

**Deliverable D3.3**

**Runtime and Hyperty Messaging Node Phase 2**

|  |  |
| --- | --- |
| Editor: | Paulo Chainho, PT Inovação |
| Deliverable nature: | <(R) Document, Report ¦ (DEM) Demonstrator ¦ (DEC) Website, press, media ¦ (OTHER) Software> |
| Dissemination level: (Confidentiality) | Public (PU) |
| Contractual delivery date: | 31/12/2016 |
| Actual delivery date: | 28/02/2017 |
| Suggested readers: | Service providers’ designers and developers |
| Version: | 1.0 |
| Total number of pages: |  |
| Keywords: | Hyperty, Microservices, Edge Computing, Runtime, Messaging Middleware, Protofly, Reporter – Observer, Data Synchronisation |

***Abstract***

This Deliverable contains documentation that accompanies Phase 2 release of reTHINK Core Framework components published as source code in reTHINK Github repositories. The reTHINK Core Framework Phase 2 release includes the Hyperty Runtime (Core Runtime Components that are reused in Hyperty Browser Runtime and Hyperty NodeJS Runtime) and three Message Node implementations: Vertx Message Node, NoMatrix Message Node, and NodeJS Message Node, as well as the Hyperty Service Framework featuring a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties. This deliverable includes a full suite of tutorials targeting external developers; a summary of major changes done in the main specification of reTHINK Core Framework components; and Github repository links to the documentation of released components. The reTHINK Core Framework Phase 2 components are used to implement Phase 2 reTHINK scenarios that will be used to validate reTHINK Framework with Telcos and start-ups.

[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 3.2 Hyperty runtime prototype, Task 3.5 Hyperty Messaging Node, Task 3.4 Hyperty Services Framework**

Document title: **Runtime and Hyperty Messaging Node Phase 2**

Editor: **Paulo Chainho, Altice Labs**

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

Copyright notice

2017 Participants in project RETHINK

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

# Executive summary

This Deliverable accompanies Phase 2 release of the Core Framework components published in reTHINK Github repositories, including the Hyperty Runtime (Core Runtime Components that are reused in Hyperty Browser Runtime and Hyperty NodeJS Runtime) and three Message Node implementations: Vertx Message Node, NoMatrix Message Node, and NodeJS Message Node. The Hyperty Service Framework is also released and it features a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties. A full suite of documentation targeting external developers is included in this report. Besides, this document provides an overview of changes done in the main specification of reTHINK Core Framework components according to feedback taken from the implementation activity.

The reThink source code repositories associated with this deliverable are publicly available on GitHub:

* Hyperty Core Runtime: <https://github.com/reTHINK-project/dev-runtime-core>
* Hyperty Runtime Browser: <https://github.com/reTHINK-project/dev-runtime-browser>
* Hyperty Runtime NodeJS: <https://github.com/reTHINK-project/dev-runtime-nodejs>
* Vertx Message Node: <https://github.com/reTHINK-project/dev-msg-node-vertx>
* NoMatrix Message Node : <https://github.com/reTHINK-project/dev-msg-node-nomatrix>
* NodeJS Message Node: <https://github.com/reTHINK-project/dev-msg-node-nodejs>
* Hyperty Service Framework: <https://github.com/reTHINK-project/dev-service-framework>

All code is provided under the Apache 2 license to assure most flexible use in commercial environments that may build upon reTHINK components without being obliged to publish modifications and enhancements.

# List of authors

|  |  |  |
| --- | --- | --- |
| Company | Author | Contribution |
| PTIN | Paulo Chainho  Miguel Mesquita | Syncher, Runtime Loader, Message Bus, MiniBus, RuntimeUA, Sync Manager, Vertx Messaging Node, Address Allocation Tutorials |
| Fraunhofer | Robert Ende  Mark Emmelmann | atalogue Data Object Factory, Runtime Catalogue, Tutorials |
| Quobis | Anton Roman Portabales  David Vílchez | Browser Runtime, Standalone Runtime, Storage Manager, Tutorials |
| Apizee | Jamal | NodeJS Messaging Node, NodeJS Runtime |
| Deutsche Telekom | Steffen Druesedow | NoMatrix Messaging Node, Tutorials |
| Orange Labs | Simon Bécot  Kevin Corre | IdP Proxy, Tutorials |
| INESC-ID | Ricardo Chaves  Nuno Santos  Ricardo Pereira | Identity Module, IdP Proxy, Runtime Registry, Policy Engine, Domain Registry Connector, Registry Data Object Factory, Discovery, Tutorials |
|  |  |  |
|  |  |  |

# Table of Contents

[Executive summary 4](#_Toc475459504)

[List of authors 5](#_Toc475459505)

[Table of Contents 6](#_Toc475459506)

[Abbreviations 8](#_Toc475459507)

[1 Introduction 10](#_Toc475459508)

[1.1 Objectives and Overview 10](#_Toc475459509)

[1.2 Structure 11](#_Toc475459510)

[2 Tutorials 12](#_Toc475459511)

[2.1 Getting Started 12](#_Toc475459512)

[2.2 reTHINK Framework Overview 12](#_Toc475459513)

[2.3 Survey 12](#_Toc475459514)

[2.4 Hyper-linked Entities - Hyperties 13](#_Toc475459515)

[2.4.1 Decentralized Messaging Framework overview 14](#_Toc475459516)

[2.5 Protocol on-the-fly (protofly) and Protostubs 16](#_Toc475459517)

[2.6 P2P Data Synchronisation: Reporter - Observer Model 17](#_Toc475459518)

[2.6.1 Data Object URL address 18](#_Toc475459519)

[2.6.2 Data Object Schema 18](#_Toc475459520)

[2.6.3 Parent - Children Resources 18](#_Toc475459521)

[2.6.4 Syncher and Sync Manager 19](#_Toc475459522)

[2.7 Trust Management and Security 20](#_Toc475459523)

[2.7.1 User Identity 21](#_Toc475459524)

[2.7.2 Runtime Sandbox 28](#_Toc475459525)

[2.8 Hyperty Development 29](#_Toc475459526)

[2.8.1 Hyperty Concept 29](#_Toc475459527)

[2.8.2 Criteria to use the Hyperty Concept 30](#_Toc475459528)

[2.8.3 Getting Started 30](#_Toc475459529)

[2.9 Application Development 32](#_Toc475459530)

[2.9.1 Application vs Hyperty 32](#_Toc475459531)

[2.9.2 How to use Hyperties 33](#_Toc475459532)

[2.9.3 How to adapt existing Applications 34](#_Toc475459533)

[2.10 Message Node and Protostubs design recommendations 37](#_Toc475459534)

[2.11 Protostub specification 40](#_Toc475459535)

[2.11.1 Stub construction and activation 40](#_Toc475459536)

[2.11.2 Integration with the Messaging Bus of the Runtime 41](#_Toc475459537)

[2.11.3 Auto connect mechanism 41](#_Toc475459538)

[2.11.4 Connection events (TODO: update to same event-types as used for the P2P stubs?) 42](#_Toc475459539)

[2.11.5 The ProtoStub API 42](#_Toc475459540)

[2.12 IdP Proxy Development 44](#_Toc475459541)

[2.13 Interworking stub development 44](#_Toc475459542)

[2.13.1 Stub conceptual view: 44](#_Toc475459543)

[2.13.2 Interworking stub conceptual view: 45](#_Toc475459544)

[2.13.3 Communication flows 45](#_Toc475459545)

[2.13.4 Interworking identity proxy development 45](#_Toc475459546)

[3 Specification changes 46](#_Toc475459547)

[3.1 Hyperty Runtime 46](#_Toc475459548)

[3.2 Messaging Node 47](#_Toc475459549)

[4 References 49](#_Toc475459550)

# 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 aims to demonstrate a radical new way to develop and deliver real time communication services. reTHINK concepts and architecture follows edge computing principles [18] pushing as much as possible the Business logic to end-user devices and network edge servers. reTHINK Business Capabilities are provided by cooperative Microservices [17] executing on behalf of users, called Hyperlinked Entities, named Hyperties. Hyperties are independently deployable components, each one providing a small set of business capabilities, using the smart endpoints and dumb pipes philosophy. Billions of devices are already Hyperty-enabled and ready to participate in a reTHINK ecosystem, since Hyperties are programmed in Javascript ECMA5/6. The User Identity associated to a Hyperty instance is decoupled from the Hyperty Service Provider through WebRTC Identity Management mechanisms.



Figure - Core Framework Components scope in reTHINK Architecture

This deliverable accompanies phase 2 release of the Core Framework components published in reTHINK Github repositories [1]. The implementation of Core Framework components follows D2.3 [2] reTHINK Architecture design and D3.5 Core Framework detailed specification [3]. Phase 2 of the Core Framework includes the Hyperty Runtime (Core Runtime Components [6] that were reused in Hyperty Browser Runtime [7] and Hyperty NodeJS Runtime[8]) and three Message Node implementations: Vertx Message Node[9], Matrix Message Node[10] and NodeJS Message Node[11]. Finally, the Hyperty Service Framework [12] is also released featuring a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties, Protostubs and IdP Proxies. A full suite of documentation specially written to facilitate reTHINK embracing by web developers, which is also published in GitHub pages, is included in this report, as well as a short summary of major changes done in the specification of reTHINK Core Framework components provided in D3.5, according to feedback taken from the implementation activity.

It should be noted that the Network Platform components supporting Specialised Network Services were already released in D3.4, as originally planned.

This deliverable complements deliverable D4.3 (Management and Security features implementation [5]), which accompanies Phase 2 release of reTHINK Support Services.

## Structure

This report starts with an introduction and the list of pointers to all components’ source code and manuals reported in this deliverable (Chapter 2). A set of Tutorials (Chapter 2) targeting reTHINK Web Developers are provided in Chapter 3 while Chapter 4 summarises major changes done in phase 2 specification.

# Source Code and Manuals

The reThink source code and associated manuals reported in this deliverable are publicly available on reTHINK GitHub repositories, namely at:

* Hyperty Core Runtime Source Code and Manuals:

<https://github.com/reTHINK-project/dev-runtime-core>

* Hyperty Runtime Browser Source Code and Manuals:

<https://github.com/reTHINK-project/dev-runtime-browser>

* Hyperty Runtime NodeJS Source Code and Manuals:

<https://github.com/reTHINK-project/dev-runtime-nodejs>

* Vertx Message Node Source Code and Manuals:

<https://github.com/reTHINK-project/dev-msg-node-vertx>

* NoMatrix Message Node Source Code and Manuals:

<https://github.com/reTHINK-project/dev-msg-node-nomatrix>

* NodeJS Message Node Source Code and Manuals:

<https://github.com/reTHINK-project/dev-msg-node-nodejs>

* Hyperty Service Framework Source Code and Manuals:

<https://github.com/reTHINK-project/dev-service-framework>

All code is provided under the Apache 2 license to assure most flexible use in commercial environments that may build upon reTHINK components without being obliged to publish modifications and enhancements.

# Tutorials

This chapter provides the full set of documentation targeting external developers that is published in reTHINK Github together with the Hyperty Service Framework source code. These documents will be updated and completed along the implementation of Phase 2 demonstration scenarios. A getting started document is provided followed by an overview of the Hyperty Concept, the Messaging Framework used to support Hyperty's interaction, the higher level Data Synchronisation Reporter - Observer communication mechanism, as well as the Hyperty Trust and Security Model. Then, detailed Tutorials are provided to guide on the development of Hyperties, Applications and Message Nodes.

## reTHINK Framework Overview

The reTHINK Framework is a [decentralised communication infrastructure](http://www.rand.org/pubs/research_memoranda/RM3420.html)(licensed with Apache 2.0) that enables developers to easily build and integrate, communication services that are faster, more effective , more trustful and inherently inter-operable. Write once, deploy anywhere, together with seamless cross domain Interoperability, gives developers and service providers much more freedom to really focus on users' expectations. Ultimately, reTHINK framework is an alternative to current [dominant walled garden](%5Bdominant%20walled%20garden%20communication%20networks%5D(https://www.theguardian.com/technology/2012/apr/17/walled-gardens-facebook-apple-censors)) communication networks that prevent new developers and new service providers from entering in the market and, at the same time, [empowers the users](https://techcrunch.com/2016/10/09/a-decentralized-web-would-give-power-back-to-the-people-online/) with the choice and the management of their private data and identities.

The reTHINK Framework provides the tools to build a global decentralised network of [Hyperlinked Entities (hyperties)](https://github.com/reTHINK-project/specs/blob/master/tutorials/hyperty.md) that are [executed at the edge](https://github.com/reTHINK-project/specs/blob/master//runtime) and [trustfully](https://github.com/reTHINK-project/specs/blob/master//trust-management) communicates through a [Decentralised Messaging Framework](https://github.com/reTHINK-project/specs/blob/master/messaging-framework/readme.md).

Developers are all invited to join our [Slack channel](https://rethink-project.slack.com/signup) (send an email to contact@rethink-project.eu ), experiment our demos and build on top of it new Hyperties and / or Applications:

* [Webinars](https://www.youtube.com/channel/UC4xTKj2ZvhUyJosA_fLeAhg)
* [Demos](https://hysmart.rethink.ptinovacao.pt)
* [Available Hyperties](https://github.com/reTHINK-project/dev-hyperty)
* [Quick start to develop Hyperties](https://github.com/reTHINK-project/dev-hyperty-toolkit)
* [Quick start to develop Applications with Hyperties](https://github.com/reTHINK-project/dev-app)
* [Tutorials](https://github.com/reTHINK-project/specs/blob/master/tutorials/readme.md)
* [Complete Specification](https://github.com/reTHINK-project/specs)

## Survey

In order to improve reTHINK framework, your feedback is precious to us:

* [App development survey](https://docs.google.com/forms/d/e/1FAIpQLSeLpZ79p2qKtXEuNk-d_roCWV8W5PuA3jNygBNsOV9LI1Q6lg/viewform)
* [Hyperty development survey](https://docs.google.com/forms/d/e/1FAIpQLSeNvRJ2JC43rkkby0osANqCzVj_a8se8eB3bQ1ymJ_BncHjMg/viewform)

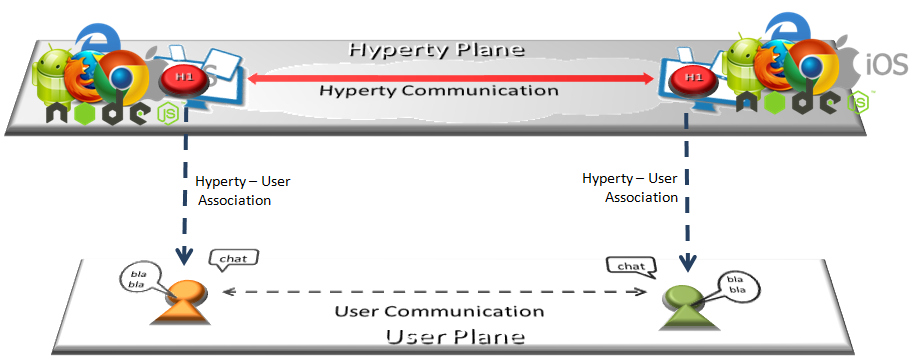
## Hyper-linked Entities - Hyperties

This document provides an overview about the Hyperty concept and it should be the starting point for any new developer. After this document, all developers should also read:

* the [Hyperty Messaging Framework overview](../messaging-framework/readme.md)
* the [Reporter - Observer Data Synchronisation model](../messaging-framework/p2p-data-sync.md)
* the [Hyperty Trust and Security Model](../trust-management/readme.md)

Hyperties are software modules that are maintained by service providers' catalogues, hence can be accepted as trusted software that is well maintained and secure. These modules follow [Microservices](http://martinfowler.com/articles/microservices.html) architectural patterns meaning they are independently deployable components each one providing a small set of business capabilities, using the smart endpoints and dumb pipes philosophy i.e. Hyperties don't depend on complex and sophisticated communication middleware like Enterprise Service BUS (ESB). Instead, Hyperties rely on a very light but powerful [Messaging Framework](../messaging-framework/readme.md) concept). On the other side, Hyperties follow emerging [Edge](https://en.wikipedia.org/wiki/Edge_computing) and [Fog](https://en.wikipedia.org/wiki/Fog_computing) computing paradigms as opposed to more popular Cloud Computing. Hyperties can also be executed in Network Servers for specific Business Capabilities (e.g. Media Servers) or when End-user devices don't have enough capabilities in terms of computing resources and/or power.

Currently, Hyperties are programmed in Javascript but in future it is expected the launch of [Hyperty Runtimes](../runtime/readme.md) supporting more programming languages like Go and Java.



Hyperty Concept and Edge Computing

Hyperties follows Decentralized Communications principles providing a few unique advantages:

**Hyperties are inherently inter-operable**

Hyperties communicate each other through a decentralized messaging framework that leverage the Protocol on-the-fly concepts and the [Reporter - Observer communication pattern](../messaging-framework/p2p-data-sync.md) to support cross-domain interoperability without requiring the standardization of network protocols or service APIs.

For example, as soon as there is new measurement collected from a sensor, the data is set in a associated JSON Object, which is immediately propagated to any authorised Observer Hyperty. In this way, the JSON Object handled by Observer Hyperty is always synchronised with the JSON Object owned by the Reporter Hyperty.

**Hyperties are Trustful**

Hyperties use an **Error! Hyperlink reference not valid.** enabling independent identity management from the communication provider and the front-end application. The identity is managed by the user and is verified by an independent trusted entity selected by the user. This identity provider delivers authentication tokens, which are sent to any involved stakeholder. This independent ID would be used for any participating service and users will be free to change service providers, where the subscription changes do not affect user reachability. On enquiry, these IdPs vouch for users’ authenticity and return the URL of the users’ domain, which enables finding destination users. The users’ identities are based on their personal and confidential data, which is verified by other solicited data, but such private information is only divulged under user-controlled privacy rules. More information is provided in [the Hyperty Security and Trust Model document](../trust-management/readme.md).

**Write once, deploy anywhere**

The development of Hyperties and Applications is very easy and flexible, giving the freedom to the developer to select its favorite programming framework. Currently Hyperties are programmed in Javascript ECMA5/6, i.e. any existing device featuring a Browser or a NodeJS can be used today to execute Hyperties without requiring the installation of any new software. This means, billions of devices are already Hyperty enabled and ready to participate in the reTHINK ecosystem.

**Hyperties are more Effective and Faster**

Hyperties follow edge computing principles, promoting a more effective usage of computing and network resources, as well as decreasing communication latency.

**Hyperties are Application Domain agnostic**

Hyperties can be used on any Application Domain, but they are specially suitable for Real Time Communication Apps (eg Video Conference and Chat) as well as IoT Apps.

### Decentralized Messaging Framework overview

This document gives an overview on a Decentraized Messaging Framework technical solution that is used by the higher level [Data Synchronisation Reporter - Observer communication mechanism](p2p-data-sync.md).

The Decentraized Messaging Framework is designed on top of a Resource Oriented Messaging model that supports publish/subscribe as well as request/response messaging patterns. Messages are used to perform CRUD operations on resources handled by communication endpoints, for example to create or update a WebRTC connection. In addition, Subscribe operations on resources are also supported in order to be notified on any change performed to such resource. The current reference implementation uses JSON for the message format. Each message has a fixed set of header fields and a variable message body. These are the common header fields that are required to route Messages:

**id**: an identifier used to associate RESPONSE messages to the initial REQUEST message

**from**: the address of the entity that sends the message.

**to**: the address of the recipient.

**type**: identifies the operation to be performed by the recipient when it receives the messages, including *create*, *read*, *update*, *delete*, *subscribe* and *unsubscribe*.

The Message **Body** is composed by different types of information depending on the type of operation to be performed. For example for *create* type messages, the body contains a value field containing a data resource to be created by the recipient:

{  
"id" : 1  
"type" : "create",  
"from" : "hyperty-runtime://<sp-domain>/<runtime-instance-identifier>/registry",  
"to" : "domain://registry.<sp-domain>",  
"body" : { "value" : <RegistryDataObject> }  
}

The Message delivery is based on a simple message Router functionality that performs a lookup for listeners registered to receive the Message. The Message is posted to all found listeners, which can be other Routers or the final recipient end-point. Thus, the Messaging Framework is comprised by a network of Routers where each Router only knows adjacent registered Routers or end-points.



Hyperty Messaging Delivery Network

Listeners are programmatically registered and unregistered by Routing Management functionalities, which take their decisions according to a higher level view of the Routing Network.



Hyperty Message Routing Management

The Messaging Framework works at three layers:

At the Runtime Sandbox level where services are executing, message delivery is provided by the [MiniBUS component](https://github.com/reTHINK-project/dev-runtime-core/blob/master/src/bus/MiniBus.js).

At the Runtime level where Sandboxes are hosted (e.g. in a Browser or in a NodeJS instance), message delivery is provided by the [Message BUS component](https://github.com/reTHINK-project/dev-runtime-core/blob/master/src/bus/MessageBus.js), which is an extension of the MiniBUS.

At Domain Level, message delivery is provided by the [Message Node](msg-node.md) functionality by using the [Protofly mechanism](protofly.md), i.e. communication between Message BUS and Message Nodes and among Message Nodes are protocol agnostic. This also means that the Message Node can be provided by any Messaging solution as soon as there is a Protostub available. Currently, a [Vertx Message Node](https://github.com/reTHINK-project/dev-msg-node-vertx), a [Matrix Message Node](https://github.com/reTHINK-project/dev-msg-node-matrix) and a [NodeJS Message Node](https://github.com/reTHINK-project/dev-msg-node-nodejs) are provided. These are just reference implementations of Message Nodes and anyone is free to develop its own Message Node. Check the [Message Node design guide](msg-node-design.md) and the [Protocol Stub specification](stub-specification.md) for more details.



Adhoc Messaging Oriented Middleware Routing Layers

At runtime level (MessageBUS and MiniBUS), it is used a standard CRUD based [JSON Message Model](../datamodel/core/message/readme.md), which is easily mapped into Restfull APIs.

More detailed descriptions about Decentralized Messaging Framework can be found for:

* [Protocol on-the-fly](protofly.md) mechanism
* [Message Node](msg-node.md) functional description
* [Peer to peer Message Delivery](p2p-msg-delivery.md) procedure
* [Data Synchronisation Reporter - Observer communication mechanism](p2p-data-sync.md)

## Protocol on-the-fly (protofly) and Protostubs

Protocol on-the-fly leverages the code on-demand support by Web runtimes (eg Javascript), to dynamically select, load and instantiate the most appropriate protocol stack during run-time. Such characteristic enables protocols to be selected at run-time and not at design time, enabling protocol interoperability among distributed services, promoting loosely coupled service architectures, optimising resources spent by avoiding the need to have Protocol Gateways in service's middleware as well as minimising standardisation efforts. The implementation of the protocol stack, e.g. in a javascript file, that is dynamically loaded from the Service Provider and instantiated at run-time is called **Protostub:**. For security reasons, Protostubs are executed in isolated sandboxes and are only reachable through the Runtime MessageBUS and the Protostub Sandbox MiniBUS.



Protocol on-the-fly and Protostubs

## P2P Data Synchronisation: Reporter - Observer Model

This document gives an overview on how Hyperties cooperate with each other through a Data Synchronisation model called Reporter - Observer.

While the [Protocol on-the-fly](protofly.md) provides transport interoperability without requiring the standardization of messaging protocols, the Reporter - Observer communication pattern enables semantic interoperability between Services without having to standardize Service APIs.

It extends existing Observable communication patterns by using a P2P data stream synchronization solution for programmatic Objects e.g. JSON Objects, hereafter simply called Data Objects.

To avoid concurrency inconsistencies among peers, only one peer is granted writing permissions to the Data Object - the Reporter service - and all the other service instances only have permissions to read the Data Object - the Observers.

As soon as the Reporter performs changes to Data Objects, they are immediately propagated to any authorized Observer by using the messaging framework. In this way, the Data Object monitored by the Observer is always synchronized with the Data Object owned by the Reporter.



Reporter-Observer Communication Pattern

Full interoperability is achieved between two service instances by only having to agree on the usage of common formats for the Data Objects.

To be noted that, conceptually, more complex semantic interoperability and data synchronization technologies, like Semantic Web and Operational Transformation, can be used.

### Data Object URL address

Data Objects addresses follow [Information-centric networking](https://irtf.org/icnrg) design principles to support Data Object portability (between Communication end-points) and group communication. These characteristics are supported by the usage virtual addresses separately allocated to Data Objects, which are agnostic of the network addresses and the execution environment.

### Data Object Schema

Each Data Object is formally described by a JSON-Schema that is identified by a Catalogue URL. This allows to check whether two different services are compliant by checking each supported Data Object schema. At this point the following Data Object schemas are defined:

* [**Connection Data Schema**](../datamodel/data-objects/connection) : services supporting this schema are able to handle [WebRTC Peer Connections](https://developer.mozilla.org/en-US/docs/Web/Guide/API/WebRTC/Peer-to-peer_communications_with_WebRTC) between the Hyperty Runtime instances where they are running independently of the signalling protocol used. The URL Scheme for Connection Data Objects is "connection" (example: "connection://example.com/alice/bob201601290617").
* [**Communication Data Schema**](../datamodel/data-objects/communication) : services supporting this schema are able to handle different communication types including Textual Chat, Audio, Video, Screen Sharing and File sharing. Such communication can be supported on top of WebRTC protocol streams by using the Connection Data Schema. The URL Scheme for Communication Data Objects is "comm" (example: "comm://example.com/group-chat/rethink201601290617").
* [**Context Data Schema**](../datamodel/context) : services supporting this schema are able to produce or consume Context Data, usually collected from sensors. The URL Scheme for Communication Data Objects is "ctxt" (example: "ctxt://example.com/room/temperature201601290617").

### Parent - Children Resources

In order to allow use cases like Group Chat where all involved services are able to write in the Sync Data Object, the Parent - Child Data Sync Objects are introduced.

A Data Object Child belongs to a Data Object Parent children resource and can be created by any Observer of the Data Object Parent as well as by its Reporter. The Reporter - Observer rules still apply to Data Object Child i.e. there is only one Reporter that can update the Data Object Child, which can be an Observer of the Data Object Parent, as mentioned earlier.



Parent - Child Sync

The creation, update and delete of an Data Object Child is performed in the Data Object Parent itself:

\*Data Object Child creation, update and delete code snippet\*

All other services observing or reporting the Data Object Parent, will be notified every time a new Data Object Child is created, updated or deleted.

At this point, Data Object Child can't also be a Data Object Parent of another Sync Data Object, i.e. Data Object composition is limited to one level.

### Syncher and Sync Manager

This section, gives an overview on how the Data Object synchronisation transparently works on top of the [Messaging Framework](readme.md).

The Data Object synchronisation is provided by two components in the Runtime:

The [Syncher](https://github.com/reTHINK-project/dev-service-framework/blob/master/src/syncher/Syncher.js) is a singleton Component co-located with the Service Instance, which is in charge of handling all required procedures to manage data synchronisation at the Service instance side, as a Reporter or a Observer service.

The [Runtime Sync Manager](https://github.com/reTHINK-project/dev-runtime-core/blob/master/src/syncher/SyncherManager.js) is a Core Runtime Component, which is in charge of handling authorisation requests to create Sync Data Objects from Reporters and subscription requests to Sync Data Objects from Observers. As soon as authorisation is granted the Sync Manager handles all required MessageBUS listeners in order to setup the Data Sync Stream routing path among Reporters and Observers. I.e., the Runtime Sync Manager provides a [Messaging Framework](raedme.md) Routing Manager functionality.

The [Message Node Sync Manager](https://github.com/reTHINK-project/dev-service-framework/blob/master/src/syncher/Syncher.js) is a Message Node functionality, which is in charge of handling requests from Runtime Sync Managers in order to setup the Data Sync Stream routing path between the Reporter Runtime and Observers Runtimes. I.e., the Message Node Sync Manager also provides a [Messaging Framework](readme.md) Routing Manager functionality..



Routing Management for Hyperty Data Syncronisation

A detailed description of the Data Synchronisation procedures are provided [here](../dynamic-view/data-sync/readme.md)

## Trust Management and Security

This document gives an overview on the Hyperty Trust Model as well as on Hyperty Sandbox runtime execution environment.

Hyperties are securely associated to User Identities selected by the end-user himself. Hyperty Users are human beings (including group of human beings e.g. corporation) or things (including group of things and physical spaces e.g. a smart home or smart building).

Hyperty Trust Model extends [WebRTC Identity model](https://w3c.github.io/webrtc-pc/#sec.identity-proxy) where Identity tokens are generated, inserted in intercepted Messages sent by Hyperties and validated by recipient Hyperty Runtimes before delivered to the target Identity. These identity management procedures are performed according to applicable policies managed by the end-user.



Hyperty Trust Management

### User Identity

In our modern society, technology is ubiquitous, and transactions are evermore accomplished using digital technologies without the need to involve physical contact. An example of this situation can be observed in money transactions, whilst a few years ago if someone needed to make a bank transfer, it would require that person to move personally into a bank agency to order it, and in current days these money transfers can be performed using a smartphone. To achieve this, we need digital credentials to prove who we are and what we are allowed to do in remote communication. Therefore two important information security mechanisms must be implemented: authentication and authorization.

Authentication is verifying the identity claimed by an actor. Usually, to authenticate users, credentials make use of one or several factors among:

* something a user knows (such as a PIN code or a password),
* something a user owns (such as a SIM card),
* something a user is (such as a fingerprint, or a voice sample),
* something a user does (such as typing characteristics).

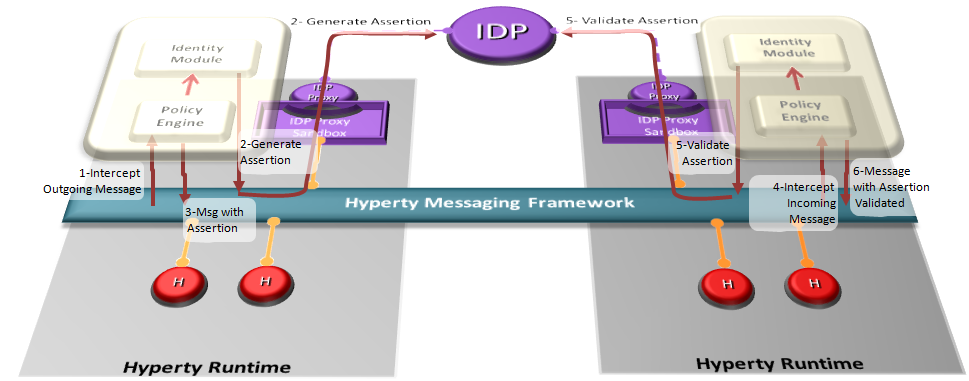
Authorization is deciding whether a given identity may execute or access a certain resource. Access control to a service or system, can be achieved based on access rights or policies that allow or deny a particular action based on an identifier, a role (RBAC), or an attribute (ABAC).

Hyperty trust model relies on service-independent authentication that is global and non-service-bound. In implementations, each Hyperty service provider may include their own user recognition (i.e. their own internal user accounts) and service authorization (i.e. level of permissions to use the service), over and above the initial user identification.

#### Identity Module and IdP Proxy

The Identity Module (Id Module) is the [Core Runtime](https://github.com/reTHINK-project/dev-runtime-core) component responsible for handling the user identity and the association of this identity with the Hyperty instances. The identity in the reTHINK project is not fixed to a unique Identity source (espically it is not bound to a single Service Provider), but obtained through several Identity P£rovider. In this approach, the Id Module provides the user the option to choose the preferred IdP for authentication. For example, a user with a Google account can use Google as an Identity Provider to retrieve Identity Tokens that may be associated by the Identity Module with a Hyperty instance. Following the flexible and adaptable approach fostered by WebRTC standards (from both [IETF](https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-12#section-5.6.2) and [W3C](https://www.w3.org/TR/webrtc/#sec.identity-proxy)) IdP Proxies are downloaded from each Identity Provider to act as intermediaries between the Identity Module and the aforementioned Identity Providers. The IdP Proxy is the [IdP ProtoStub](hyperty-messaging-framework.md#protocol-on-the-fly-protofly-and-protostubs) that is responsible to handle the communication between the Identity Module and a specific Identity Provider.

The following figure illustrates this interaction:



Interaction between the Identity Module and the Identity Provider

*NB. The Policy Engine is the Core Runtime component rsponsible to manage and apply policies to perform access control. More details are found in the* [*Policy Management*](https://github.com/reTHINK-project/specs/tree/master/policy-management) *section.*

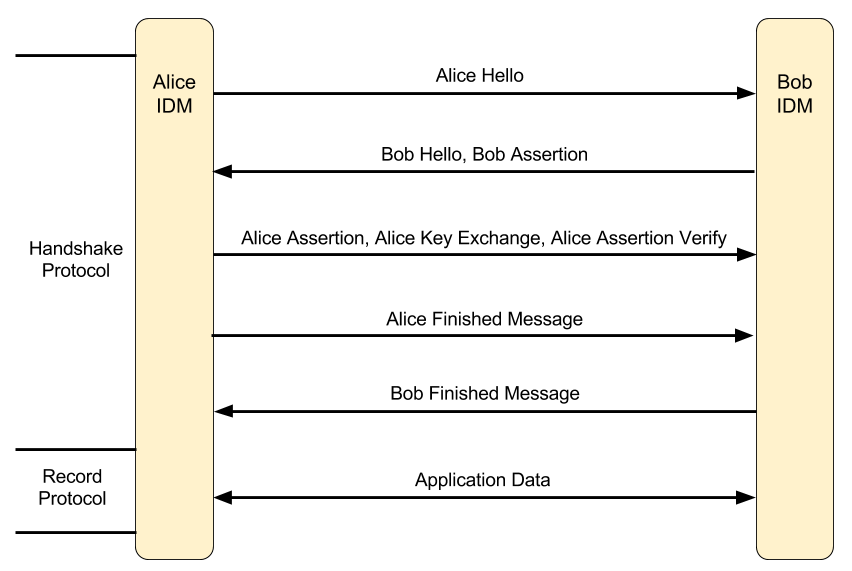
#### Mutual Authentication

Identity management relies upon managing life-cycle of identity-related security tokens. Therefore, identity assertions are used in reTHINK to identify a user and to prove that he was authenticated on an IdP. The IdP asserts a particular content for the user, provided during the request for authentication by the IdM, after a successful authentication this same user.

reTHINK architecture is designed to operate in a peer-to-peer architecture, and as a result there is no centralised service that proceeds to authenticate the users in reTHINK. Because of that, the identity assertions play a very important role in reTHINK, by enabling mutual authentication between users: here agin reTHINK promotes ideas similar to [WebRTC working group](https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-12). Using IdP protocols like [OpenID Connect](http://openid.net/developers/specs/) it is indeed possible to request the IdP to assert a particular content on the request to authenticate the user. The identity token received after the user's successful authentication contains the user identity assertion and the content provided during the authentication request, also asserted. In reTHINK, the content to be asserted by the IdP is a public key specified by the Identity Module, later used to prove the user's identity to a third party. This way, the identity assertion provided by the IdP acts as a digital certificate, where the IdP plays the role of a Certification Authority (CA).

As such, whenever a user intends to initiate a communication with another user, the mutual authentication protocol is triggered so that users can authenticate each other mutually. In order for the mutual authentication to be successful, all the messages are required to have an identity assertion, which will work as a digital certificate. Therefore, to authenticate to a message, the sender identity assertion obtained through the IdPs is attached to the message, containing the user public key. To confirm that the public key actually corresponds to the claimed identity, the receiving user contacts the IdP to validate the content of the Identity assertion. With this public key, the receiver can validate the sender message digital signature and encrypt the response to the sender challenge to conclude and successfully authenticate the sender identity. For mutual authentication, the roles invert and the receiving users becomes the one who must prove his identity, using the same procedure. For user privacy assurance, the Identity Module may frequently request the generation of a new Identity assertions, each with different public keys.

Taking the mutual authentication process described above, the IdM performs this authentication process and along with it, generates the symmetric session keys to be used for the protection of the messages exchanged after the mutual authentication process. This is done in an identical manner as in the TLS protocol. By doing this, we provide the same procedures and the same security properties of the TLS, including the security assurance for the message integrity and confidentiality. The proposed solution used in rethink for authentication is also separated in the handshake and record phases. Some simplifications are introduced in the authentication protocol compared to TLS, mainly because there is no need to support some of the features. The negotiation steps of cryptographic methods in TLS are not taken in consideration since it is the Identity Module that defines the cryptographic methods to be used, and all devices running the reTHINK application will use the same version of the Identity Module. Additionally, compression will not be considered, since this protocol is already running on top another TLS communication (i.e. the communication of each client with the Message Node).



reTHINK mutual authentication flow

The figure above illustrates all the messages exchanged to provide mutual authentication between the users Alice and Bob. The following provides an explanation of the overview on the developed mutual authentication flow.

* **Alice Hello**: This message contains a number generated randomly by Alice.
* **Bob Hello, Bob Assertion**: This message contains a random number generated randomly by Bob and his identity assertion with his public key.
* **Alice Assertion, Alice Key Exchange, Alice Assertion Verify**: In case Alice is not in anonymous mode, she sends her identity assertion with her public key and sends a randomly generated premaster key encrypted with Bob's public key. In case Alice identity assertion is sent, the assertion verification must also be sent in the message to prove the ownership of the public key. This verification consists in the signature of the hash of all the previous messages exchanged and the content of this message.
* In this phase both Alice and Bob generate a master key using the premaster key and both random values exchanged in the beginning. With the master key the following keys are computed: Alice's MAC Key, Bob's MAC Key, Alice's encryption Key, Bob's encryption Key.
* **Alice Finished Message**: This is the first message to use the keys generated previously, and contains the MAC from the result of a pseudo random function that receives the master secret and all the handshakes messages exchanged previously as argument. This result is also encrypted with the Alice's encryption key.
* **Bob Finished Message**: This message is the response to the Alice finished message. It uses the keys generated previously to generate a MAC and encrypt that value. The MAC is obtained from the result of a pseudo random function that receives the master secret and all the handshakes messages exchanged previously as argument. This result is also encrypted with the Bob's encryption key.
* In this phase the authentication or mutual authentication is complete with all the necessary keys generated. This symmetric key allows for a secure communication between two users.
* **Application Data**: This message contains the body of the message encrypted with the encryption key of the sender to grant confidentiality, and a MAC of the entire message with the sender's MAC key to grant integrity.

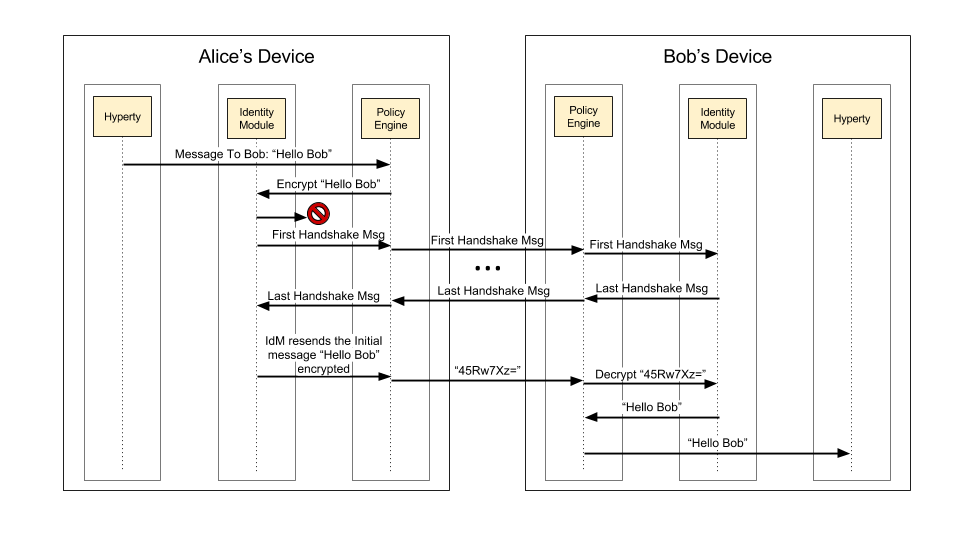
*NB. Despite the emphasis of the protocol developed for the IdM being in the mutual authentication it also supports anonymous communication, where the user who initiates the call starts it in anonymous mode. The only difference in the protocol flow is that the user who starts the communications does not send his identity assertion, not allowing the user who receives the request for communication to verify the caller's identity. The decision of accepting or not the anonymous communication rests with the policy engine.*

##### When does the mutual authentication occur?

The mutual authentication protocol is mandatory so that a secure communication channel can be established. Whenever some user intends to start a communication with another user the mutual authentication protocol must be triggered. The mutual authentication can start in two situations: when a message from a Hyperty URL to a Hyperty URL is sent for the first time between those Hyperties, and when an IdM method is called to explicitly start a mutual authentication between two Hyperties URLs.

The IdM method that triggers the mutual authentication protocol is called in a specific case, being when the Policy Engine receives a request to accept a subscription to a group chat, or when the current user invites another user to subscribe his Data Object.

Regarding the activation triggered by sending a message for a Hyperty for the first time, the Figure below illustrates the message flow for the first message sent between two Hyperties, one from Alice and one from Bob sent via the Message Bus. Only the essential components to demonstrate the flow are illustrated, which in this case are the Hyperty, the Identity Module (IdM) and the Policy Engine (PE). It is assumed that all messages sent via the Message Bus are intercepted by the Policy Engine, in order to be filtered according to the defined policies.



Example of Alice sending a message to Bob, for the first time

Taking data flow as an example, Alice starts by sending for the first time, the message "Hello Bob" to Bob. The PE intercepts the message and sends it to the IdM, to encrypt the message. The IdM tries to find the cryptographic keys generated for the secure communication between Alice and Bob. Since it is the first communication between Alice and Bob no cryptographic keys exists and the identity of Bob has not yet been authenticated. In this case, the IdM suspends the transmission of that message and starts immediately the mutual authentication protocol.

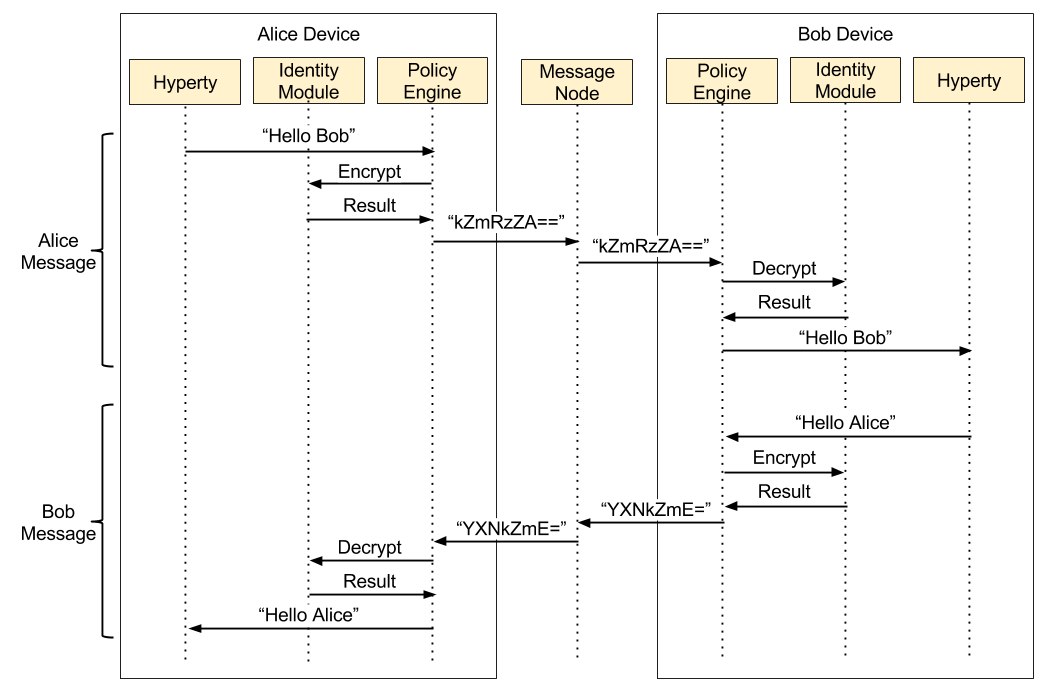
After receiving the last message of the handshake protocol and if Bob's identity is verified, Alice's IdM unblocks the initial message that triggered the mutual authentication protocol. However, the message is now sent encrypted with her symmetric key and with the integrity of the message protected by adding the hash of the message. This message is forward to the Bob's device and is intercepted by the PE, which requests Bob's IdM to decrypt the message and verify the hash. The IdM then uses Alice's symmetric keys to decrypt and verify the hash of the message, and passes the message with the original value "Hello Bob" sent by Alice. To conclude, the PE receives the message, decrypted and validated, and forwards it to the destination Hyperty.

#### Secure Communication

The communication between users is one of the major characteristic in reTHINK. As such, the authentication of each user takes a big role to ensure that no personification can occur. Every time a user starts a communication with another user, the process of mutual authentication, described above, is initiated by the Identity Module. This mutual authentication is not only useful for the authentication of the users, but is also essential for the exchange of the symmetric keys used in the established secure communications. In case, one of the users starts a communication anonymously, the authentication of the other user is made, so it can be possible to establish a secure channel between the two users.

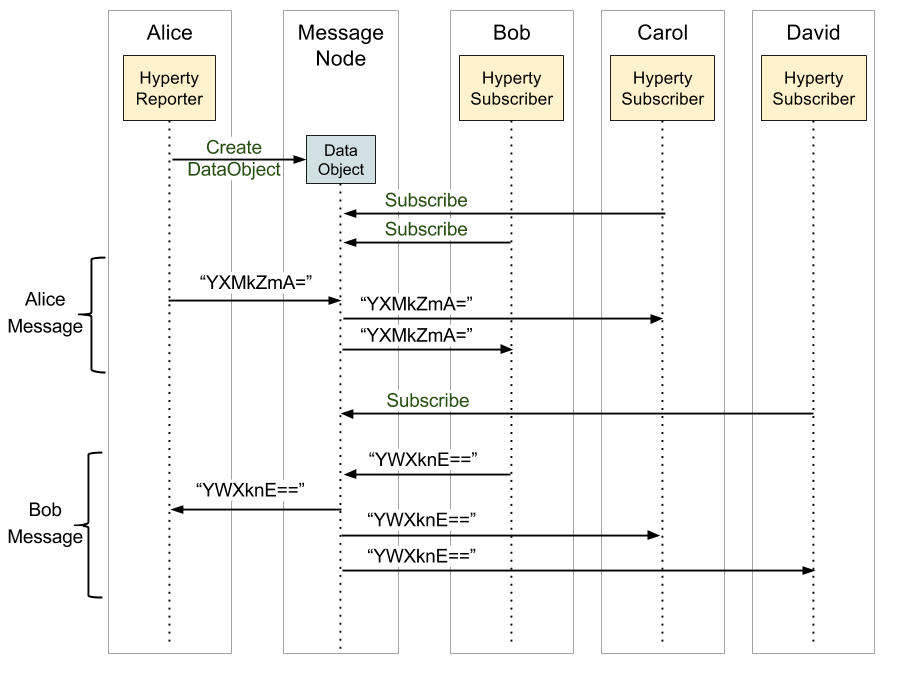
The reTHINK framework provides support for two types of communication, a direct communication between two users through their respective Hyperties and a group chat communication, using Data Objects, where the messages are exchanged between all participants in the group, subscribed to that Data Object. The encryption of these communications is optional, depending on the user preferences regarding secure channels. This secure channel consists on the creation of a HMAC of the message followed by the encryption of the message, to ensure confidentiality and integrity.

To implement the secure channel on a direct communication between two Hyperties, the Identity Module needs to capture these messages, in order to secure the contents of the messages exchanged. Using Figure 5 as an example of a secure communication between Alice and Bob, when Alice sends a message through her Hyperty, this message is intercepted by the Policy Engine, since all messages passe by it. The Policy Engine sends that message to the Identity Module, to be encrypted with Alice's session key and authenticated with the Alice MAC's key, with these two keys generated during the mutual authentication process. After the message manipulation, the Identity Module returns the message to the Policy Engine, to be sent to the public Message Node. The Message Node then forwards the message to the Bob device, where the Policy Engine running in the Bob device, intercepts it and applies the same steps used in the protection phase, but in this time to decrypt the message. After the Policy Engine receives the decrypted message by the Identity Module, returns it to the Hyperty of Bob in plaintext. When Bob sends a message, the same flow is used, with the only difference being in the key used to encrypt and to authenticate, which in this case uses the set keys of Bob.



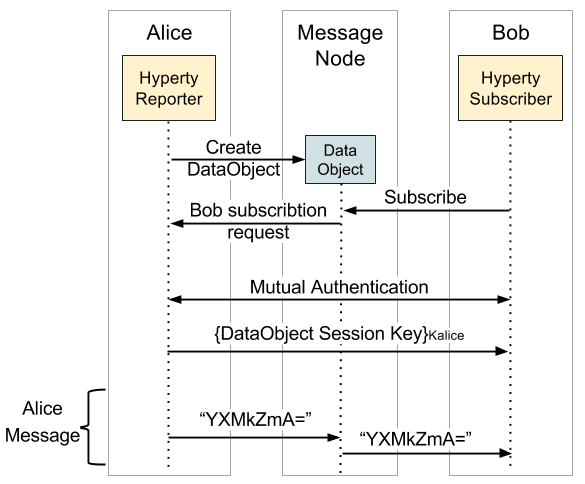
Hyperty to Hyperty communication

In a group chat communication, the communication starts with the creation of a Data Object by the Hyperty Reporter and the subscription from other Hyperties to that Data Object. After that, when a Hyperty sends a message, it is broadcasted to all Hyperties subscribed. The encryption of these messages is optional, but in case of opting to use a secure channel, these messages are encrypted with a symmetric key shared by all participants in the group chat.



Group chat communication

The Hyperty reporter, in a group chat, is responsible for managing the session key and the authentication of others Hyperties that join that chat group. Following the creation of a Data Object by the reporter, he generates a symmetric key and associates it to the Data Object. As illustrated in the Figure below, after the creation of the Data Object, when a Hyperty makes a request for subscription, this request is forward to the reporter, which starts the mutual authentication between the reporter and that subscriber. When the authentication is completed with success, the reporter encrypts the symmetric key associated to the Data Object with the reporter session key, obtained through the mutual authentication process and shared by both Hyperties reporter and subscriber. After that, the subscriber has the key which allows him to decrypt all messages exchanged. In the end, all Hyperties that successfully subscribe the same Data Object will end with the same symmetric key for that session.



Hyperty subscription flow

#### Identity selection

The Identity Module is responsible to obtain, manage the identities in reTHINK, and provide the identities when the Runtime Registry requires an identity to associate with a Hyperty. For the users to be able to select the desired identity, a GUI is mandatory. An identity GUI is part of the reTHINK web browser application, and allows the user the act of registering a Hyperty to select the identity he wants to associate with it.

The identity GUI can be presented in two distinct ways: by clicking in the settings button on the reTHINK web application, or by clicking in the button to register a Hyperty that will trigger the appearance of the identity GUI. This GUI allows the user to add new identities from a list of IdPs, choose an identity from the existing ones and remove identities from the list of previously registered identities. For example, when a user using the reTHINK web application click the button to register a new Hyperty the identity GUI appears listing all identities previously registered, and the option to add new ones. After the user selects the desired identity or chooses to obtain a new one, the Hyperty registration process continues.

### Runtime Sandbox

The hyperty runtime implements sandboxing mechanisms that ensure the correct isolation of client JavaScript code (i.e., Hyperties, ProtoStubs, and Applications). Isolation means that client code is confined to execute within the address space of an independent sandbox. As a result, sandboxes prevent potentially malicious code from interfering with client code instances in co-located sandboxes or from accessing external resources in the surrounding environment (e.g., files, network, etc.). Communication outside the sandbox is possible through well defined channels. Both sandboxing mechanisms and communication channels implemented by the hyperty runtime are available to the application programmer throught specific APIs and are dependent on the targeted platform.

For the browser platform, sandboxing is enforced by leveraging native mechanisms provided by the browser API. The core runtime components execute inside an iFrame. The iFrame implements the core sandbox by isolating the code of the core runtime from the main window in which the application Javascript code is executed. Application code is therefore prevented from accessing directly to the memory address space of the core runtime. Communication between application and core runtime is possible only through a single and well defined entrypoint which allows them to exchange messages: method postMessage(). Hyperties and protoStub downloaded from different domains are executed in independent Web Workers. Web Workers effectively isolate their internal states from each other and from the core runtime. The postMessage() method constitutes the only communication bridge between the these components.

For the standalone platform, the sandboxing mechanisms we employ are similar to the browser platform. The main difference is that, instead of using a browser, we leverage Crosswalk to support standalone applications. Crosswalk is an HTML application runtime that allows us to execute the hyperty runtime as native mobile applications in Android and iOS devices without the need to install a full-blown browser. Mobile applications only need to be bundled with both Crosswalk webviews and the hyperty runtime code. Since a Crosswalk webview implements a Chromium-based runtime, it can seamlessly execute the hyperty runtime code that was implemented for the browser platform.

## Hyperty Development

This document provides guidelines for developers of Hyperties. It is recommended that you have already read:

* [An Overview of the Hyperty Concept](hyperty.md)
* [An Overview of the Hyperty Messaging Framework](hyperty-messaging-framework.md)
* [An overview on how Hyperties cooperate with each other through a Data Synchronisation model called Reporter - Observer](p2p-data-sync.md)
* [An overview on the Hyperty Security and Trust Model](hyperty-trust.md)

### Hyperty Concept

The Hyperty Concept is introduced [here](hyperty.md) as a secure user associated [microservice](http://martinfowler.com/articles/microservices.html), which can be deployed either on a web runtime environment, on an end-user device or on a networked server. The main characteristics of a Hyperty include:

* *Modular*: A Hyperty should be a self-contained module providing reusable service logic implementions in form of a script (e.g. a JavaScript file)
* *User association* : A Hyperty instance is associated to a “User” (e.g. Human beings, physical objects, physical spaces, organizations) through an Identity, even if this User can be anonymous in some cases.
* *Data Synchronization Communication*: Hyperties interact with each other through data synch objects by using the Reporter – Observer communication pattern.
* *Protocol Agnostic*: Through the protocol-on-the-fly concept, Hyperties are network protocol agnostic. In other words, the data synchronization communication between Hyperties is not dependent on a specific network protocol. Communication is accomplished via a common data schema that describes the data synch objects used.
* *GUI independent*: Hyperty should not provide Graphical User Interfaces.
* *APIs*: A Hyperty can expose Javascript APIs within the runtime environment that can to be used by web applications

While designing and specifying service logics, it should be noted that Hyperties are not suitable for all use cases. In some case, making use of a simple reusable JavaScript file as library may suffice. The next section explains the criteria under which the decision to use a Hyperty or not could apply.

### Criteria to use the Hyperty Concept

These are guidelines to help developers decide if they should provide specific service logic as Hyperty or via a simple JavaScript library. Consider these as guidelines and not instructions. Before you embark on a new feature development, ask yourself the following questions:

* Is the feature delivery directly associated with a "User" (e.g. Human beings, physical objects, physical spaces, organizations)?
* Does the feature delivery involve communication between "Users"?
* Can the feature be delivered in cross-domain scenarios (i.e interoperability with other domains)?

If at least you have one "YES" answers to the above questions then most likely, you should go for the Hyperty Concept. Some examples are:

* Video Chat between human beings;
* a Human-being or a Data Analysis service collecting data from sensors in a Room.
* ..

The reTHINK Service Framework is what you want to look at next. The Service Framework provides APIs for developers to facilitate the development of Hyperties.

### Getting Started

So you have decided for the Hyperty Concept and now ask yourself where to start. This section describes the basic steps any developer needs to undertake to develop Hyperties. The source code of the Hello World Hyperty used as example, can be found [here](https://github.com/reTHINK-project/dev-hyperty/tree/master/src/hello-world)

1. Clone the [Hyperty Toolkit Repo](https://github.com/reTHINK-project/dev-hyperty-toolkit/blob/master/README.md), the [Hyperty Repo](https://github.com/reTHINK-project/dev-hyperty) and Install the toolkit:

npm install

Create a new folder for your Hyperty at src folder and work there.

1. Select or specify the descriptor of the Hyperty Data Objects in json-schema. Let's assume we define a simple Hello Hyperty object:

{  
 "$schema": "http://json-schema.org/draft-04/schema#",  
 "id": "HelloObject",  
 "type": "object",  
 "required": ["scheme","hello"],  
 "additionalProperties": false,  
 "properties": {  
 "scheme": {  
 "constant": "hello"  
 },  
 "hello": {  
 "type": "string"  
 }  
 }  
}

1. Then we can start developing the Hyperty Reporter of the previously defined Hello Hyperty Object. First the constructor where we are going to instantiate the Hyperty syncher. For that, we take as input parameter the Hyperty address (HypertyURL), the MiniBus used to receive and send data synchronization messages (bus), and any required configuration data.

constructor(hypertyURL, bus, configuration) {  
  
 let syncher = new Syncher(hypertyURL, bus, configuration);  
}

1. Then we prepare the creation of the Hyperty Hello Data Object by defining the Catalogue URL from where the previously defined Hyperty Data Object schema can be retrieved and initial data to be used in the creation:

\_this.\_objectDescriptorURL = 'hyperty-catalogue://example.com/.well-known/dataschemas/HelloWorldDataSchema';  
  
let hello = {  
 hello : "Hello World!!"  
};

1. The object is created and another Hyperty is invited to be an observer.

syncher.create(\_this.\_objectDescriptorURL, [invitedHypertyURL], hello).then(function(dataObjectReporter) {  
  
 console.info('Hello Data Object was created: ', dataObjectReporter);

1. From the Observer side, in order to observe the Hello Data Object, it has to subscribe it, by passing its URL and the Catalogue URL from where its schema can be retrieved. As soon as the subscription is accepted the most updated version of the Hello Data Object is received ...

syncher.subscribe(\_this.\_objectDescriptorURL, ObjectUrl).then(function(dataObjectObserver) {}  
  
 console.info( dataObjectObserver);

1. Any change performed by the Reporter in the the object.

dataObjectReporter.data.hello = "Bye!!";

1. ... will be immediately received by the Observer:

dataObjectObserver.onChange('\*', function(event) {  
 // Hello World Object was changed  
 console.info(dataObjectObserver);  
 });

1. Define the Hyperty descritor. The most important attributes are:

* constraints to be fullfilled by the Runtime, in this case the only constraint is to be executed in a browser: browser: true
* hypertyType or the schemes of the data objet: "hypertyType": ["hello"]
* the catalogue URL of the data object schemas supported by the Hyperty, in this case: "dataObjects": ["https://%domain%/.well-known/dataschema/HelloWorldDataSchema"]

The complete descriptor is:

{  
 "language": "javascript",  
 "signature": "",  
 "configuration": {},  
 "constraints": {  
 "browser": true  
 },  
 "hypertyType": [  
 "hello"  
 ],  
 "dataObjects": [  
 "https://%domain%/.well-known/dataschema/HelloWorldDataSchema"  
 ]  
}

1. develop an App to test your Hyperty by using the [template used in "dev-hyperty" repo](https://github.com/reTHINK-project/dev-hyperty/tree/master/examples/hello-world).
2. test and run your Hyperty,

start the toolkit for browser:

npm run start:browser

Open https://catalogue.localhost/ and accept certificate

Open https://localhost/ and select your Hyperty to be tested

Any changes performed in your Hyperty source code will be automaticaly loaded by the toolkit to be tested.

## Application Development

### Application vs Hyperty

A Hyperty is a module of software logic that is dynamically deployed in web runtime environments on end user devices, to execute session control and media flow management in a peer to peer manner. They are ready-to-use modules which are instantiated within the reTHINK runtime when required by the application. From the App developer perspective, Hyperties are similar to common Javascript libraries. Hyperty unique characteristics as described [here](hyperty.md) are transparent for the App developer. Through the Core Runtime, the required Hyperties and Protostubs are downloaded from the Catalogue server.

The Protostubs are the pieces of code which allows to interact with different messaging protocols. They are downloaded dynamically when the hyperty tries to reach a hyperty belonging to a different Content Service Provider (CSP) which uses a specific protocol for its messaging nodes.

This process is transparent for the developer of the final application, and of course, also for the final user of the application. If the Application requires some functionality or service provided by a Hyperty which has not been downloaded and instantiated yet, the runtime can get the code and instantiate it on the fly. The Application Developer only has to know in advance the Catalogue URL from where Hyperties are downloaded.

### How to use Hyperties

The usage of Hyperties is very simple and similar to the usage of any common Javascript library:

1- the App has to ensure the runtime is instantiated:

You can use through the npm module like:

npm install npm install github:reTHINK-project/dev-runtime-browser#master --save

import rethink from 'runtime-browser';

or you can load through the html script tag

<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Your App Name</title>  
</head>  
<body>  
 <script src="https://<domain>/.well-know/runtime/rethink.js"></script>  
</body>  
</html>

For both methods you need to do:

let config = {  
 "development": true,  
 "runtimeURL": "hyperty-catalogue://hybroker.rethink.ptinovacao.pt/.well-known/runtime/Runtime",  
 "domain": "hybroker.rethink.ptinovacao.pt"  
}  
  
rethink.install(config).then(function(result) {  
  
 runtime = result;  
   
 // your code  
  
}).catch(function(reason) {  
 console.error(reason);  
});

2- then load and deploy the required Hyperty from the Catalogue

runtime.requireHyperty('hyperty-catalogue://catalogue.example.com/.well-known/hyperty/HelloWorldReporter').then(  
  
 startUsingHpertyDeployed;  
  
 ).catch(function(reason) {  
 errorMessage(reason);  
});

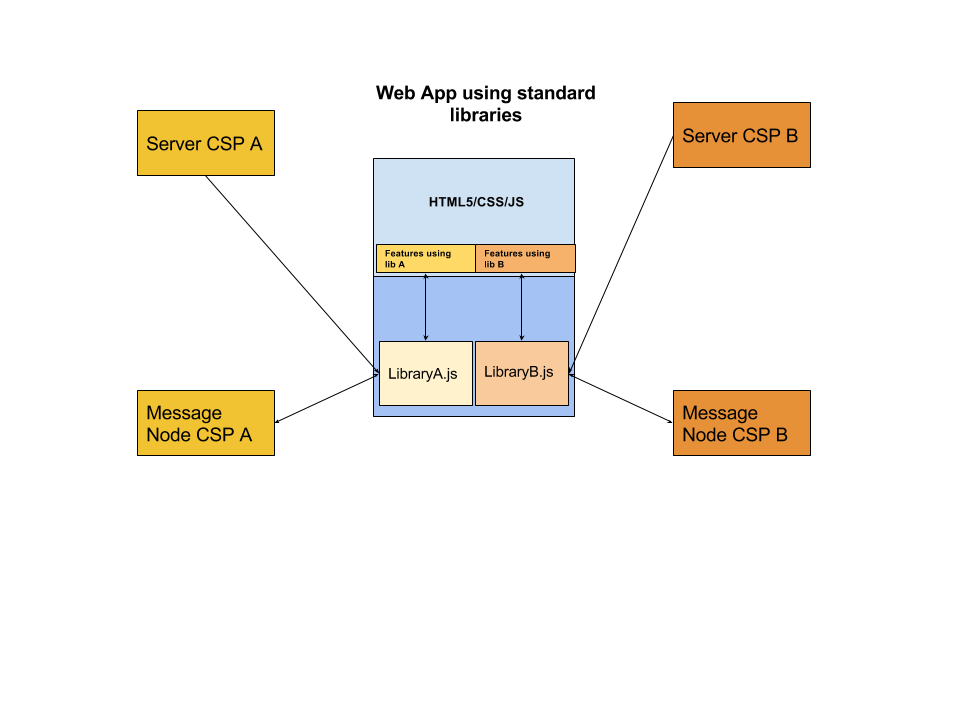
3- and invoke Hyperty functions from its API as a common Javascript Lib:

startUsingHpertyDeployed(hyperty){  
 hyperty.instance.hello();  
}

### How to adapt existing Applications

#### Standard web application

The diagram below shows how a standard application, which interacts with services provided by a service provider, currently looks like. There are several points which are not being considered such as Identity Management issues. If you need to interact with the service provided by CSP A, you need to use the library it provides, you need to authenticate to that specific service and you need to provide the logic in the Web Application to be able to produce and consume data from it.



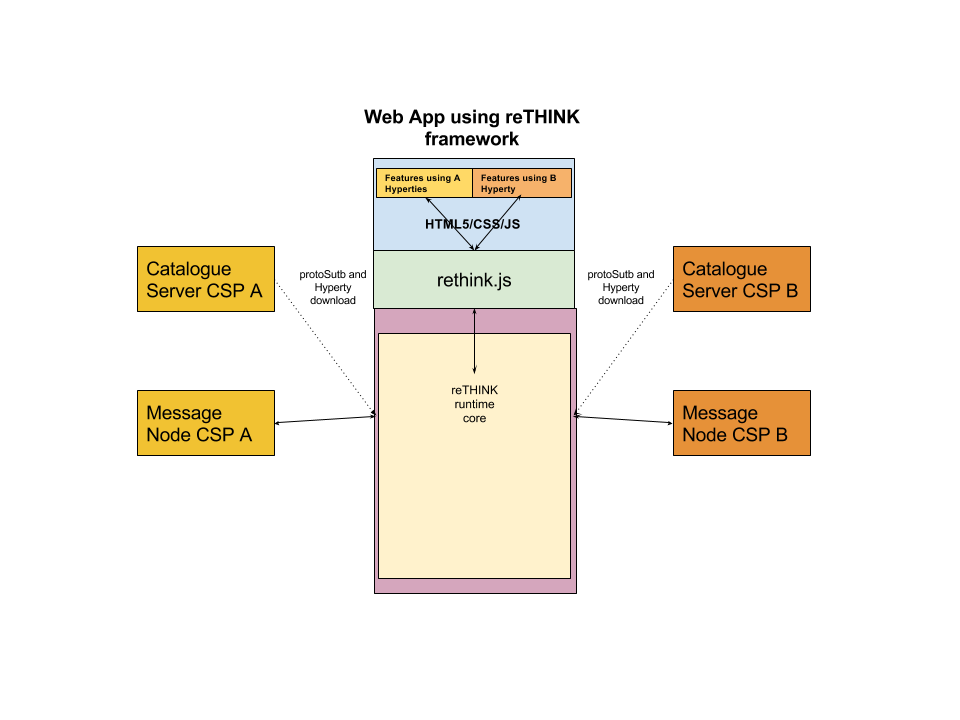
Standard App Diagram

If the Web App needs to interact with more than one CSP or service, typically it will need to download as many libraries as services and it may need to authenticate against the service provider with several identities and mechanisms. The code from different CSPs is running on the same sandbox so it can potentially interact with code from other libraries. This risk has been mitigated in the reTHINK architecture.

On the other side, we have only considered here Web Apps, but services provided by CSPs can be potentially used from a wide range of devices including constrained devices for M2M application. That is the reason why the reTHINK client libraries has been to be executed also in M2M scenarios where no web browsers are involved.

#### reTHINK application

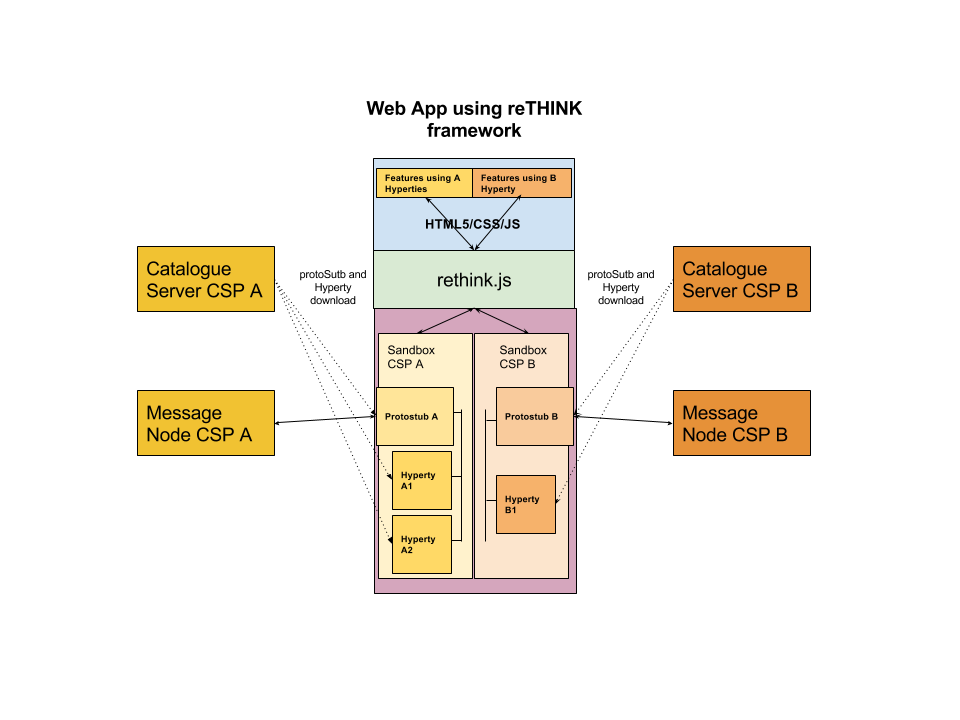
reTHINK web applications are similar to a common Web Application and have no impact at interface level, being impossible for an average user to distinguish between a standard application and a reTHINK application. However there are some settings available that gives the end user the power to make some decisions independently of the App consumed namely in terms of Identity to be used, Contact Lists and personal policies in general.



reTHINK App Diagram Black-box diagram

In the standard application, the developers needs to know in advance the libraries and versions it has to download to build the application. In reTHINK the functionality is provided by hyperties which are dynamically loaded in the runtime on-demand. The interaction with the Hyperties which can be executed in different sandboxes is done through commands and events from the Web application.

The diagram below depicts an Higher level picture of the Core Runtime architecture which is transparent to App developers. This diagram has been created for didactic purposes and it does not include all the elements. For example it does not contain the modules in charge of dealing with identity management.



reTHINK App Diagram

#### Relevant concepts

##### Security: sandboxes

The Web developer does not have to deal with low level details of the architecture. The sandboxes and the management of hyperties and protostubs is done by the core framework. The sandboxes allows to isolate code from different providers reducing the risk of suffering cross-site scripting attacks. There are special cases where the Hyperty needs to be executed in the same sandbox as the main web application, for example, when the Hyperty needs to access the WebRTC API. These hyperties which are executed in the same sandbox as the main app are called **App Hyperties**. The hyperties which are executed in a different sandbox (a Web Worker in the case of the runtime browser) are called **Service Hyperties**.

##### Compatibility: Protostubs

In a standard Web Application, the developers needs to know in advance with which services providers it will be necessary to interact. The number of protocols an application can speak is limited in implementation time and it can not change without modifying the code. In reTHINK the protocol-on-the-fly concept is used. If you need to interact with a service which uses protocol A, the framework will provide you on-the-fly a piece of code called protostub which will be executed in the right sandbox. This protostub will speak protocol A and it will expose a common API to the Hyperty Core Runtime. The Web Developer will not need to deal with this complexity.

##### Identity management.

The identity management is normally coupled to the service logic and there are many different standard protocols for authentication and Identity management that makes it harder to achieve interoperability between different services. The reTHINK framework decouples the service logic from the Identity Management logic and provides a common Identity Management API that is agnostic of protocols used. The protocol-on-the-fly mechanism is also used to provide you on-the-fly the right protostub (here called IDP Proxy) to interact with each Identity Provider selected by the end-user.

#### Admin GUIs

With rethink, you get some services like identity management or policy management. Some of this services can be configured by the user of your application. Rethink injects an iframe on your application that contains the administration GUIs. As an example, you can take a look at [policy manangement administration GUI](https://github.com/reTHINK-project/specs/blob/master/policy-management/user-interface.md).

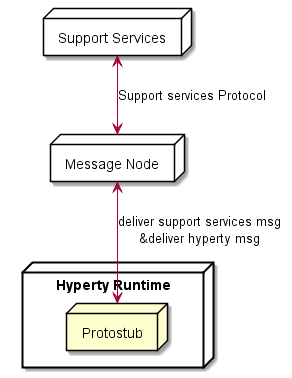
## Message Node and Protostubs design recommendations

The protocol stubs (AKA protostub) play a central role in the protocol on-the-fly concept. A stub is the piece of code that a reTHINK runtime downloads, instantiates and executes on-the-fly in order to exchange messages with a backend system from the own or a foreign domain. From the runtime's point of view the stub is the required "glue" between the reTHINK Message Model and the backend domain's protocols. The stub implements a well defined interface for the bi-directional asynchronous exchange of messages and hides all potential complexity of protocol translations for the interoperability with the backend domain.

The communication endpoint of a stub in a domains backend is the Messaging Node (MN). The MN and the stub build a unit that shall be designed and implemented together. The implementor of a protocol stub and the corresponding MN has to take some decisions:

1. is the Message Node able to route native reTHINK JSON Messages and no protocol translation is required in the protostub?
2. is it possible to add (with a reasonable effort), Message address allocation and Subscription Management support services functionalities as well as required connectors to interact with reTHINK back end services (including Domain Registry, Global Registry and Identity Management support services) to the Message Node?

If answers to above questions are yes, probably the most appropriate Message Node topology is the standalone message node model where all functionalities are provided by a single message node service and its associated protostub. This topology is used by [Vertx Message Node](https://github.com/reTHINK-project/dev-msg-node-vertx) and [NodeJS Message Node](https://github.com/reTHINK-project/dev-msg-node-nodejs).



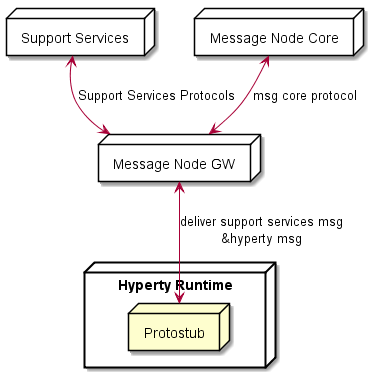
Message Node Standalone Topology

If answer to question 1 is negative, the potential complexity to be placed in the stub itself should be evaluated: Shall the stub do everything that is necessary to translate the protocol to the backend domains specifics? Or shall the stub just forward messages and let the MN perform the major tasks of the protocol translations? These are some hints that the developer should take into account:

1. Does the stub have dependencies to additional libraries? This might blow up the size of the stub and may complicate its deployment. Perhaps there is a chance to avoid some external dependencies?
2. Do any parts of the stub and it's dependencies underlie special restricting licenses or do they contain code that holds intellectual properties that shall be protected? Since the code is downloaded to an unknown, "strange" environment this might be an issue.
3. How much resources (network, processing, memory etc.) does the stub require? Are these requirements compatible with all addressed runtime platforms?

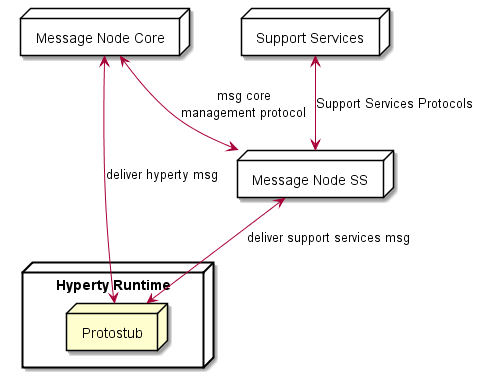
These questions shall be kept in mind, when the design decisions for a stub/MN couple are made. If one of the above questions can be answered with yes, then it might perhaps be an option to implement a basic stub that uses a simple connection mechanism like a WebSocket or similar to forward the reTHINK messages directly to the MN. In this case the MN itself would be responsible for the required protocol translations on the server side for its domain.

An example for such a situation is the [Matrix.org based MN](https://github.com/reTHINK-project/dev-msg-node-matrix) and its stub. The decision was made to let the stub just forward reTHINK messages and therefore keep it simple and small. The implementation of the Matrix.org client logic was done on the MN side. If the stub had implemented a full Matrix.org client, there would have been a set of dependent SDK-libraries with their own set of dependencies each. Furthermore a Matrix.org client produces additional overhead traffic that should be restricted to the MN internal system and therefore be kept away from the runtime device.



Message Node Standalone Topology

Another aspect to be taken into account is whether the Message Node is based on an existing Messaging solution that is already in production (e.g. core IMS, cloud messaging like pubnub, firebase, etc). In this case Messaging Node specific functionalities (allocation manager, subscription manager, registry connector) can be provided by a separated Support Service server, while Hyperty messages are delivered to the messaging core. This means, the protostub would handle two different protocols connections as shown below.



Message Node with separated Support Services

## Protostub specification

### Stub construction and activation

Stubs are provided by different vendors and developers and of course they have different naming conventions. In order to provide a common instantiation scheme a convention was defined additionally to the interface that ProtoStubs have to implement. The convention is that each stub modules must export a default activation function that is used by the runtimes to obtain a stub instance with a given set of parameters.

export default function activate(url, bus, config) {  
 return {  
 name: 'MatrixProtoStub',  
 instance: new MatrixProtoStub(url, bus, config)  
 };  
}

This activation function hides the internal naming and just returns an object that provides an implementation of the methods defined in the ProtoStub interface. The parameters of this function correspond directly to the previously described parameters of the Stub constructor.

### Integration with the Messaging Bus of the Runtime

Protocol stubs are tightly integrated with the messaging bus of the runtime. This integration is bi-directional. A reference to the messaging bus is provided as second paramenter of the stub constructor.

In order to receive messages from the runtime's messaging bus, the stub has to add itself as a listener. This can be done directly in the stubs constructor by adding such a code snippet:

this.\_bus.addListener('\*', (msg) => {  
 this.\_assumeOpen = true;  
 this.\_sendWSMsg(msg);  
});

Whenever the stub receives a message through this listener callback it invokes its own implementation specific functionality to potentially anaylyze, process and finally send an outgoing message to its MN (either the un-modified or a transformed message, depending on the MN type). In the code example above it would just forward them via a websocket connection.

For every message that is received from the MN, the stub forwards this message to the bus like shown here:

// parse msg and forward it locally to the runtimes messaging bus  
\_onWSMessage(msg) {  
 this.\_deliver(JSON.parse(msg.data));  
}

Also messages received from a MN might be anaylyzed, processed and potentially transformed by the Stub before they are delivered to the runtimes message bus, depending on the MN type.

### Auto connect mechanism

The stubs are expected to support an auto connect mechanism. This is because the runtime will not explicitely trigger the stub connection process itself by invoking a "connect" method. Instead it just sends messages via the messaging bus to the stub and assumes that the stub takes care of its own connection state.

A simple approach to implement this behavior in the stub is to maintain a flag that indicates whether the connection to the MN shall be kept open or not. This flag could be set to TRUE, as soon as the first message is being sent and to FALSE if the stub receives a "disconnect" command from the runtime. If for instance a network problem causes an interruption of the connection between stub and MN, the stub would attempt to re-connect as soon as the next message shall be sent.

This is, how the method to send a message could look like:

\_sendWSMsg(msg) {  
 if ( this.\_assumeOpen )  
 this.connect().then( () => {  
 this.\_ws.send(JSON.stringify(msg));  
 });  
}

If there is an explicit invocation of the "disconnect" method of the stub the stub will close the connection to the MN and set the keep alive flag off.

disconnect() {  
 this.\_assumeOpen = false;  
 this.\_ws.close();  
}

### Connection events (TODO: update to same event-types as used for the P2P stubs?)

In order to synchronize the state of the stubs with the runtime, each stub is expected to emit status messages to the bus whenever its connection state changes. The following method can be used to encapsulate this:

\_sendStatus(value, reason) {  
 let msg = {  
 type: 'update',  
 from: this.\_runtimeProtoStubURL,  
 to: this.\_runtimeProtoStubURL + '/status',  
 body: { value: value }  
 };  
 if (reason) {  
 msg.body.desc = reason;  
 }  
  
this.\_bus.postMessage(msg); }

The Message is of type "update". Its "from" attribute must be the runtimeProtoStubURL that was provided as first parameter of the constructor. The "to" attribute is the runtimeProtoStubURL extended with "/status". The expected "value" parameter is either "connected" or "disconnected". Optionally a reason can be specified that will be placed in the body of the message.

If the connection to the MN is established via a Websocket, then the sending of the corresponding event messages can be triggered in the "open" and "close" handlers of the Websocket.

\_onWSOpen() { this.\_sendStatus("connected"); }  
  
\_onWSClose() { this.\_sendStatus("disconnected"); }

### The ProtoStub API

The interface that a protocol stub has to implement is kept very small and simple by intent.

A protocolStub is constructed with a set of parameters that ensures that the stub can be uniquely identified, connect to its backend Messaging Node and can communicate with the messaging bus in the runtime.

new ProtoStub(runtimeProtoStubURL, busPostMessage, configuration)

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| runtimeProtoStubURL | URL.RuntimeURL | A URL allocated by the runtime that uniquely identifies this protocolStub. |
| busPostMessage | Message.Message (???) | The runtime BUS postMessage function to be invoked on messages received by the protocol stub. |
| configuration | ProtoStubDescriptor.ConfigurationDataList | Configuration data that is retrieved from the protocolStub descriptor. This data is implementation-specific and ensures that the Stub can address and connect its own Messaging Node. |

#### Methods

connect(identity)

The connect method establishes the connection between the protocol stub and the backend messaging node.

**Note:** The "connect" method will not be directly invoked by the runtime implementation. Rather it is expected that the stub maintains its connection state internally. Whenever the runtime intents to send a message via the postMessage method, the stub shall auto-connect to the Messaging Node and attempt to keep this connection open until it explicitly receives a "disconnect" invocation.

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| identity | IDToken | An optional identity token that can be used to authenticate this stub connection against the backend messaging node |

disconnect()

The disconnect method is used to explicitly disconnect a stub from its messaging node. Such a disconnect can be used to release and clean up resources in the stub and also on the backend side in the messaging node.

The connect method establishes the connection between the protocol stub and the backend messaging node.

\_filter(msg)

This message must be present in order to avoid message cycles for messages beeing sent out to the MN. It is used to filter out messages that have been sent already via the protocol stub, by checking a "via" field in the message body. If this field exists and equals the stubs runtimeProtoStubURL then this message must return false, otherwise true.

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| msg | Message | The original message from the MessageBus |

*Returns:*

**true**, if the message does not contain a body.via field matching the own runtimeProtoStubURL

**false** otherwise

\_deliver(msg)

This message must be present in order to perform the actual delivery of a message towards the MN. It must insert the body.via field into the message, which existence is checked in the "*filter" method.*

*Parameters:*

|  |  |  |
| --- | --- | --- |
| name | type | description |
| msg | Message | The original message from the MessageBus |

#### Events

A protocol stub emits events to communicate its own connection state to the runtime. Whenever the stub gets connected or disconnected, it uses the "busPostMessage" to send a message to the runtimes message bus. These Event messages are defined as follows:

{  
 "type": 'update',  
 "from": runtimeProtoStubURL,  
 "to": runtimeProtoStubURL + '/status',  
 "body": {  
 "value": "connected|disconnected"  
}

The runtimeProtoStubURL is the URL that was provided as first parameter of the Stub constructor. The value in the message body is either "connected" or "disconnected".

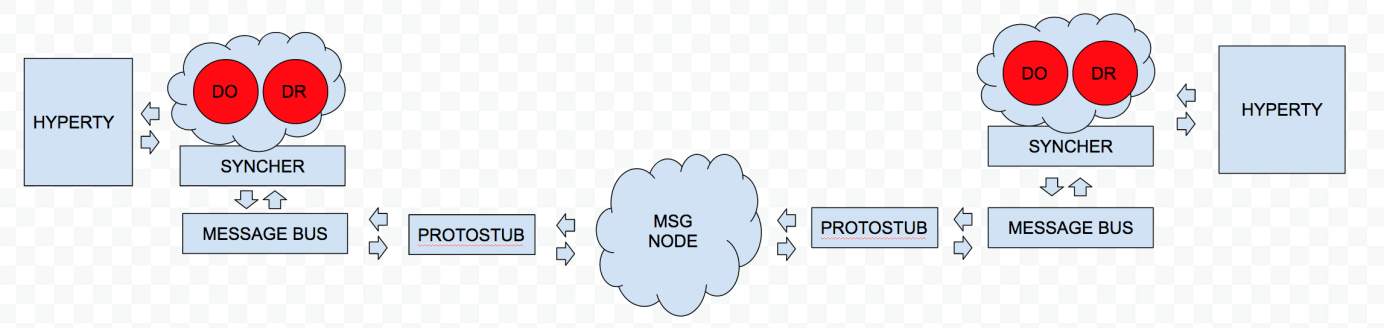
## IdP Proxy Development

*provide here guidelines to be used to develop IdP Proxies. Use* [*protostub dev*](https://github.com/reTHINK-project/specs/blob/master/messaging-framework/stub-specification.md) *as reference. The current* [*IdP Proxy template*](https://github.com/reTHINK-project/dev-protostubs/blob/master/src/idpproxy/ProxyStubTemplate.js) *could also be useful.*

## Interworking stub development

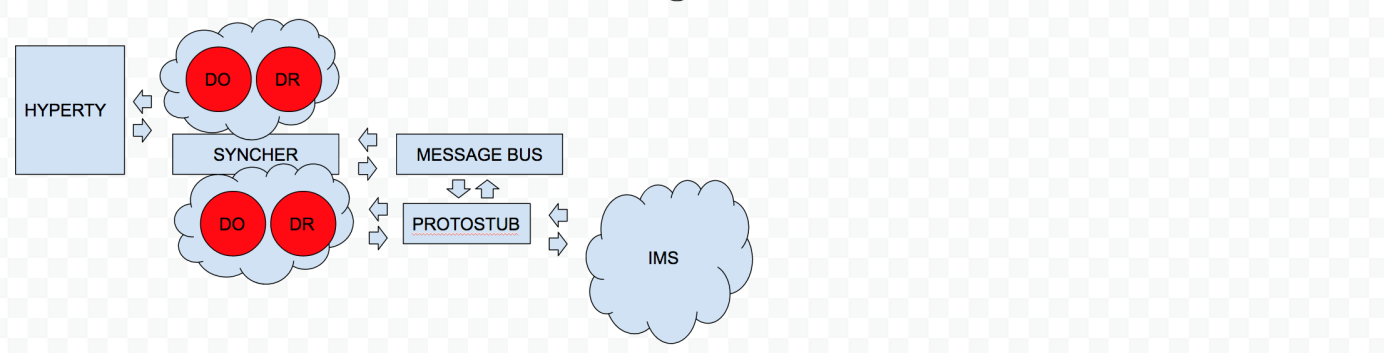
In a general way, there are not many differences between developing a stub or an interworking stub. The same [specs](https://github.com/reTHINK-project/specs/blob/master/messaging-framework/stub-specification.md) still apply. The big difference could be a conceptual one.

### Stub conceptual view:



Stub conceptual view

### Interworking stub conceptual view:



Interworking stub conceptual view

As you can see on the diagrams an interworking stub is a stub that communicates with the hyperty using the [Data Synchronisation Reporter - Observer communication mechanism](https://github.com/reTHINK-project/specs/blob/master/messaging-framework/p2p-data-sync.md). It still has to listen to the message bus in order to catch the message from the hyperty but then it subscribes to the syncher and gets the dataobjectreporter with the data. In this way, we can keep the hyperty interoperable with other protostubs without changing it.

### Communication flows

**Communication from the hyperty to the protostub:**

miniBus.addListener('\*', (msg) => {  
 switch (msg.type) {  
 case 'create':  
 if(this.\_filter(msg) && msg.body.schema) {  
 let dataObjectUrl = msg.from.substring(0, msg.from.lastIndexOf('/'))  
 this.\_syncher.subscribe(this.schema, dataObjectUrl)  
 .then(dataObjectObserver => {

**Communication from the protostub to the hyperty:**

miniBus.addListener('\*', (msg) => {  
 switch (msg.type) {  
 case 'register':  
 this.\_connection.connect(msg.body.identity.access\_token)  
 this.source = msg.body.source  
 this.schema = msg.body.schema  
 break

TODO: Implement a way to register the hyperty in the legacy protostub without changing it.

**Communication from the protostub to a legacy domain**

A protostub is an adapter so it implements and isolates the communication with third parties. The implementation will be different from one to another.

For the sake of cleanness and organization, in the examples, you can find a new object, ConnectionController, responsible for this communication.

### Interworking identity proxy development

No special consideration are required. [Identity proxy development](https://github.com/reTHINK-project/specs/blob/master/trust-management/idp-proxy-development.md)

# Specification changes

This chapter provides a short summary of major changes done in the specification of the Runtime and Message Node, according to Phase 2 implementation feedback.

## Hyperty Runtime

The Hyperty Runtime supports the execution of Hyperties. The Hyperty Runtime architecture follows a security by design approach since it was highly influenced by a careful security analysis. Different types of components with different origins are deployed and executed in isolated sandboxes. As highlighted in Figure 19, comparing with the original specification, two new components were added to the Hyperty Core Runtime:

The **Address Allocation Manager** was added to provide a consistente and more complete Address Allocation management functionality.

The **Discovery** was added to support cross domain discovery features through Graph Connector, Global Registry Support Service and Discovery Support Service

The **QoS User Agent** that manages network QoS was removed from the Core Runtime and is provided as a Service Framework lib.

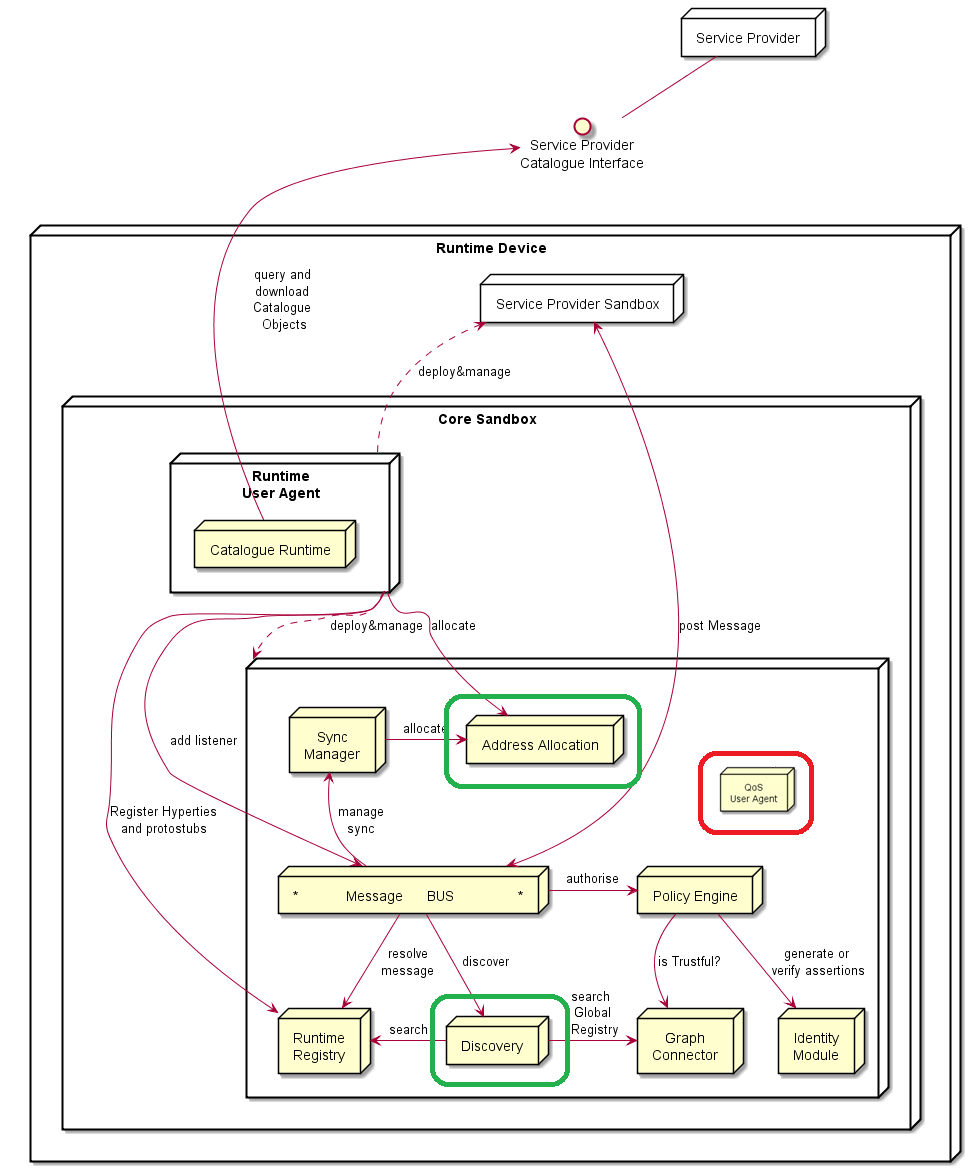


Figure – Changes in the Hyperty Runtime Core

## Messaging Node

The Messaging Node main functionality is to route Hyperty Messages among Hyperty Runtimes. As shown in Figure 20, there are no changes to the original phase 2 specification.

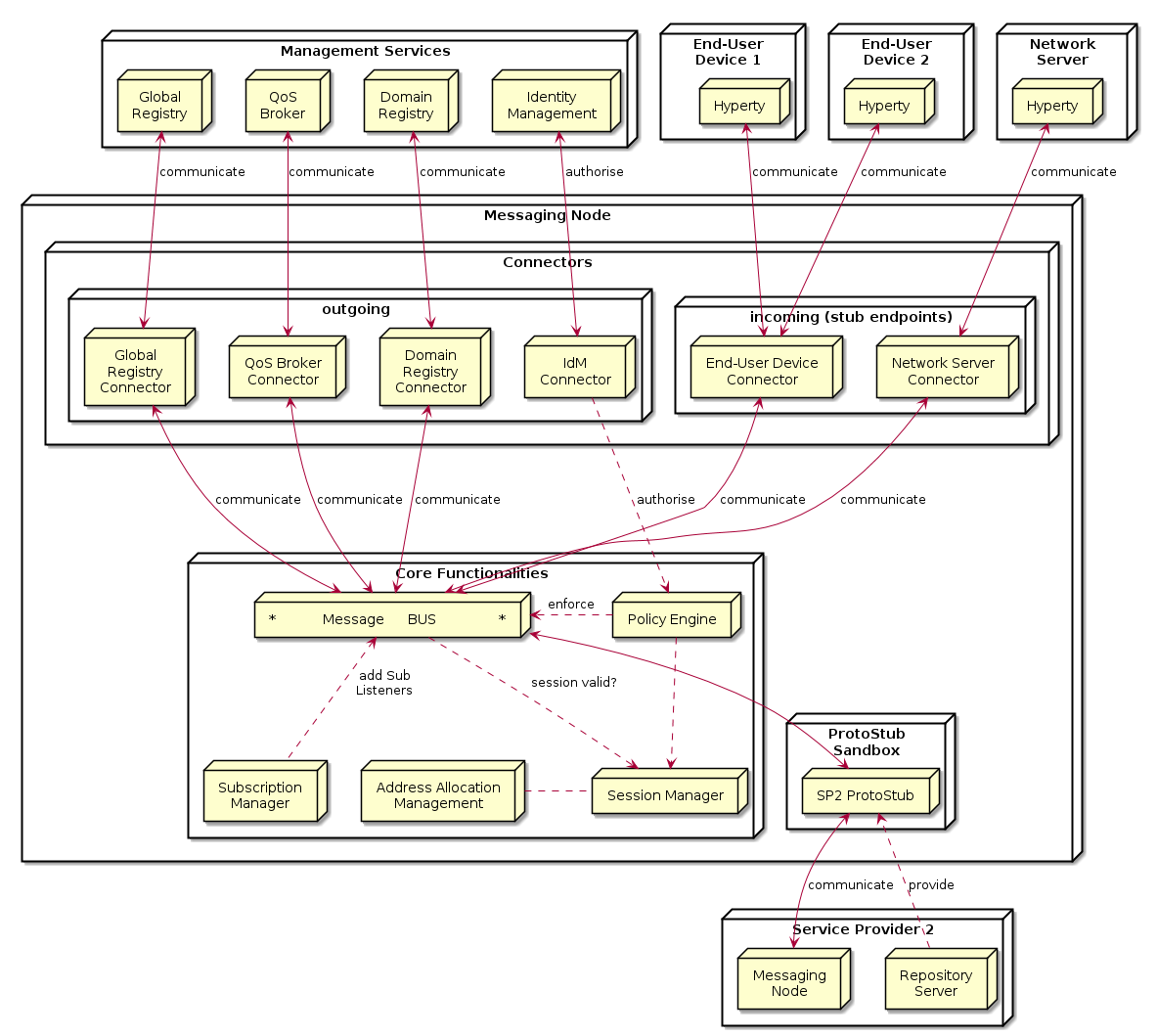


Figure – Changes in the Messaging Node Architecture

# References

1. https://github.com/reTHINK-project
2. Deliverable D2.3, “Final Design Architecture”, 31-10-2016.
3. Deliverable D3.5, “Hyperty Runtime and Hyperty Messaging Node Specification - Revision”, 30-11-2016.
4. Deliverable D4.5, “Management and Security features specifications - Revision”, 30-11-2016.
5. Deliverable D4.3, “Implementation of Governance and identity management components for Phase 2”, 28-02-2017.
6. <https://github.com/reTHINK-project/dev-runtime-core>
7. <https://github.com/reTHINK-project/dev-runtime-browser>
8. <https://github.com/reTHINK-project/dev-runtime-nodejs>
9. <https://github.com/reTHINK-project/dev-msg-node-vertx>
10. <https://github.com/reTHINK-project/dev-msg-node-matrix>
11. <https://github.com/reTHINK-project/dev-msg-node-nodejs>
12. <https://github.com/reTHINK-project/dev-service-framework>
13. 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.
14. <https://github.com/reTHINK-project/dev-catalogue>
15. <https://github.com/reTHINK-project/dev-registry-domain>

1. <http://martinfowler.com/articles/microservices.html>
2. <https://en.wikipedia.org/wiki/Edge_computing>
3. https://en.wikipedia.org/wiki/Fog\_computing
4. https://developers.google.com/v8/
5. https://nodesecurity.io/resources