins availability. The Plugins module provides some out-of-the-box plugins to be integrated or extended into new applications developed on top of Janus core. The already available plugins perform media record and play and establish streaming, voice mail or SIP connections, just to mention some useful examples. There is also a Plugin API which offers potential plugin developers an overview of how these plugins are implemented and also instructions of how to develop a new one.

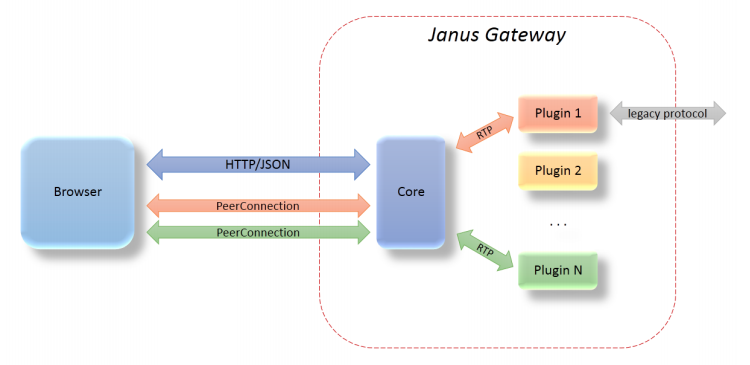


Figure 20 Janus Gateway architecture

### Janus Gateway and runtime requirements

#### [The Runtime should be deployable in the most used Devices and Operating Systems](https://github.com/reTHINK-project/core-framework/issues/1)

Janus Gateway web servers can currently be deployed on Linux systems only, and a cross-platform version is not going to be developed for now. For client-side, Google Chrome and Mozilla Firefox are the supported browsers.

#### [The Runtime should support W3C WebRTC APIs including](https://github.com/reTHINK-project/core-framework/issues/2)

Janus Gateway implementation of the WebRTC stack supports the W3C WebRTC APIs.

#### [The runtime must support standard Javascript (ECMAScript)](https://github.com/reTHINK-project/core-framework/issues/3)

Janus Gateway supports standard JavaScript in its implementation. Concretely, it offers a JavaScript library (janus.js) which allows developers to access both the REST and WebSocket interfaces. This library facilitates the task of establishing sessions between clients and the gateway, attaching plugins to clients, exchange events between them and so on.

#### [The Runtime should support Web Socket](https://github.com/reTHINK-project/core-framework/issues/4)

As said above, Janus Gateway offers a WebSocket API which developers may choose instead of the default plain HTTP REST API to interact with the gateway.

#### [The Runtime should support Standardised Messaging Notifications](https://github.com/reTHINK-project/core-framework/issues/5)

Janus can send events and notifications at any time through the long poll channel or the related push mechanisms made when using WebSockets. On plain HTTP, a user has to explicitly subscribe to notifications, repeating that as soon as an event has been received. However, the push mechanisms used by the WebSockets makes this task easier: as soon as a client creates a session through a WebSocket, that channel becomes automatically subscribed for events related with that session, and these events of interest will be then received through the same channel.

#### [The Runtime must have a good performance](https://github.com/reTHINK-project/core-framework/issues/6)

Measuring the performance of Janus is a complicated task, since it is just a gateway. Thus, the performance of plugins and applications written by third-party developers and working with Janus takes an important role on the measurements.There was a [recent study](http://dl.acm.org/citation.cfm?id=2749223) [30] on the performance of the Janus Gateway, when applied to several use-cases. Also, when [comparing the Video MCU conferencing plugin](http://www.rtc-conference.com/wp-content/uploads/gravity_forms/2-2f7a537445fa703985ab4d2372ac42ca/2014/09/Romano_Janus.pdf)[31] with other similar systems like [Jitsi](https://jitsi.org/) [26] and [Licode](http://lynckia.com/licode/), it revealed strong improvements on CPU and memory usage, both on client and server sides.

#### [The effort to introduce new capabilities in the runtime should be reasonable](https://github.com/reTHINK-project/core-framework/issues/8)

Due to its modular architecture, in which plugins can be seen as "bricks" in an application, introducing new features like a policy engine or hyperty registry should not be a very hard task.

## Kurento Media Server

### Overview

[Kurento](http://www.kurento.org/) is an Open Source Software WebRTC media server, that can be used to manage media flows :

* Send / Receive
* Recording
* Transcoding
* Augmented reality
* Mixing
* broadcasting

It can be used to handle different type of communications applications : 1 to 1, N to N, 1 to N (Real time exchanges or streaming)

### Architecture

Kurento is mainly composed of the two elements : - Kurento media server - Kurento Application

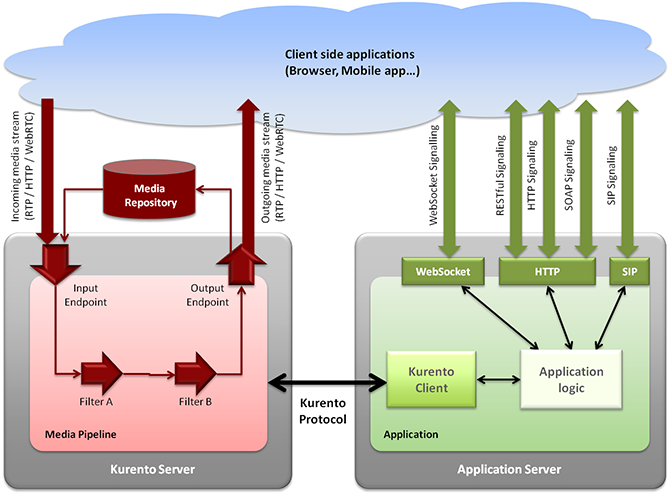


Figure 21: Kurento Architecture

Application developers can use Kurento Clients or Kurento API directly for creating their multimedia enabled applications. Developpers can use Javascript clients, Java Client or Kurento Protocol. This is interesting as it can easily be integrated with NodeJs

### APIs

*Available APIs for developers*

Media server API is describe here : http://www.kurento.org/docs/current/mastering/kurento\_API.html

Kurento Clients are also available for application developers :

JAVA : http://www.kurento.org/docs/current/langdoc/javadoc/index.html

JavaScript : http://www.kurento.org/docs/current/langdoc/jsdoc/kurento-client-js/index.html http://www.kurento.org/docs/current/langdoc/jsdoc/kurento-utils-js/index.html

### Integration in Rethink

Multiparty conversations supported with MCU/SFU for Star topologies can be supported with server side Hyperties running in the MCU/SFU ie there would be protofly in the MCU/SFU.

Kurento Media Server can be connected through a NodeJs Client : it will be possible to add protOfly interface on nodeJs to then connect to the MCU.

# Messaging SOTA

## Vert.x 2.0

### Overview

Vert.x 2.0 [57] is an application framework developed by VMWare in 2011. The application framework provides possibilities to develope loosely coupled network service applications.

The concept of the framework is summarized as follows: \* **Polyglot (supports several languages)**: Vert.x framework runs on the Java Virtual Machine. However, Java is not required to use Vert.x. As well as languages based on JVM operation, such as Java or Groovy, Vert.x can be used with Ruby, Python, and even JavaScript. In addition, Scala and Closure are planned to be supported. \* **Super Simple Concurrency model**: When building an application by using Vert.x, users can write code as a single thread application. That means that the multi-thread programming effect can be achieved without synchronization, lock, or volatility. However, Vert.x allows to create multiple threads based on the number of CPU cores whlie only one process is executed. It handle the multi-threading so users can focus on implementing business logic. \* **Provides Event Bus**: The main concept of Vert.x is not only to produce a ‘one server process DAEMON'. Vert.x aims to make a variety of Vert.x-built server programs communicate well with each other. For this, Vert.x provides Event Bus. Therefore, functions such as Point to Point or Pub/Sub can be used (to provide Event Bus function, Vert.x uses Hazelcast, an In-Memory Data Grid). With this Event Bus, a server application built with different languages can easily communicate with each other. \* **Module System & Public Module Repository**: Vert.x has a module system. This module system can be understood as a type of component. That means the Vert.x-built server application project itself is modularized. It aims at reusability. Modules can be registered to Public Module Repository. Through the Public Module Repository, the module can be shared

### Architecture

This subsection highlights the main building blocks of the Vert.x architecture.

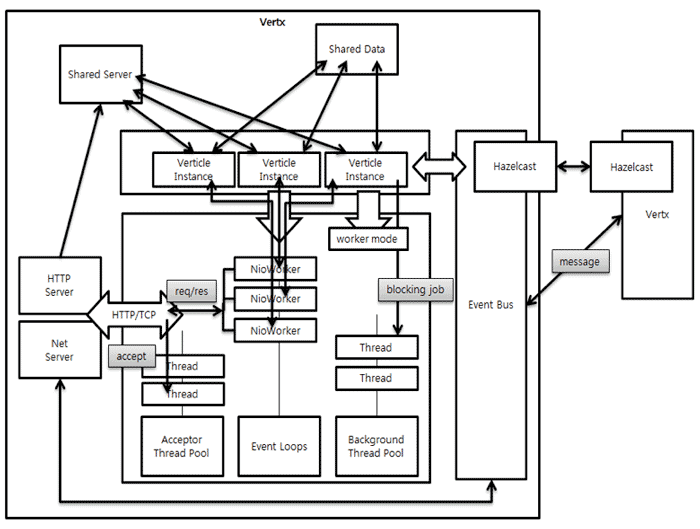


Figure 35: Vertx Architecture Diagram

### Addressing

Messages are sent on the event bus to an address. Vert.x instances are not bound to any addressing schemes. An address is simply a string, any string is valid. Some examples of valid addresses are europe.news.feed1, acme.games.pacman, sausages, and X. As a convention the names of the packages that implement certain functionalities should also be represented on the event bus and should be combined with a meaningful event/operation name, e.g. org.acme.MyPackage.MyClass.doSomething

### Handlers

A handler is an entity that receives messages from the event bus. You register a handler at an address. Many different handlers from the same or different modules can be registered at the same address. A single handler can be registered at many different addresses at the same time.

### Messaging Schemes

The Event Bus supports the following modes of operation: \* *Publish / subscribe messaging*: Publishing means delivering the message to all handlers that are registered at that address. This is the familiar publish/subscribe messaging pattern. \* *Point to point and Request-Response messaging*: Messages are routed to just one of the handlers registered at an address. They can optionally be replied to. \* *Remote Procedure Call (RPC)*: This mode of operation is implemented on top of the Request-Response model, basically by enforcing certain conventions on requests and responses

This example shows the Event Bus can be instantiated, how a Handler can be defined and registered on the Event Bus and how the Event Bus can subsequently publish a message for the defined Handler:

EventBus eb = vertx.eventBus();  
  
Handler<Message> myHandler = new Handler<Message>(){  
  
 public void handle(Message message){  
 System.out.println("I just recieved a message "+ message.body);  
 }  
};  
//test.address is the address at which this handler will be registered  
eb.registerHandler("test.address", myHandler);  
  
...  
//publishing a message. The message will be delivered to all handlers registered against the address  
eb.publish("test.address", "hello world");  
//point-2-point sending of message.   
//Only one handler registered at the address receiving the message.   
//The handler is chosen in a non strict round-robin fashion  
eb.send("test.address", "hello world");  
  
...  
  
eb.unregisterHandler("test.address", myHandler);

### Types of Messages

Messages that you send on the event bus can be as simple as a string, a number or a boolean. It is also possible to send Vert.x buffers or JSON messages. It's highly recommended to use JSON messages to communicate between verticles. JSON is easy to create and parse in all the languages that Vert.x supports. For RPC messages, JSON is enforced.

### Verticle

The unit of execution for Vert.x applications is called a Verticle. Verticles can be written in multiple languages (JavaScript, Ruby, Java, Groovy or Python). Many verticles can be executed concurrently in the same Vert.x instance. An application might be composed of multiple verticles deployed on different nodes of the network communicating by exchanging messages over the Vert.x Event Bus. For trivial applications verticles can be run directly from the command line, but more usually they are packaged up into modules.

### Module

Applications within the framework comprise of one or more modules. The framework allows packaging of applications or other re-usable functionality into modules, which can be deployed or used by other code. Module can also by catalogue in the Vert.x module registry so others can discover and use it. The framework offers the possibility to automatically download and install modules from any repository given the module identifier. Each module has a unique identifier. The identifier is a string that is composed of three parts: A module can contain any number of (including zero) verticles and can depend on other modules (and their verticles) in turn. Creating a module with no verticles makes sense to provide only library support for other modules. Modules are described by a descriptor file: mod.json. A minimal descriptor looks like this:

{  
 "owner": "org.acmecorp",  
 "name": "myReThinkAdapterModule",  
 "version": "0.1"  
}

Additionally, three more fields are optionally recognized: \* worker indicates if this is a worker module. See below under event loop. \* main Indicates the startup routine for this module. \* includes Additional module dependencies as a comma-separated string.

### Event Loop

By default, all verticles run in an asynchronous event loop. When developing a verticle, it is essential not to block the event loop. Blocking here means either doing any kind of blocking I/O or even doing any kind of computational intensive work. Modules that do either of these should indicate that they are so called worker modules by setting "worker": true in their *mod.json* file. The advantage of an event loop is that it is enormously scalable. Instead of waiting for I/O operations to complete, the executing thread will rather do other stuff (e.g. servicing the next request) in the meantime. This is achieved by using a callback driven style of programming. Imagine the following scenario: *We want to read some data in an I/O intensive operation (function readData)* We want to do something with that data (function doSomething) *We want to do something completely different (function doSomethingUnrelated)* In the traditional blocking world we would do something like the following:

def doSomething(data):  
 # do something with data  
data = readData()  
doSomething(data)  
doSomethingUnrelated()

What happens here is the following:

After the data is read, the program waits for the operation (readData) to complete (which is consuming the event loop thread lifetime). As soon as readData returns, we have our data and can go on to do something with it (doSomething(data)). Finally, when that is done, we can go on and do other stuff (doSomethingUnrelated).

In the asynchronous world, we do something like this:   
def doSomething(data):  
 # do something with data  
readData(callback = doSomething)  
doSomethingUnrelated()

As can be seen, the result of readData is not received in the functions return value. Instead doSomething is passed in the handler method as a callback. The framework will take care that this handler is called asynchronously as soon as the data is available

### APIs

Vert.x provides the different APIs which are implemented in various languages:

**Core API** \* TCP client/Server API \* HTTP client/Server API \* Transport Protocol (Websocket, SockJS(provides websocket-like API through http), UDP, TCP) \* File System Access \* DNS client API \* Shared Data \* Event Bus API \* JSON API

**Container API** \* Deploy and undeploy verticles \* Deploy and undeploy modules \* Retrieve verticle configuration \* Logging

### Requirements Analysis

*According to Component Type addressed by the solution ie Messaging Node, Runtime, Network QoS and Framework*

#### [Autentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)

External Authentication and Authorisation are supported through the usage of an Authorisation module:

container.deployModule("io.vertx~mod-auth-mgr~2.0.0-final");

The Authorisation module can be the front-end to interact with an external vertx service eg with restful APIs or could be attached to the vertx-io event bus.

**Authorisation to Send/publish a Message**

* SockJSServer, where we need a bridge configuration
* InboudPermitted must have: vertx.basicauthmanager.login and clients handler ```java JsonArray inboundPermitted = new JsonArray();

JsonObject inboundPermitted1 = new JsonObject().putString("address", "vertx.basicauthmanager.login"); inboundPermitted.add(inboundPermitted1); JsonObject inboundPermitted2 = new JsonObject().putString("address", "aliceHandler").putBoolean("requires\_auth",true); inboundPermitted.add(inboundPermitted2); ``` Inboundpermitted allows the use of the "requires\_auth" flag. When it is true, messages will be first forwarded to the authorization module, where decisions to send or not the messages are taken.

**receive a Message**

* OutboundPermitted must have: clients handler. java outboundPermitted.add(new JsonObject().putString("address", "aliceHandler"));
* sockJSServer.bridge(new JsonObject().putString("prefix", "/eventbus"), inboundPermitted, outboundPermitted);

**Example: communication between two javascript clients connected via SockJS**

Both client applications perform log-in on the EventBus.

eb.login('alice','alice123', function(reply){console.log(reply);});

Then the client A(alice) wants to send messages to the client B(bob), so client B(bob) needs to register a handler. Before the client A(alice) can send a message to client B(bob), B must first register himself.

eb.registerHandler('bobHandler', function(reply){console.log(reply);});

After that client A (alice) can publish messages on client B (bob) handler

eb.publish('bobHandler','Hello bob from alice');

When client A publishes a message to client B handler, this message will be first forwarded to the authorization module because of inbounpermitted configuration.

**subscribe / register handlers to be notified about published messages**

In the SockJSServer configuration we can set a Hook (Registers functions to be called when certain events occur on an event bus bridge).

ServerHook hook = new ServerHook(logger);  
sockJSServer.setHook(hook);

ServerHook takes some keyword arguments for example:

* pre-register: Called before a client handler registration is processed.
* public boolean handlePreRegister(SockJSSocket sock, String address) {  
  logger.info("handlePreRegister, sock = " + sock + ", address = " + address);  
  return true;  
   }

In this way handlers registration can be controlled.

#### [Unstable Connections](https://github.com/reTHINK-project/core-framework/issues/15)

Since vertx is based on http://hazelcast.org/ we can use it to cache some info including the sessionId

#### [Carrier grade deployment features (Resilience, DoS and DDoS protection, Service Assurance)](Messaging%20Node%20with%20carrier%20grade%20deployment%20features)

* Resilience: Vert.x provides resilience through the "automatic failover" and "HA group" options. When a module is run with HA, if the Vert.x instance where it is running fails, it will be re-started automatically on another node of the cluster. An HA group denotes a logical grouping of nodes in the cluster. Only nodes with the same HA group will failover onto one another.
* DoS and DDoS Protection: Vert.x 2.x. has no support for this, BUT Vert.x 3.0 provides built-in core functiionality for this core
* Service Assurance: Modules can be deployed in clusters, and Vert.x provides an internal Load Balancer for routing messages within the cluster. Also the above mentioned "auomatic failover" and "HA group" options contribute to enforce service assurance.

#### [Scalability] (https://github.com/reTHINK-project/core-framework/issues/16)

Verticle instances, except advanced multi-threaded worker verticles are almost always single threaded. what this implies is that, a single verticle instance can at most utilise one core of the server. In order to scale across cores, several verticles which are responsible for the same task can be instantiated and the runtime will distribute the workload among them (load balancing), this way taking full advantage of all SPU cores without much effort. Verticles can also be distributed between several machines. This will be transparent to the application code. The Verticles use the same mechanisms to communicate as if they would run on the same machine. This makes it very easy to scale applications.

#### [Messaging Transport Protocols] (https://github.com/reTHINK-project/core-framework/issues/20)

* Websockets - Yes supported
* SockJS - Yes supported
* HTTP Long-Polling - Yes
* HTTP Streaming - ? (Not sure what this means, clarification needed)

#### [Message delivery reliability] (https://github.com/reTHINK-project/core-framework/issues/17)

No. Vert.x uses the Event Bus to send messages through pub/sub mechanism or point-2-point mechanism. In both cases, there is no feedback to the sender if the message was recieved and processed or if it was not recieved at all. In the end reliability will boil down to the application logic service build on top of vert.x.

## Vert.x 3.0

### Overview

This evaluation will evidence differences between version 2 [57] and 3 [58] of vert.x. It will not describe all the architecture as in the version 2 evaluation.

The concept of the framework is summarized as follows: \* **Polyglot (supports several languages)**: Vert.x framework runs on the JVM. However, Java is not required to run a Verticle. Main languages supported in version 3 are Java, JavaScript, Groovy and Ruby. \* **Concurrency model**: Concurrency model has not changed between versions. \* **Provides Event Bus**: Event bus is still available and is an essential part of vert.x engine for communication between programs, even when written in different languages. The event bus even penetrates into in-browser JavaScript allowing you to create effortless so-called real-time web applications. \* **Module System & Public Module Repository**: It seems that this feature is dropped in version 3. There is no more methods like deployModule(..) available. Now you create your verticles and package them into standard java jars. These jars can be pushed to Maven repositories just like any Maven artifact. For static module imports this is enough, however if we need dynamic module maintenance, an OSGi container could be used instead.

### Architecture

This subsection highlights the main building blocks of the Vert.x architecture. (diagram not supplied)

### Vert.x Runtime (Java 8 only)

Vert.x 3.0 is Java 8 only. This is to take advantage of new language features in Java 8, the most important of which is Lambdas which make developing against event based APIs so much nicer than in previous versions of Java. And is also chosen so that it can use Nashorn - the new high performance JavaScript engine that it contains.

### Addressing

Messages are sent on the event bus to an address. Vert.x instances are not bound to any addressing schemes. An address is simply a string, any string is valid. Some examples of valid addresses are europe.news.feed1, acme.games.pacman, sausages, and X. As a convention the names of the packages that implement certain functionalities should also be represented on the event bus and should be combined with a meaningful event/operation name, e.g. org.acme.MyPackage.MyClass.doSomething

### Handlers

A handler is an entity that receives messages from the event bus. You register a handler at an address. Many different handlers from the same or different modules can be registered at the same address. A single handler can be registered at many different addresses at the same time.

### Messaging Schemes

The Event Bus supports the following modes of operation: \* *Publish / subscribe messaging*: Publishing means delivering the message to all handlers that are registered at that address. This is the familiar publish/subscribe messaging pattern. \* *Point to point and Request-Response messaging*: Messages are routed to just one of the handlers registered at an address. They can optionally be replied to. \* *Remote Procedure Call (RPC)*: This mode of operation is implemented on top of the Request-Response model, basically by enforcing certain conventions on requests and responses

This example shows the Event Bus (in version 3) can be instantiated, how a Handler can be defined and registered on the Event Bus and how the Event Bus can subsequently publish a message for the defined Handler:

final EventBus eb = vertx.eventBus();  
  
/\*Create a message consumer against the specified address.  
The returned consumer is not yet registered at the address, registration will be effective when MessageConsumer.handler(..) is called.  
\*/  
final MessageConsumer<JsonObject> msgConsumer = eb.consumer("test.address");  
  
//register handler for the MessageConsumer  
msgConsumer.handler(message -> {  
 System.out.println("I just recieved a message "+ message.body());  
});  
  
...  
//publishing a message. The message will be delivered to all handlers registered against the address  
eb.publish("test.address", "hello world");  
  
//point-2-point sending of message.   
//Only one handler registered at the address receiving the message.   
//The handler is chosen in a non strict round-robin fashion  
eb.send("test.address", "hello world");  
  
...  
//unregister handler for the MessageConsumer  
msgConsumer.unregister();

### Types of Messages

Messages that you send on the event bus can be as simple as a string, a number or a boolean. It is also possible to send Vert.x buffers or JSON messages. It's highly recommended to use JSON messages to communicate between verticles. JSON is easy to create and parse in all the languages that Vert.x supports. For RPC messages, JSON is enforced.

### Verticle

The unit of execution for Vert.x applications is called a Verticle. Verticles can be written in multiple languages (Java, JavaScript, Groovy and Ruby). Many verticles can be executed concurrently in the same Vert.x instance. An application might be composed of multiple verticles deployed on different nodes of the network communicating by exchanging messages over the Vert.x Event Bus. You can now (version 3) instantiate verticles and deploy verticle instances programmatically. The platform manager API has been removed, and methods for deploying verticles have moved to the Vertx interface. The API for deploying verticles is much simpler, so this should simplify things when embedding.

### Module

The Vert.x 3 module system has gone. It's advisable to use already available options like Maven or Gradle. Non Java verticles (e.g. JavaScript) can also be packaged in jars and pushed as Maven artifacts. It will also support resolving in other ways and from other places (e.g. at run-time and from npm modules) before 3.0.final.

### Event Loop

By default, all verticles run in an asynchronous event loop. When developing a verticle, it is essential not to block the event loop. Blocking here means either doing any kind of blocking I/O or even doing any kind of computational intensive work. Modules that do either of these should indicate that they are so called worker modules by setting "worker": true in their *mod.json* file. The advantage of an event loop is that it is enormously scalable. Instead of waiting for I/O operations to complete, the executing thread will rather do other stuff (e.g. servicing the next request) in the meantime. This is achieved by using a callback driven style of programming. Imagine the following scenario: *We want to read some data in an I/O intensive operation (function readData)* We want to do something with that data (function doSomething) *We want to do something completely different (function doSomethingUnrelated)* In the traditional blocking world we would do something like the following:

def doSomething(data):  
 # do something with data  
data = readData()  
doSomething(data)  
doSomethingUnrelated()

What happens here is the following:

After the data is read, the program waits for the operation (readData) to complete (which is consuming the event loop thread lifetime). As soon as readData returns, we have our data and can go on to do something with it (doSomething(data)). Finally, when that is done, we can go on and do other stuff (doSomethingUnrelated).

In the asynchronous world, we do something like this:   
def doSomething(data):  
 # do something with data  
readData(callback = doSomething)  
doSomethingUnrelated()

As can be seen, the result of readData is not received in the functions return value. Instead doSomething is passed in the handler method as a callback. The framework will take care that this handler is called asynchronously as soon as the data is available

### APIs

Vert.x provides the different APIs which are implemented in various languages. The two main modules are "Core API" and "Apex API":

**Core API** \* Deploy and undeploy verticles \* Logging \* TCP client/Server API \* HTTP client/Server API \* File System Access \* DNS client API \* Shared Data \* Event Bus API \* JSON API

**Apex API** \* Routing (based on method, path, etc) \* Event-bus bridge \* SockJS support \* Session support - both local (for sticky sessions) and clustered (for non sticky) \* Basic Authentication and Redirect based authentication \* User/role/permission authorisation

### Requirements Analysis

#### Messaging Node Requirements Analysis

##### [Autentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)

External Authentication and Authorisation are supported through Maven artifacts: vertx-apex and vertx-auth-service

The Authorisation module can be the front-end to interact with an external vertx service eg with restful APIs or could be attached to the vertx-io event bus.

**Authorisation to Send/publish a Message** \* SockJSHandler, where we need a bridge configuration \* Inbound and outbound options have specific bridge configuration classes. java final BridgeOptions options = new BridgeOptions(); options.addInboundPermitted(new PermittedOptions().setAddress("chat.to.server").setRequiredPermission("tim")); options.addOutboundPermitted(new PermittedOptions().setAddress("chat.to.client")); Inboundpermitted with "setRequiredPermission" or "setRequiredRole" will force an authenticated session to send into that address.

* Configure authentication handler and provider java final AuthProvider authProvider = //implement this interface for authentication and authorization control final AuthHandler basicAuthHandler = BasicAuthHandler.create(authProvider); AuthProvider is similar to a SPI (Service Provider Interface) with 3 basic methods: login(..), hasRole(..), hasPermission(..). It's available for custom implementations, so that it's possible to interop with other parts of the system (like a database). AuthHandler can also be rewritten, but in this case we use simple browser authentication.

**Receive a Message** \* SockJS handler is needed with bridge options.

java final SockJSHandler sockJSHandler = SockJSHandler.create(vertx); sockJSHandler.bridge(options); \* Configure Apex Router with SockJS handler for "/eventbus/\*" uri

```java //required Cookie and Session handlers for every address final Router router = Router.router(vertx); router.route().handler(CookieHandler.create()); router.route().handler(SessionHandler.create(LocalSessionStore.create(vertx)));

//apply AuthHandler handler to an address (order of this handler is important) router.route("/eventbus/\*").handler(basicAuthHandler);

//apply SockJS handler to an address router.route("/eventbus/*").handler(sockJSHandler); ```*  EventBus handler for "chat.to.server" address, every message sent to this address will be processed in this handler

java final EventBus eb = vertx.eventBus(); eb.consumer("chat.to.server").handler(message -> { //user code... });

**Example: simple communication send/receive in the same client connected via SockJS** \* Register handler to receive messages in "chat.to.client" address

javascript eb.registerHandler('chat.to.client', function(msg) { console.log('Message Received: ' + msg); });

* Send message to "chat.to.server" address javascript eb.send('chat.to.server', {name: 'tim', age: 35});

**Subscribe / register handlers to be notified about published messages**

EventBusBridgeHook is not yet available in version 3, however it's possible to override the SockJSHandlerImpl class and bypass this limitation.

ServerHook takes some keyword arguments for example: \* pre-register: Called before a client handler registration is processed.

java public boolean handlePreRegister(SockJSSocket sock, String address) { out.println("handlePreRegister, sock = " + sock + ", address = " + address); return true; } \* message-handler: it's possible in this version to discovery the user that has sent the message (available in apex Session)

java public boolean handleSendOrPub(SockJSSocket sock, boolean send, JsonObject msg, String address) { msg.put("principal", sock.apexSession().getPrincipal()); return true; }

In this way handlers registration can be controlled, and the user information can be sent to the EventBus.

##### [Unstable Connections](https://github.com/reTHINK-project/core-framework/issues/15)

Since vertx is based on http://hazelcast.org/ we can use it to cache some info including the sessionId

##### [Carrier grade deployment features (Resilience, DoS and DDoS protection, Service Assurance)](Messaging%20Node%20with%20carrier%20grade%20deployment%20features)

* Resilience: Vert.x provides resilience through the "automatic failover" and "HA group" options. When a module is run with HA, if the Vert.x instance where it is running fails, it will be re-started automatically on another node of the cluster. An HA group denotes a logical grouping of nodes in the cluster. Only nodes with the same HA group will failover onto one another.
* DoS and DDoS Protection: Vert.x 2.x. has no support for this, BUT Vert.x 3.0 provides built-in core functiionality for this core
* Service Assurance: Modules can be deployed in clusters, and Vert.x provides an internal Load Balancer for routing messages within the cluster. Also the above mentioned "auomatic failover" and "HA group" options contribute to enforce service assurance.

##### [Scalability] (https://github.com/reTHINK-project/core-framework/issues/16)

Verticle instances, except advanced multi-threaded worker verticles are almost always single threaded. what this implies is that, a single verticle instance can at most utilise one core of the server. In order to scale across cores, several verticles which are responsible for the same task can be instantiated and the runtime will distribute the workload among them (load balancing), this way taking full advantage of all SPU cores without much effort. Verticles can also be distributed between several machines. This will be transparent to the application code. The Verticles use the same mechanisms to communicate as if they would run on the same machine. This makes it very easy to scale applications.

##### [Messaging Transport Protocols] (https://github.com/reTHINK-project/core-framework/issues/20)

* Websockets - Yes supported
* SockJS - Yes supported
* HTTP Long-Polling - Yes
* HTTP Streaming - ? (Not sure what this means, clarification needed)

##### [Message delivery reliability] (https://github.com/reTHINK-project/core-framework/issues/17)

No. Vert.x uses the Event Bus to send messages through pub/sub mechanism or point-2-point mechanism. In both cases, there is no feedback to the sender if the message was recieved and processed or if it was not recieved at all. In the end reliability will boil down to the application logic service build on top of vert.x.

#### Runtime Requirements Analysis

From a runtime perspective the Vert.x is transparent to JVM8 (nashorn). Nashorn supports the full ECMAScript 5.1 specification. In that regard there are no browser API's such as: HTML5 canvas, HTML5 audio, WebWorkers, WebSockets, WebGL...

Initialization of the runtime engine:

final ClassLoader classLoader = TestClass.class.getClassLoader();  
final NashornScriptEngineFactory factory = new NashornScriptEngineFactory();

Nashorn is built on top of Java and takes advantage of standard Java security measures. Fine-grained security is enabled within applications. We can control the class load mechanism, effectively building a sandbox: ```java final ScriptEngine engine = factory.getScriptEngine(name -> { if(name.equals(TestClass.class.getName())) { return true; //OK, Java TestClass available from JavaScript }

return false; //everything else fails...

}); ```

Binding variables to the JavaScript scope is just one line of code. We can expose the Vert.x EventBus and use it like if we were in the JVM:

engine.getBindings(ScriptContext.ENGINE\_SCOPE).put("eb", vertx.eventBus());

Running a JavaScript file is also just a line of code:

engine.eval(new FileReader(classLoader.getResource("myjs.js").getFile()));

Although there are no WebSockets in the Nashorn runtime, it's possible to simulate a WS interface connecting directly through the EventBus, delegating the actual connection with the Vert.x.

Found some performance measures on: http://ariya.ofilabs.com/2014/03/nashorn-the-new-rhino-on-the-block.html

## Realtime backends

Realtime backends (aka noBackend or BackendAsAService(BaaS)) is a concept related to real time databases. It is a way to build web architectures without necessarily defining and standardizing data structures or interworking protocols if they are really needed. The backend and its remote framework is taking into account all low level mechanism of client-server dialogue, allowing developer to concentrate in service logic, in its local runtime [54]. The realtime backend concept would allow to define and manage interworking with other services, in a way that entirely belongs to each application. It can also be a solution to manage and maintain user preferences, endpoints registration status … and also to manage discovery.

For instance, in IoT domain, nobody is able now to identify every kind of objects that will be available in a close future, the way they will communicate, their need of security, level of authentication … It’s the reason why a solution that still allows in the future to define or modify data structures is the best way to have an evolving solution, well understood and adopted by a large number of developers.

[“Real-Time Web Technologies Guide”](http://www.leggetter.co.uk/real-time-web-technologies-guide/), from Phil Leggetter gives an overview of the different tools that are currently offered [55]. Among them we can site PubNub, Firebase, recently acquired by Google, and many others.

The main difference with other Messaging Nodes is the fact that all the synchronisation layer is managed by the service, so the developper only manages local objects that are immediatly syncrhonized by the library.

* Here is an example of code given by the firebase site:

// Use YOUR Firebase URL (not the one below)

var fb = new Firebase("https://.firebaseio.com");

/\* Remember to include firebase JS Library

\*/

* Save data:

fb.set({ name: "Alex Wolfe" });

* Update in real time

fb.on("value", function(data) {

var name = data.val() ? data.val().name : "";

alert("My name is " + name);

});

Ref: http://en.wikipedia.org/wiki/Real-time\_database

http://www.leggetter.co.uk/real-time-web-technologies-guide/

http://www.leggetter.co.uk/2013/12/09/choosing-realtime-web-app-tech-stack.html

### Requirements Analysis

#### [Messaging Node Requirements](https://github.com/reTHINK-project/core-framework/labels/Messaging%20Node%20Requirement)\*\*

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* As protocol on the fly is based on data representation, it is very convient as the realtime back ends are data oriented.
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* No, as the transport protocol is hidden from the remote machine.
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes - Several logging modules available...
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Depending of the product.
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Depending of the product.
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Yes - Depending of the product.
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes - Low level feature taken into account regarding the product
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes - Depending of the product, some are available in SaaS mode.
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes - Depending of the product
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* Yes.
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes

## Matrix.org

From [matrix.org spec](http://www.matrix.org/)[56]: > *The end goal of Matrix is to be a ubiquitous messaging layer for synchronising arbitrary data between sets of people, devices and services - be that for instant messages, VoIP call setups, or any other objects that need to be reliably and persistently pushed from A to B in an interoperable and federated manner.*

### Overview

In the scope of the reTHINK project, matrix.org is a candidate technology for the **Messaging Node**

### Architecture

Following picture shows the main data flow in a matrix architecture.

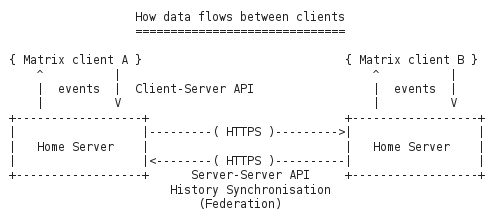


Figure 22: Main data flow in a matrix architecture

The core components are the Home Servers (HS) which can federate to sync and maintain the history of shared communication sessions among domains. Home Servers resolv each other via DNS.

Some general points: \* **Every** communication requires a room. Even for a simple chat message to a dedicated receiver a room MUST be created first and the receiver MUST be invited and join it \* Rooms are persistent. They can be re-entered after successive login sessions. \* A set of event/message types is defined in the API, own extensions can be made. \* Focus of matrix is on federation and consistency and history of session states.

### APIs

Matrix.org defines 3 types of APIs with different scope. All these APIs are REST APIs using JSON Objects as payload.

* [Client Server API v1](http://www.matrix.org/docs/spec/#client-server-api-v1)
* for implementation of application frontends
* functions for:
  + Registration and Login
  + sending and receiving of Events
  + management of communication rooms
* well documented, easy to implement
* version 2 is currently in development - will be backward compatible
* [Federation API (Server-Server API)](http://www.matrix.org/docs/spec/#id100)
* API for the inter-domain communication between Home Servers
* uses HTTPS GET and PUT requests
* transaction based
* requests are authenticated by PK signatures
* [Application Service API](http://www.matrix.org/docs/spec/#id79)
* API to provide custom server-side behaviour (e.g. gateways, filters, extensible hooks etc)
* so far only "Passive Application Services" are specified, i.e. services that can only monitor but not block or modify events

some words about Identifiers: \* Users are identified as: **(???)** (with an optional **:port** suffix) \* Rooms are identified as: **#roomalias:host.domain** (with an optional **:port** suffix) \* Home servers are identified and resolved by their FQDN like: **https://host.domain:port**

The Matrix concept includes the concept of an "Identity Server", which is intended to map 3rd party entities to matrix ids: \* no documentation \* testing didn't work in our lab environment

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Partially
* matrix is based on REST, example client uses long-polling to receive Events
* wrapping of Events/message in other transport protocols would require potential changes on HomeServer side
* (since "synapse", the ref-impl of Matrix HS, is implemented on top of Twisted, it should be possible to do that for e.g. Websockets)
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* this is kind of a core feature
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes
* could be done via an attached passive Application Service
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* transmission errors are returned to clients
* messages are re-delivered to clients from internal history of HS
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* No/(Perhaps via load balancers)
* from spec: "HS SHOULD implement rate limiting ..."
* experiments showed that already simple forwarding of every individual candidate of a WebRTC call can trigger this rate limitiation
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Yes
* since all communication is initiated by the client and HS addressing happens via DNS or static IP:port it is up to the client to re-connect after network interruptions
* HS always has the complete history of the session - nothing is lost
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* No
* HTTP(S) is not the most efficient protocol in terms of latency
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* rather Yes (not tested on embedded device yet)
* current ref-impl is based on Twisted (Python)
* overall installation size 94MB (including python and webclient)
* ==> environments supporting python should also support a Matrix HS
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* No
* matrix includes concept of external "Identity Server", but only for mapping of 3rd party IDs to internal IDs
* AA is performed internally
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes, via room-IDs

## RabbitMQ Evaluation

RabbitMQ [59] is defined as a robust and easy to use messaging platform that can work synchronously an asynchronously.

From rabbitmq.com: > \*RabbitMQ is a messaging broker - an intermediary for messaging. It gives your applications a common platform to send and receive messages, and your messages a safe place to live until received.

### Overview

In the scope of the reTHINK project, RabbitMQ is a candidate technology for the Messaging Node

##### Highlights:

* **Reliability** - RabbitMQ offers a variety of features to let you trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability.
* **Flexible Routing** - Messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic. For more complex routing you can bind exchanges together or even write your own exchange type as a plugin.
* **Clustering** - Several RabbitMQ servers on a local network can be clustered together, forming a single logical broker.
* **Federation** - For servers that need to be more loosely and unreliably connected than clustering allows, RabbitMQ offers a federation model.
* **Highly Available Queues** - Queues can be mirrored across several machines in a cluster, ensuring that even in the event of hardware failure your messages are safe.
* **The Polyglot Broker** - RabbitMQ supports messaging over a variety of messaging protocols including STOMP, MQTT, or AMQP.
* **Tracing** - If your messaging system is misbehaving, RabbitMQ offers tracing support to let you find out what's going on.

### Architecture

The core component is the **Broker** that routes the messages from the **producers** to the **consumers** as depicted in the following image.

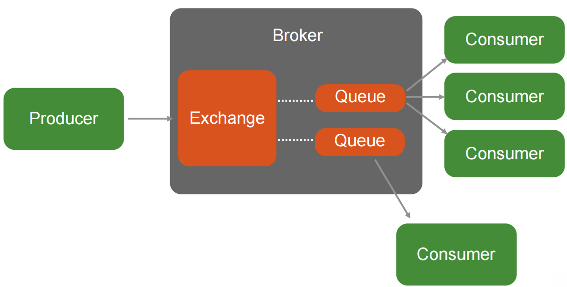


Figure 23: RabbitMQ Architecture2

The **Broker** has two main componentes: \* **Exchange**: accepts and routes messages from producer to clients based on the message information such as keys, bindings, filtering or broadcast.

* **Queue**: a FIFO queue of messages

Message routing can be based on topic / wild-card, as shown in the following image.

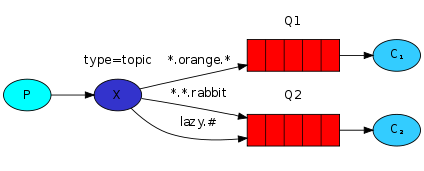


Figure 24: RabbitMQ Architecture

Image Source: [www.rabbitmq.com](https://www.rabbitmq.com/getstarted.html)

### APIs and Documentation

See [60] for the [AMQP protocol specification](http://www.amqp.org/).

RabbitMQ provides the following APIs: \* [Java and JVM](https://www.rabbitmq.com/devtools.html#java-dev) \* Java \* Spring Framework \* Scala \* Groovy \* Grails \* Clojure \* JRuby \* [Ruby](https://www.rabbitmq.com/devtools.html#ruby-dev) \* [Python](https://www.rabbitmq.com/devtools.html#python-dev) \* [.NET / C#](https://www.rabbitmq.com/devtools.html#dotnet-dev) \* [PHP](https://www.rabbitmq.com/devtools.html#php-dev) \* [PERL](https://www.rabbitmq.com/devtools.html#perl-dev) \* [C / C++](https://www.rabbitmq.com/devtools.html#c-dev) \* [Node.js](https://www.rabbitmq.com/devtools.html#node-dev) \* [Go](https://www.rabbitmq.com/devtools.html#go-dev) \* [Erlang](https://www.rabbitmq.com/devtools.html#erlang-dev) \* [Haskell](https://www.rabbitmq.com/devtools.html#haskell-dev) \* [**Web Messaging**](https://www.rabbitmq.com/devtools.html#web-messaging) \* [SockJS-erlang](https://github.com/sockjs/sockjs-erlang) SockJS-erlang is compatible with SockJS client \* [rabbitmq-chat](https://github.com/videlalvaro/rabbitmq-chat) A Web chat implemented with RabbitMQ and Websockets \* [rabbithub](https://github.com/tonyg/rabbithub) RabbitHub provides an HTTP-based interface to RabbitMQ. \* [VorpalBunny](https://github.com/myYearbook/VorpalBunny) PHP for talking to RabbitMQ's JSON-RPC-Channel Plugin

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Partially
* Has support for :
  + AMQP 0.9.1, 0.9, 0.8
  + STOMP via plugin
  + MQTT via plugin
  + AMQP 1.0 via plugin
  + Web-STOMP using WebSockets (SockJS) via plugin
  + Non-Reliable HTTP API to send and receive messages via the management plugin
  + JSON-RPC via plugin (synchronous)
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* Core feature
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes
* RabbitMQ has a built-in tracer feature that is able to see every message that is published, and every message that is delivered on a per-node.
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* For AMQP:
  + Acknowledgements can be used in both directions - to allow a consumer to indicate to the server that it has received/processed a message and to allow the server to indicate the same thing to the producer.
  + Use of acknowledgements guarantees at-least-once delivery. Without acknowledgements, message loss is possible during publish and consume operations and only at-most-once delivery is guaranteed.
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Yes
* In a RabbitMQ cluster all data/state required for the operation of a broker is replicated across all nodes, for reliability and scaling, with full ACID properties. Queues may be located on a single node, or mirrored across multiple nodes. A client connecting to any node in a cluster can see all queues in the cluster, even if they are not located on that node.
* Brokers tolerate the failure of individual nodes
* 1 million messages/second was benchmarked using a cluster of 30 Nodes [[Link1]](http://blog.pivotal.io/pivotal/products/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine)[[Link2]](http://googlecloudplatform.blogspot.pt/2014/06/rabbitmq-on-google-compute-engine.html)
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* No
* The client should use heartbeats for detecting dead TCP connections that have a timeout interval.
* Business logic must be implemented on the application layer in order to deal with this case.
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* Using the HTTP management API
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* Has low latency derived from its Erlang message oriented implementation. [[Link1]](http://www.rabbitmq.com/blog/2012/04/17/rabbitmq-performance-measurements-part-1/)[[Link2]](http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/)
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Depends on the protocol used:
  + AMQP,STOMP, HTTP - depends (some computation is needed) (>256K RAM)
  + MQTT - yes (Designed especially for IoT)
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* Yes
* Using SSL client certificates via plugin
* Using LDAP and Federation via pugins
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes
* Core functionality, including topic/wildcard based filtering and broadcasting.

## ZeroMQ Evaluation

### Overview

In the scope of the reTHINK project ZeroMQ [65] is a candidate technology for the Messaging Node.

It is a high-performance, low level, asynchronous messaging library originally written in C++, that now has multiple native Implementations. It is used as a thin layer between the application and transport layers.

##### Highlights:

* Connect your code in any language, on any platform. 40+ Language Bindings
* Carries messages across inproc, IPC, TCP, TIPC, multicast.
* Smart patterns like pub-sub, push-pull, and router-dealer can be combined together to form powerful architectures.
* High-speed asynchronous I/O engines, in a tiny library ((20k lines of C++)).
* Backed by a large and active open source community.
* Supports every modern language and platform.
* Any architecture: centralized, distributed, small, or large.
* Multicore Optimized
* Automatic TCP (re)connect
* Fast: 8M msg/sec, usec latency
* Faster than TCP, for clustered products and supercomputing
* Fast for development thanks to many useful abstraction layers, native languages, bindings and huge open source community.

### Architecture

The following figure represents the six basic types of communication patterns that ZeroMQ supports.

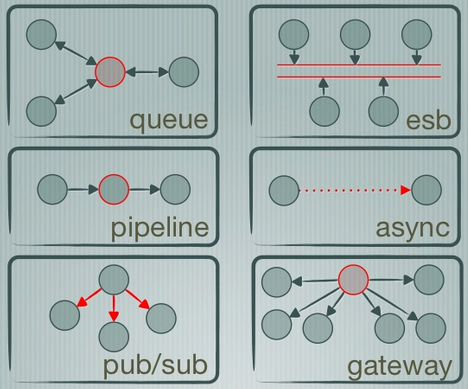


Figure 25: ZeroMQ types of communication patterns

Image Source: [http://www.slideshare.net](http://www.slideshare.net/IanBarber/zeromq-is-the-answer)

#### Built-in core ZeroMQ patterns

* Request-reply: connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
* Pub-sub: connects a set of publishers to a set of subscribers. This is a data distribution pattern.
* Pipeline: connects nodes in a fan-out/fan-in pattern that can have multiple steps and loops. This is a parallel task distribution and collection pattern.
* Exclusive pair: connects two sockets exclusively. This is a pattern for connecting two threads in a process, not to be confused with "normal" pairs of sockets.

On top of the built-in core ZeroMQ patterns high-level messaging patterns are defined [here](http://zguide.zeromq.org/page:all). They are not part of the core library, do not come with the ZeroMQ package, and exist in their own space as part of the ZeroMQ community. For example the Majordomo pattern ([Reliable Request-Reply Patterns](http://zguide.zeromq.org/page:all#reliable-request-reply)), sits in the GitHub Majordomo project in the ZeroMQ organization. One of the things that ZeroMQ also aims to provide is a set of high-level patterns, both small (how to handle messages sanely) and large (how to make a reliable pub-sub architecture).

These can be used as "fabric" to make very powerful architectures. The next image shows an example of this modularity. [A pub/sub multi-cluster Architecture](http://zguide.zeromq.org/page:all#Scaling-to-Multiple-Clusters)

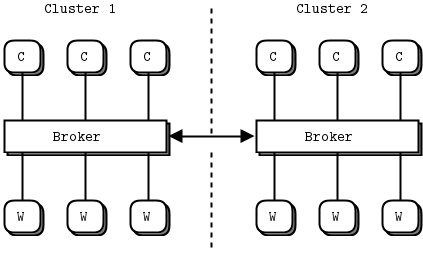


Figure 26: ZeroMQ Clusters

Image Source: [http://zguide.zeromq.org](http://zguide.zeromq.org/page:all#Scaling-to-Multiple-Clusters)

The internal Architecture in more detail can be found [here.](http://zeromq.org/whitepapers:architecture)

### APIs and Bindings

#### Native Implementations of the library

* [C/C++](https://github.com/zeromq/libzmq) represents the State of the Art.
* [Java](https://github.com/zeromq/jeromq) JeroMQ. Fully compatible at both API and protocol level but sans encryption or PGM. *what is PGM?*
* [.NET](https://github.com/zeromq/netmq) same constraints as JeroMQ
* [Erlang](https://github.com/zeromq/ezmq)
* [Python](https://github.com/caedesvvv/zmqproto)
* [C](https://github.com/zeromq/libzmtp) designed for small devices you would not normally expect messaging technology to fit
* [Netty](https://github.com/spotify/netty-zmtp)

#### Language Bindings

* [Ada, C, Chicken Scheme, Common Lisp, C#(.NET & Mono), C++, D, delphi binding, Eiffel, Erlang, F#, Felix, Flex (ActionScript), Fortran77, Go, Guile, Haskell, Haxe, Java binding, JavaScript (Flash), Julia, LabVIEW, Lua bindings, Nimrod, Node.js, Objective-C, Objective Caml binding, ooc, Perl s, PHP binding, Python binding, Q, Racket, R, RE, RE, Red, Ruby, Ruby(FFI), Scala, Smalltalk, Tcl, Twisted (Python), XPCOM](http://zeromq.org/bindings:_start), and more [on github.com](https://github.com/search?utf8=%E2%9C%93&q=zmq&type=Repositories&ref=searchresults)

#### Web Clients/Servers

* [JSMQ](https://github.com/zeromq/JSMQ) Javascript client for ZeroMQ/NetMQ over WebSockets
* [NullMQ](https://github.com/progrium/nullmq) ZeroMQ semantics in the browser [Link1](http://www.slideshare.net/progrium/nullmq-pdx) [Link2](http://avalanche123.com/blog/2012/02/25/interacting-with-zeromq-from-the-browser/)
* [ZmqSocket.js](http://zeromq.org/bindings%3ajavascript) talk to zmq sockets from your JavaScript code.
* [SockJSProxy](https://bitbucket.org/vladev/sockjsproxy/) a simple proxy server that proxies message from SockJS to a ZeroMQ.
* [Zerogw](https://github.com/tailhook/zerogw) HTTP to zeromq gateway
* [XARP](http://rfc.zeromq.org/spec:40) (Draft) Extensible Resource Access Protocol (XRAP), a RESTful protocol built over ZeroMQ
* [Zato](https://zato.io/docs/index.html) ESB, SOA, REST, APIs and cloud integrations
* [Malamute](https://github.com/miska/malamute) All the enterprise messaging patterns in one box.

[ZeroMQ API Reference](http://api.zeromq.org/)

[Other Projects](http://zeromq.org/docs:labs)

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Yes
* Has support for Javascript and WebSockets using:
  + [JSMQ](https://github.com/zeromq/JSMQ)
  + [NullMQ](https://github.com/progrium/nullmq)
  + [ZmqSocket.js](http://zeromq.org/bindings%3ajavascript)
  + [SockJSProxy](https://bitbucket.org/vladev/sockjsproxy/)
  + [Zerogw](https://github.com/tailhook/zerogw)
  + [XRAP](http://rfc.zeromq.org/spec:40)
  + [Zato](https://zato.io/docs/index.html)
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* Using the [Titanic Service Protocol](http://rfc.zeromq.org/spec:9)
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes
* Using the [Titanic Service Protocol](http://rfc.zeromq.org/spec:9)
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* Using [Reliable Patterns](http://zguide.zeromq.org/page:all#Chapter-Reliable-Request-Reply-Patterns)
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Yes
* Using scalable patterns such as a [brokerless design](http://zeromq.org/whitepapers:brokerless)
* 9,5 Million Messages / second were benchmarked on a 16 core machine[Link1](http://zeromq.org/results:0mq-tests-v03) [Other Tests](http://zeromq.org/results:_start)
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Partial
* Business logic can be developed to deal with this issue
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* Very low usec latency
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes
* ZeroMQ runs on everything of interest, from 32KB embedded chips to z/OS mainframes running IBM dialects of Unix. [Link](http://zeromq.org/docs:features)
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* Yes
* Implemented:
  + [ZAP - ZeroMQ Authentication Protocol (PAM, LDAP, Kerberos, passwd, etc)](http://rfc.zeromq.org/spec:27)
  + CurveZMQ Authentication and Encryption Protocol [Link1](http://curvezmq.org/) [Link2](http://rfc.zeromq.org/spec:26)
  + [GSSAPI (Kerberos)](http://rfc.zeromq.org/spec:38) or [CURVE](http://curvezmq.org/)
  + [SRP](http://rfc.zeromq.org/spec:34)
* Supports the ability to be extended
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes
* Using the [ZeroMQ Publish-Subscribe Pattern](http://rfc.zeromq.org/spec:29)

## Redis Evaluation

### Overview

In the scope of the reTHINK project Redis [63] is a candidate technology for the Messaging Node.

From http://redis.io/: > Redis is an open source, BSD licensed, advanced key-value cache and store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs.

##### Highlights:

* Very Fast in memory data-store
* Usually used as Cache or MQ
* Scalable
* Supports strings, hashes, lists, sets, sorted sets, bitmaps and hyperloglogs as data structures.

### Architecture

* The main Redis architecture can be resumed as a in memory key-value data-store for service decoupling.

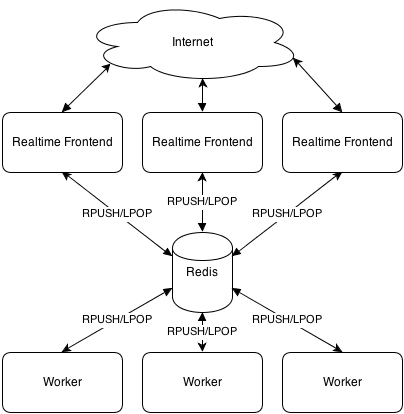


Figure 27: Redis architecture

Image Source: [http://mrjoes.github.io](http://mrjoes.github.io/2013/06/21/python-realtime.html)

* Another possibility is a PubSub model with message filtering based on prefix matching.

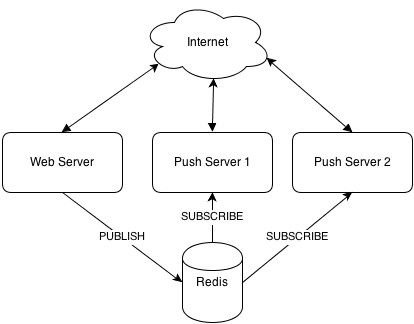


Figure 28: PubSub model

Image Source: [http://mrjoes.github.io](http://mrjoes.github.io/2013/06/21/python-realtime.html)

### APIs and Bindings

#### Language Bindings

* [C, C#, C++, Clojure, Common Lisp, D Dart, Elixir, emacs, lisp, Erlang, Fancy, GNU Prolog, Go Haskell, haXe, Io, Java, Lua, Matlab, Nimrod, Node.js, Objective-C, OCaml, Perl, PHP, Pure Data Python, Rebol, Ruby, Rust, Scala, Scheme, Smalltalk, Tcl, VCL](http://redis.io/clients)

#### Other

* [webdis](https://github.com/nicolasff/webdis) A Redis HTTP interface with JSON output
* [BankersBox](https://github.com/twilio/BankersBox) A redis-like wrapper for javascript data storage
* [Bone](https://github.com/solutious/bone) Rudimentary Redis over HTTP(S)

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* No
* An external wrapper should be used
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* Core feature
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* No
* Should be implemented externally
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* Atomic message processing
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Yes
* Using:
  + Clustering [Link1](http://redis.io/topics/cluster-tutorial) [Link2](http://redis.io/topics/cluster-spec)
  + Partitioning [Link1](http://redis.io/topics/partitioning)
  + Sentinel provides high availability [Link1](http://redis.io/topics/sentinel)
  + Replication [Link1](http://redis.io/topics/replication)
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Partial
* Business logic can be developed to deal with this issue
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* Built-in
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* Very low usec latency [Link1](http://redis.io/topics/benchmarks)
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* No
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes
* Core functionality, including prefix filtering

## XMPP

From http://xmpp.org/ [64]:

The Extensible Messaging and Presence Protocol (XMPP) is an open technology for real-time communication, which powers a wide range of applications including instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.

### Overview

In the scope of the reTHINK project XMPP is a candidate technology for the Messaging Node.

##### Highlights:

* **Open** - the XMPP protocols are free, open, public, and easily understandable; in addition, multiple implementations exist in the form clients, servers, server components, and code libraries.
* **Standard** - the Internet Engineering Task Force (IETF) has formalized the core XML streaming protocols as an approved instant messaging and presence technology. The XMPP specifications were published as RFC 3920 and RFC 3921 in 2004, and the XMPP Standards Foundation continues to publish many XMPP Extension Protocols. In 2011 the core RFCs were revised, resulting in the most up-to-date specifications (RFC 6120, RFC 6121, and RFC 6122).
* **Proven** - the first Jabber/XMPP technologies were developed by Jeremie Miller in 1998 and are now quite stable; hundreds of developers are working on these technologies, there are tens of thousands of XMPP servers running on the Internet today, and millions of people use XMPP for instant messaging through public services such as Google Talk and XMPP deployments at organizations worldwide.
* **Decentralized** - the architecture of the XMPP network is similar to email; as a result, anyone can run their own XMPP server, enabling individuals and organizations to take control of their communications experience.
* **Secure** - any XMPP server may be isolated from the public network (e.g., on a company intranet) and robust security using SASL and TLS has been built into the core XMPP specifications. In addition, the XMPP developer community is actively working on end-to-end encryption to raise the security bar even further.
* **Extensible** - using the power of XML, anyone can build custom functionality on top of the core protocols; to maintain interoperability, common extensions are published in the XEP series, but such publication is not required and organizations can maintain their own private extensions if so desired.
* **Flexible** - XMPP applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, remote systems monitoring, web services, lightweight middleware, cloud computing, and much more.
* **Diverse** - a wide range of companies and open-source projects use XMPP to build and deploy real-time applications and services; you will never get “locked in” when you use XMPP technologies.

### Architecture

A typical XMPP network consists of several Servers/Domains connected together. The mechanism is similar to email where the servers are used as relays for the messages. Every entity on the XMPP network is addressed using a JabberID (JID). It has the form : username@domain/resource where domain is the domain name of the XMPP server, and username identifies an account on that server.

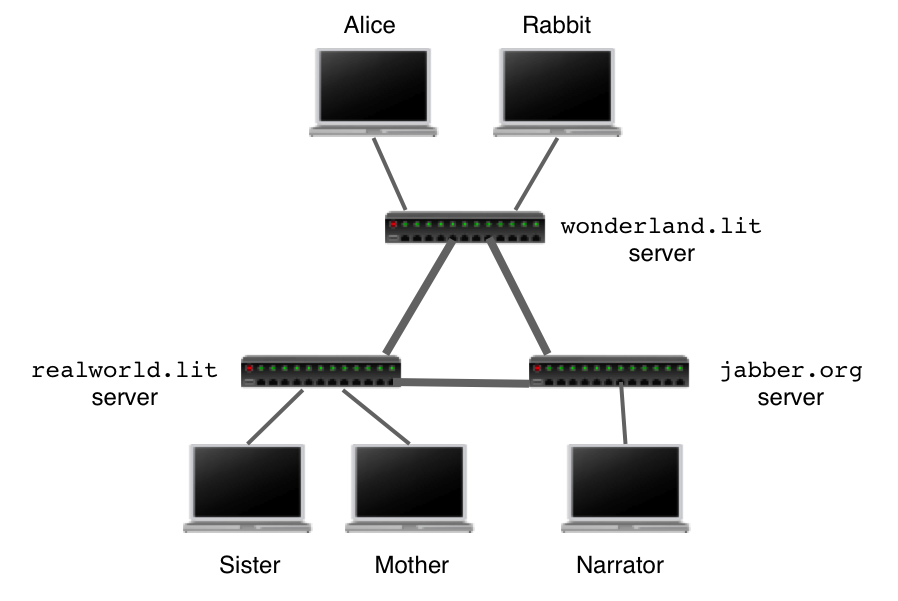


Figure 29: XMPP Architecture

Image Source: [https://el-tramo.be](https://el-tramo.be/documents/beautiful-xmpp-testing/)

Server can have adapters/gateways to be able to talk to other protocols such as those used by ICQ, AIM, Yahoo and others.

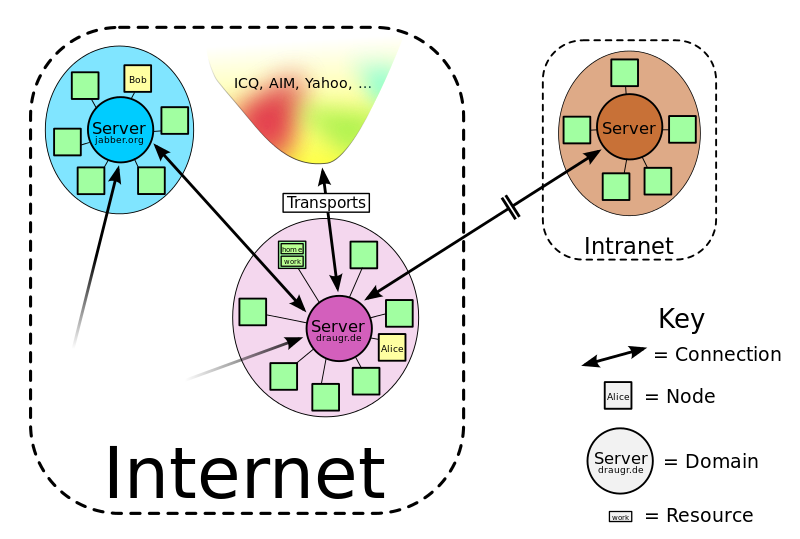


Figure 30: XMPP Protocols

Image Source: [https://en.wikipedia.org](https://en.wikipedia.org/wiki/XMPP)

Sometimes the servers are used only for signaling, in order to establish a media connection. [XEP-0166: Jingle](http://xmpp.org/extensions/xep-0166.html) is a protocol extension for initiating and managing peer-to-peer media sessions between two XMPP entities.

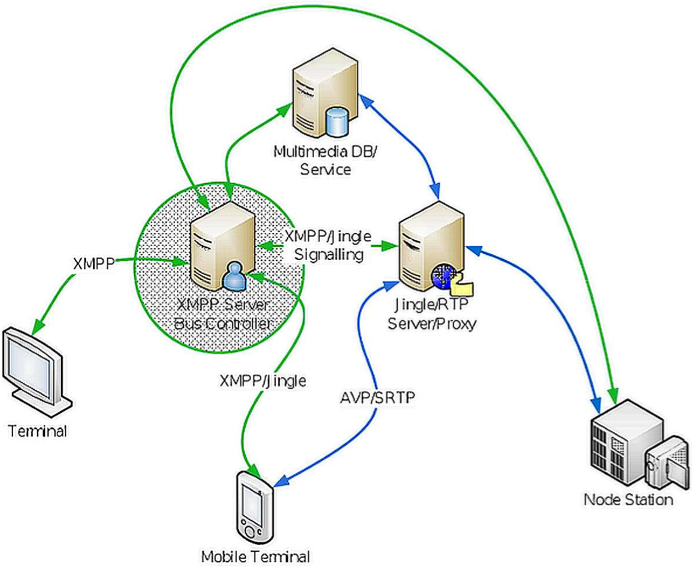


Figure 31: XMPP Jingle

Image Source: [http://xmppjingle.blogspot.pt](http://xmppjingle.blogspot.pt/2010/09/jingle-nodes-proxy-and-eu-surveillance.html)

### APIs, Bindings and Extensions

#### APIs

|  |  |  |
| --- | --- | --- |
| Name | Language(s) | Details |
| agsXMPPSDK | C#/.NET/Mono | [Link](http://www.ag-software.net/agsxmpp-sdk/) |
| AnyEvent::XMPP | Perl | [Link](http://www.ta-sa.org/projects/net_xmpp2.html) |
| as3xmpp | Flash/ActionScript | [Link](http://code.google.com/p/seesmic-as3-xmpp/) |
| asmack | Java(Android) | [Link](https://github.com/flowdalic/asmack) |
| AXMPP | Ada | [Link](http://forge.ada-ru.org/axmpp) |
| Babbler | Java | [Link](http://babbler-xmpp.blogspot.de/) |
| Babylon | Ruby | [Link](https://github.com/bryanwoods/babylon) |
| Blather | Ruby | [Link](http://adhearsion.com/blather) |
| cl-xmpp | Lisp | [Link](http://common-lisp.net/project/cl-xmpp/) |
| CoversantSoapBoxSDKStudio | C#/.NET/Mono/C | [Link](http://www.coversant.com/products/soapbox-sdk) |
| dojox.xmpp | JavaScript | [Link](http://api.dojotoolkit.org/jsdoc/1.3/dojox.xmpp) |
| dxmpp | C | [Link](http://deusexmachinae.se/dxmpp) |
| EchomineFeridian | Java | [Link](http://freecode.com/projects/feridian) |
| Eiffel | PHP | [Link](https://github.com/jocelyn/bricabrac/wiki/Eiffel-XMPP) |
| emite | Java | [Link](https://github.com/EmiteGWT) |
| exmpp | Erlang | [Link](http://exmpp.org/) |
| frabjous | JavaScript | [Link](https://github.com/theozaurus/frabjous) |
| gloox | C | [Link](http://camaya.net/gloox) |
| goexmpp | Go | [Link](http://code.google.com/p/goexmpp/) |
| headstock | Python | [Link](https://github.com/Lawouach/headstock) |
| hsxmpp | Haskell | [Link](http://חנוך.se/hsxmpp/) |
| hxmpp | haXe | [Link](http://hxmpp.disktree.net/) |
| iksemel | C | [Link](http://code.google.com/p/iksemel/) |
| IP\*WorksInternetToolkit | ActiveX C C# .NET Mono Delphi Java | [Link](http://www.nsoftware.com/ipworks/) | | Iris | C | [Link](http://delta.affinix.com/iris/) | | jabber-net | C#/.NET/Mono | [Link](http://code.google.com/p/jabber-net/) | | jabber.py | Python | [Link](http://jabberpy.sourceforge.net/) | | JabberLib | Tcl | [Link](http://coccinella.im/jabberlib) | | JabberStreamObjects(JSO) | Java | [Link](http://java.net/projects/jso/) | | JAXL | PHP | [Link](http://code.google.com/p/jaxl/) | | jQuery-XMPP-plugin | JavaScript | [Link](http://github.com/maxpowel/jQuery-XMPP-plugin) | | Jreen | C/Qt | [Link](http://qutim.org/jreen) | | JSJaC | JavaScript | [Link](http://blog.jwchat.org/jsjac/) | | libstrophe | C | [Link](http://strophe.im/) | | Lightr | PHP | [Link](http://code.google.com/p/lightr/) | | Loudmouth | C | [Link](http://groups.google.com/group/loudmouth-dev) | | Loudmouth | Ruby | [Link](http://groups.google.com/group/loudmouth-dev/web/loudmouth-ruby) | | MatriX | C#/.NET/Mono | [Link](http://www.ag-software.net/matrix-xmpp-sdk/) | | Net::XMPP | Perl | [Link](http://search.cpan.org/dist/Net-XMPP/) | | node-xmpp | JavaScript | [Link](http://github.com/astro/node-xmpp) | | oajabber | C | [Link](http://www.openaether.org/oajabber.html) | | PontariusXMPP | Haskell | [Link](https://github.com/pontarius/pontarius-xmpp/) | | pyxmpp | Python | [Link](http://pyxmpp.jajcus.net/) | | QXmpp | C | [Link](http://code.google.com/p/qxmpp/) | | seesmic-as3-xmpp | Flash/ActionScript | [Link](http://code.google.com/p/seesmic-as3-xmpp/) | | SleekXMPP | Python | [Link](http://github.com/fritzy/SleekXMPP/wiki) | | Smack | Java | [Link](http://igniterealtime.org/projects/smack/) | | stanza.io | JavaScript | [Link](https://github.com/legastero/stanza.io) | | strophe.js | JavaScript | [Link](http://strophe.im/) | | StropheCappuccino | Objective-J | [Link](http://github.com/primalmotion/strophecappuccino) | | Swiften | C | [Link](http://swift.im/swiften/) | | Tinder | Java | [Link](http://igniterealtime.org/projects/tinder/) | | txmpp | C | [Link](http://github.com/silas/txmpp) | | TwistedWords | Python | [Link](http://twistedmatrix.com/trac/) | | Ubeity | C# | [Link](https://github.com/ubiety/xmpp) | | Verse | Lua | [Link](http://matthewwild.co.uk/projects/verse/home) | | XIFF | Flash/ActionScript | [Link](http://igniterealtime.org/projects/xiff/) | | xmpp-psn | Python | [Link](http://code.google.com/p/xmpp-psn/) | | jaxmpp2 | Java/Android/GoogleWebToolkit | [Link](https://projects.tigase.org/projects/jaxmpp2) | | xmpp4js | JavaScript | [Link](http://xmpp4js.sourceforge.net/) | | XMPP4R | Ruby | [Link](http://home.gna.org/xmpp4r/) | | xmpp4r-simple | Ruby | [Link](http://code.google.com/p/xmpp4r-simple/) | | xmppframework | ObjectiveC | [Link](https://github.com/robbiehanson/XMPPFramework) | | xmpphp | PHP | [Link](http://code.google.com/p/xmpphp/) | | xmppy | Python | [Link](http://xmpppy.sourceforge.net/) | | XMPP-FTW | JavaScript | [Link](https://github.com/lloydwatkin/xmpp-ftw) | | Z-XMPP | JavaScript | [Link](http://ivan.vucica.net/zxmpp/) | |  |  |

#### Relevant Extensions

* [XEP-0166: Jingle](http://xmpp.org/extensions/xep-0166.html) Protocol extension for initiating and managing peer-to-peer media sessions (e.g., voice chat, video chat, file transfer) with a wide variety of transport methods (e.g., TCP, UDP, ICE, application-specific transports)
* [XEP-0343: Signaling WebRTC datachannels in Jingle](http://xmpp.org/extensions/xep-0343.html) Defines how to use the ICE-UDP Jingle transport method to send media data using WebRTC DataChannels.
* [XEP-0060: Publish-Subscribe](http://www.xmpp.org/extensions/xep-0060.html) Protocol extension for generic publish-subscribe functionality
* [XEP-0045: Multi-User Chat](http://xmpp.org/extensions/xep-0045.html) Protocol extension for multi-user text chat.
* [XEP-0072: SOAP Over XMPP](http://www.xmpp.org/extensions/xep-0072.html) Defines methods for transporting SOAP messages over XMPP
* [XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)](http://xmpp.org/extensions/xep-0124.html) bidirectional TCP connection between two entities (such as a client and a server) by efficiently using multiple synchronous HTTP request/response pairs.
* [XMPP over WebSocket](http://tools.ietf.org/html/rfc7395) Binding for the XMPP protocol over a WebSocket transport layer. It provides higher performance than the current HTTP binding for XMPP.
* [XEP-0030: Service Discovery](http://xmpp.org/extensions/xep-0030.html) Protocol extension for discovering information about other XMPP entities.

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Yes
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* Using [XEP-0203: Delayed Delivery](http://xmpp.org/extensions/xep-0203.html)
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes
* Using [XEP-0313: Message Archive Management](http://xmpp.org/extensions/xep-0313.html) and [XEP-0136: Message Archiving](http://xmpp.org/extensions/xep-0136.html)
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* Using [XEP-0184: Message Delivery Receipts](http://xmpp.org/extensions/xep-0184.html) and/or [XEP-0079: Advanced Message Processing](http://xmpp.org/extensions/xep-0079.html)
* [More info](http://www.isode.com/whitepapers/reliable-xmpp.html)
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Yes
* Using scalable Erlang-based servers [mongooseim](https://www.erlang-solutions.com/products/mongooseim-massively-scalable-ejabberd-platform) or [ejabberd](http://docs.ejabberd.im/architect/) clusters can handle tens of millions of users.
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Yes
* If Using [XEP-0198: Stream Management](http://xmpp.org/extensions/xep-0198.html)
* [More info](http://www.isode.com/whitepapers/reliable-xmpp.html)
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* [ms latency](https://www.ejabberd.im/benchmark)
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes
* Clients based on C libraries can run on embedded systems
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* Yes
* Using [XEP-0178: Best Practices for Use of SASL EXTERNAL](http://xmpp.org/extensions/xep-0178.html)
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes
* Using [XEP-0060: Publish-Subscribe](http://www.xmpp.org/extensions/xep-0060.html)

## MQTT Evaluation

MQ Telemetry Transport (MQTT) [61] is a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. These characteristics make it ideal for use in constrained environments such as machine-to-machine (M2M)/"Internet of Things" connectivity scenarios.

MQTT-SN (MQTT for Sensor Networks) is a variation of the main protocol aimed at embedded devices on non-TCP/IP networks, such as ZigBee.

### Overview

In the scope of the reTHINK project, MQTT is a candidate technology for the Messaging Node.

#### Highlights:

MQTT was designed for low-bandwidth, high latency networks in the late 1990s/early 2000s. As a result, the designers made a number of key choices which influenced the way it "looks and feels".

* Simplicity, simplicity, simplicity! Don't add too many "bells and whistles" but provide a solid building block which can easily be integrated into other solutions. Be simple to implement.
* Publish/subscribe messaging. Useful for most sensor applications, and enables devices to come online and publish "stuff" that hasn't been previously known about or predefined.
* Zero administration (or as close as possible). Behave sensibly in response to unexpected actions and enable applications to "just work", e.g. by dynamically creating topics when needed.
* Minimise the on-the-wire footprint. Add an absolute minimum of data overhead to any message. Be lightweight and bandwidth efficient.
* Expect and cater for frequent network disruption (for low bandwidth, high latency, unreliable, high cost-to-run networks)... -> [Last Will and Testament](http://www.hivemq.com/mqtt-essentials-part-9-last-will-and-testament/)
* Continuous session awareness -> [Last Will and Testament](http://www.hivemq.com/mqtt-essentials-part-9-last-will-and-testament/)
* Expect that client applications may have very limited processing resources available.
* Provide traditional messaging qualities of service where the environment allows. Provide "quality of service"
* Data agnostic. Don't mandate content formats, remain flexible.

## Architecture

MQTT has a client/server model, where every node is a client that connects to a server, know as a broker, over TCP. MQTT is message oriented. Messages are published to a topic. Clients may subscribe to multiple topics or use a wildcard matching based subscription.

MQTT supports three quality of service levels, "Fire and forget", "delivered at least once" (QoS1) and "delivered exactly once" (QoS2).

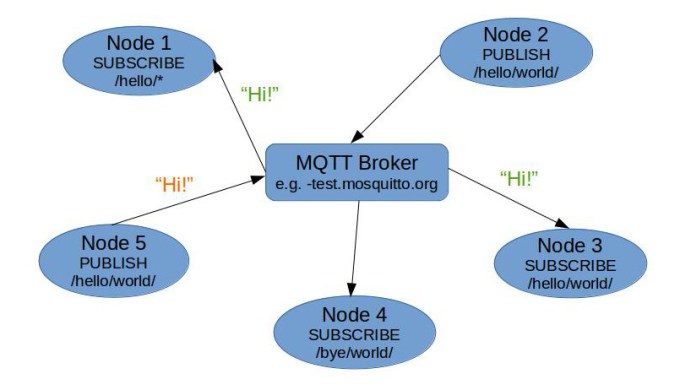


Figure 32: MQTT Architecture

Image Source: [https://sakshambhatla.wordpress.com](https://sakshambhatla.wordpress.com/2014/08/11/simple-mqtt-broker-and-client-in-python/)

## APIs and Bindings

#### Specification

The current formal protocol specification can be found at:

[MQTT v3.1.1 specification](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html)

#### Device-Specific

* [Arduino](https://github.com/knolleary/pubsubclient) ([more information](http://knolleary.net/arduino-client-for-mqtt/))
* [mbed](https://github.com/yilun/MQTT-client-on-mbed) ([more information](http://ceit.uq.edu.au/content/mqttclient-mbed-version-20))
* [mbed (simple port of the Arduino pubsubclient)](http://mbed.org/users/jwende/code/MQTT/)
* [mbed (native implementation)](http://mbed.org/users/Nim65s/code/niMQTT/)
* [mbed (Paho Embedded C++ port)](http://developer.mbed.org/teams/mqtt/code/MQTT/) ([more information] (https://www.eclipse.org/paho/clients/c/embedded/))
* [mbed (Paho Embedded C port)](http://developer.mbed.org/teams/mqtt/code/MQTTPacket/) ([more information](https://www.eclipse.org/paho/clients/c/embedded/))
* [Nanode](http://github.com/njh/NanodeMQTT/)
* [Netduino](https://github.com/danielan/NetduinoMQTT)
* [M2MQTT (works with .Net Micro Framework)](https://m2mqtt.codeplex.com/)

(see also [devices](things) page for more on hardware with built-in support)

#### Actionscript

* [as3MQTT](https://github.com/yangboz/as3MQTT)

#### Bash

* see [Shell Script](#shell-script), below

#### C

* [Eclipse Paho C] (https://www.eclipse.org/paho/clients/c/)
* [Eclipse Paho Embedded C] (https://www.eclipse.org/paho/clients/c/embedded/)
* [libmosquitto](http://mosquitto.org)
* [libemqtt](https://github.com/menudoproblema/libemqtt) - an embedded C client

#### C++

* [Eclipse Paho C++] (https://www.eclipse.org/paho/clients/cpp/)
* [libmosquittopp](http://mosquitto.org)
* [Eclipse Paho Embedded C++] (https://www.eclipse.org/paho/clients/c/embedded/)

#### Clojure

* [Machine Head](http://clojuremqtt.info)
* [Clojure MQTT Codec for Netty](https://github.com/xively/clj-mqtt/)

#### Dart

* [mqtt.dart](http://pub.dartlang.org/packages/mqtt)

#### Delphi

* [TMQTTClient](http://jamiei.com/code/TMQTTClient.zip) ([more information](http://jamiei.com/blog/code/mqtt-client-library-for-delphi/))

#### Erlang

* [erlmqtt](https://github.com/squaremo/erlmqtt)
* [emqttc](https://github.com/emqtt/emqttc) - Erlang MQTT Client
* [mqtt4erl](http://code.google.com/p/mqtt4erl/)
* [my-mqtt4erl](http://code.google.com/p/my-mqtt4erl/) - updated fork of mqtt4erl

#### Elixir

* [hulaaki](https://github.com/suvash/hulaaki)

#### Go

* [Eclipse Paho Go](http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.golang.git/)

#### Haskell

* [mqtt-hs](http://hackage.haskell.org/package/mqtt-hs)

#### Java

* [Eclipse Paho Java](http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.java.git/)
* [Xenqtt](http://xenqtt.sf.net) Includes a client library, mock broker for unit/integration testing, and applications to support enterprise needs like using a cluster of servers as a single client, an HTTP gateway, etc.
* [MeQanTT](https://github.com/AlbinTheander/MeQanTT)
* [Fusesource mqtt-client](https://github.com/fusesource/mqtt-client)
* [moquette](http://code.google.com/p/moquette-mqtt/)
* ["MA9B" zip of 1/2 dozen mobile clients source code. Includes Android-optimized Java source that works with Android notifications, based on Paho](http://www-933.ibm.com/support/fixcentral/swg/selectFix?product=ibm%2FWebSphere%2FWebSphere+MQ&fixids=1.0.0.1-WS-MQCP-MA9B&source=dbluesearch&function=fixId&parent=ibm/WebSphere)
* [IA92](http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24006006&loc=en_US&cs=utf-8&lang=en) - *deprecated* IBM IA92 support pack, use Eclipse Paho GUI client instead. A useful MQTT Java swing GUI for publishing & subscribing. The Eclipse Paho GUI is identical but uses newer client code

#### Javscript / Node.js

* [Eclipse Paho HTML5 JavaScript over WebSocket.](http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.javascript.git/)
* [mqtt.js](https://github.com/adamvr/MQTT.js)
* [node\_mqtt\_client](https://github.com/yilun/node_mqtt_client) ([more information](http://ceit.uq.edu.au/content/simple-mqtt-cient-nodejs))
* [IBM-provided PhoneGap / Apache Cordova MQTT plug-in for Android](http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24033580&loc=en_US&cs=utf-8&lang=en) - JavaScript API is identical to Eclipse Paho HTML5 JavaScript
* [Ascoltatori](https://github.com/mcollina/ascoltatori) - a node.js pub/sub library that allows access to Redis, AMQP, MQTT and ZeroMQ with the same API.

#### LotusScript

* [MQTT From LotusScript](https://tingenek.wordpress.com/2011/11/30/mqtt-with-lotus-notes/)

#### Lua

* [Eclipse Paho Lua](http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.lua.git/)

####.NET / dotNET

* [MqttDotNet](http://sourceforge.net/projects/mqttdotnet/)
* [nMQTT](https://github.com/markallanson/nmqtt)
* [M2MQTT](https://m2mqtt.codeplex.com/)
* [KittyHawkMQ] (http://www.kittyhawkmq.com/)

#### Objective-C

* [mqttIO-objC](https://github.com/m2mIO/mqttIO-objC)
* [libmosquitto](https://mosquitto.org) - via wrappers ([example](https///github.com/njh/marquette))
* [MQTTKit](https://github.com/jmesnil/MQTTKit) ([sample app](https///github.com/jmesnil/MQTTExample))
* ["MA9B" zip of 1/2 dozen mobile clients source code including Objective-C](http://www-933.ibm.com/support/fixcentral/swg/selectFix?product=ibm%2FWebSphere%2FWebSphere+MQ&fixids=1.0.0.1-WS-MQCP-MA9B&source=dbluesearch&function=fixId&parent=ibm/WebSphere)

#### OCaml

* [mqtt\_client](https://github.com/philtomson/mqtt_client)

#### Perl

* [net-mqtt-perl](https://github.com/beanz/net-mqtt-perl)
* [anyevent-mqtt-perl](https://github.com/beanz/anyevent-mqtt-perl)
* [WebSphere-MQTT-Client](http://search.cpan.org/dist/WebSphere-MQTT-Client/)
* Net::MQTT::Simple [cpan](https://metacpan.org/pod/Net::MQTT::Simple) [github](https://github.com/Juerd/Net-MQTT-Simple)

#### PHP

* [phpMQTT](http://github.com/bluerhinos/phpMQTT)
* [Mosquitto-PHP](https://github.com/mgdm/Mosquitto-PHP)
* [sskaje's MQTT library](http://github.com/sskaje/mqtt)

#### Python

* [Eclipse Paho Python](http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.python.git/) - originally the mosquitto Python client
* [nyamuk](https://github.com/iwanbk/nyamuk)
* [MQTT for twisted python](https://github.com/adamvr/MQTT-For-Twisted-Python)

#### REXX

* [REXX MQTT](https://github.com/DougieLawson/REXX_MQTT)

#### Ruby

* [ruby-mqtt](https://github.com/njh/ruby-mqtt)
* [em-mqtt](https://rubygems.org/gems/em-mqtt)
* [mosquitto](https://github.com/xively/mosquitto)

#### Shell Script

* [bish-bosh](https://github.com/raphaelcohn/bish-bosh), supports bash, ash (including BusyBox), pdksh and mksh.

#### Tcl

* [tcl-mqtt](https://github.com/Tingenek/tcl-mqtt)

### Brokers

Server | QoS 0 | QoS 1 | QoS 2 | auth | [bridge](bridge_protocol) | [$SYS](conventions#$sys) | SSL | [dynamic topics](are_topics_dynamic) | cluster | websockets | plugin system ------ | ----- | ----- | ----- | ---- | ------------------------- | ------------------------ | --- | ------------------------------------ | ------- | ---------- | ------------- | [mosquitto](mosquitto_message_broker) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ | ✔ | ✔ | [RSMB](http://mqtt.org/wiki/doku.php/really_small_message_broker) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ | ✔ | ✘ | ✘ | ? | [WebSphere MQ](http://www-03.ibm.com/software/products/en/wmq/) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ? | ? | ? | [HiveMQ](http://www.hivemq.com) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | [Apache Apollo](http://activemq.apache.org/apollo) | ✔ | ✔ | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ | ? | ✔ | ? | [Apache ActiveMQ](http://activemq.apache.org/) | ✔ | ✔ | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | [my-Channels Nirvana Messaging](http://www.my-channels.com/products/nirvana.html) | ✔ | ✔ | ✔ | § | ✘ | ✘ | ✔ | ✘ | ? | ? | ? | [RabbitMQ](http://www.rabbitmq.com/blog/2012/09/12/mqtt-adapter/) | ✔ | ✔ | ✘ | ✔ | ✘ | ✘ | ✔ | ✔ | ? | ? | ? | [MQTT.js](https///github.com/adamvr/MQTT.js) | ✔ | ✘ | ✘ | § | ✘ | ✘ | ✔ | ✔ | ✘ | ? | ✘ | [moquette](http://code.google.com/p/moquette-mqtt/) | ✔ | ✔ | ✘ | ? | ? | ? | ? | ? | ✘ | ✘ | ✘ | <mosca> | ✔ | ✔ | ✘ | ✔ | ? | ? | ? | ? | ✘ | ✔ | ✘ | [IBM MessageSight](http://www-03.ibm.com/software/products/en/messagesight/) | ✔ | ✔ | ✔ | ✔ | ✘ | ✔ | ✔ | ✔ | § | ✔ | ✘ | [2lemetry](http://2lemetry.com/platform/) | ✔ | ✔ | ✔ | ✔ | ✔ | § | ✔ | ✔ | ✔ | ✔ | ✘ | [GnatMQ](http://mqttbroker.codeplex.com/) | ✔ | ✔ | ✔ | ✔ | ✘ | ✘ | ✘ | ✔ | ✘ | ✘ | ✘ | [JoramMQ](http://mqtt.jorammq.com) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | [ThingMQ](https://thingmq.com) | ✔ | ✔ | ✔ | ✔ | ✔ | ✘ | ✔ | ✔ | ✔ | ✔ | ✔ | [VerneMQ](https://verne.mq) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | Key: ✔ supported ✘ not supported ? unknown § see limitations

## Requirements Analysis

Analysis against **Messaging Node** Requirements

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* the Client Server API could be wrapped in a protocol stub, that can be downloaded at runtime
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Yes
* Using [MQTT over WebSockets](http://www.hivemq.com/mqtt-essentials-special-mqtt-over-websockets/)
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* Yes
* Using [Persistence Sessions](http://www.hivemq.com/mqtt-essentials-part-7-persistent-session-queuing-messages/)
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes
* Using a QoS1 or QoS2 level.
* [More Info](http://www.hivemq.com/mqtt-essentials-part-6-mqtt-quality-of-service-levels/)
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Yes
* Some Brokers can be clusters
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Yes
* Using [Persistence Sessions](http://www.hivemq.com/mqtt-essentials-part-7-persistent-session-queuing-messages/)
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes
* Called : [Last Will and Testament](http://www.hivemq.com/mqtt-essentials-part-9-last-will-and-testament/)
* And Using [MQTT Keep Alive and Client Take-Over](http://www.hivemq.com/mqtt-essentials-part-10-alive-client-take-over/)
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes, because of the small overhead and simple architecture
* [In the usec range and 273M mobile messages/sec per rack](https://mobilebit.wordpress.com/2013/05/03/rest-is-for-sleeping-mqtt-is-for-mobile/)
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes
* Core Feature
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* [Yes](http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718111)
* Depends on the Broker, can be implemented on a higher abstraction layer [Link1](https://auth0.com/docs/scenarios/mqtt) [LInk2](http://www.hivemq.com/mqtt-security-fundamentals-oauth-2-0-mqtt/)
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* Yes
* Core Feature

## PSYC

PSYC [62] is a mostly text-based protocol, aiming at providing a decentralized global messaging infrastructure for unicast/multicast chatting and social media exchanging. Its goal is to replace the popular IRC protocol, which currently suffers from unscalability, lack of security and requires complex bureaucracy. This is because in IRC each server node may take responsibility over control aspects of a given network channel he participates in, adding this way unneccessary complexity and also raising security issues, like when an evil server takes the authority of the channel for malicious purposes. PSYC solves this by implementing a context master node (or well-known distributed set of nodes), that the protocol recognizes as the authority node(s). All other server nodes distributing this context to their members may perform jobs on behalf of the context master, but authority questions like allowing/banning users to/from entering the channel is always up to the master. Also, unlike current protocols like talk or msend, PSYC intends to be more people-oriented, identifying them through Uniform Network Identifications (UNI) (e.g. psyc://psyc.kanzlerant.de/~gerhard) or Uniform Network Locations (UNL) in case of a shared messaging room or conference.

### Architecture and main functions

On a top-level view, PSYC combines an IRC-like topology with the concepts of context, logical targets and packet IDs. Multiple PSYC nodes can be deployed in a tree to help in routing messages from the source to the destination, as it happens in IRC. Context slaves allow better routing options to be automatically discovered when multiple recipients ask to receive data from a given source (UNI/UNL), lowering the amount of traffic. Logical targets are the end-users to whom a given message is targeted. Finally, packet IDs allow PSYC to use redundant multicast strategies, when more than one may fit our needs. This way, duplicate packets due to multiple strategies can be caught and ignored. On a conference server, the minimalistic control module can be also used to deliver group messages in a peer to peer manner, by maintaining on each member a list of other members and how to reach them.

### Evolution to PSYC2

Due to the deprecated state of some key-concepts PSYC relies on, such as its uniform-based routing layer and federation architecture, the PSYC project is moving onto its second version: PSYC2. This new version combines the PSYC old message syntax with a pseudonymous routing technology which defines that an entity's address is its public key itself instead of a string in a uniform, and these public keys should be looked up in a Distributed Hash Table (DHT), avoiding this way the misunderstanding and spoofing problems that the uniform-based solution presented. Apart from that, the federation concept stated that each entity should run its own server and applications had the responsibility of connecting to the appropriate servers. This brings privacy and trust issues, since two entities exchanging messages on a server would need to trust the server owner, which generally is one of the reasons why users continued using the centralized messaging services offers instead of these free solutions. Also, there is a scalability problem due to the lack of resources on entity-owned servers. For example, an average linux-server could not efficiently distribute a multicast message to millions of recipients, in contrary to the powerful servers supporting big companies' services. To overcome these problems, PSYC developers are working on a fully-distributed end-to-end privacy-enabled solution, [secushare](http://secushare.org), a distributed social network operating on top of the public-key routing method explained above combined with PSYC messaging logic.

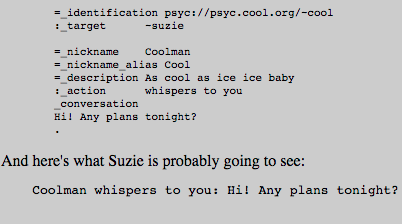


Figure 33: PSYC Message

### Requirements Analysis

Analysis against **Messaging Node** Requirements

* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* The PSYC2 project implementation, Secushare, comes with an [API for pubsub](http://secushare.org/pubsub)
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* PSYC has its own authentication method, inherited in PSYC2 specification. A request\_authentication\_method is used to query a UNI if a given network entity is actually a linked location of that UNI. This method can have different arguments (\_location, \_host\_IP, \_nonce, \_password) to help the querying entity to take a decision. However, there is no evidence in the documentation that PSYC is able to accept external authentication/authorisation methods other than this.
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* The performance of PSYC has been studied, and its wiki relies on benchmarks to say that PSYC is the fastest, yet extensible text-based protocol they are aware of. However, the benchamrk results are not available at this time.
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* PSYC server (psyced) can currently be deployed on Linux, Mac OSX and Windows (on a Cygwin environment) systems.
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* PSYC applies some techniques in order to reduce message delivery latency. First, by avoiding negotiations between nodes "talking" the same protocol between them. Since PSYC supports IRC and XMPP, if two nodes are exchanging messages through XMPP protocol, PSYC suggests the protocol switch in order to reduce latency. Also, PSYC avoids resource discovery (disco on XMPP) by pushing information to possibly interested recipients in advance. However, by applying TLS for encrypted PSYC and techniques for DoS prevention on psyced, a certain degree of latency is, therefore, inevitable.
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* PSYC currently features notification interfaces for software versioning systems (CVS and Git), syslog daemon events, MediaWiki page edits, phpBB forum events and IRC chat messages. Currently, there is no reference to notifications on clients' connection to and disconnection from messaging nodes.
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* TODO
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* TODO
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* PSYC provides three message families to inform clients about problems on message delivery. The \_error method family features methods like \_error\_invalid, \_error\_illegal, \_error\_duplicate and informs the client of a problem occuring on his side, rather on the server side. Basically, it is the server telling a client "It's your fault, not mine". The \_failure method family informs a message sender about a problem on the receiving side. This method family features methods like \_failure\_deliver or \_failure\_redirect (when a given destination changed its address). Finally, the \_warning method family means that a message was processed and sent, but maybe not as intended. An example is \_warning\_usage, which indicates a possible mistake on the message syntax, presented on Fig. 1, and has a single variant for each of the syntax fields.
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* On PSYC, each server running psyced implements the concept of "log of last messages" for every UNI registered on that server. It is used to store messages received by the server, and to let every user have its last session backlog whenever he logs in. It is possible to tune the log size for each UNI and to export the chat history of a room to a webpage.
* [Message Caching](https://github.com/reTHINK-project/core-framework/issues/19)
* PSYC does not have any reference to Store and Forward, probably because it goes against real time communication. And about caching in general, it states it's oriented towards using push events instead of caching. So, whenever a push event regarding a given resource is received by the server, anyone accessing the resource at that time will see it refresh, in order to present always the most recent version.
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Current implementations of PSYC do not support WebSockets nor HTTP Live Streaming. About HTTP Long-Polling, it does not make much sense in the context of PSYC, since it models all data distribution based on an event push system. So, whenever some potentially interesting information for a recipient is available at the server, it is automatically sent, overcoming this way the need of something like HTTP Long-Polling or even REST.
* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* The psyced implementation of PSYC has a negotiation feature of protocol switch advertising. This way, each node has information about supported protocols on all the nodes it is communicating with. However, this could be achieved, since the Client-Server API could be wrapped in a protocol stub, that can be downloaded at runtime.

## Node.js

### Overview

In the scope of the reTHINK project NodeJs [22] is a candidate technology for the Messaging Node.

From https://nodejs.org/ :

Node.js® is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Several additional packages are available to be used with nodeJs : https://www.npmjs.com/

It can be used with component like Redis, socket.io to enhance connectivity and redundancy

A fork of nodejs exist : iojs but a merge seems to be ongoing.

NodeJs is already used in several WebRTC platform or product. (NodeJs can be used in front of Kurento)

### Architecture

NodeJS is divided into two main components: the core and its modules. The core is built in C and C++. It combines Google’s V8 JavaScript engine with Node’s Libuv library and protocol bindings including sockets and HTTP.

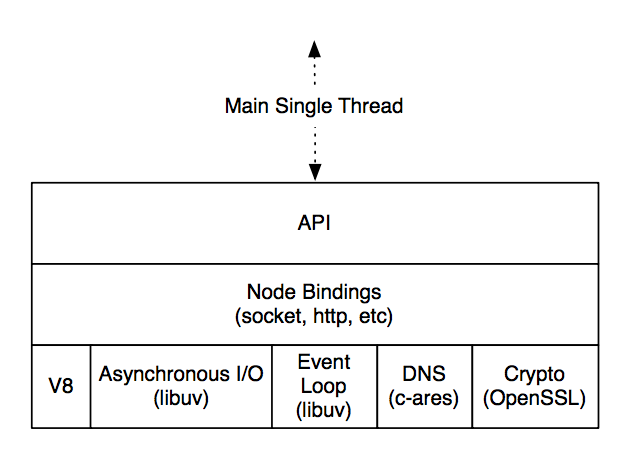


Figure 34: NodeJS Architecture

More information are avaialble in this document : http://mcgill-csus.github.io/student\_projects/Submission2.pdf

### APIs

Available API are described on nodejs website : https://nodejs.org/api/

But this can be completed thanks to the different packages available on https://www.npmjs.com/

### Requirements Analysis

#### [Messaging Node Requirements](https://github.com/reTHINK-project/core-framework/labels/Messaging%20Node%20Requirement)\*\*

* [It should be possible to support Protocol on-the-fly](https://github.com/reTHINK-project/core-framework/issues/21)
* Yes
* ProtOFly connector can be developped. JS connector can be develop on top of NodeJs to enble protofly on server side. This connector will be for example reusable to connect Kurento Media Server
* [Messaging Transport Protocols](https://github.com/reTHINK-project/core-framework/issues/20)
* Yes (socket.io). Socket.io enables the usage of different transport protocol to establish connection between user and server. (Long polling, WebSocket ...)
* [Message Node logging](https://github.com/reTHINK-project/core-framework/issues/18)
* Yes - Several logging modules available : log4js, winston, bunyan ... Logs can be dispalyed in console, store in file with log rotate, send to a network entity ...
* [Message delivery reliability](https://github.com/reTHINK-project/core-framework/issues/17)
* Yes - Socket.io enables message acknowledgement
* [Messaging Node deployments with carrier grade scalability](https://github.com/reTHINK-project/core-framework/issues/16)
* Using:
  + Cluster Mode
  + Redis cluster : it is possible to use Redis Cluster with PUB/SUB mechanism : several NodeJs entities can be connected through the redis cluster : this can enable load balancing, redundancy
* [Messaging Node should be tolerant to unstable connections](https://github.com/reTHINK-project/core-framework/issues/15)
* Yes - socket.io can manage reconnection with different configurable parameters (timeout, retries ...)
* reconnection whether to reconnect automatically (true)
* reconnectionDelay how long to wait before attempting a new reconnection (1000)
* reconnectionDelayMax maximum amount of time to wait between reconnections (5000). Each attempt increases the reconnection by the amount specified by reconnectionDelay.
* timeout connection timeout before a connect\_error and connect\_timeout events are emitted (20000)
* [Events about clients connection / disconnection from Messaging Node](https://github.com/reTHINK-project/core-framework/issues/14)
* Yes - using socket.io different event are fired on connection status :
* connect. Fired upon connecting.
* error. Fired upon a connection error
* disconnect. Fired upon a disconnection.
* reconnect. Fired upon a successful reconnection.
* reconnect\_attempt. Fired upon an attempt to reconnect.
* reconnecting. Fired upon an attempt to reconnect.
* reconnect\_error. Fired upon a reconnection attempt error.
* reconnect\_failed. Fired when couldn’t reconnect within reconnectionAttempts
* [Messaging Node must support very low message delivery latency](https://github.com/reTHINK-project/core-framework/issues/13)
* Yes
* [Messaging Node must be deployable in the most used Virtual Machines](https://github.com/reTHINK-project/core-framework/issues/12)
* Yes - NodeJs is available on Linux, windows, mac
* [Messaging Node should require minimal computing resources](https://github.com/reTHINK-project/core-framework/issues/11)
* Yes
* [Messaging Node must support external authentication and Authorisation](https://github.com/reTHINK-project/core-framework/issues/10)
* Yes. Module like Passport : http://passportjs.org/ enables to use external authentication like facebook, twitter, google .. (We will have to check if passport can be used as it seems to require Express which may not be relevant in rethink case)
* [Messaging Node must support pub/sub](https://github.com/reTHINK-project/core-framework/issues/9)
* No - Yes with Redis Pub/Sub mechanism : http://redis.io/topics/pubsub

#### [Runtime Requirements](https://github.com/reTHINK-project/core-framework/labels/Runtime%20Requirement)

**Web Sockets on node.js**

This suplemments the lack of implementation of Web Sockets inside v8.

npm install ws

IT is easy to use, fast and up-to-date against current HyBi protocol drafts. IT can send and receive typed arrays (ArrayBuffer, Float32Array) as binary data. IT passes the extensible Autobahn test suite: client report ( Autobahn WebSockets v0.4.5.), server report ( Autobahn WebSockets v0.4.5.).

It has extensive benchmarks relative to its overhall performance in node.js.

In fact there are extensive implementations of Web Sockets over node.js: websocket-node faye-websocket-node engine.io socket.io sockjs

The most performante is claimded to be ws.

Socket.io seems to be the most complete regarding feature (connection type, acknowledgement, configuration ..)

# QOS SOTA

## coturn

### Overview

The TURN [47] protocol is defined as an extension of the STUN [48] protocol.

TURN servers act as media relays and are directly placed in the media path. TURN server gets all the media packets from one endpoint and sends them towards the remote peer (remote endpoint) or to another TURN server, if it is used by the remote endpoint.

TURN server acts as a relay for the packets. In the most common case, a TURN server is in the public Internet and the hosts are behind NATs or restrictive firewalls.

The study about TURN servers is very up-to-date since there is an ongoing work on this subject in IETF. The recently formed IETF group – TRAM (TURN Revised and Modernized) focuses on improving TURN implementations and features in order to make STUN and TURN more suitable for WebRTC [49].

Coturn is an open source TURN server implementation [50]. It is a separate branch of the previous implementation rfc5766-turn-server-project, and is dedicated for testing new protocols. As a result, it supports more specifications than the previous version.

The asset can be used for network QoS.

### Requirements Analysis

To use coturn, download the latest official version, at least 4.4.

TURN-based solution would allow a collaborative approach as it can be placed in a media path and be used for flow steering. It can impact different network types and segments and thanks to its compatibility with existing solutions, it can be introduces incrementally.

# Service Frameworks SOTA

### AngularJS Framework

#### Overview

AngularJS [51] is a Javascript MVC Framework developed and promoted by Google, to build well architectured and maintainnable web applications. It is usually mistaken for a library due to its lightweight than normal frameworks. It is entirely based on Javascript and a client side framework. It is supported by multiple browsers.

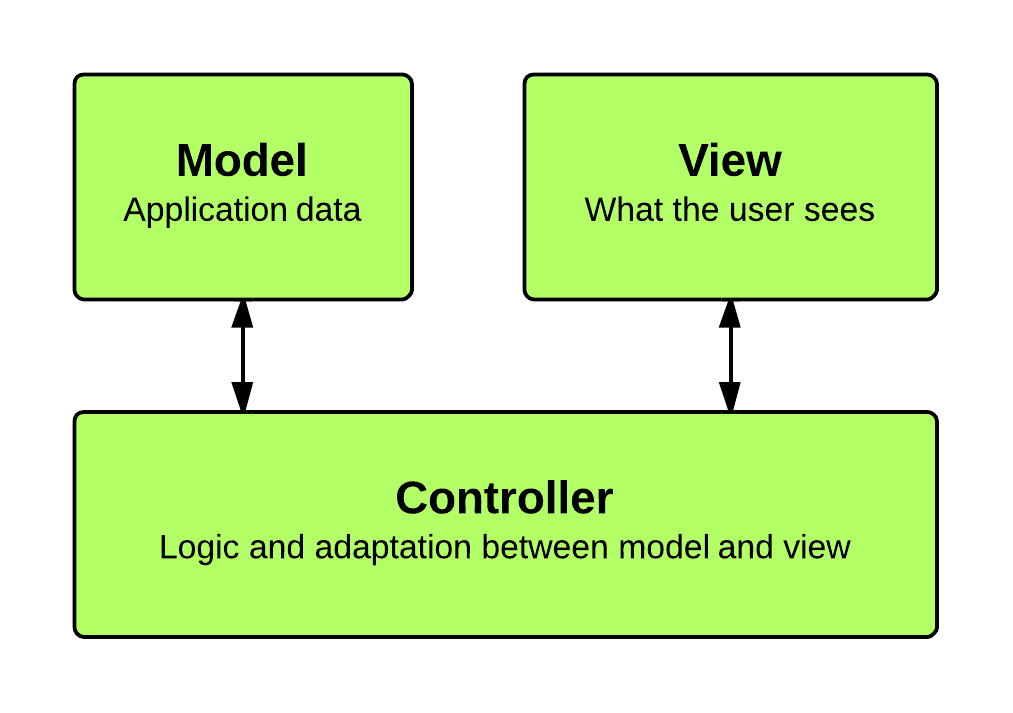


Figure 36: AngularJS Framework

**Main Features**

* *Compiler* : Compiler is an angular service which traverses the DOM looking for attributes. It allows the developer to teach the browser new HTML syntax. The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior. Angular calls these behavior extensions directives. The compilation process happens in two phases. Compile: traverse the DOM and collect all of the directives. The result is a linking function. Link: combine the directives with a scope and produce a live view. Any changes in the scope model are reflected in the view, and any user interactions with the view are reflected in the scope model. This makes the scope model the single source of truth.
* *Directive* : Directives can be placed in element names, attributes, class names, as well as comments.  Directives are a way to teach HTML new tricks. A directive is just a function which executes when the compiler encounters it in the DOM <input ng-model='name'>. It is possible to define custom derective <span draggable>Drag ME</span>.
* ng-app Directive: Angular uses this directive to auto-bootstrp an application. Only one ng-app directive can be used per HTML document. <html ng-app>
* Expression : Expressions are JavaScript-like code snippets that are usually placed in bindings such as {{ expression }} ```
* 1+2={{1+2}}

\* \*Form & Control\* : Forms and controls provide validation services, so that the user can be notified of invalid input. This provides a better user experience, because the user gets instant feedback on how to correct the error.``  
Modules: Modules declaratively specify how an application should be bootstrapped. There can be multiple modules in an app  
Those could be interdependent too. Modules are configured with routes, controllers, models etc.

var myAppModule = angular.module('myApp', [--here goes the dependent Modules--]); ```

* *Routing* : It Is used for deep-linking URLs to controllers and views (HTML partials). It watches $location.url() and tries to map the path to an existing route definition.
* *Scope* : Scope is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watch expressions and propagate events.
* *Dependency Injection* : Dependency Injection (DI) is a software design pattern that deals with how code gets hold of its dependencies
* *Filters* : Angular filters format data for display to the user. Custon filters can be created as well

#### Evaluation

##### Positive

* Encourages MVC design pattern
* Two way data binding allows for automation synchronization of data bewteen model and view components
* Written for testability and with modularized code which enables clarity , extensibiliy and strong built-in-services
* Active project with a large eco-system, leading to higher rates of inquries answered and new developments
* End-to end integration testing and Unit testing
* Fast Development once familiar with structure and concept
* Very expressive, leading to less code for same result as with other libraries

##### Negative \* Complex Directives API \* Good for Single Page Apps (SPA), SO, is not the best option to go for Hyperty Developments \* Complex Directives API \* Runtime configuration only only before Bootstrap procedure. No configuration possible after. \* Scopes are easy to use but difficult to debug

#### Requirement Analysis

Analysis against **Service Framework** Requirements

* [Service Framework **MUST** be Message Node agnostic](https://github.com/reTHINK-project/core-framework/issues/44): **Yes**
* [The Service Framework **MUST** avoid any JavaScript conflicts](https://github.com/reTHINK-project/core-framework/issues/43): **No** This is a namespace issue and potential conflicts with other frameworks are likely to occur
* [Service Framework **MUST** be Modular in nature](https://github.com/reTHINK-project/core-framework/issues/42): **Yes** The framework is modular, and offers developers possibility to write their own modules or import third party modules.
* [The Service Framework **MUST** be open source](https://github.com/reTHINK-project/core-framework/issues/39): **Yes**

AngularJS is an open-source framework maintained by Google and by a community of individual developers and corporations - [Service Framework **SHOULD** be device agnostic](https://github.com/reTHINK-project/core-framework/issues/38): **Yes** It runs on all devices and operating systems envisioned to host the Hyperty Runtime (Android, iOS, Raspberry PI, Linux VM, Windows VM) - [Service Framework **MUST** be light weight and fast](https://github.com/reTHINK-project/core-framework/issues/37): **yes** The current minimized gzipped web version is 172KB while the minimized gzipped web app version is 146KB. Gzip compresses it down to one third and Data per page has small memory footprint

* [Service Framework **SHOULD** support Model-View-Controller design pattern](https://github.com/reTHINK-project/core-framework/issues/36): **Yes** The Angular framework provides a powerful MVC design Pattern

### BackboneJS Framework

#### Overview

BackboneJS [52] is a web frontend framework that provides structure to web applications by specifying models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

##### Models and Views

The model looks as follows:



Figure 37: BackboneJS Framework

##### Model

* Orchestrates data and business logic,
* Loads and saves from the server,
* Emits events when data changes.

A Model manages an internal table of data attributes, and triggers "change" events when any of its data is modified. Models handle syncing data with a persistence layer — usually a REST API with a backing database. Design your models as the atomic reusable objects containing all of the helpful functions for manipulating their particular bit of data. Models should be able to be passed around throughout your app, and used anywhere that bit of data is needed.

##### View

* Listens for changes and renders UI.
* Handles user input and interactivity.
* Sends captured input to the model.

A View is an atomic chunk of user interface. It often renders the data from a specific model, or number of models — but views can also be data-less chunks of UI that stand alone. Models should be generally unaware of views. Instead, views listen to the model "change" events, and react or re-render themselves appropriately.

##### Collections

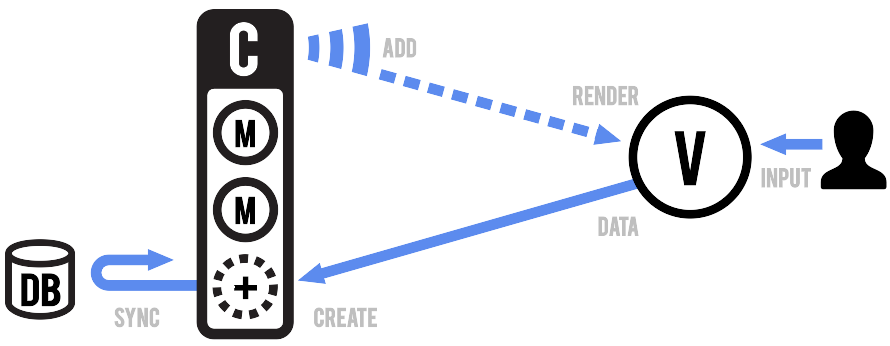


Figure 38: Backbone Collections

A Collection helps you deal with a group of related models, handling the loading and saving of new models to the server and providing helper functions for performing aggregations or computations against a list of models. Aside from their own events, collections also proxy through all of the events that occur to models within them, allowing you to listen in one place for any change that might happen to any model in the collection.

#### Overall Evaluation

* No concept of separated Controller
* functionality is basic
* provides no "golden pattern" for structuring of an application
* leaves lots of structuring decisions to the developer
* well suited for creation of own frameworks
* **no two-way data binding --> lots of boilerplate code required**
* **views change DOM directly by looking up css class names**
* --> changing CSS or modifications in DOM (wrapping, nesting) requires updates in code

#### Requirement Analysis

Analysis against **Service Framework** Requirements

* [Service Framework **MUST** be Message Node agnostic](https://github.com/reTHINK-project/core-framework/issues/44): **Yes** -[Service Framework SHOULD support Model-View-Controller design pattern](https://github.com/reTHINK-project/core-framework/issues/36)
* NO
* The Backbone framework provides a MV Pattern with direct interaction between Models and Views.
* The controller part is mainly done in the code of the models and also the views.
* [Service Framework MUST be light weight and fast](https://github.com/reTHINK-project/core-framework/issues/37)
* YES
* minimized gzipped version is very small (approx 5.6kb)
* mandatory dependcies to underscore.js (5kb) and jQuery (32kb) or Zepto(9,1kb, a JQuery clone)
* small memory footprint
* [Service Framework **SHOULD** be device agnostic](https://github.com/reTHINK-project/core-framework/issues/38)
* rather YES
* every view is tight to its own root-DOM element and responsible for the tree below it
* Therefore Backbone.js relies on runtimes that provide a DOM tree.
* But this does NOT have to be a "real" DOM tree --> can be used with e.g. React virtual DOM (React has implemented a browser-independent events and DOM system)
* --> needs special additions in non-browser runtime environments
* [Service Framework MUST be Modular in nature](https://github.com/reTHINK-project/core-framework/issues/42)
* rather NO
* Backbone itself lacks a Controller concept and Views and Models are relatively tightly coupled
* therefore also resulting modules should be tightly coupled to view elements and not easily portable (would need further investigations)
* Models stand-alone and their synchronization capabilities with backend storages should be portable and fulfill this requirement

### MeteorJS Framework

#### Overview

[Meteor](http://docs.meteor.com/#/full/quickstart) [39] is a Javascript plataform, offering a complete full-stack framework for delivering web and mobile apps;

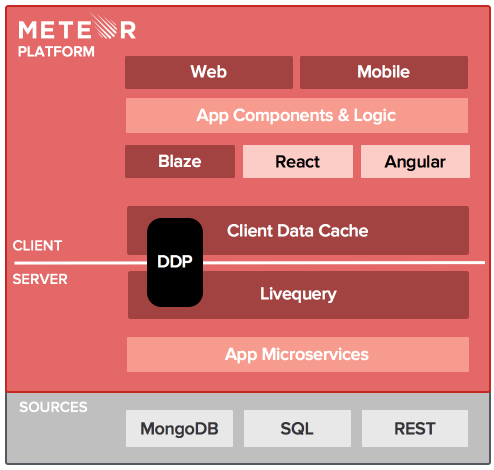


Figure 39 Meteor Plataform Overview

**Main Features**

* It allows rapid prototyping and produces cross-platform code (Web, Android, iOS).
* Meteor uses Blaze library for the front-end which fulfills the same purpose as Angular, Backbone or Ember but is much simpler to use and can be used with any of these libraries.
* Blaze is also faster than Angular in high-throughput rendering tests.
* Meteor works well with REST APIs using DDP, a simple protocol for fetching structured data from a server, and receiving live updates when that data changes.
* The underlying database is MongoDB which Meteor works in real-time.
* While Angular is a front-end library and Sails a backend one, Meteor is a full-stack framework, comparable with Derby.
* The MVC model is replaced by MVM (Model-View-Mapper) which replace controller by ORM code that supports what views need to display data. Meteor controllers are on the client side.

#### Special directories

With this directories we can split the server code from the client code, for a better code organization, and also, what we want to share from what we want to make private;

* **/client** Any files here are only served to the client. This is a good place to keep your HTML, CSS, and UI-related JavaScript code.
* **/server** Any files in this directory are only used on the server, and are never sent to the client. Use /server to store source files with sensitive logic or data that should not be visible to the client.
* **/public** Files in /public are served to the client as-is. Use this to store assets such as images. For example, if you have an image located at /public/background.png, you can include it in your HTML with or in your CSS with background-image: url(/background.png). Note that /public is not part of the image URL.
* **/private** These files can only be accessed by server code through Assets API and are not accessible to the client.

**or** we can, in same javascript file, wrapping our code in

app.js  
  
if (Meteor.isClient) {  
 // code will run only on client side  
}  
  
if (Meteor.isServer) {  
 // code will run only on server side  
}

#### Code convention

Client connection:

Tasks = new Mongo.Collection("tasks");

Insert:

Template.body.events({  
 "submit .new-task": function (event) {  
 // This function is called when the new task form is submitted  
 var text = event.target.text.value;  
 Tasks.insert({  
 text: text,  
 });  
 }

Query:

Template.body.helpers({  
 tasks: function () {  
 // Show newest tasks first  
 return Tasks.find({}, {sort: {createdAt: -1}});  
 }  
)};

Delete:

Template.task.events({  
 "click .delete": function () {  
 Tasks.remove(this.\_id);  
 }  
)};

#### Evaluation

##### Positive

* Full Stack package system
* Mobile version with Cordova/Phonegap
* Multiplataform Distribution
* Sofisticaded UI
* [Atmosphere Tool](https://atmospherejs.com/) list all available packages to MeteorJS

##### Negative

* Meteor Development Group has built specifically for a reactive front-end experience
* For applications with very large numbers of concurrent connections (like facebook), Meteor might require larger amounts of RAM than other solutions, but can be optimized

#### Requirement Analysis

* [Service Framework **MUST** be Message Node agnostic](https://github.com/reTHINK-project/core-framework/issues/44): **No**

It has its own "message node", who is called [DDP (distributed Data Protocol)](https://www.meteor.com/ddp).

* [The Service Framework **MUST** avoid any JavaScript conflicts](https://github.com/reTHINK-project/core-framework/issues/43): **Yes**

The platform is constructed in pure javascript, don't have any other frameworks dependencies;

* [Service Framework **MUST** be Modular in nature](https://github.com/reTHINK-project/core-framework/issues/42): **Yes**

The platform is complete modular, we can write our own modules or import third party modules.

* [The Service Framework **MUST** be open source](https://github.com/reTHINK-project/core-framework/issues/39): **Yes**

*“Meteor is a complete open source platform for building web and mobile apps in pure JavaScript.”*

* [Service Framework **SHOULD** be device agnostic](https://github.com/reTHINK-project/core-framework/issues/38): **Yes**
* [Service Framework **MUST** be light weight and fast](https://github.com/reTHINK-project/core-framework/issues/37): **yes**

Meteor is build on top of NodeJs and MongoDB, and both are fast;

* [Service Framework **SHOULD** support Model-View-Controller design pattern](https://github.com/reTHINK-project/core-framework/issues/36): **Yes/No**

**No**, it isn't MVC pattern is more a MVM(Model-View-Mapper), but **Yes** we can have MVC, in other way:

*"The MVC pattern in Meteor is dead simple. The Model is coded up in HTML, the Controller is coded in Javascript, and the View is coded up in CSS. It's that simple."*

Model HTML What Is Displayed   
View CSS How It Is Displayed   
Controller Javascript When It Is Displayed

### StapesJS Framework

#### Overview

StapesJS [53] is a little web framework designed to be agnostic about the setup and style of coding. It is very flexible and it allows to use MVC paradigm combined with libraries such as jQuery, Zepto, React and Rivets. Stapes provides the necessary building blocks to build a apps in a short-time.

#### Advantages and main features

* It allows class creation, custom events, and data methods. It only has 20 methods compared to other web-frameworks like Backbone which has more than 75.
* It is very light so the loading time will be very short. It is just 2kb (minified and gzipped) so it is suitable for web sites designed for mobile devices.
* It has around 600 lines of codes, so if any issue is find or it is necessary to add some new feature it can be done at a small cost.

#### Drawbacks

* It is less complete than other frameworks so some functionality may require some development.
* The last commit was done the Summer of 2014 so its contributors are not very active in Github. This may be a problem if any issue is found.

#### Requirement Analysis

* It is an Open Source project. It has MIT license which is a permissive free software license. meaning that it permits reuse within proprietary software provided all copies of the licensed software include a copy of the MIT License terms and the copyright notice. Such proprietary software retains its proprietary nature even though it incorporates software under the MIT License. It can be used without restrictions in the project.
* ReTHINK develeopments are not expected to be very complex so StapesJS could be a right choice. Additionally its small size will reduce the overhead in the applications which will help to reduce the load time.

1. OMNA Network
2. WebRTC API evolution
3. Web Push Architecture
4. Main flows of events for subscription, push message delivery, and unsubscription
5. HTTP/2 Push
6. HTTP/2 Framing
7. HTTP/2 Streams
8. signalling on-the-fly concept
9. Wonder Library Main Classes
10. WebRTC.org architecture scheme
11. OpenWebRTC Architecture
12. V8 Architecture
13. V8 Multiple Contexts
14. Firefox OS Architecture
15. Jitsi Videobridge Architecture
16. Video Bridge Component
17. OpenFire VideoBridge and Jicofo Components
18. Docker Architecture
19. Janus Gateway architecture
20. Kurento Architecture
21. Main data flow in a matrix architecture
22. RabbitMQ Architecture2
23. RabbitMQ Architecture
24. ZeroMQ types of communication patterns
25. ZeroMQ Clusters
26. Redis architecture
27. PubSub model
28. XMPP Architecture
29. XMPP Protocols
30. XMPP Jingle
31. MQTT Architecture
32. PSYC Message
33. NodeJS Architecture
34. Vertx Architecture Diagram
35. AngularJS Framework
36. BackboneJS Framework
37. Backbone Collections
38. Meteor Plataform Overview