

**Deliverable D3.1**

**Hyperty Runtime and Hyperty Messaging Node Specification**

|  |  |
| --- | --- |
| 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: | 1.0 |
| 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 comprehensivee 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. The core of this report contains a detailed specification of the Hyperty Runtime API and of the main procedures to support use cases, requirements and concepts defined in previous reports, providing the basis for the implementation tasks.

[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 number and title: **D3.1 Hyperty Runtime and Hyperty Messaging Node Specification**

Editor: **Paulo Chainho, Altice Labs**

Work-package leader: **Paulo Chainho**, company: **Altice Labs**

Copyright notice

2016 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/3.0http://creativecommons.org/licenses/by-nc-nd/3.0http://creativecommons.org/licenses/by-nc-nd/3.0](http://creativecommons.org/licenses/by-nc-nd/3.0/)

# Executive summary

This document describes the technical details and the information needed by developers to start prototyping reTHINK Core Framework, which is comprised of the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. This document takes as input the conceptual foundations, data models and interfaces definitions from deliverables D2.1 (The reThink Framework Architecture) and D2.2 (the reTHINK Data Model). This report complements deliverable D4.1 (Management and Security features specifications), which specifies reTHINK Support Services, namely: Policy Management, Governance, Identity Management, Graph Connector, and Hyperty Directory services (Catalogue and Registry). The core of this document is dedicated to the detailed specification of the Hyperty Runtime describing in detail, the Hyperty Runtime architecture and the Core Runtime components required to support the execution of Hyperties. The Hyperty Runtime architecture follows a security by design approach since it was highly influenced by a careful security analysis where different types of components are executed in isolated sandboxes. Thus, components downloaded from a specific Service Provider are executed in sandboxes that are different from the sandboxes used to execute components downloaded from another service provider. Communication between components running in different sandboxes is only possible through messages exchanged through a Message Bus functionality provided by the Hyperty Runtime Core Sandbox. The access to the Message BUS functionality is controlled by a Policy Engine which is also located in the Core Runtime sandbox. On the other hand, and according to the ProtoOFly concept introduced in D2.1, the protocol stub is executed in isolated sandbox and provides the bridge for the Hperty Runtime to communicate with associated Service Provider.

The design of the Hyperty Runtime APIs progressed along the design of the main procedures to be performed in order to validate it with the most important use cases that were already used in D2.1 and originally described in D1.1. Thus, basic procedures (e.g. message routing and Hyperty deployment), Identity Management Procedures (e.g. registration and login of users) and Human to Human communication procedures were detailed, including the definition of the data sets and messages as defined in D2.2. The Hyperty Runtime design was also partially validated with Machine to Machine communication and Human to Machine communication use cases, which will be fully reported in D3.2.

Special attention was given on the design of components involved in the Reporter-Observer data synchronisation communication pattern introduced in D2.2, which complements the ProtOFly concepts to support seamless interoperability between domains at service layer. The access control to synchronised objects, through the Reporter-Observer communication pattern, is enforced by the Core Policy Engine. More sophisticated and proprietary data synchronisation algorithms can be used, by enabling the deployment of other Policy Enforcer in the Hyperty Runtime, which will be executed in isolated sandboxes.

A reference design for the Messaging Node Architecture is also provided in this report. 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.

Together, the Hyperty Runtime and the Messaging Node specifications are based on a set of design principles to support Hyperty Instance Mobility (between Network Interfaces and also between Devices), Data Object portability (between Hyperty Instances) and group communication. These characteristics are supported by the usage of different virtual addresses separately allocated to Hyperty Instances and Data Objects, which are agnostic of the network addresses. Hyperties communicate each other by publishing messages on the target Hyperty Instance virtual address, or, in case the Reporter-Observer communication pattern is used, on the synchronised data object virtual address. Any Hyperty Instance granted with authorisation to listen on those virtual addresses, will receive the messages. The separation of concern design principle was also used in order to let Hyperty developers focus on its service logic and leaving business related decisions to product managers, as well as giving the users more control on how service is delivered. As a consequence of this principle, by default, the different security tokens used (including ID Tokens and Access Tokens) are handled by the Core Runtime and not by the Hyperty Instances.

The reTHINK Core Framework detailed specification is achieved by a comprehensive effort on web runtime design state of the art research with special attention given to Security in Web Runtime and relevant W3C and IETF standards. A comprehensive report about the procurement of existing open source solutions to be used to prototype reTHINK Core Framework components, is also presented, mainly in terms of Web Runtime Solutions and Real Time Messaging Solutions.

Taking as input the procurement report, some solutions were selected and some implementation considerations are presented for the Hyperty Runtime and for the messaging solutions.

Some preliminary design guidelines are provided for the implementation of the Hyperty Service Framework. The Hyperty Service Framework is a Software Development Toolkit (SDK) that will feature a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties.

It should be noted that the Network Platform specification supporting Specialised Network Services is an ongoing work that will be reported later in D3.4, as originally planned.

# List of authors

|  |  |
| --- | --- |
| Company | Author |
| PTIN | Paulo Chainho  Vitor Silva  Luis Duarte |
| Fraunhofer |  |
| Quobis |  |
| Apizee |  |
| Deutsche Telekom |  |
| Orange Labs |  |
| INESC-ID |  |
| IMT |  |
|  |  |

# Table of Contents

[Executive summary 3](#_Toc457918018)

[List of authors 5](#_Toc457918019)

[Table of Contents 6](#_Toc457918020)

[Abbreviations 7](#_Toc457918021)

[1 Introduction 9](#_Toc457918022)

[1.1 Objectives and Overview 9](#_Toc457918023)

[1.2 Structure 11](#_Toc457918024)

[2 Core Framework Specification update 12](#_Toc457918025)

[2.1 Runtime Specification Update (Paulo) 12](#_Toc457918026)

[2.1.1 Runtime Architecture 12](#_Toc457918027)

[2.1.2 Runtime Main Procedures 19](#_Toc457918028)

[2.1.3 Runtime Implementation Considerations 47](#_Toc457918029)

[2.2 Messaging Framework Specification Update (Steffen) 54](#_Toc457918030)

[2.2.1 Messaging Node Architecture 54](#_Toc457918031)

[2.2.2 Vertx Specification 56](#_Toc457918032)

[2.2.3 Node.js based Messaging Node Specification 58](#_Toc457918033)

[2.2.4 Matrix.org based Messaging Node Specification 62](#_Toc457918034)

[3 New Features specification 66](#_Toc457918035)

[3.1 Runtime Trust Management Specification (Ricardo Chaves/Nuno) 66](#_Toc457918036)

[3.2 P2P Protofly Specification (Paulo) 66](#_Toc457918037)

[3.3 QoS Control specification (Marc) 66](#_Toc457918038)

[3.4 Multiparty WebRTC Connections specification (Arnaut) 66](#_Toc457918039)

[3.5 Interworking with Legacy Services (Anton) 66](#_Toc457918040)

[4 Conclusions 67](#_Toc457918041)

[5 References 68](#_Toc457918042)

# Abbreviations

|  |  |
| --- | --- |
| API | Application Programming Interface |
| COAP | Constrained Application Protocol |
| CRUD | Create, Retrieve, Update and Delete |
| CSP | Communication service provider |
| DDoS | Distributed Denial of Service Attacks |
| DoS | Denial of Service |
| H2H | Human to Human communication |
| ICE | Information and Content Exchange |
| IETF | Internet Engineering Task Force |
| JSON | JavaScript Object Notation |
| LWM2M | LightweightM2M |
| M2M | Machine to Machine communication |
| ORTC | Object Real-Time Communications |
| QoS | Quality of Service |
| REST | Representational State Transfer |
| STUN | Session Traversal Utilities for NAT |
| TURN | Traversal Using Relay NAT |
| UML | Unified Modelling Language |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
| W3C | World Wide Web Consortium |
| WHATWG | Web Hypertext Application Technology Working Group |
| SPPE | Service Provider Policy Enforcer |
| PEE | Policy Enforcer Engine |
| TRAM | TURN Revised and Modernized |
| HTTP | Hypertext Transfer Protocol |
| TCP | Transmission Control Protocol |
| QUIC | Quick UDP Internet Connections |
| XMPP | Extensible Messaging and Presence Protocol |
| ORTC | Object Real-Time Communications |
| COAP | Constrained Application Protocol |
| LWM2M | Lightweight M2M |
| SDT | Smart Device Template |
| HGI | Home Gateway Iniative |
| SFU | Selective Forwarding Unit |
| MCU | Multipoint Control Unit |
| TLS | Transport Layer Security |
| MQTT | MQ Telemetry Transport |
| WAC | WebRTC Application Controller |
| AAA | Authentication, Authorization and Accounting |
| OSS | Operations Support System |
| BSS | [business support systems](https://en.wikipedia.org/wiki/Business_support_system) |
| RCS | Rich Communication Services |
| UC | Unified Communications |
| CRM | [Customer Relationship Management](https://en.wikipedia.org/wiki/Customer_relationship_management) |
| JSONoWS | JSON over Web Sockets |
| IdP | Identity Provider |
| TCB | Trusted Computing Base |
| PDP | Policy Decision Point |
| PEP | Policy Enforcement Point |

# Introduction

## Objectives and Overview

Project reTHINK proposes a radical transformation on how real time communication services are thought. reTHINK concepts and architecture represents a significant paradigm change for the communication services domain. The reTHINK approach enables the fulfilment of real-time communications requirements that so far have been considered impossible to achieve: trustful identities, interoperable endpoints, agility of introducing new services, and fast moving innovation. Previous Deliverables D2.1 [38] and D2.2 [15] have already started enlightening the path to reach such objectives. A new web service paradigm, the so-called Hyperlinked Entities - Hyperties – was introduced to enable a global network of trustful services executing in web runtime environment, on end-user devices or edge-network servers. Communication between Hyperties is based on the protocol-on-the-fly (ProtoFly) concept that avoids creating or modifying standard network protocols, but utilizes instead standard APIs. Interoperability between Hyperties and Support Services (Registry, Catalog, and Identity Management) are assured by a detailed and extensible data model, combined with the principle of Hypermedia as the Engine of Application State (HATEOAS) as defined in D2.2.

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. This report complements deliverable D4.1 (Management and Security features specifications)[109], which specifies reTHINK Support Services, namely: Policy Management, Governance, Identity Management, Graph Connector, and Hyperty Directory services (Catalogue and Registry). 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 (See Figure 1).



Figure 1 - Specification Scope

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

The reTHINK Core Framework specification provided in this report, is compliant with reTHINK Data Model, Hyperty Management interfaces, Stream Interface and Messaging Interface designed in D2.2 [15]. It should be noted that, according to Protocol On-the-fly concept, the Messaging Interface is defined by the Message Model defined in [15].

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 comprehensive 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.

An exhaustive study of relevant IETF, W3C standards and others that facilitate the fulfillment of previously analysed requirements, is reported. 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 in reTHINK testbeds, interoperability between different Hyperties domains that use different Message Nodes, namely Vertx, Node.js and Matrix.

The experimentations performed on JavaScript engines and WebRTC implementations have shown to be very difficult to extend existing runtimes 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 runtime solutions notably with existing Web Browsers like Chrome and JavaScript platforms like Node.js.

The Runtime design enables reuse 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 Hyperty Runtime architecture follows a security by design approach where different types of components are executed in isolated sandboxes. Communication between different sandboxes is only possible through a Message Bus and is subject to access control. Communication with remote Hyperties is provided by protocol stubs executed in isolated sandboxes.

The design of the Hyperty Runtime APIs is validated with the most important use cases that were already used in D2.1 and originally described in D1.1. The Hyperty Runtime procedures were described for basic procedures (e.g. message routing and Hyperty deployment), Identity Management Procedures (e.g. registration and login of users) and Human to Human communication. Although, the Hyperty Runtime is designed to also support Machine to Machine communication and Human to Machine communication use cases, its procedures will be fully reported in D3.2.

The Messaging Node Reference Architecture is described to provide some guidelines for Messaging Node implementation. Thanks to the protocol-on-the fly concept, a detailed specification of Messaging Node APIs as provided for the Hyperty Runtime, is not required. Instead, a more detailed specification is provided for each messaging solution selected during the procurement activity namely for Vertx.io, Node.js and Matrix.

The main functionalities to be provided by the Hyperty Service Framework, which will be used by Hyperty Developers, is provided at the end. The Hyperty Service Framework is a Software Development Toolkit (SDK) that will feature a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties.

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.

The final specification for Messaging Node and Hyperty Runtime will be reported in D3.3 (Hyperty Runtime and Hyperty Messaging Node Phase 2 – Dec 2016).

## Structure

This report starts with an introduction and, in Chapter 2, requirements that are more specific to the reTHINK Core Framework are clearly identified. In chapter 3 a summary of the State of the Art and Procurement work is given. The full State of the Art and Procurement report can be found in Annex A. The core part of this report is located in Chapter 4, which details the specification of the Hyperty Runtime, and in Chapter 5, the specification of the Messaging Node. This reports concludes with a short description of functionalities to be provided by the Hyperty Service Framework to be used by Hyperty Developers.

# Core Framework Specification update

## Runtime Specification Update (Paulo)

This Chapter contains the detailed specification of the Hyperty Runtime, where Hyperties are executed. It describes in detail the Hyperty Runtime architecture and the Core Runtime components required to support the execution of Hyperties. The Hyperty Runtime architecture follows a security by design approach since it was highly influenced by a careful 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 Hyperty 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 originally 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, procedures performed to route messages among Hyperties and procedures to setup a Reporter-Observer data object synchronisation communication.
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 asserts 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.

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 Figure 10. 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) are executed in sandboxes that are different from the sandboxes used to execute components downloaded from another service provider (e.g. Service Provider 2). 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 is 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 Hyperty Runtime to communicate with associated Service Provider. For example, in Figure 10, 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.

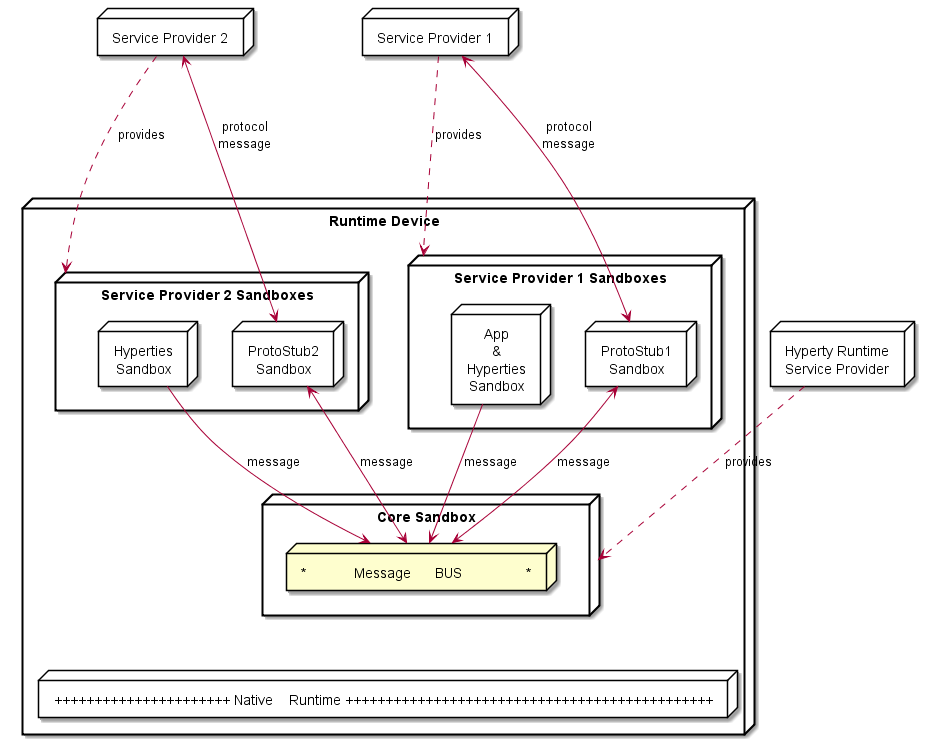


Figure 10: High Level Runtime Architecture with trusted Hyperties

According to [Hyperty Definition](https://github.com/reTHINK-project/architecture/blob/master/docs/concepts/Hyperty.md) introduced in D2.1 [38], an Hyperty is a web software that can be reused by Web Applications through a local (Javascript) API (the Hyperty Application API), which is not subject to standardisation. The Application and the Hyperty can be delivered by the same Service Provider or by different Service Providers, i.e. Hyperty is delivered by an (Hyperty) Service Provider and the Application is delivered by an Application Service Provider. These two different situations impacts the level of trust between the Application and the Hyperty, that should be handled by the Hyperty Runtime accordingly.

In Figure 10, the Application and the Hyperty Instances it consumes, are downloaded from the same Service Provider. Thus, it is assumed they trust each other and that they can be executed in the same sandbox with no impact on how the Application consumes the Hyperty Application API. In Figure 11, 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. In this case, the Hyperty Application API is no longer local and the application is only able to reach the Hyperty Instance through the Message BUS. It is desirable to abstract the Application developer from these situations and to let the Application developer call the Hyperty Application API as if they are always local. This implies that the Core Runtime and the Sandbox implementation, is able to support a Remote Procedure Call (RPC) communication when the Application and the Hyperty Instance are in different sandboxes.

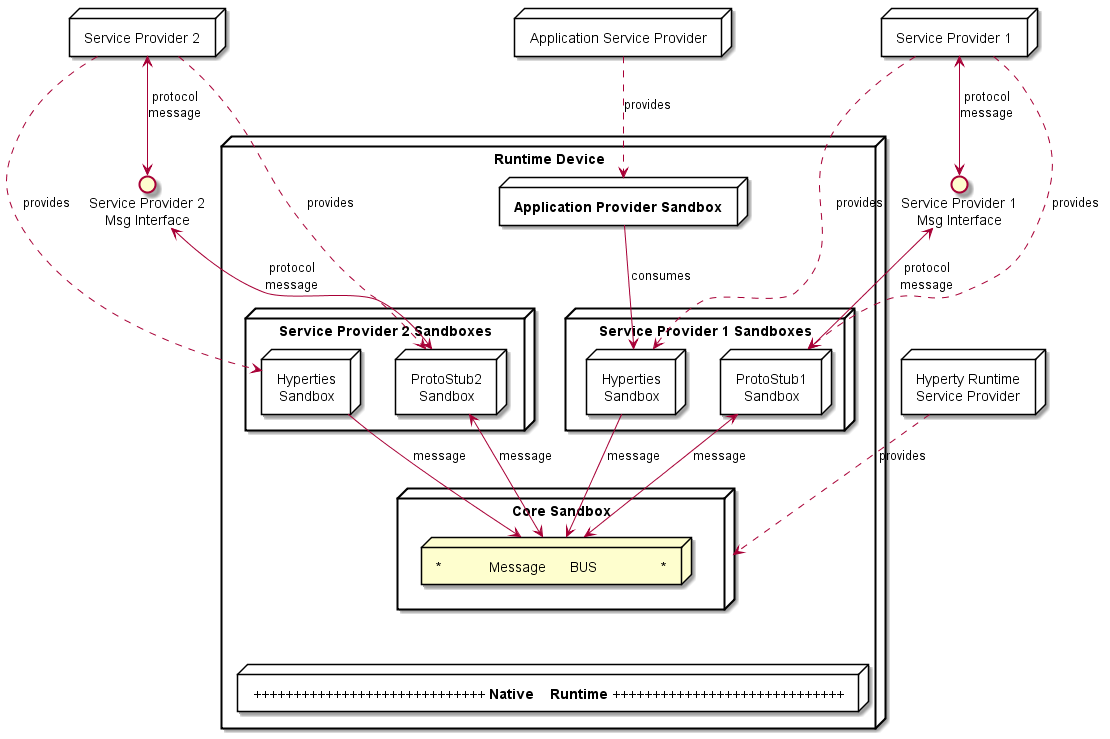


Figure 11: High Level Runtime Architecture with untrusted Hyperties

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 Figure 12) enabling the enforcement of proprietary policies.

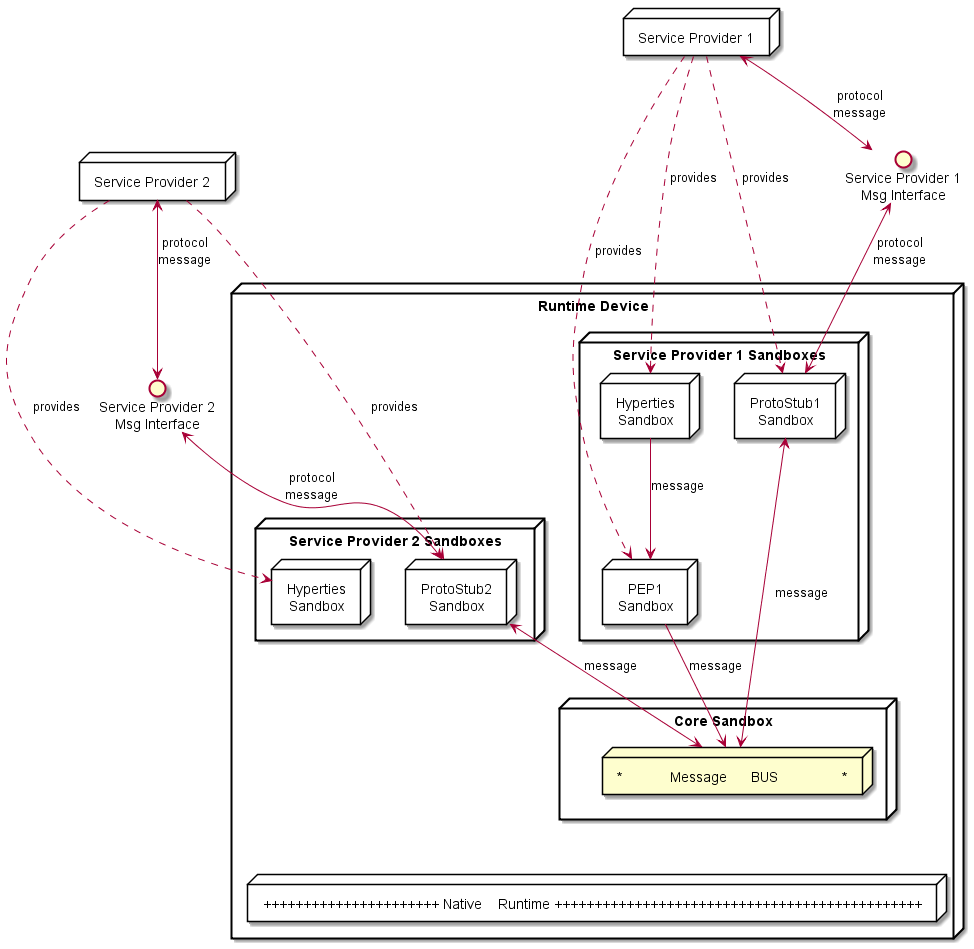


Figure 12: High Level Runtime Architecture with domain specific Policy Enforcer

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 Core Policy Engine.

In addition, Core Policy Engine 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 [15] 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 Core Policy Engine.

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 12. The Reporter-Observer pattern is supported by the exchange of messages between Reporter Syncher and Observer Syncher as defined in the reTHINK Message Model [15].

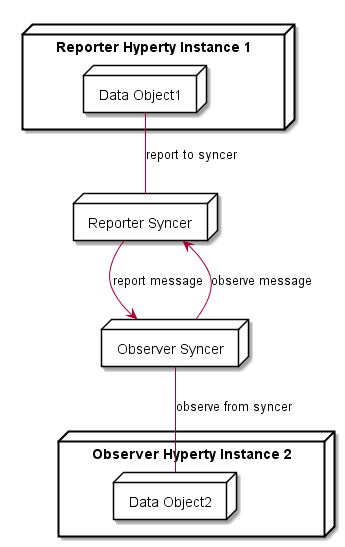


Figure 13: Reporter-Observer Communication Pattern

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

Hyperty communication through data object synchronisation is provided by the Syncher component running in the Hyperty Sandbox. Data object synchronisation should take advantage of the emerging [JavaScript Object.observer API](http://www.html5rocks.com/en/tutorials/es7/observe/) [106].

##### Policy Enforcer

Policy Enforcer complements the Core Policy Engine functionality enabling the enforcement of proprietary or closed Policies in the Hyperty 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 the Service Provider's 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 Figure 14.

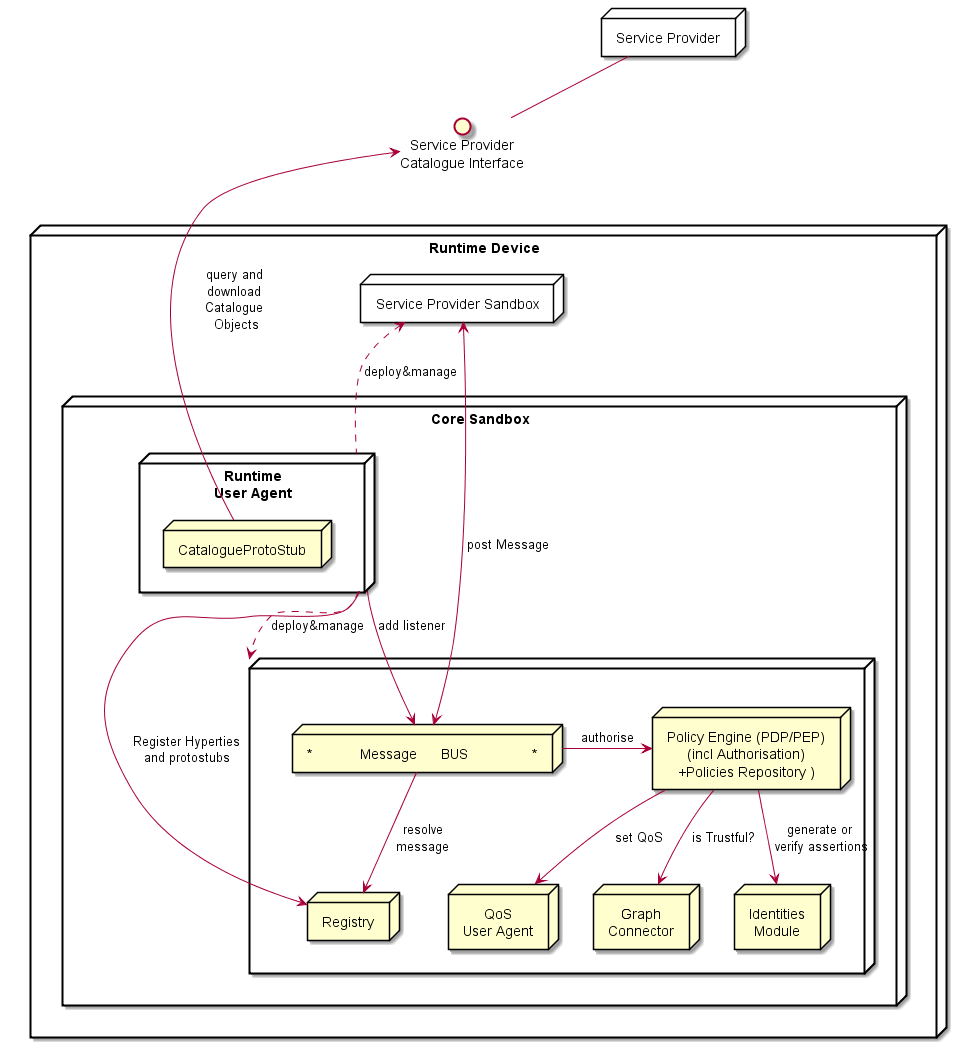


Figure 14: Runtime Core Architecture

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

The 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 previously 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.

##### Core Policy Engine

The Core Policy Engine provides Policy decision and Policy Enforcement 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.

The verification or generation of identity assertions, to get valid Access tokens, are two examples of actions ruled by policies.

##### 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 trustfully 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) [107] 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)[15].

##### 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.

##### Graph Connector

The Graph Connector is a local address book maintaining a list of trustful communication users. This functionality is further detailed in deliverable D4.1 [109].

#### 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.

### 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 D2.1 and originally described in D1.1.

The presented data flows, use the Hyperty Runtime APIs, 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 deployment and operation of Hyperties in the Hyperty 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
* Generic procedure to setup a Reporter-Observer data object synchronisation communication.
* 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 flow to support the deployment of the Hyperty Core Runtime is depicted in the diagram below.

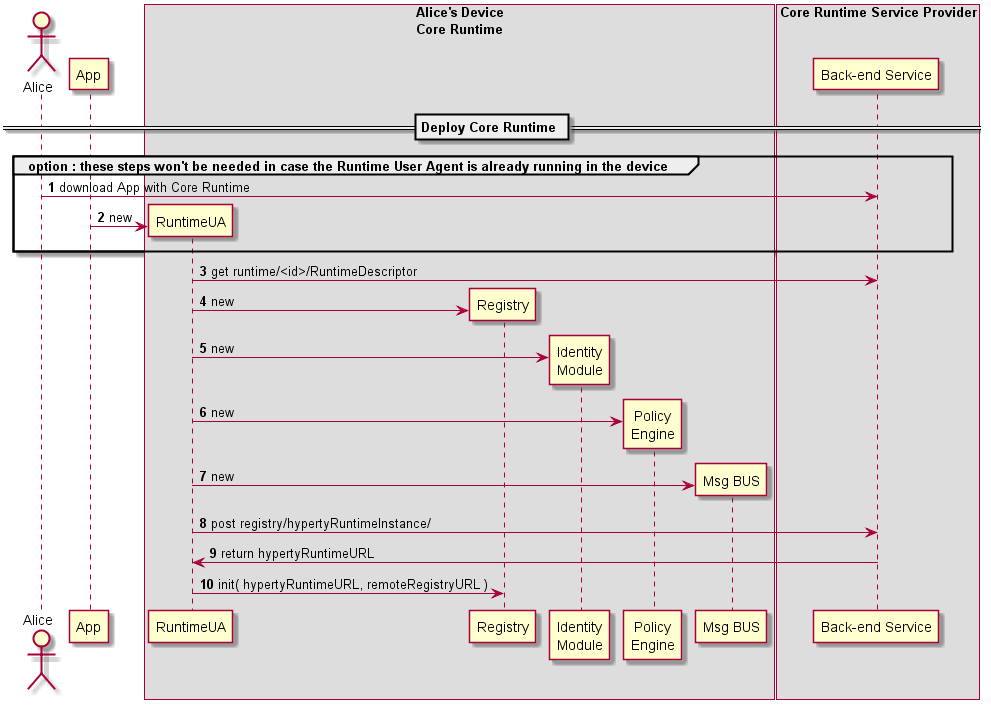


Figure 21: Deploy Core Runtime Components in the Native Runtime

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 [110] 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 Policy Engine and the Message BUS.

Steps 9 - 10: 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.

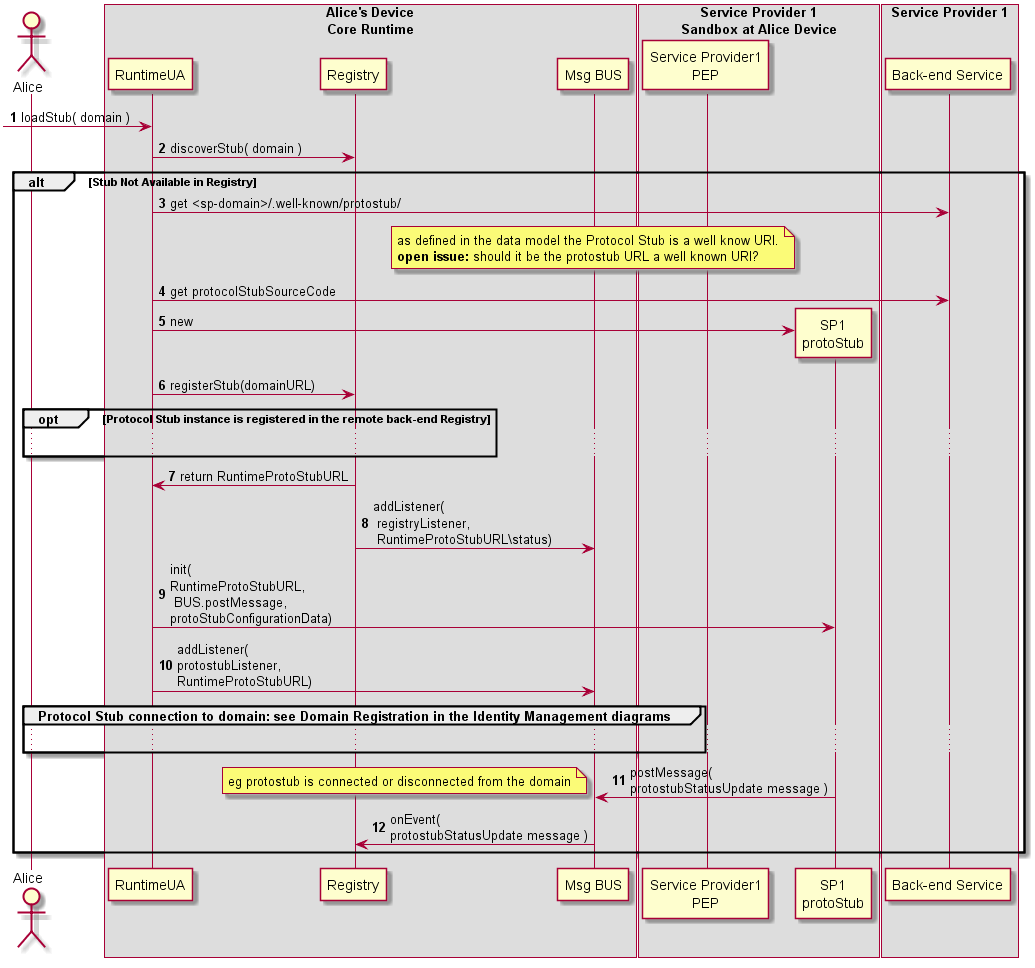


Figure 22: Deploy Protocol Stub

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. Such trigger may take advantage of some existing libraries like require.js [110]. The Runtime UA only downloads and deploys requested Protocol Stub after checking in the Registry that there is no Protocol Stub available in the Hyperty 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 returns 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 initializes the new Protocol Stub with configuration data contained in its descriptor. Depending on the sandbox implementation, the initialization 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.

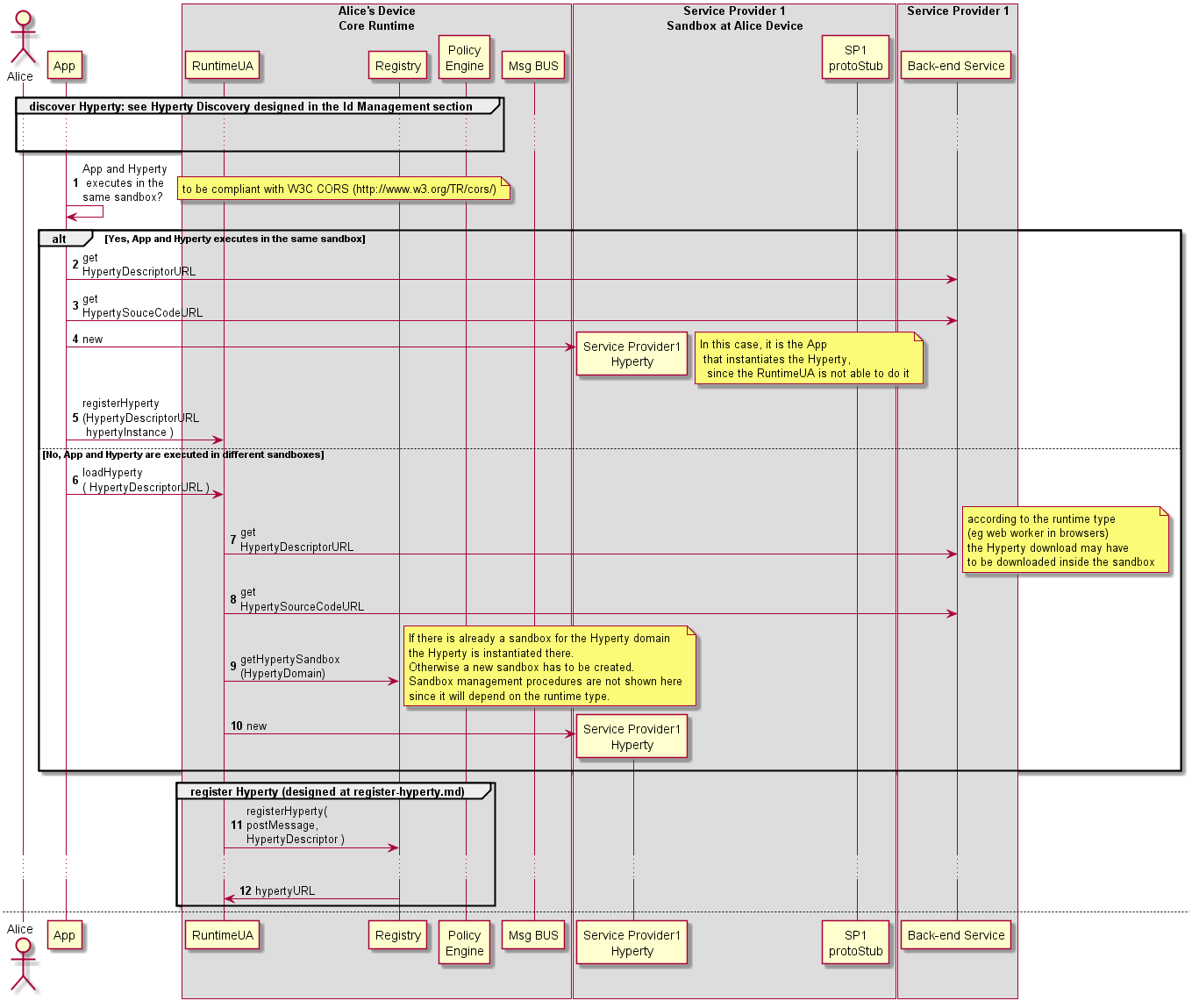


Figure 23: Deploy Hyperty (part1)

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 instantiated 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](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\d31\register-hyperty.md) by the Runtime Registry. See section 4.3.1.4 for more details.

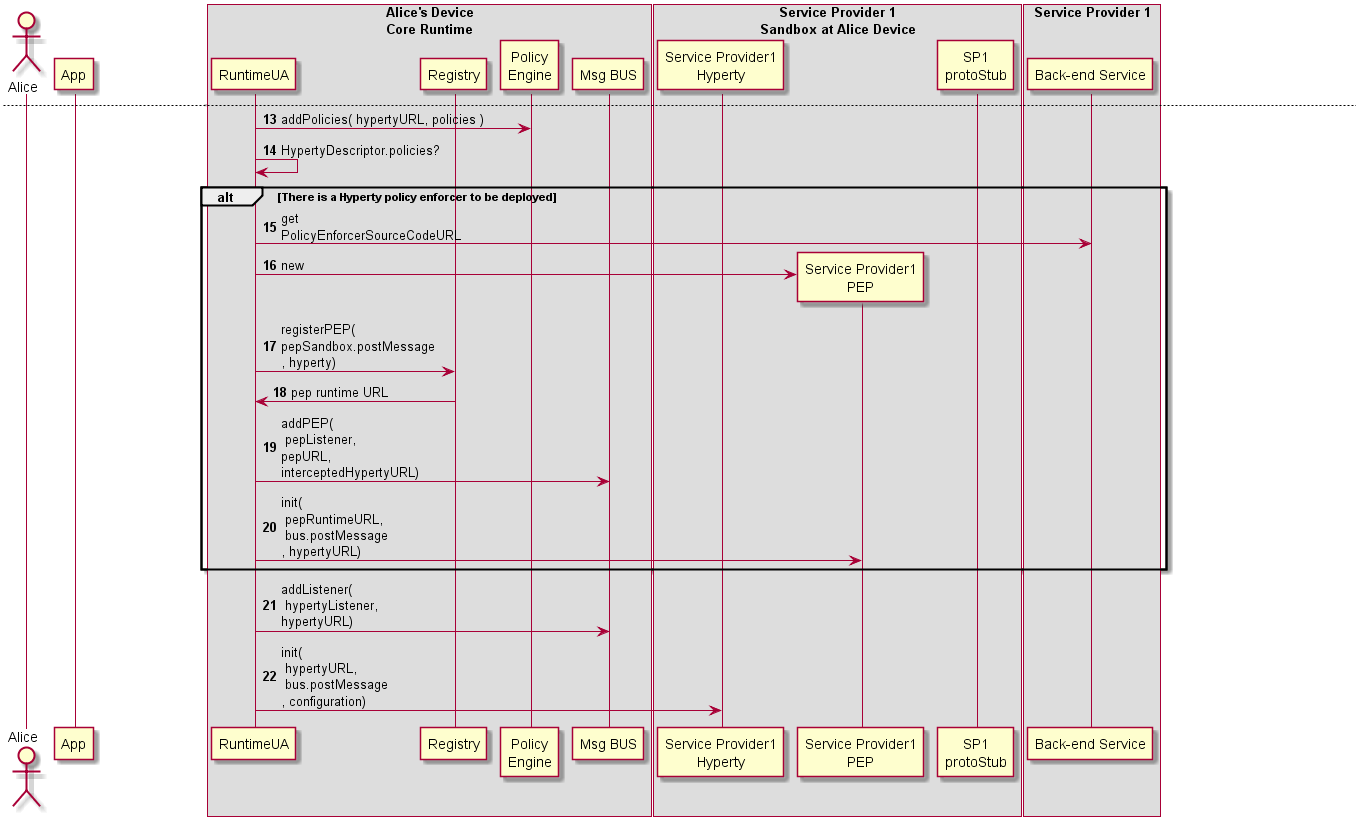


Figure 24: Deploy Hyperty (part2)

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 an 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 an 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.

##### Register Hyperty

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

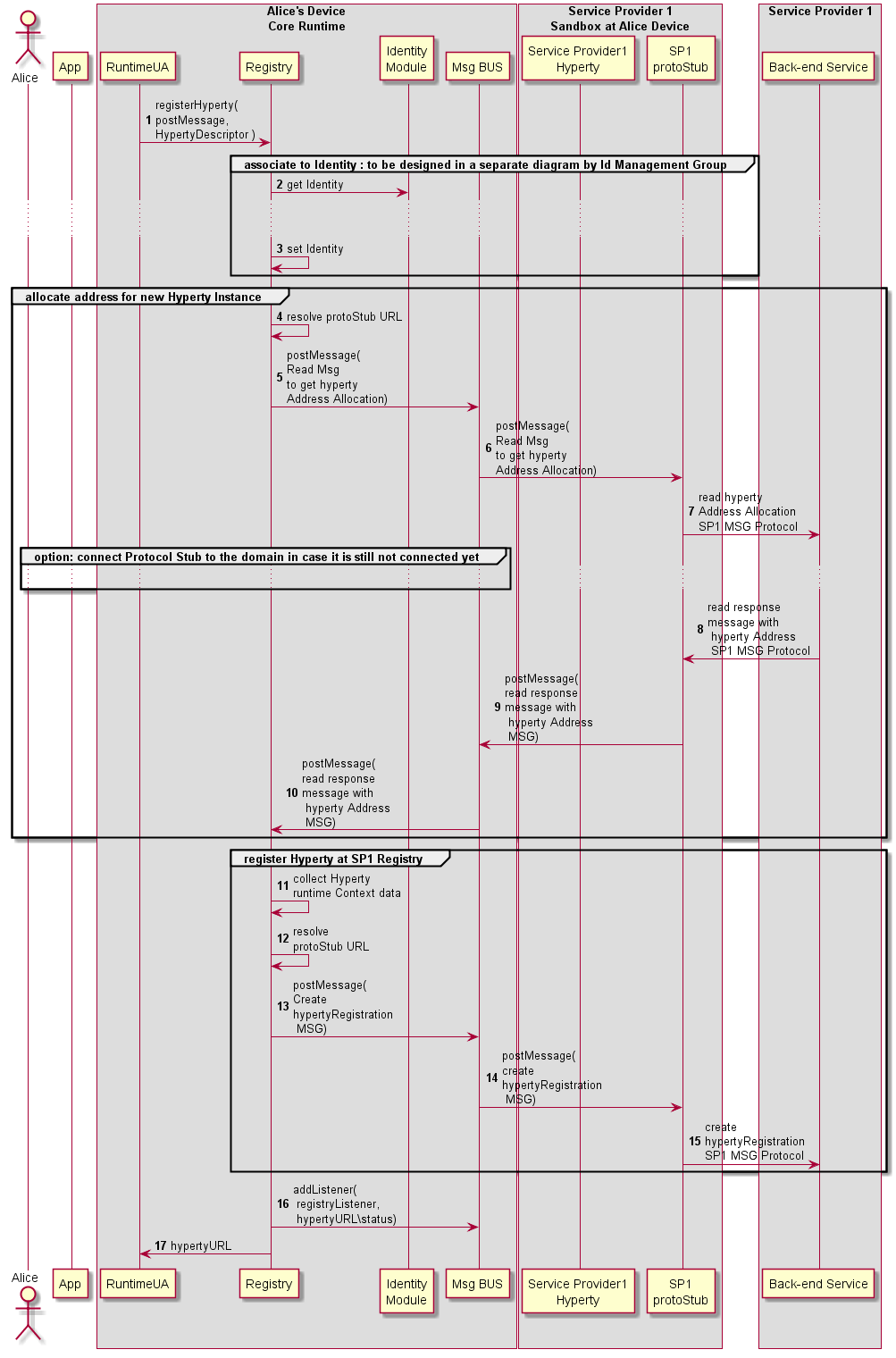


Figure 25: Register Hyperty

Step 1: the Hyperty registration is requested by the Runtime UA triggered by the [Hyperty Deployment process](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\d31\deploy-hyperty.md) (section 4.3.1.3).

Steps 2 and 3: The Hyperty is associated to a certain [identity](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\identity-management\user-to-hyperty-binding.md) (section 4.3.2.3)

Steps 4 - 10: an Hyperty URL address is allocated in cooperation with the Msg Node Address Allocation functionality.

**Message to request address allocated for new Hyperty Instance**

|  |
| --- |
| "id" : "1" "type" : "CREATE", "from" : "hyperty-runtime://sp1/runalice/registry", "to" : "sp1/msg-node/address-allocation", "body" : { "hypertyUrl" : "hyperty://sp1/hy123" } |

**Response Message returning the requested Hyperty Instance address**

|  |
| --- |
| "id" : "1" "type" : "RESPONSE", "from" : "sp1/msg-node/address-allocation", "to" : "hyperty-runtime://sp1/runalice/registry", "body" : { "hypertyInstanceURL" : "hyperty-instance://sp1/alice/hy123" } |

Steps 11 - 15: the Hyperty instance is registered in the back-end Registry

**Message to Register new Hyperty Instance**

|  |
| --- |
| "id" : "1" "type" : "CREATE", "from" : "hyperty-runtime://sp1/runalice", "to" : "sp1/registry", "body" : { "hypertyURL" : "hyperty://sp1/hy123", "hypertyInstanceURL" : "hyperty-instance://sp1/hy123, "hypertyRuntimeURL" : "hyperty-runtime://sp1/runalice, ...} |

Steps 16 - 17: The runtime Registry adds its listener to be notified about Hyperty instance status and returns the Hyperty URL to the runtime UA

##### Message Routing in Message BUS

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

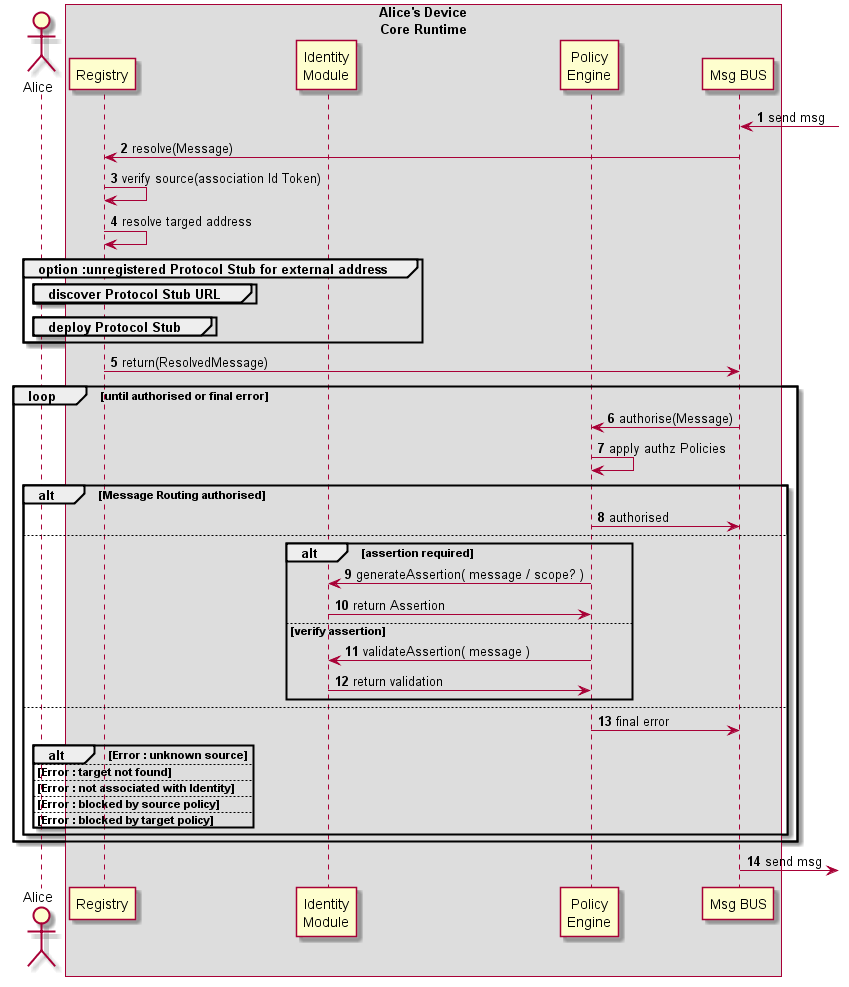


Figure 26: Message Routing in Message BUS

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](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\d31\deploy-protostub.md) (section 4.3.1.2) is triggered, in case it is not available yet.

Steps 6 - 7: in case the message requires authorisation, the Core PDP applies applicable policies to authorise its routing.

Steps 8 - 12: The Core Policy Enforcer enforces authorisation policies (including generation of Assertions or verification of assertions) in case the Runtime PDP requests it. In case policy enforcement is performed successfully, routing authorisation is requested again (step 6).

Step 13: 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.

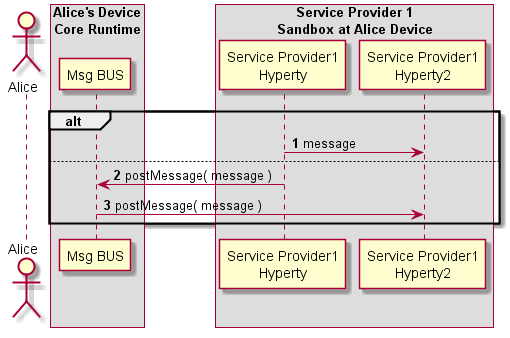


Figure 27: Intra-domain Local Communication

##### Intra-domain Remote Communication

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

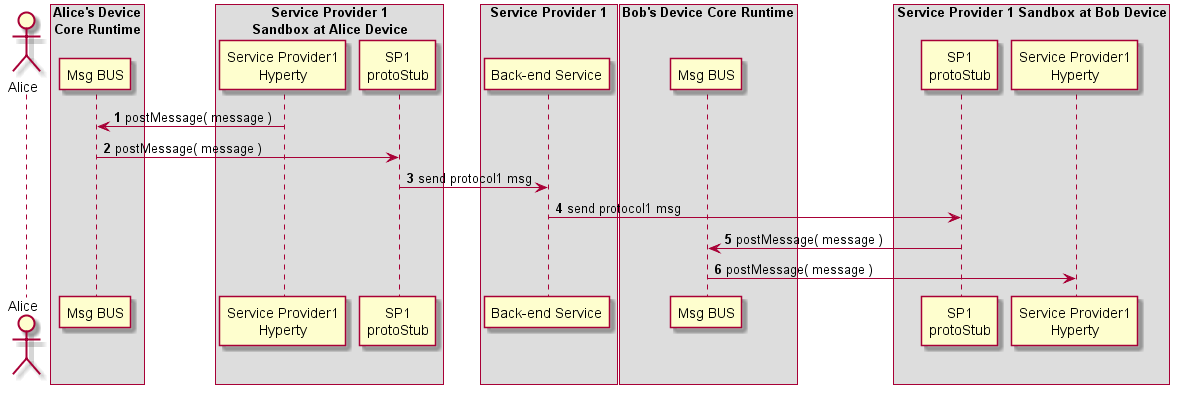


Figure 28: Intra-domain Remote Communication

##### 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.

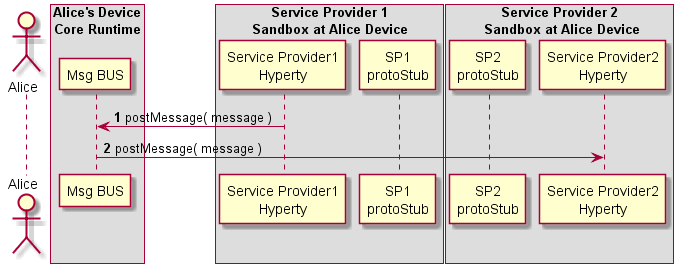


Figure 29: Inter-domain Local Communication

##### Inter-domain Remote Communication

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

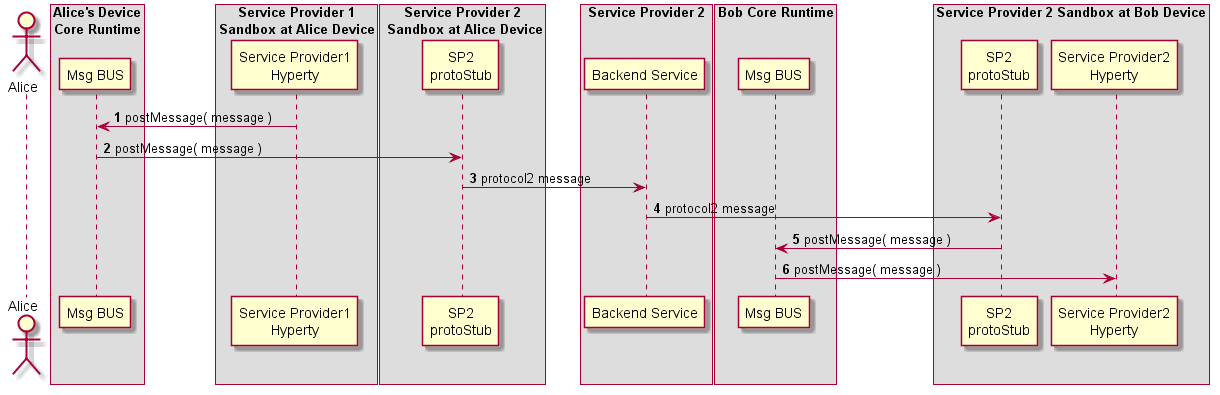


Figure 30: Inter-domain Remote Communication

##### Setup of Data Object Synchronisation with Reporter-Observer communication pattern

This MSC diagrams shows the most relevant steps to support the setup of data object synchronisation.

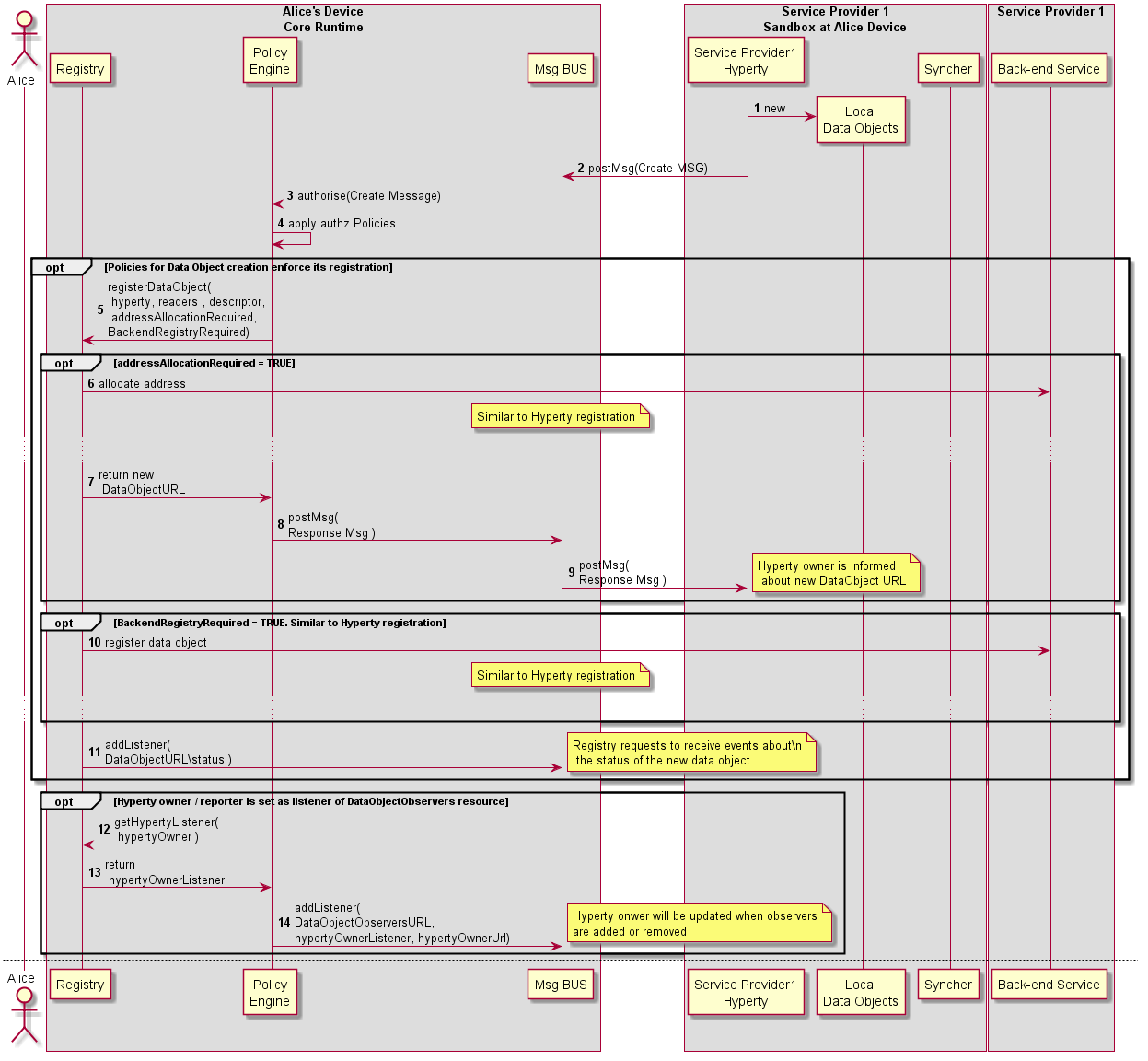


Figure 31: Request to create a Sync Data Object

Steps 1-2: The Data Object reporter post a Create Message to initiate the setup of the Data Object synchronisation.

[**Create Message sent by Reporter**](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 > , "schema" : "hyperty-catalogue://sp1/dataObjectSchema/schema123" } |

Steps 3-4: The Core Police Engine applies policies to check whether Alice has permissions to create the data object.

Step 5: optionally, and according to applicable policies, the new data object is registered in the Registry. In Step 11, the Registry adds a listener in the MessageBUS to be notified about status change events of the new Data Object.

Steps 6-9: optionally, and again, according to applicable policies, a new address might have to be allocated to the Data Object by the Messaging Node address allocation functionalities, to ensure the new data object is globally reachable. The new address allocated to the Data Object is informed back to the Reporter with a 3XX response message.

[**Response Message by Core PEP to inform Hyperty Owner about new allocated Data Object URL**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#createmessagebody)

***note:*** usually 3XX requires to send a new request message. In this case a new request message is not required.

|  |
| --- |
| "id" : "1" "type" : "RESPONSE", "from" : "hyperty-runtime://sp1/core/pep", "to" : "hyperty-instance://sp1/alicehy123", "contextId" : "qwertyuiopasdfghjkl", "body" : { "code" : "308", "value" : "{ "resource" : "comm://sp1-msg-node/alice/123456" } } |

Steps 10: optionally, and again, according to applicable policies, the new Data Object is also registered in the Back-end (Domain) Registry.

Steps 12 - 14: optionally, and again, according to applicable policies, the Reported Hyperty is added as a listener to be notified when new Observers to its Data Object are added or removed. The DataObjectObserversURL is handled by the Policy Engine.

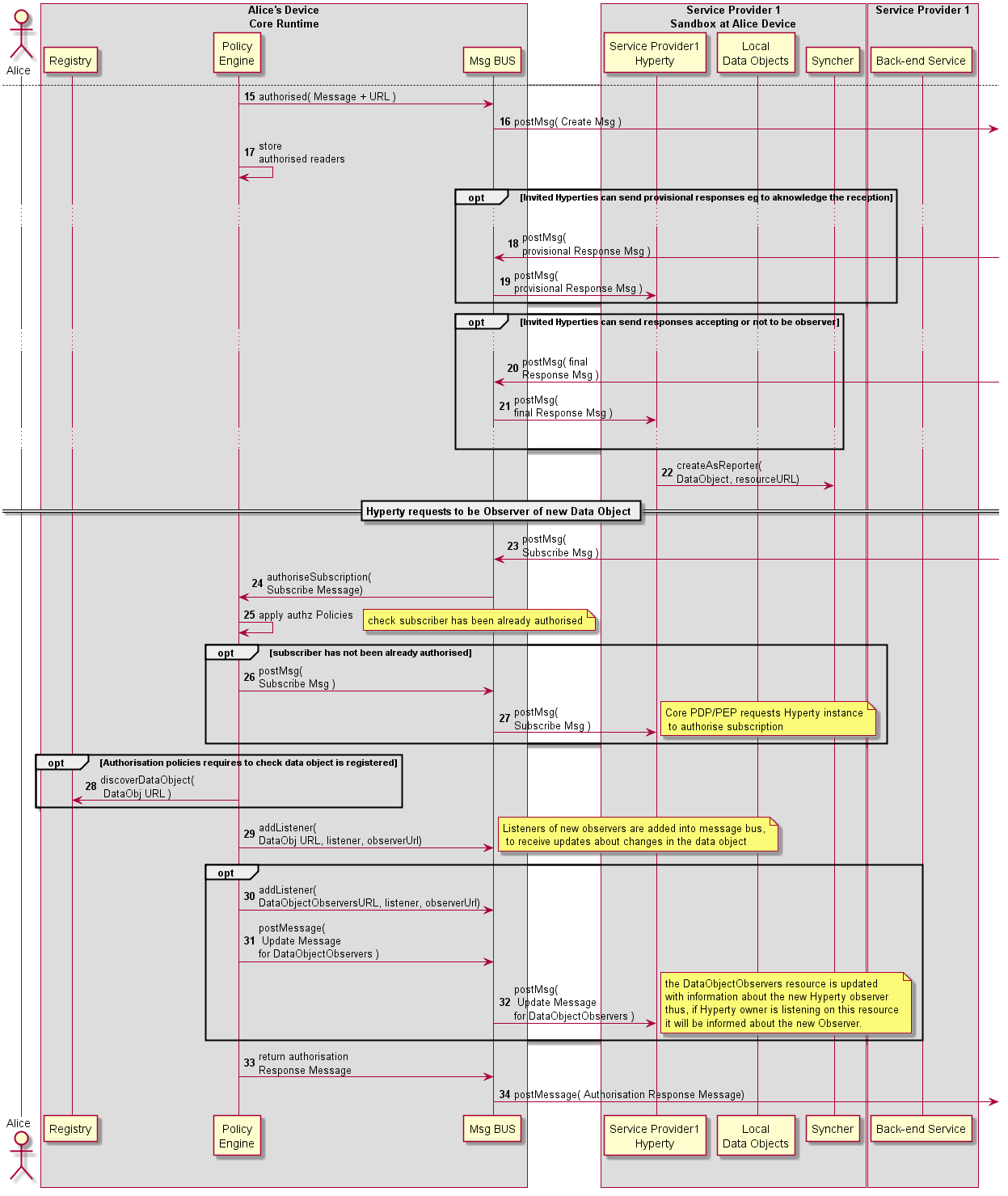


Figure 32: Data Object synchronisation is authorised and Observers added

Steps 15-17: In case the data object creation is authorised, the Message BUS forwards the data object creation message to invited Observers and the Core Policy Engine takes the Hyperty Instance URLs set in the to header field, to set as the list of addresses that are previously authorised to be Observer for the new data object.

Steps 19-21: optionally, invited Observers may respond with provisional responses e.g. to aknowledge the reception of the invitation and to accept or not the invitation to be an Observer.

[**Provisional Response Message sent by inviter Observer**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#responsemessagebody)

|  |
| --- |
| "id" : "1" "type" : "RESPONSE", "from" : "hyperty-instance://sp2/bobhy123", "to" : "hyperty-instance://sp1/alicehy123", "contextId" : "qwertyuiopasdfghjkl", "body" : { "code" : "1XX" } |

Step 22: as soon as the Reporter receives the information that the data object synchronisation was authorised, it can request the Syncher to start reporting data object changes by posting UPDATE messages to the Data Object URL resource. It should be noted that, according to Hyperty Service logic, this step can be performed later, e.g. after the Reporter is notified an Observer has been added.

[**Data Synchronisation UPDATE Message sent by Reporter Syncher**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#updatenmessagebody)

|  |
| --- |
| "id" : "2" "type" : "UDATE", "from" : "hyperty-instance://sp2/bobhy123", "to" : "comm://sp1-msg-node/alice/123456", "contextId" : "qwertyuiopasdfghjkl", "body" : { "value" : "changed value" } |

Steps 23-25: to be an Observer of a Data Object, a Subscription message is sent to the Runtime Core component managing subscription authorisation, in this case it is assumed it is the Policy Engine. The Policy Engine applies message to decide on the received subscription request namely if subscription requester has been previously authorised in step 17.

[**Subscription Message sent by inviter Observer**](https://github.com/reTHINK-project/architecture/tree/master/docs/datamodel/message#subscriptionmessagebody)

|  |
| --- |
| "id" : "1" "type" : "SUBSCRIPTION", "from" : "hyperty-instance://sp2/bobhy123", "to" : "hyperty-runtime://sp1/core/pep", "contextId" : "qwertyuiopasdfghjkl", "body" : { "listener" : "bobhy123.postMessage" } |

Steps 26-27: in case subscription requester has not been previously authorised, and according to applicable policies, the subscription request message can be forwarded to the Reporter Hyperty that will take the final decision.

Step 28: according to applicable policies, the Registry can be queried to check if the data object was previously registered.

Step 29: in case the subscription request is authorised, a listener of the new observer, contained in the SUBSCRIPTION message body, is added into the Message BUS to receive messages on the Data Object resource URL.

Step 30: optionally, the new Observer listener is added in the Message BUS to be notified about new Observers.

Steps 31-32: optionally, it is posted an UPDATE message into the DataObjectObserversURL with information about the new added Observer.

Steps 33-34: the subscription requester is informed about the subscription authorisation with a RESPONSE message.

#### 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 Identity Registration

This section, describes the main procedures for the registration of a new Identity in the Hyperty Runtime. It is assumed that an account was already created by the user on the IdP through an out of scope mechanism.

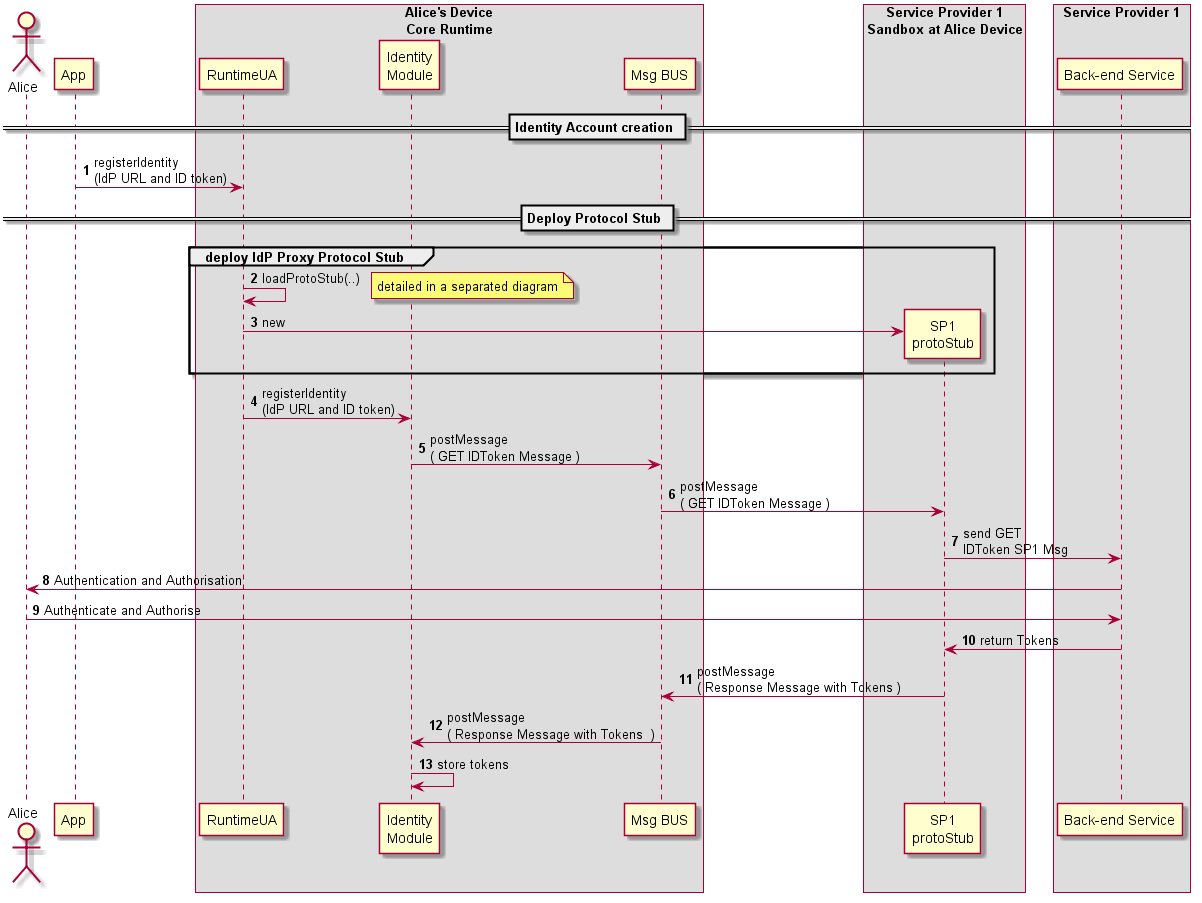


Figure 33: User registration

Steps 1: the App request the RuntimeUA to register the new Identity, providing the IdP URL and the IdP user identifier.

Steps 2-3: The RuntimeUA [deploys the IdP Proxy protocol stub](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\basics\deploy-protostub.md)(see section 4.3.3.2) required to support the connection with back-end IdP server.

Steps 4-7: the RuntimeUA requests the IdModule to register a new Identity. The IdModule requests the Service Provider back-end IdM to authorise the new Identity creation by sending a message through IdP Proxy Protocol Stub.

Steps 8 - 9: optionally, the back-end IdM requests the user to authenticate and authorise the new identity set in the Runtime via a separated channel (e.g. SMS)

Steps 10 - 13: assuming the identity set in the runtime is successfully authorised, the IdM back-end service returns a set of tokens, which are stored by the IdModule.

##### Domain Login

This section describes the main procedures to support domain login.

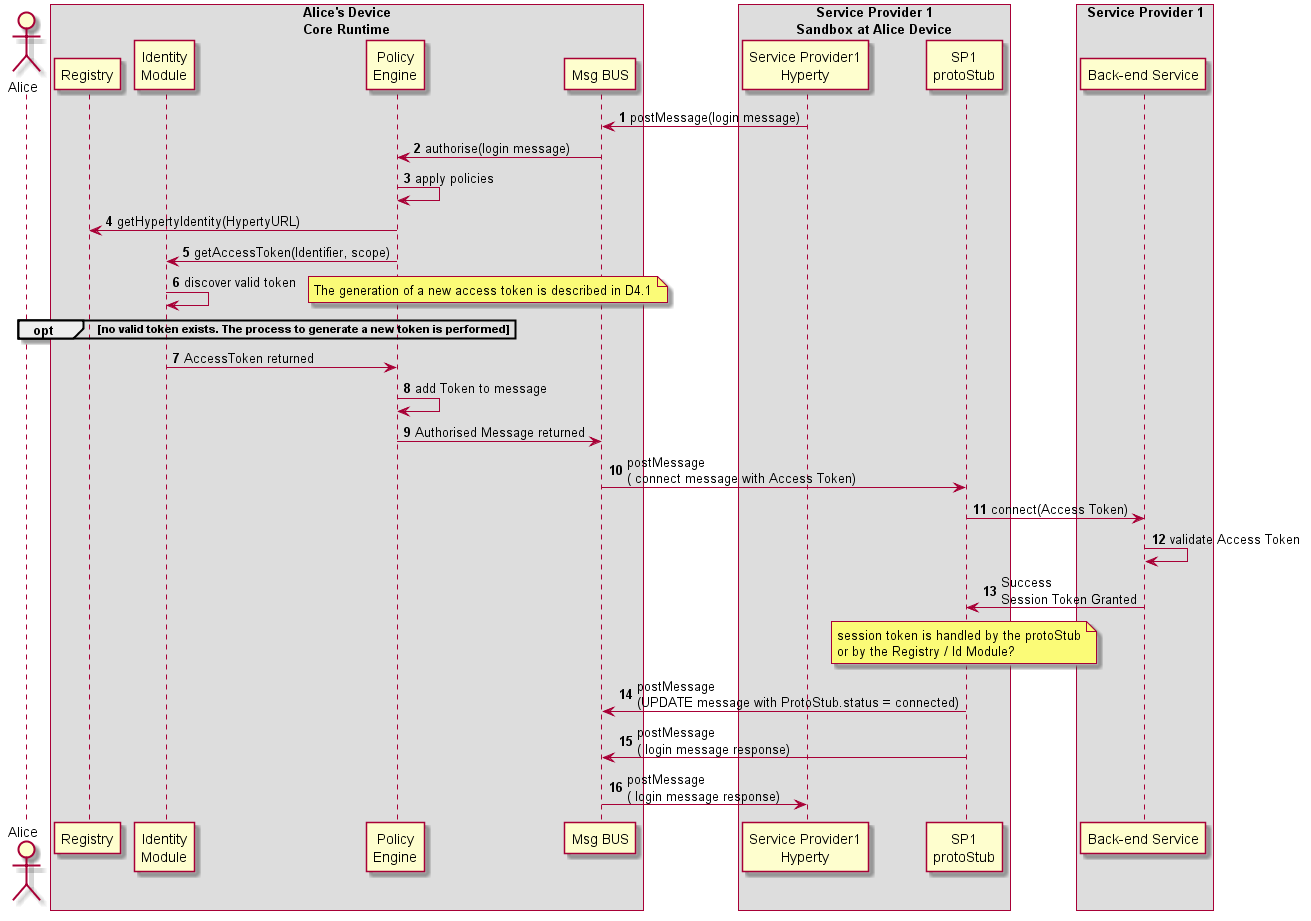


Figure 34: Explict Domain Login

A first option is the Hyperty to explicitly ask to connect (see Figure above):

Steps 1-3: Hyperty requests to connect to domain with a GET message sent to DomainURL which is subject for Authorisation by the Core Policy Engine.

Steps 4-5: according to applicable policies the Policy Engine request the Identity Module for an Access Token to be used in the login message, providing the Identity identifier associated to the hyperty and the scope (login to domain).

Steps 6-7: Identity Module returns a valid Access Token to be used in the domain login. To be noted that this may imply the generation of a new token in case there is no valid token stored in the Identity Module. In this case, the Identity Module may have to interact with an IdP back-end server through an IdP (proxy) Protocol Stub. The Access Token generation is described in D4.1.

Steps 8-10: the returned token is added to the login message by the Policy Engine, which is forwarded to the Protocol Stub by the Message BUS.

Steps 11-13: the Protocol Stub uses the Access Token to request to connect to the domain back-end server. If successful a Session Token is granted and returned back to the Protocol Stub. (it is assumed the session token is handled by the Protocol Stub and not by Core Runtime)

Steps 14-16: as soon as the Protocol Stub is connected, its status is updated (UPDATE message posted to its status URL resource) and Response message is sent back to the Hyperty.

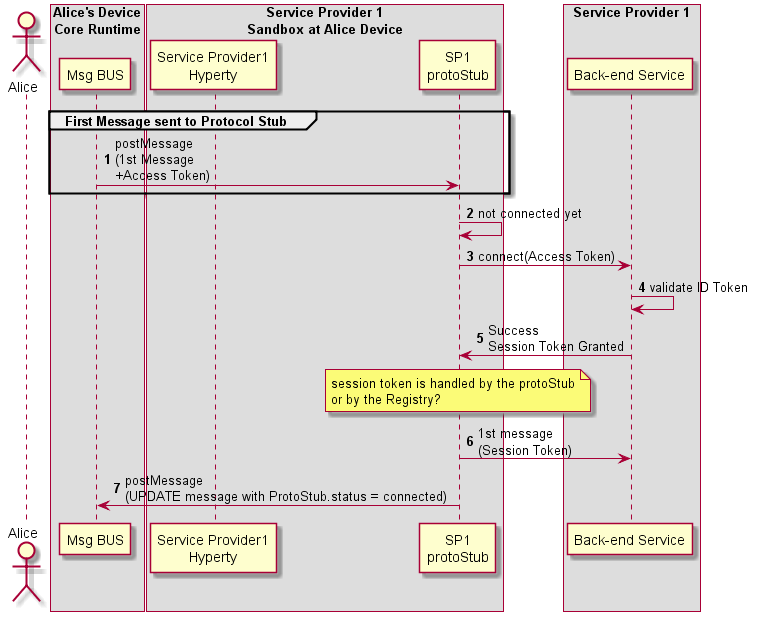


Figure 35: Implict Domain Login

In a second option (see Figure above), the ProtoStub only connects when requested to send the first message. The Access Token used in the connection request is provided like it is in the first option.

##### Associate User Identity to Hyperty Instance



Figure 36: 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 an 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 an identity Token has already exists for the requested user. If it does not, a [Domain Login](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\d31\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.

This question has to be further investigated.

#### 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 descriptions 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 Use Case, six procedures are performed:

1. Alice invites Bob
2. Bob receives Invitation from Alice
3. Alice is acknowledged 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.

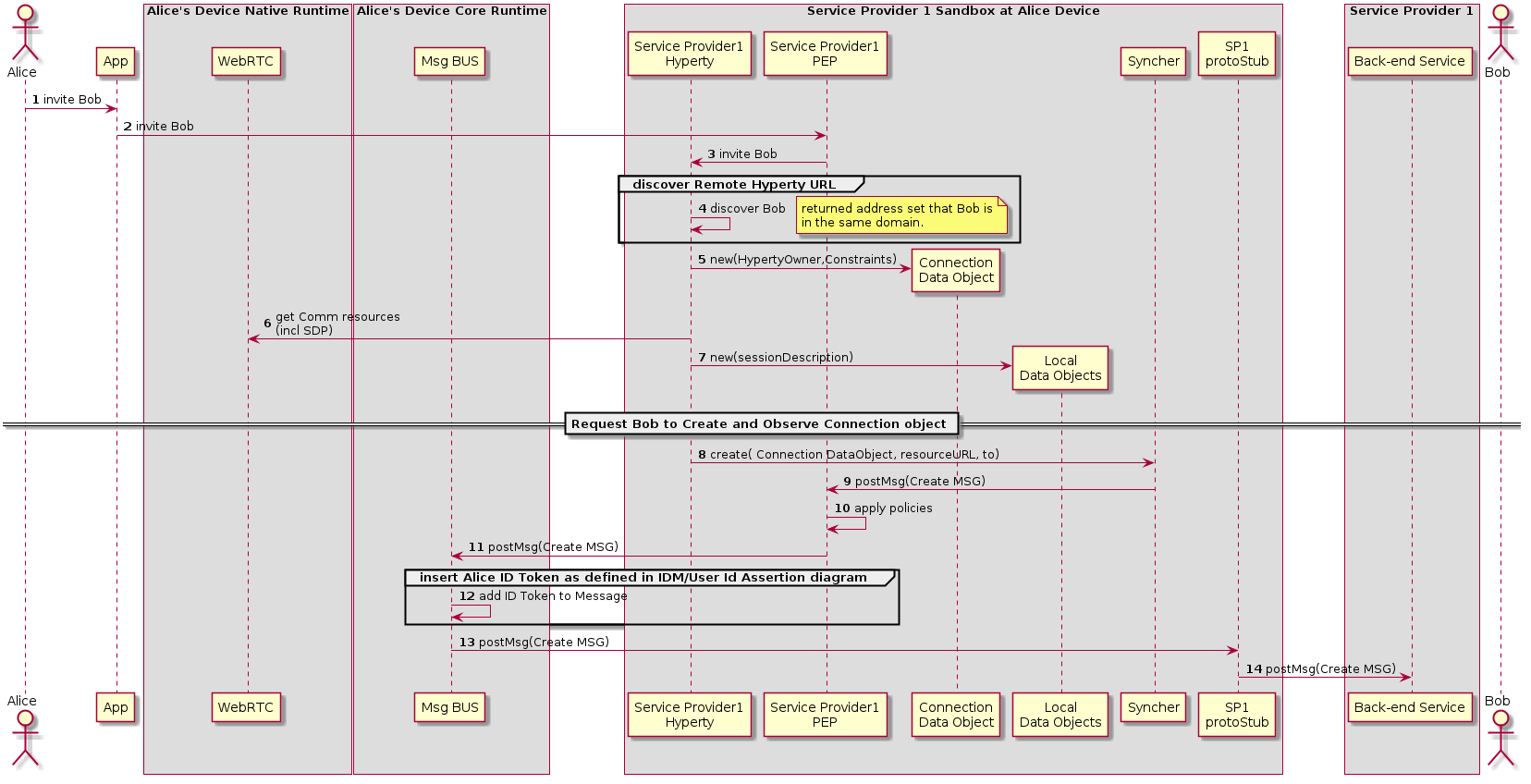


Figure 37: Alice invites Bob for a communication

(Steps 1 - 4): Alice decides to invite Bob for a communication.

(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 Messaging Service.

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

This MSC diagrams shows how Bob receives invitation from Alice.

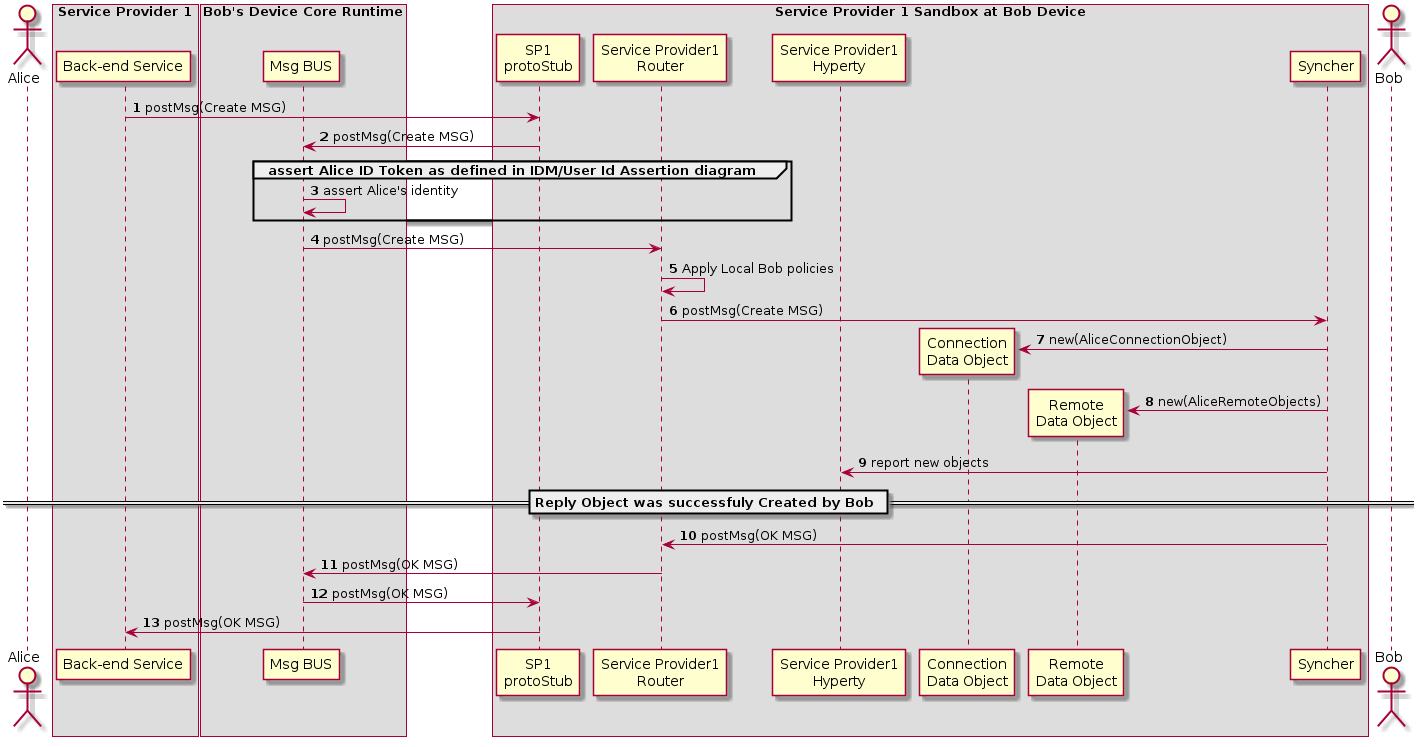


Figure 38: Bob receives invitation

(Steps 1 - 4): Service Provider Back-end Messaging 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 acknowledged that Bob received the invitation

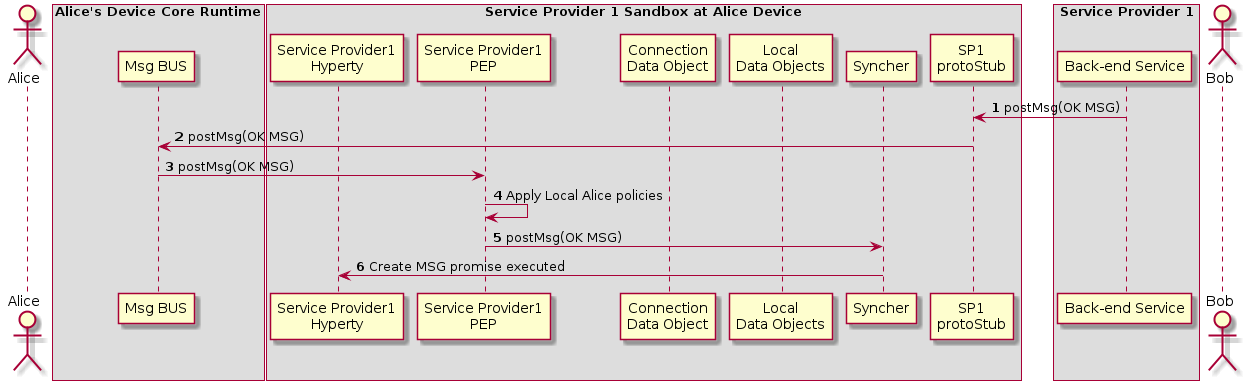


Figure 39: Acknowledged that Bob received the invitation

(Step 1 - 3) : Service Provider Back-end Messaging 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 40: 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 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

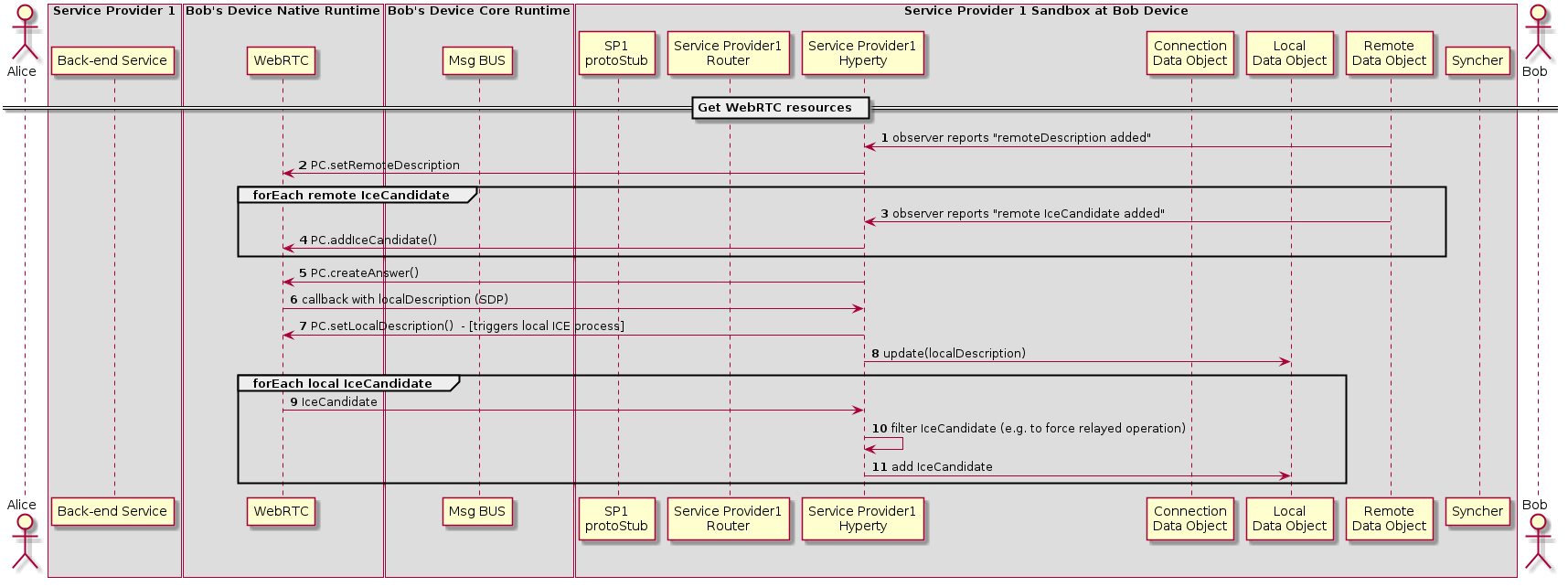


Figure 41: Bob gathers WebRTC resources

(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

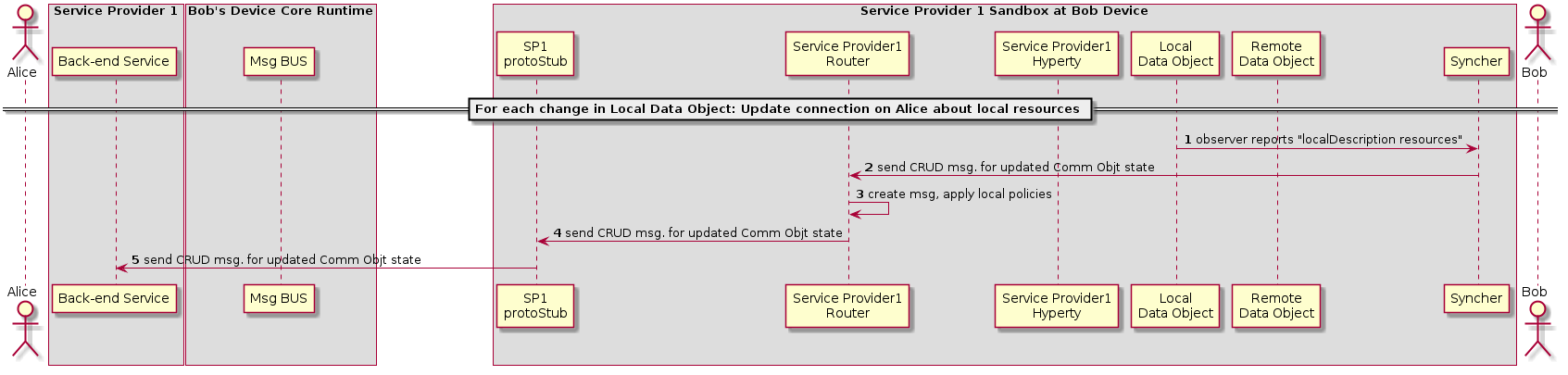


Figure 42: Synchronization of Alice's Data object

(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.

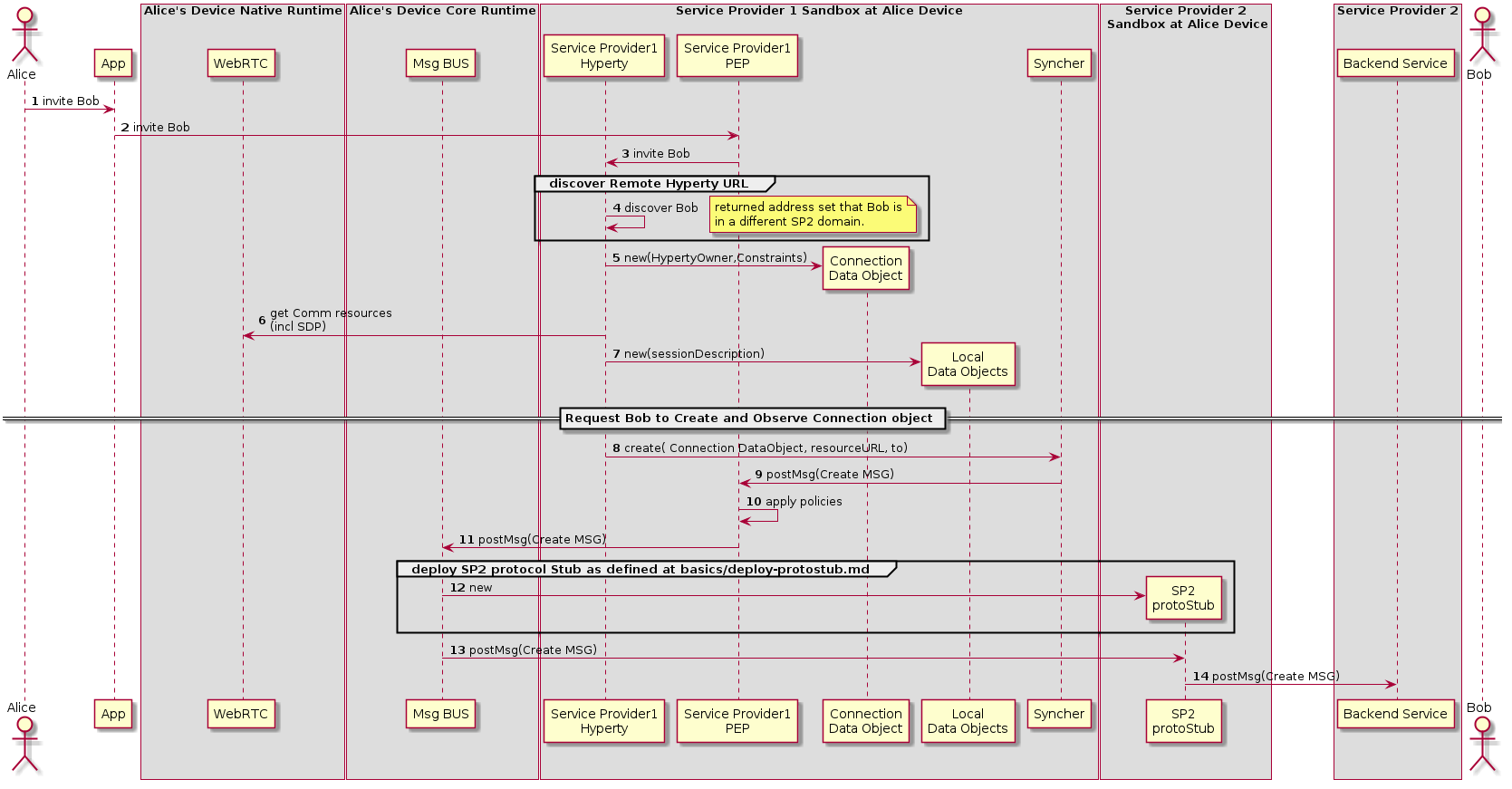


Figure 43: H2H Inter domain Communication : create communication

(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.

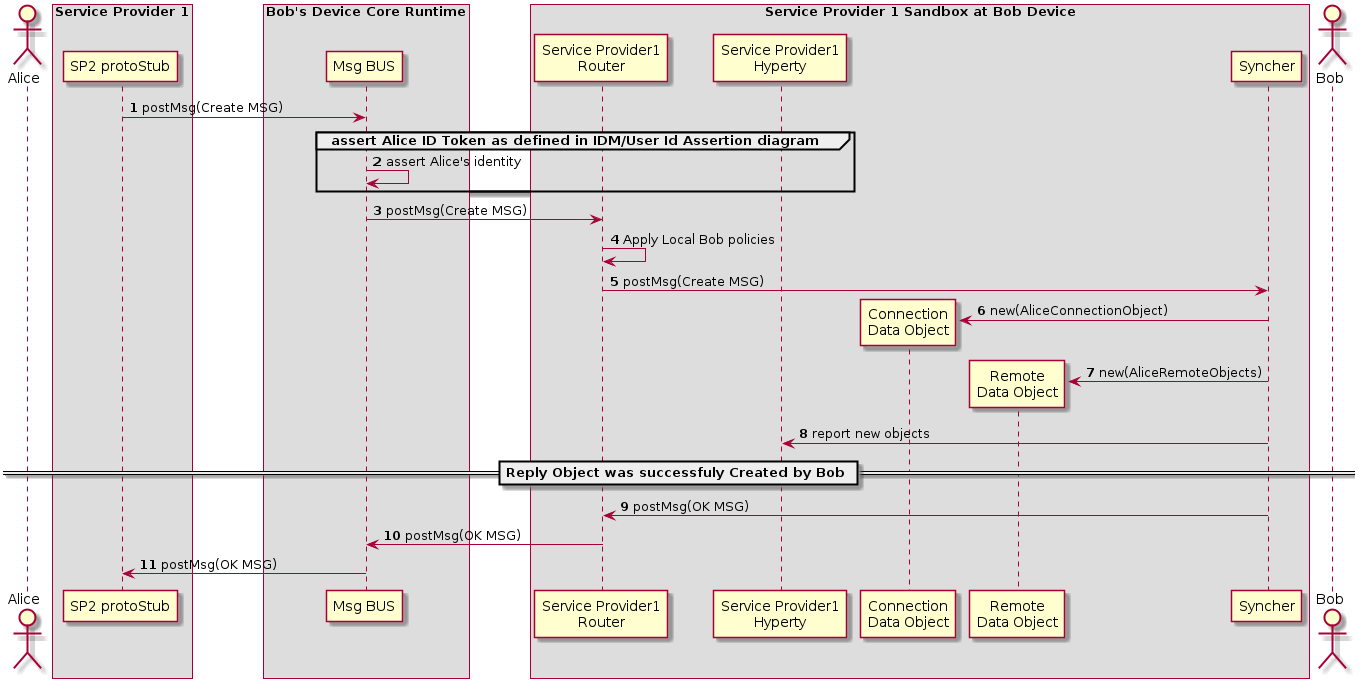


Figure 44: H2H Interdomain Communication: bob receives invitation

(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 acknowledged that Bob received the invitation

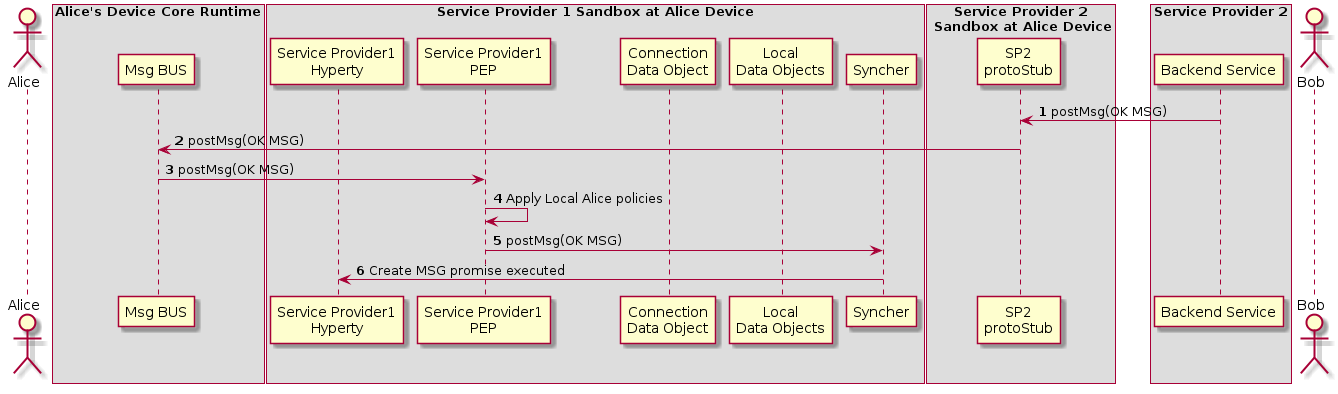


Figure 45: H2H Interdomain Communication : Alice is Acknowledged

(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.



Figure 46: 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.

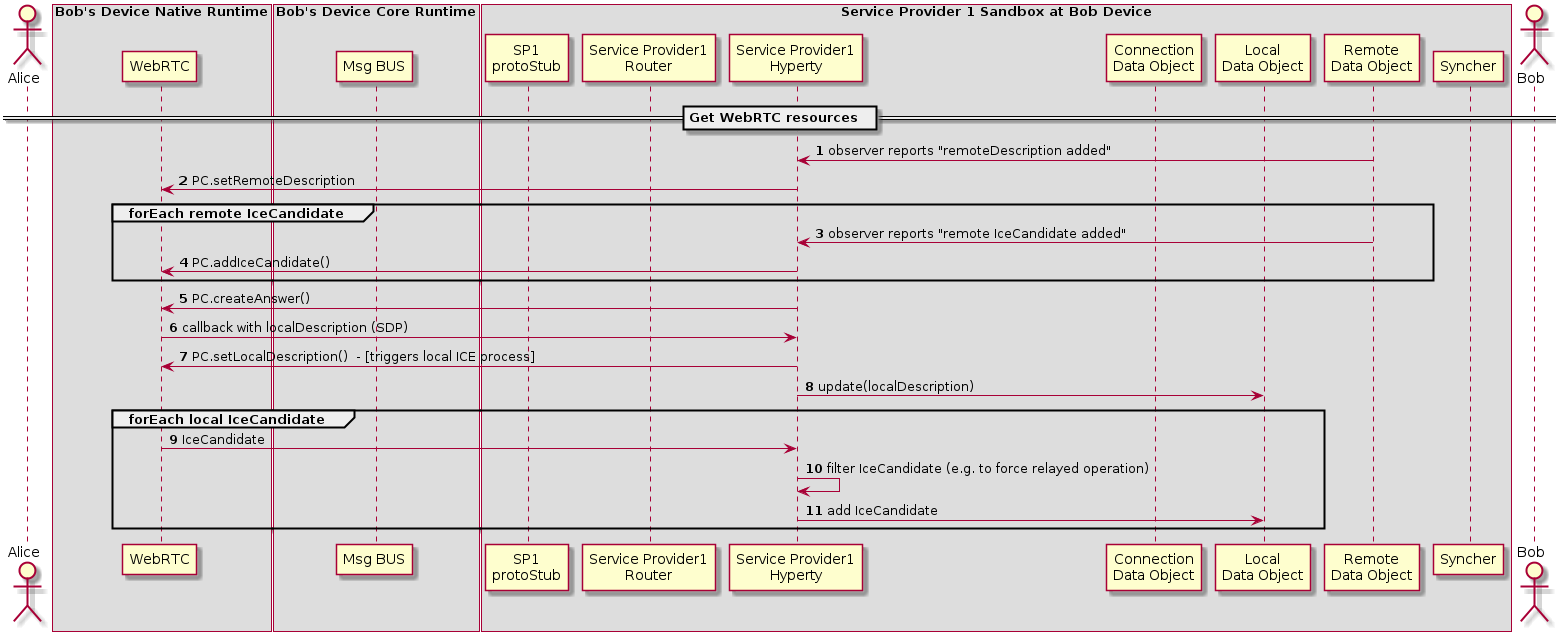


Figure 47: H2H Interdomain Communication: Bob gathers WebRTC resources

(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

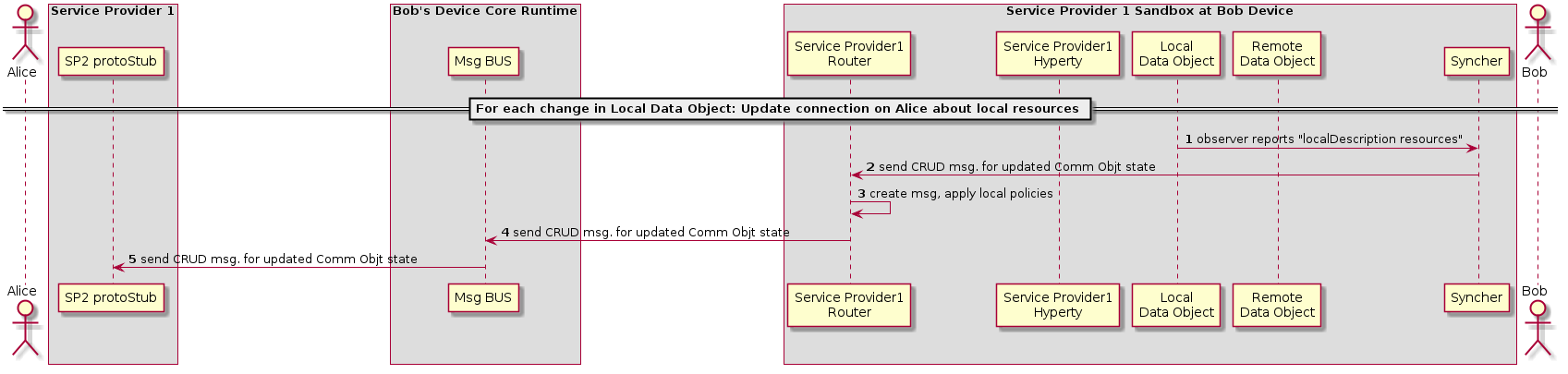


Figure 48: H2H Interdomain Communication: Synchronization of Alice's Data object

(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 decisions 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.

##### Proposed implementation design.

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

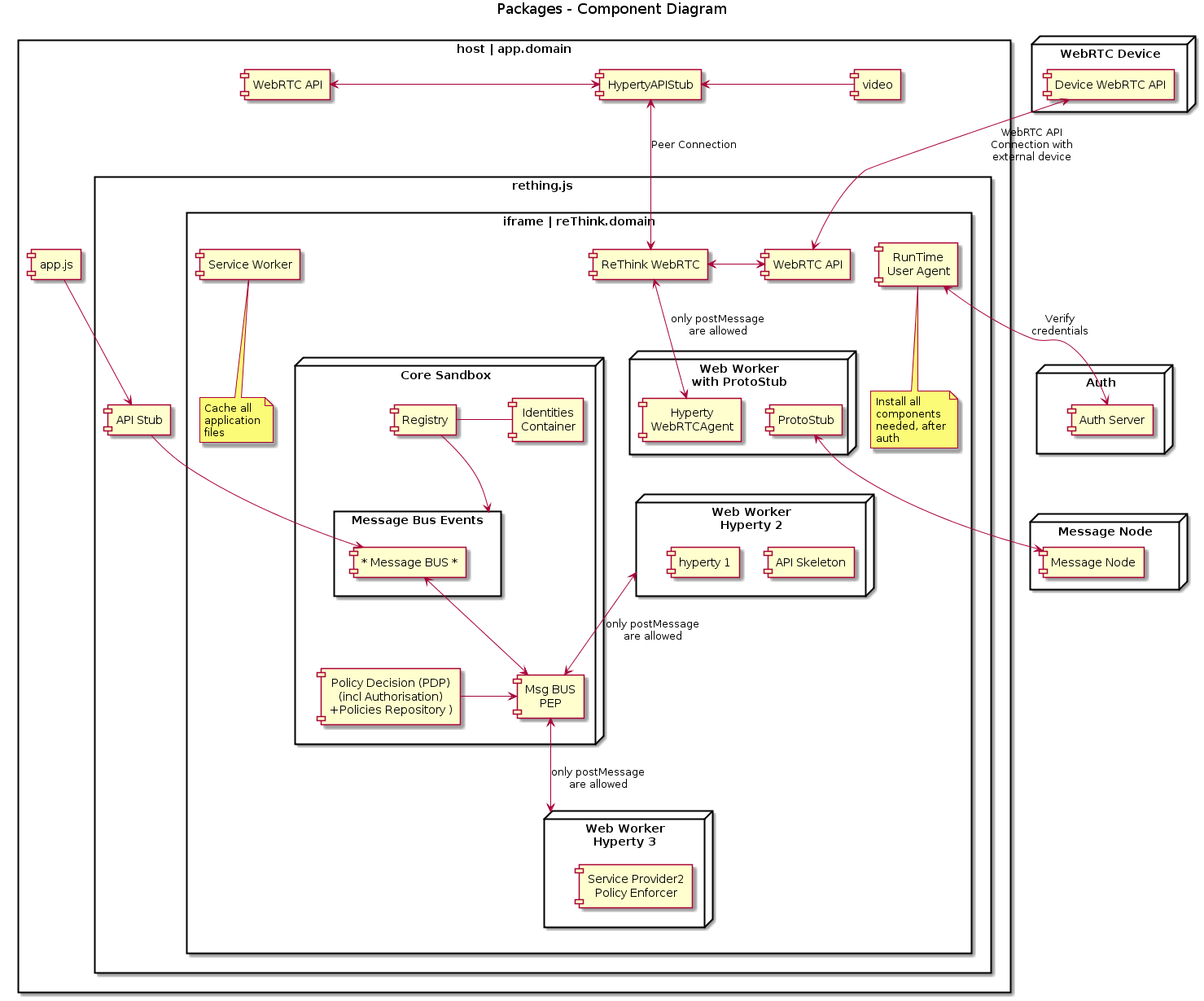


Figure 49: Runtime browser implementation

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 the CSP which provides the 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 section 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 the execution of 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 an 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 independently 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 messages coming from the different elements and sending them to the right destination 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 intended 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 objects 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 <video> and <audio> 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 [111]
* cordova / phonegap / ionic [112]

##### 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 and 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 50: 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 functional schema



Figure 51: Cordova functional 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

Some Cordova plugin examples are:

* **iosRTC** [113]: Iosrtc is a wrapper around Google’s WebRTC library and simply provides PeerConnection, getMediaDevices and getUserMedia APIs , without any limitations or artificial constraints.
* [**phoneRTC**](https://github.com/alongubkin/phonertc) [114]

###### 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 successfully used by companies from reTHINK consortium to develop WebRTC hybrid applications. Thus, 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 separately 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 incompatibility.

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

##### OpenWebRTC

OpenWebRTC [19] is also an option. See Annex A for OpenWebRTC evaluation.

##### Selected solutions for the implementation

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

Android

Crosswalk: integrate chromium in the application with different possible integration:

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

iOS :

iOSRTC, cordova plugin [113] - Usage of Cordova will enables us to reuse the components that will be developed on the browser runtime.

Android & iOS :

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

Hybrid solution will be selected for the project as it enables us 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 interesting as it should enable the possibility to build complete native and hybrid application.

#### Runtime implementation M2M standalone application

Node.js is considered one of the options for implementing the Runtime API for platforms like Raspberry PI and [Beagle Board](http://beagleboard.org/bone) [115]:

##### Node.js Installation

For [installing Node.js on Raspberry Pi](http://weworkweplay.com/play/raspberry-pi-Node.js/) [116], 2 steps are required:

download the debian package and then install it:

wget <http://node-arm.herokuapp.com/node_latest_armhf.deb>

sudo dpkg -i node\_latest\_armhf.deb

The [installation of Node.js on BeagleBoard](http://beagleboard.org/Support/BoneScript)[117] requires to compile it from scratch [118] or install it in a similar way as for Raspberry using one of the versions from the download page [119].

An important package based on Node.js is [Cylon](http://cylonjs.com/) [120] 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 Node.js 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. One possible Node,js solution is provided in [gf3](http://gf3.github.io/sandbox/) [121].

For the Runtime UA a module implementing the protocol LWM2M [is already available for Node.js](https://github.com/telefonicaid/lwm2m-node-lib) [122].

##### Code Snippets

For creating several sandboxes using gf3 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/) [123]. It currently supports Raspberry Pi2 and STM32F4-Discovery + BB as hardware platforms and Linux and [Nuttx](http://nuttx.org/) [124] 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 probably it won't be used in reTHINK prototype. Nevertheless, during the development phase, the iotJs will be considered and tests will be performed to validate the support of ioJs are envisioned.

## Messaging Framework Specification Update (Steffen)

This Chapter contains the functional design of the Messaging 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, Node.js 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 52: 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 Vert.x 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 this should 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 Vert.x 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, fewer 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 and 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. Vert.x 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, which 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, which 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 GUID, 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 expose 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 contains all the dependencies it needs to run on Vert.x.

##### 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) [125]. This vertx module provides multiple authentication mechanisms (OAuh, CAS, HTTP, OpenID, SAML2.0 and OpenIDConnect) for different IdPs.

### Node.js based Messaging Node Specification

For each [functional block](file:///C:\Projectos\reTHINK\WP3\git\core-framework\docs\deliverables\d31\msg-node-architecture.md) existing Node.js modules were identified, which can be either reused or extended.

#### Core Functionalities

This section attempts to match the functional blocks of the Messaging Node architecture to features and functional blocks of the Node.js and Redis architecture.

##### Message BUS

The message bus can be implemented with [Redis](http://redis.io) [63]. 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 Node.js

[Redis integrate a PUB/SUB mechanism](http://redis.io/topics/pubsub) [126]:

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 splited 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 Node.js and Redis can be managed by a Nodes.Js Redis [client module](https://github.com/NodeRedis/node_redis) [127]. Redis instance can be a single instance or a Redis cluster.

##### Access Control

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

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

[PassportJs](http://passportjs.org/) [105], which is an interesting middleware that could enable us to add third party authentication should be used. An authentication can also be done between Node.js and Redis.

##### Session Management

For a complete session management on Node.js, it will be interesting to use [ExpressJS](http://expressjs.com/) [128] which is a Web framework for Node.js.

##### Address Allocation Management

This component will have to be developed on a Node.js server

##### Protocol Stub & Connectors

Connectors will be Node.js processes to be developed. The protoStub/protoFly mechanism Goal can be used to facilitate the integration with other servers.

###### IdM Connector

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

The authentication against the Identity Provider has to be done at the beginning.

###### Registry Connector

The implementation of this Connector requires further study.

###### End-User Device Connector

Communication between Users and Node.js 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 implementation of this Connector requires further study.

###### Node Sandbox framework

[Node-sandbox](https://www.npmjs.com/package/node-sandbox) module [129] 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).

#### Node.js implementation architecture

**Architecture : Node.js and Redis :**

Here is the description of the architecture with Redis :

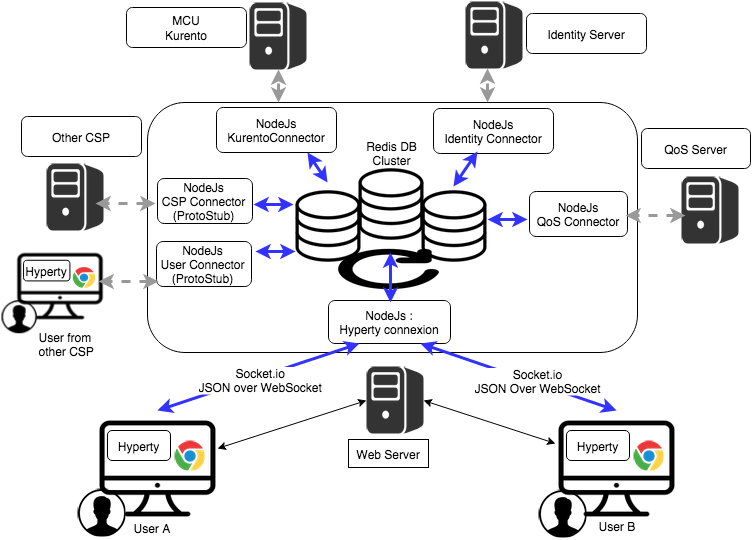


Figure 53: Messaging Node implementation with Node.Js and Redis

**Architecture : Integration in ReThink :**

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

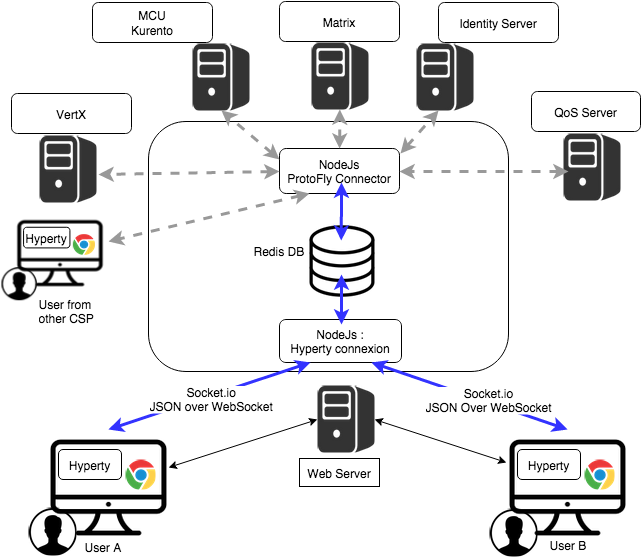


Figure 54: Integration of Node.Js based Messaging Node implementation with reTHINK

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

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

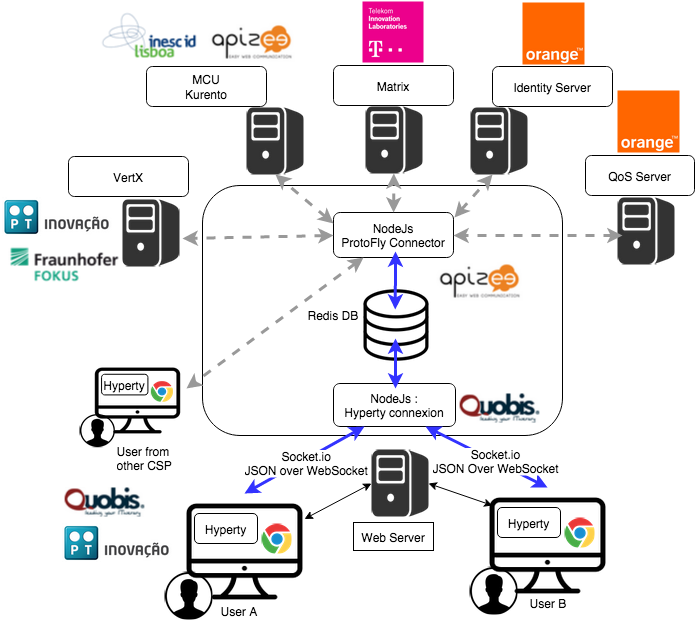


Figure 55: Integration of Node.Js based Messaging Node implementation with reTHINK partners

### Matrix.org based Messaging Node Specification

This section matches the requirements for the functional blocks of the Messaging 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.

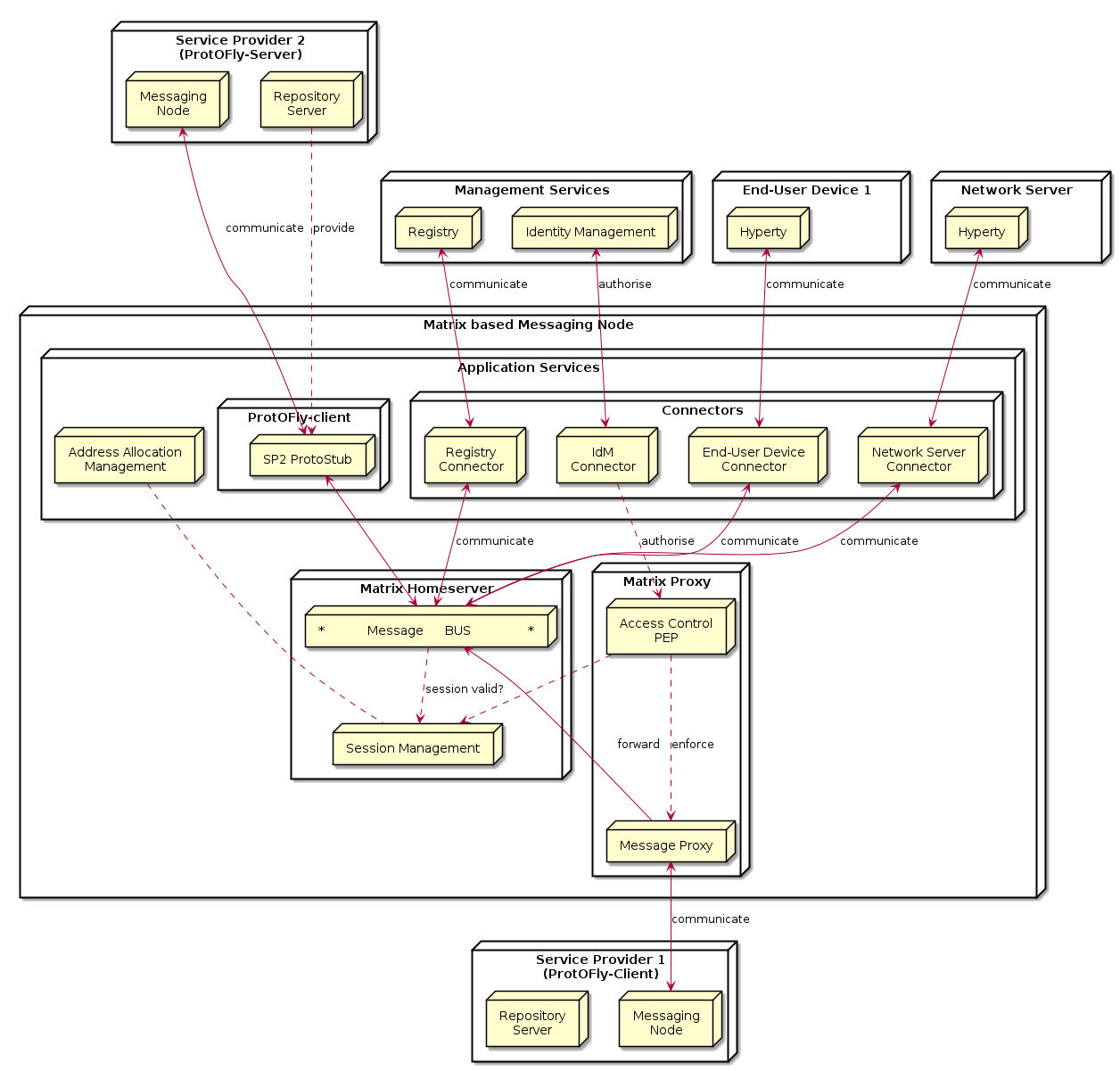


Figure 56: Matrix Messaging Node Architecture

##### 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.

# New Features specification

## Runtime Trust Management Specification (Ricardo Chaves/Nuno)

## P2P Protofly Specification (Paulo)

## QoS Control specification (Marc)

## Multiparty WebRTC Connections specification (Arnaut)

## Interworking with Legacy Services (Anton)

# Conclusions

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

The core of the document (Chapter 4 and 5) provided a detailed specification of the Hyperty Runtime architecture and the Core Runtime components required to support the execution of Hyperties. The Hyperty Runtime architecture was designed with a security by design approach where different types of components can be executed in isolated sandboxes.

The design of the Hyperty Runtime APIs were validated with detailed descriptions of the main procedures to be supported by the Hyperty Runtime, namely basic procedures (e.g. message routing and Hyperty deployment), Identity Management Procedures (e.g. registration and login of users) and Human to Human communication procedures.

At the end, detailed design was also validated from the data models and interfaces design specified in D2.2 and a few improvements were made.

The reTHINK Core Framework specification is sustained by a comprehensive state of the art research on web runtime and real-time messaging with special attention given to security as well as by an exhaustive work in terms of procurement of existing open source solutions to be used to prototype reTHINK Core Framework components. Taking as input the procurement report, some solutions were selected and some implementation considerations were made. This approach, positions reTHINK prototypes at the forefront of technology with its new functionalities. At the same time it also promotes 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.

The specification will evolve along the implementation phase and it will be also completed with the definition of additional procedures required by the scenarios implementation tasks. Thus, additional procedures are expected to be defined to handle Machine to Machine communication and Human to Machine communication use cases (partial done at the time of this writing), as well as trust and context management procedures.

The Hyperty Runtime APIs were designed to be Developer friendly hiding many complexities from the developer. For example, the complex mechanisms required to manage ID and Access tokens is provided out of the box by the Core Runtime. The same applies to the mechanisms implemented by the Core Runtime to enable out of the box seamless interoperability by using the ProtOFly concept. Developers only have to deal with a couple of functions MessageBUS.postMessage() and the Syncher API. Nevertheless, the Hyperty Service Framework - an Hyperty Software Development Toolkit (SDK) - was also introduced in this report in order to further increase the levels of productivity of Hyperty developers.

The Network Platform specification supporting Specialised Network Services is an ongoing work that will be reported later in D3.4, as originally planned.

# 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] - Deliverable D2.2 “Data Models and Interface Specification of the Framework ”, 30-08-2015

[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://Node.js.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] 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] - https://xmpp.org/xmpp-software/libraries/

[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/

[81] https://http2.github.io/

[82] RFC7540 - Hypertext Transfer Protocol version 2

[83] Object RTC - http://ortc.org/

[84] draft-tsvwg-quic-protocol-01 : QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 - http://tools.ietf.org/html/draft-tsvwg-quic-protocol-01

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

[86] https://whatwg.org/

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

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

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

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

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

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

[93] http://www.w3.org/2011/04/webrtc/

[94] https://w3c.github.io/webrtc-pc/

[95] http://www.w3.org/TR/mediacapture-streams/

[96] http://www.w3.org/TR/mediastream-recording/

[97] http://www.w3.org/TR/image-capture/

[98] http://w3c.github.io/mediacapture-depth/

[99] http://www.w3.org/TR/mediacapture-fromelement/

[100] http://www.w3.org/TR/audio-output/

[101] http://www.w3.org/TR/webrtc-stats/

[102] http://www.w3.org/TR/screen-capture/

[103] http://apirtc.com/api-docs/

[104] http://www.quobis.com/index.php?option=com\_content&task=view&id=285&Itemid=208

[105] http://passportjs.org/

[106] http://www.html5rocks.com/en/tutorials/es7/observe/

[107] http://w3c.github.io/WebRTC-pc/#identity

[108] https://nodesecurity.io/resources

[109] Deliverable D4.1, “Management and Security features specifications”, 30-09-2015.

[110] http://requirejs.org/

[111] https://github.com/crosswalk-project/crosswalk

[112] http://cordova.apache.org/

[113] https://github.com/eface2face/cordova-plugin-iosrtc

[114] https://github.com/alongubkin/phonertc

[115] http://beagleboard.org/bone

[116] http://weworkweplay.com/play/raspberry-pi-Node.js/

[117] http://beagleboard.org/Support/BoneScript

[118] http://www.armhf.com/node-js-for-the-beaglebone-black/

[119] http://www.armhf.com/download/

[120] http://cylonjs.com/

[121] http://gf3.github.io/sandbox/

[122] https://github.com/telefonicaid/lwm2m-node-lib

[123] http://samsung.github.io/iotjs/

[124] http://nuttx.org/

[125] https://github.com/pac4j/vertx-pac4j

[126] http://redis.io/topics/pubsub

[127] https://github.com/NodeRedis/node\_redis)

[128] http://expressjs.com/

[129] https://www.npmjs.com/package/node-sandbox