

**Deliverable D3.5**

**Hyperty Runtime and Hyperty Messaging Node Specification - Revision**

|  |  |
| --- | --- |
| Editor: | Paulo Chainho, PT Inovação |
| Deliverable nature: | (R) Document, Report |
| Dissemination level: (Confidentiality) | Public (PU) |
| Contractual delivery date: | 30/09/2016 |
| Actual delivery date: | 30/11/2016 |
| Suggested readers: | Service providers’ designers and developers |
| Version: | 1.0 |
| Total number of pages: | 53 |
| Keywords: | Fixed: Communication networks, media, information society  Free: Hyper-linked entities, Future Internet architecture, Trustful communications, Social trustful networks, WebRTC, Independent Identity, IdP, Peer-to-peer communications,H2H, M2M, Graph ID |

***Abstract***

This Report contains reviews the specification of reTHINK Core Framework components comprised by the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. The core of this document is dedicated to highlight the main changes done in the specification in order to meet the main objectives for phase 2, namely, Peer to Peer messaging delivery, interworking with legacy domains, control of quality of service and multiparty WebRTC communications.

[End of abstract]

Disclaimer

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

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

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

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

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

Impressum

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

Short project title: **reTHINK**

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

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

Document number and title: **D3.5 Hyperty Runtime and Hyperty Messaging Node Specification - Revision**

Editor: **Paulo Chainho, Altice Labs**

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

Copyright notice

2016 Participants in project RETHINK

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

Executive summary

This document provides an update of the specification needed by developers to evolve reTHINK Core Framework in order to meet Phase 2 objectives. This document takes as input the feedback collected from phase 1 trials and hackathons (D6.3 – Assessment Report) as well as the update done in data models and interfaces definitions from deliverable D2.3 (Final design of the Architecture). This report complements deliverable D4.4 that updates on Management and Security features specifications. The core of this document is dedicated to highlight the main changes done in the specification in order to meet the main objectives for phase 2, namely, Peer to Peer messaging delivery, interworking with legacy domains, control of quality of service and multiparty WebRTC communications. The document only provides a summary of the full Core Framework specification. The detailed specification can be found in the Github project repository.

List of authors

|  |  |
| --- | --- |
| Company | Author |
| PTIN | Paulo Chainho |
| Fraunhofer | Marc Emmelmann |
| Quobis | Antón Román Portabales |
| Apizee | Jamal Bolmal |
| Deutsche Telekom | Steffen Druesedow |
| Orange Labs | Simon Bécot |
| INESC-ID | Ricardo Lopes Pereira  Ricardo Chaves  Nuno Santos |
|  |  |

Table of Contents

[Executive summary 3](#_Toc466652017)

[List of authors 4](#_Toc466652018)

[Table of Contents 5](#_Toc466652019)

[Abbreviations 7](#_Toc466652020)

[1 Introduction 9](#_Toc466652021)

[1.1 Objectives and Overview 9](#_Toc466652022)

[1.1 Structure 10](#_Toc466652023)

[2 Core Framework Specification update 11](#_Toc466652024)

[2.1 Runtime Specification Update 11](#_Toc466652025)

[2.1.1 Functional Summary 11](#_Toc466652026)

[2.1.2 Main Changes performed in Phase 1 13](#_Toc466652027)

[2.1.3 Main Specification Updates for Phase 2 (around 5-10 pages) 14](#_Toc466652028)

[2.2 Messaging Framework Specification Update (Steffen) 15](#_Toc466652029)

[2.2.1 Hyperty Messaging Framework overview 15](#_Toc466652030)

[2.2.2 Protocol on-the-fly (protofly) and Protostubs 17](#_Toc466652031)

[2.2.3 P2P Data Synchronisation: Reporter - Observer Model 18](#_Toc466652032)

[2.2.4 Messaging Node 19](#_Toc466652033)

[2.2.5 Connectors 22](#_Toc466652034)

[2.3 Service Framework Specification Update (Marc) 27](#_Toc466652035)

[2.3.1 Service Framework Specification Overview 27](#_Toc466652036)

[2.3.2 Service Framework Specification updates 28](#_Toc466652037)

[2.4 Trust Management and Security 31](#_Toc466652038)

[2.4.1 User Identity 32](#_Toc466652039)

[2.4.2 Runtime Sandbox 39](#_Toc466652040)

[3 New Features specification 41](#_Toc466652041)

[3.1 P2P Message Delivery 41](#_Toc466652042)

[3.2 Quality of Service Control 42](#_Toc466652043)

[3.2.1 QoS in reTHINK 42](#_Toc466652044)

[3.2.2 reTHINK TURN services 43](#_Toc466652045)

[3.2.3 Last Hop Connectivity Broker 44](#_Toc466652046)

[3.3 Multiparty WebRTC Communication 46](#_Toc466652047)

[3.3.1 Multiparty Topologies 46](#_Toc466652048)

[3.3.2 Solution for Star Topologies 47](#_Toc466652049)

[3.4 Interworking with Legacy Services 48](#_Toc466652050)

[3.4.1 Introduction 48](#_Toc466652051)

[3.4.2 Applications 48](#_Toc466652052)

[3.4.3 Interworking strategy proposal 49](#_Toc466652053)

[3.4.4 Technical implementation 50](#_Toc466652054)

[3.4.5 IWstub implementation 51](#_Toc466652055)

[3.4.6 Token based authentication techniques 51](#_Toc466652056)

[3.4.7 IMS interworking 51](#_Toc466652057)

[3.4.8 IWstub Extensibility Considerations 52](#_Toc466652058)

[2 References 53](#_Toc466652059)

Abbreviations

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

# Introduction

## Objectives and Overview

Project reTHINK proposes a radical transformation on how real time communication services are thought. reTHINK concepts and architecture represents a significant paradigm change for the communication services domain. The reTHINK approach enables the fulfilment of real-time communications requirements that so far have been considered impossible to achieve: trustful identities, interoperable endpoints, agility of introducing new services, and fast moving innovation.

This report provides an update of the specification of reTHINK Core Framework provided in D3.1 [4] for phase 2. This report complements deliverable D4.4 which also provides an update on Management and Security features specification [3]. Thus, and according to reTHINK Architecture [38], the scope of this report includes an update on the specification of the Messaging Node providing reTHINK Messaging Services and the specification of the Hyperty Runtime that will be included in User Devices and Application Servers to deliver User Hyperties and Network Side Hyperties (See Figure 1).



Figure 1 - Specification Scope

The current report highlights the mains specification changes required to support new features planned for phase 2, having the full detailed specification in a dedicated Github repository [5].

The update of the reTHINK Core Framework specification provided in this report, is compliant with of the update of reTHINK Data Model, Hyperty Management interfaces, Stream Interface and Messaging Interface designed in D2.3 [1].

The specification reported in this deliverable, provides the basis for the implementation tasks of phase 2 but it is expected to be adjusted and to be completed along the implementation phase.

## Structure

This report starts with the current introduction chapter and the core part of this report is located in Chapter 2, which highlights the main changes done for phase 2 in the Core Framework. In chapter 3, a more detailed specification is provided for the new features planned for phase 2.

# Core Framework Specification update

## Runtime Specification Update

This section contains an overview of the Hyperty Runtime specification, where Hyperties are executed. It provides a summary of functionalities provided, main changes performed in phase 1 since the initial specification and the specification update for phase 2.

### Functional Summary

The main functionality provided by the Runtime is the safe execution of Hyperties. Different types of components (see Figure 1) with different origins are deployed and executed in isolated sandboxes including Runtime Core Components, Hyperties and Protostubs.

The Runtime Core functionalities are comprised by:

The **Runtime User Agent** that manages the lifecycle of the Runtime itself as well as of Hyperties and Protostubs, including the deployment, update and removal of these functionalities.

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

The **Runtime Catalogue** manages the descriptors of deployable components and Hyperty Data Object schemas that are downloaded from the Service Provider Catalogue.

The **Message Bus** supports local message communication in a loosely coupled manner between Service Provider sandboxes including Hyperty Instances, Protocol Stubs and Policy Enforcers.

The **Runtime Identity Module** manages ID and Access Tokens required to trustfully manage Hyperty Instances communication including the generation and validation of Identity assertions.

The **Policy Engine** provides Policy decision and Policy Enforcement functionalities for messages intercepted from the Message BUS.

The **Sync Manager** handles data synchronisation streams used by Hyperties to communicate each other.

The **QoS User Agent** Manages network QoS in the runtime.

The **Graph Connector** is a local address book maintaining a list of trustful communication users.

The only important Runtime APIs to be used by Applications and Hyperties are:

* Runtime User Agent APIs that are used by Applications to instantiate Hyperties and Protostubs
* Message Bus APIs that are used by Hyperties to send and receive messages

The remaining APIs are internal to the core runtime, thus developers of Hyperties and Applications have not to deal with them.

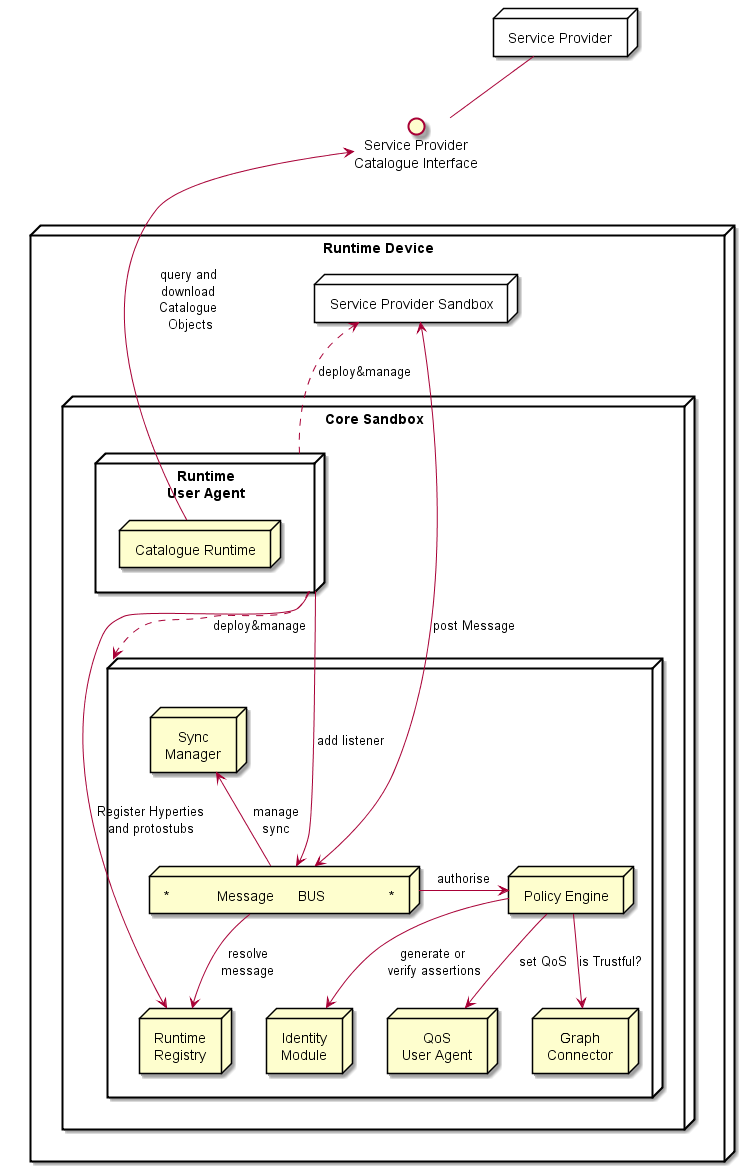


Figure 1 – Runtime Architecture

For further information, the full specification is publicaly available in the Github:

* the functional specification at [7] .
* the Runtime Main procedures at [11] .
* the detailed definition of messages at [10].

### Main Changes performed in Phase 1

As highlighted in Figure 2, comparing with the original specification, the **Sync Manager** was addedin in order to support Hyperty Data Objects synchronisation by handling creation and subscriptions requests.

In addition, the **CatalogueProtostub** from the **Runtime User Agent** was renamed to **Runtime Catalogue** and has a few more functionalities. It manages the descriptors of deployable components and Hyperty Data Object schemas that are downloaded from the Service Provider Catalogue via the Catalogue Service interface. The Runtime Catalogue ensures synchronisation with Back-end Catalogue servers.

The **QoS User Agent** and the **Graph** **Connector** will be implemented / integrated in the Runtime during phase 2.

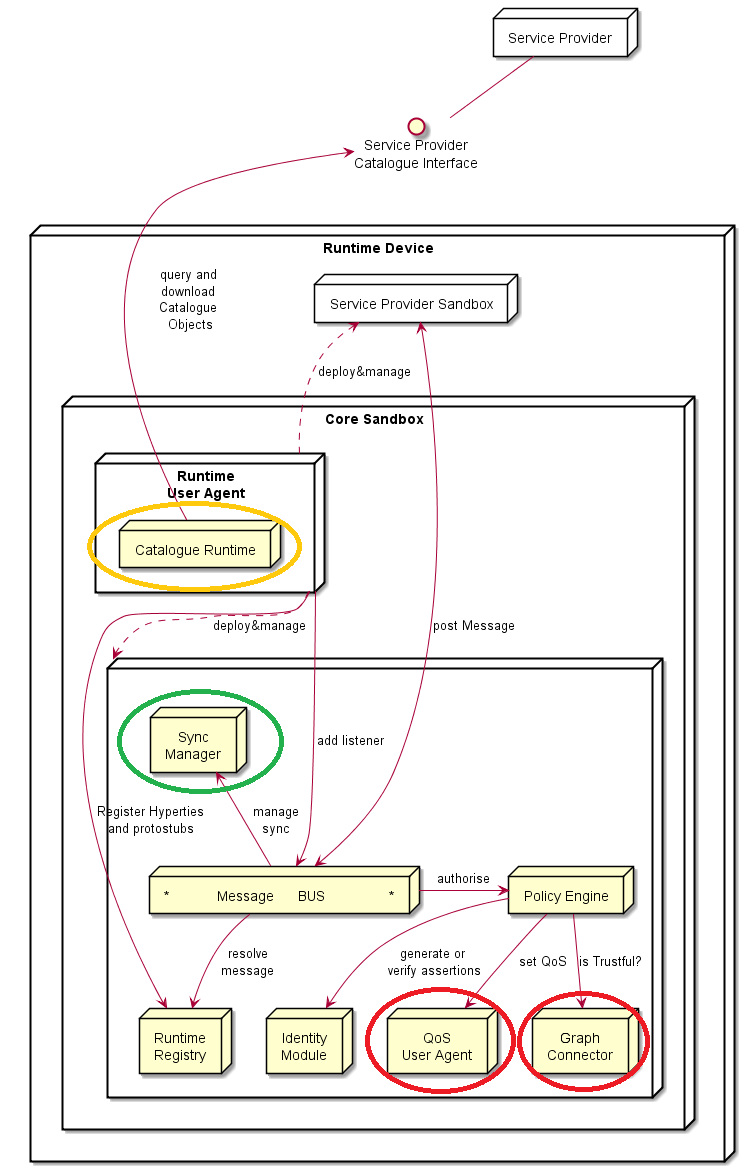


Figure 2 - Changes in the Hyperty Runtime Core

The specification of the Runtime Procedures was further elaborated and detailed, notably by fully specifying the messages to be used for each procedure [11].

### Main Specification Updates for Phase 2

For phase 2, no new Runtime Components are planned but only the extension of the components already introduced in Figure 2, namely:

* The Runtime Catalogue has to be extended in order to support some new fields introduced in the Catalogue Data Model required to support P2P Message delivery new functionality (section 3.1) as well as interworking with legacy domains (section 3.4).
* The Runtime User Agent has to be extended in order to support the management of P2P Protocol Stubs (section 3.1). In addition the Runtime User Agent has also to be extended in order to support the possibility to reuse Hyperty and Data Objects addresses as well as the sharing of the same Runtime instance among several Application instances.
* The Runtime Registry has to be extended in order to support new fields introduced in the Registry Data Model required to support some of the new features introduction in chapter 3. In addition the resolution of addresses was extended in order to support P2P Protocol Stubs (section 3.1).
* The Runtime Identity Module has to be extended in order to support the management of Identities from legacy domains (section 3.4) as well as management of Identities in a Runtime Edge Server implemented by the NodeJS Runtime.
* The Sync Manager has to be extended to support P2P Message Delivery (section 3.1) as well as the possibility to re-use data object addresses.

## Messaging Framework Specification Update (Steffen)

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

Since the protocol-on-the fly concept is used together with the message model defined in D2.2 [13], it is not required to specify in detail the Messaging Node APIs to guarantee interoperability between different domains.

### Hyperty Messaging Framework overview

Hyperties cooperate each other with a Resource Oriented Messaging model implemented by a simple Messaging Framework. The Hyperty Messaging Framework, supports different messaging patterns including publish/subscribe and request/response messaging patterns. The higher level [Reporter - Observer communication pattern](p2p-data-sync.md) works on top of these basic messaging patterns.

The Message delivery is based on a simple message Router functionality that performs a lookup for listeners registered to receive the Message (the ["Message.to" Header field](https://github.com/reTHINK-project/dev-service-framework/blob/develop/docs/datamodel/message/readme.md#to) is the only information looked up for). The Message is posted to all found listeners, which can be other Routers or end-points (Hyperties). Thus, the Hyperty Messaging Framework is comprised by a network of Routers where each Router only knows adjacent registered Routers or end-points.



Figure - Hyperty Messaging Delivery Network

Listeners are programmatically registered and unregistered by Routing Management functionalities, which decide the listeners to be added according to a higher level view of the Routing Network.



Figure - Hyperty Message Routing Management

The Messaging Framework works at three layers:

At the Runtime Sandbox level where Hyperties 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](#protocol-on-the-fly-protofly-and-protostubs), 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](#protocol-on-the-fly-protofly-and-protostubs). 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.



Figure - Adhoc Messaging Oriented Middleware Routing Layers

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

### 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 and instantiated in 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.



Figure - Protocol on-the-fly and Protostubs

### P2P Data Synchronisation: Reporter - Observer Model

Hyperties cooperate each other through a Data Synchronisation model called Reporter - Observer.

The usage of Observer models in Web Frameworks (eg [ReactiveX](http://reactivex.io/documentation/observable.html)) is becoming very popular. However, current solutions require server-side databases that are not compliant with edge computing Hyperty principles.

Hyperty Reporter - Observer communication pattern goes beyond current solutions by using a P2P data stream synchronisation solution for JSON Data Objects, here called Hyperty Data Object or Sync Data Object. To avoid concurrency inconsistencies among peers, only one peer has granted writing permissions in the Hyperty Data Object - the Reporter hyperty - and all the other Hyperty instances only have permissions to read the Hyperty Data Object - the Observer hyperty.



Figure - Reporter-Observer Communication Pattern

#### Hyperty Data Object URL address

The Hyperty Messaging Framework allocates to each new created Hyperty Data Object a Global Unique Identifier URL that is independent from the Hyperty instance creator and from the Hyperty Runtime, in order to support mobility of the Data Object between different Hyperty Runtimes and also to support delegation of the Reporter role to other Hyperty instances. However, at this point Reporter delegation is only supported between Hyperty instances from the same domain.

#### Hyperty Data Object Schema

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

* [Connection Data Schema](../datamodel/connection) : Hyperties 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/communication) : Hyperties 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) : Hyperties 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 Hyperties are able to write in the Sync Data Object, the Parent - Child Data Sync Objects is 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.



Figure - Parent - Child Sync

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

### Messaging Node

#### Messaging Node Functional Architecture

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

Compared to the phase 1 version two additional components have been added to the Messaging Node Architecture, a Subscription Manager as well as a connector for the Global Registry.

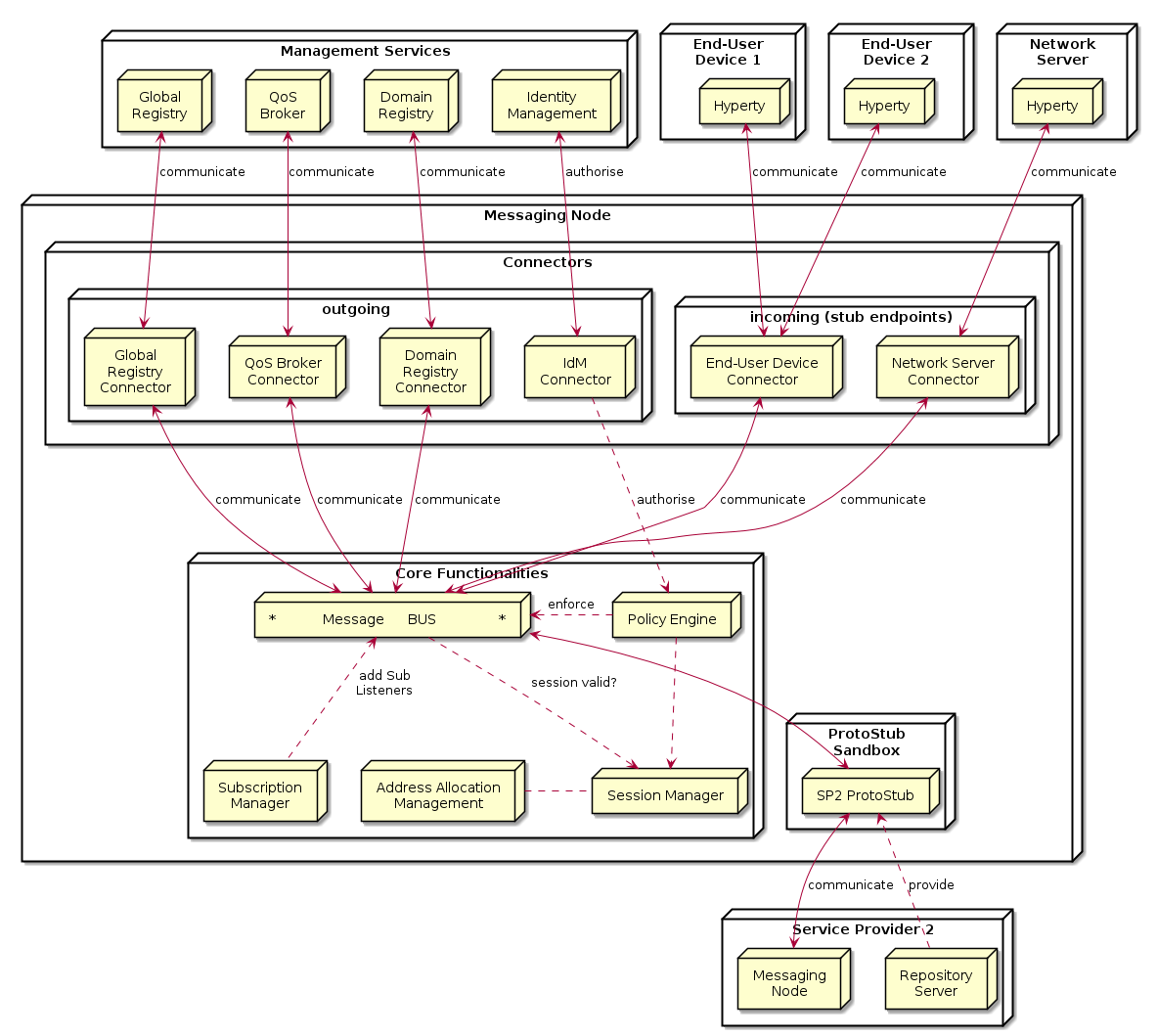


Figure - Messaging Node Architecture

##### Core Functionalities

###### Message BUS

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

###### Policy Engine

Message nodes are responsible for the interaction of runtimes in their own domain with runtimes from foreign domains by offering protocol stubs to these external runtimes. However the operators of a certain domain need a mechanism to control these domain interactions and to potentially block or limit certain combinations of message exchange.

In order to achieve this, a MN must provide a hook in the message flow that allows to apply policy based decisions to the routing. These policy must be manageable by the domain Policy Manager.

Therefore Policy Engine provides Policy decision and Policy Enforcement functionalities at Domain level for incoming and outgoing messages in cooperation with authentication and authorisation provided by Identity Management functionalities. It also provides authorisation / access control to the Message BUS.

###### Session Management

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

###### Address Allocation Management

As soon as an entity in a runtime wants to be accessible from another runtime, this entity must be addressable. Since a MN is the central message routing point for a domain it is the MNs task to create these addresses and to assign them to the requesting runtime. The resulting internal allocation table stores the relation of the allocated addresses to the stub connections and enables a proper routing of messages between the runtimes.

The Address Allocation Management functionality handles the allocation of messaging addresses to instances of Hyperties and Synchronization Data objects in cooperation with Session Management when users connect to the domain. These addresses are valid for at least the lifetime of a session. They are used by the Subscription Manager and Message BUS to take routing decisions. The specification of the messages to manage address allocations can be found at [Address-allocation-messages](https://github.com/reTHINK-project/specs/blob/master/messages/address-allocation-messages.md).

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

Address Allocation Management functionality must have listeners to receive messages for the following addresses:

domain://msg-node.<sp-domain>/address-allocation

##### Subscription Manager

A core concept in the reTHINK architecture is that Hyperties interact with each other by exchanging and synchronizing their managed data objects based on the [Reporter - Observer pattern](p2p-data-sync.md). The MN supports this concept by allowing observers (Hyperties, running in one or more runtimes) to subscribe for changes of certain allocated data object urls deployed in other runtimes. Whenever a Hyperty runtime reports a change in a monitored data object it sends a change message to the MN. The "to" address of this message will just be the allocated address of the updated data object, not the address of the subscribers directly.

In order to route such object change messages to the subscribed listeners, the MN has to maintain an own list of subscribers per allocated data object. Therefore the MN must intercept subscription messages.

A more detailed specification can be found at [Data sync messages](https://github.com/reTHINK-project/dev-service-framework/blob/d3.2-working-docs/docs/specs/messages/data-sync-messages.md). The Subscription Manager functionality must have listeners to receive messages for the following addresses:

domain://msg-node.<sp-domain>/sm

##### Protocol Stub

The basic operation mode of a MN is that it is connected by runtimes directly via the provided protocol stubs. A message received from one runtime will be forwarded to another runtime which must also be directly connected through a stub. This is a classic "triangular" messaging architecture. The triangular message flow looks like this:

RuntimeA --> StubB --> MN-B --> RuntimeB

For future iterations of the reTHINK messaging it is intended that the MNs also support a "trapezoid" architecture for inter-domain communication. In contrast to the triangle, each runtime will only have a connection with the MN from its own domain. If one runtime wants to send a message to another one from another domain, it will not be runtime itself that downloads and instantiates the stub of the foreign domain. It would be the domains MN instead that has to do this.

The trapezoid message flow will then look like this:

RuntimeA --> StubA --> MN-A --> StubB --> MN-B --> RuntimeB

and vice versa. This implies that in future versions the MN has to implement a module for the proper downloading, instantiation and operation of foreign stub in a sandboxed environment, just like the runtimes are already doing it.

From the viewpoint of the MN, each stub represents one runtime. It is the task of the MN to identify a stub connection, and to manage the life-cycle of the assigned server side resources. The actual "handshake mechanisms" between the stub and the MN are left implementation specific.

A valid method for the MN to identify a stub connection is to use the "runtimeURL", which each stub is constructed with in the runtime. If the stub provides this url during the connection handshake procedure, then the MN can identify the stub/runtime, even after a potential re-connect, e.g. due to temporary loss of network connectivity.

It is the responsibility of the MN to release resources if the "disconnect" method was invoked on the stub . This is the official indication that the runtime does not need this stub connection anymore and it has released the stub. In the alternative case, that a stub was not sending messages for a longer period, but was also not officially disconnected, it is up to the MN implementation to run a kind of garbage collection mechanism to release stale resources.

For more detailed specification of Protocol Stubs please refer to [Protocol stub specification](https://github.com/reTHINK-project/specs/blob/master/messaging-framework/stub-specification.md).

### Connectors

Connectors implements protocol stacks used to interoperate with external elements from the domains,. In general there are connectors for outgoing access to components inside or outside the own domain and on the other hand endpoints listening for incoming connections from external entities, like hyperty runtimes on end-user/network- devices. All types of Connectors can be supported by using protocol on-the-fly concept, giving more flexibility for the integration of the Messaging Node in the Service Provider infra-structure.

#### IdM Connector

The IdM connector provides access to the domains Identity Manager. The IdM functionalities support the Session Manager for a general Access Control and the Policy Manager for the validation of identity tokens in messages and the enforcement of routing policies.

It must have listeners to receive messages for the following addresses:

domain://idm.<sp-domain>

#### Domain Registry Connector

The allocation of a unique address is only the first step on the way to make an entity (hyperty or data object) discoverable and usable from another runtime. In order to make it discoverable the allocated addresses must be registered in the domain registry component. The interaction with the domain registry is also the task of the MN. The MN has to intercept messages from a runtime that address the subdomain of the MNs own url and to create a corresponding asynchronous request to the domain registry. As soon as it receives an answer, the MN has to respond this answer back to the runtime.

It handles messages for the registration, un-registration and lookup of Hyperties and Data Objects in the domain registry. The specification of these messages can be found at [Registration messages](https://github.com/reTHINK-project/specs/blob/master/messages/registration-messages.md). The Domain Registry Connector mainly acts as a “relay” between the hyperty runtimes and the domain registry. It does not actively process the messages and responses. This connector is mandatory because the direct access to the Domain registry from hyperty runtimes should be restricted.

It must have listeners to receive messages for the following addresses:

domain://registry.<sp-domain>

#### Global Registry Connector

The role of the Global Registry Connector is comparable to the connector for the Domain Registry. It acts as a relay between the hyperty runtimes and the Global Registry. This Connector is optional. It might be required in cases where the runtime itself might not be able to establish an own connection to the Global registry. In such cases it can use the Connector running on the MN of its home-domain to access it. The specification of the messages for the interaction with the global registry can be found at [Global Registry messages](https://github.com/reTHINK-project/specs/blob/master/messages/global-registry-messages.md).

It must have listeners to receive messages for the following addresses:

global://registry/

#### QoS Broker Connector

The QoS Broker Connector allows applications to communicate with the QoS Broker Component in a certain domain. The application can use this mechanism to inquire available network resources and to place resource reservations for application sessions.

#### End-User Device Connector / Network Server Connector

These Connectors provide the “server-side” for connections that are initiated by protocol stubs running inside of Hyperty runtimes. These runtimes can either be running on end-user devices (e.g. in a browser or stand-alone) or on network-server devices, for example on an embedded system that supports an IoT use case. A simple technical example for such a connector is a Websocket server that waits for connection requests from externally deployed stubs and handles them. The types of required server-side connectors correlates to the types of stubs that the MN needs to support. If a stub, for instance, needs to establish a REST like communication with the MN than the MN must operate a connector that implements the REST server endpoint.

#### 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 a foreign or even from the own 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).

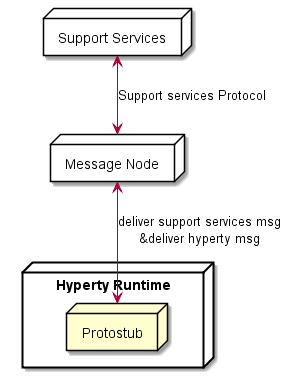


Figure - Message Node Standalone Topology

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 many 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.

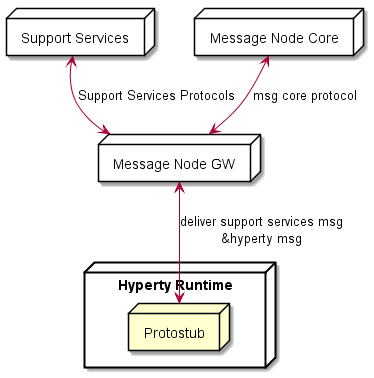


Figure - Message Node Standalone Topology

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.

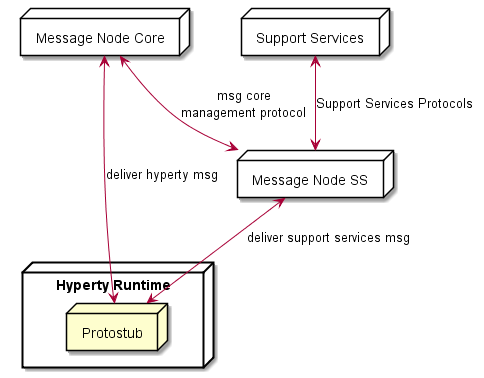


Figure - Message Node with separated Support Services

#### 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 now the stub receives a message via this listener callback it sends it forward (in this case via a Websocke connection) to its MN.

For every message that is received from the MN, the stub forwards this message to the bus.

##### Auto connect mechanism

The stubs are expected to support an auto connect mechanism. This is because the runtime will not explicitely invoke the connect method itself. 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.

##### Connection events

The stub must emit a "connect" or "disconnect" message to the bus whenever its connection state changes.

##### 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, can connect to its backend Messaging Node and can communicate with the messaging bus in the runtime.<https://github.com/reTHINK-project/specs/blob/master/tutorials/msg-node-development-recommendations.md>

## Service Framework Specification Update (Marc)

### Service Framework Specification Overview

The reTHINK Service Framework provides a comprehensive set of application program interfaces (namly APIs) and JavaScript-based libraries to support the development of Hyperties. As such, the Service Framework is agnostic with respect to the underlying messaging node, modular in design, and to the widest degree agnostic to devices and their operating systems, such as Android, iOS, Raspberry PI, Linux, or Windows. It features a comprehensive set of application program interfaces (APIs) and JavaScript libraries to facilitate the development of Hyperties within the reTHINK architecture.

As such, the reTHINK Service Framework enables the design of the Hyperty Runtime APIs to be developer-friendly, i.e., the latter only have to focus on a few core functionalities, namely:

- MsgBUS.postMessage() that is used to post messages in order to communicate with other remote Hyperty Instances and with back-end reTHINK Support Services,

- Syncher API that is used to communicate through the Reporter-Observer communication pattern, and potentially

- the implementation of the hyperty init() function, used to activate the Hyperty Instance with required configuration parameters.

This chapter updates the initial specificaiton of the reTHINK Service Framework as given in [D3.1]. For briefness, unaltered, valid descriptions are cross-referenced and updates and additions are given where appropriate. Those additions and updates focus on:

\* Service Framework Address and Message Factory

\* Synchronizaiton among Hyperties (Syncer API)

\* Discovery and Identity Manager library,

\* QoS interface and LHCB library,

\* Runtime Capabilities Manager library (new in Phase 2)

\* Storage Manager

Functionalities provided by the reTHINK Service Framework was based on closing the gaps between Runtime Core functionalities and the complete functionality requirements coming from the use cases.

### Service Framework Specification updates

#### Address Resource Factory

The Address Resource Factory creates the different types of URLs required as specified in D2.2. It is compliant with the API described by [WHATWG](https://url.spec.whatwg.org/#api) . No changes are envisaged for this library.

#### Message Resource Factory

The MessageFactory creates messages according to the [Message Data Model](https://github.com/reTHINK-project/specs/tree/master/datamodel) [13] to be sent through the Runtime Message Bus. No changes are envisaged for this library.

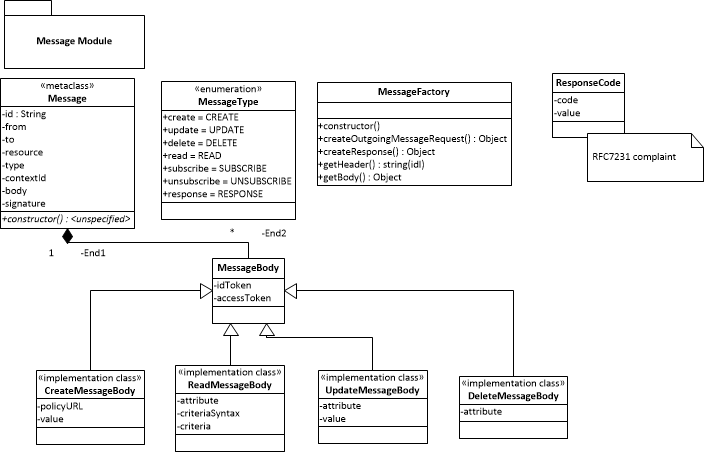
[](https://github.com/reTHINK-project/specs/blob/master/service-framework/message%20module.png)

Figure – Message Factory class diagram

#### Synchronizaiton among Hyperties (Syncer API)

The Syncher API provides data object synchronisation among Hyperties. The synchronised Data Objects are JSON data objects. This library has to be extended in phase 2 in order to support some new features described in chapter 3, notably the P2P message delivery.

The Syncher API is depicted in the following diagram:

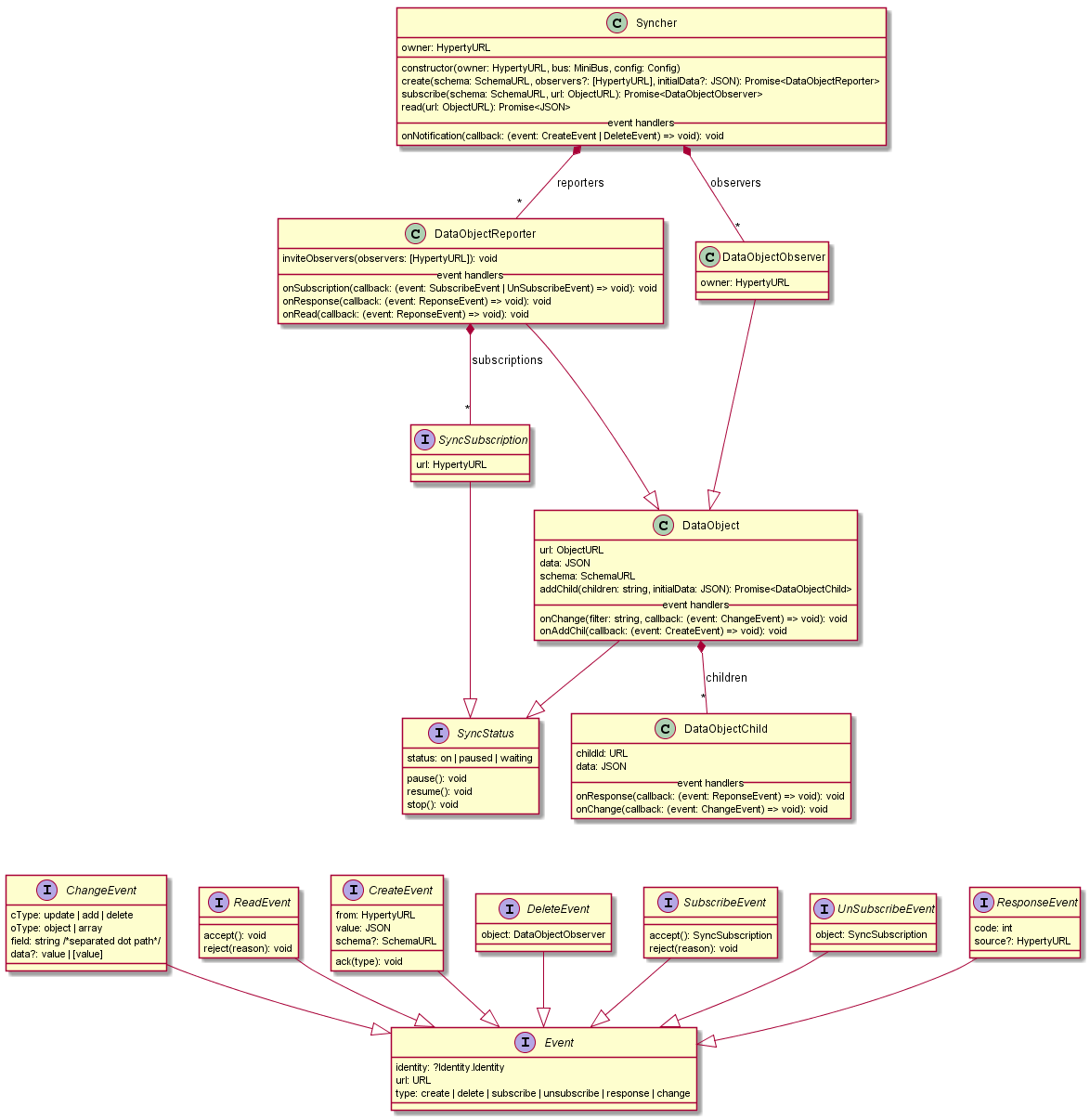
[](https://github.com/reTHINK-project/specs/blob/master/service-framework/SyncherAPI.png)

Figure – Syncher Library Class Diagram

The Syncher is a singleton owned by a Hyperty Instance that uses it to communicate with other Hyperty instance through data synchronisation. The Syncher "owns" all DataObjects (DataObject class) used by its Hyperty Instance i.e. DataObject instances (creation, destruction) are managed by the Syncher and not by the Hyperty Instance. Each DataObject is addressed by a URL - ObjectURL - that is used by the Hyperty Messaging Framework to correctly route messages required to support the data synchronisation, via the MiniBUS component. When a new Data Object (Reporter or Observed) is created, the Syncher will add listeners in the MiniBus to receive messages targeting the ObjectURL. This means, the Syncher is the end-point associated to ObjectURL and not the Hyperty Instance.

According to the Reporter-Observer pattern, there are two types of DataObjects that each Syncher can manage:

DataObjectReporter - provides functions to handle DataObjects as a Reporter i.e. the data that is written in the object by the DataObject owner, is immediately propagated to all observers. It also handles requests from other Hyperty instance to subscribe (ie request to be an Observer) or to read the Data Object.

DataObjectObserver - provides functions to handle DataObjects as a Observer i.e. it handles a "copy" of the original Data Object which is updated as soon as the Reporter changes. Changes on the DataObject are notified to the Hyperty Instance Observers.

In addition, DataObjects can be SyncObjectParents with collections of DataObjectChild. Each collection is called DataObjectChildren. Either Reporter (DataObjectReporter) or Observers (DataObjectObserver) can create DataObjectChilds in a certain children collection (addChild() function).

##### DataObjectChild

Child objects are returned from the DataObject.addChild. DataObjectChild are created in relation to a pre-existent path on the parent object schema. Child objects can be created from a Reporter or Observer and are shared between them.

##### Methods, Events and Handlers

Every object have methods, and event handlers to map to a pulling and push scheme. Methods fire actions and Handlers react to actions and respond accordingly. All events listed on the class diagram are intercepted in an event handler. From a functional perspective, methods like (accept, reject, wait, ...) are responses to an action. Since actions are represented by events, it makes sense that responses are directly related to them.

#### Discovery and Identity Manager library

Hyperty discovery is performed through the Runtime Registry which returns the Hyperty Registry entry containing its Hyperty Runtime URL, the P2P Handler Stub instance URL and the catalogue URL of P2P Requester Stub. Identities are managed by using a token-based access. A full dynamic view for discovery and identity management is provided at . These libraries have to be extended in order to cope with the extensions of the data model.

#### Runtime Capability Manager

The Runtime Capability Manager is a new library that handles a JSON data object containing the capabilities of the Hyperty Runtime that will be used to take the most appropriate decisions according to Runtime Capabilities e.g. to select the most appropriate Protostub or Hyperty to be deployed in the runtime.

#### QoS interface and LHCB library.

An initial description of the QoS support is provided in section **Error! Reference source not found.**. A full specification of the QoS framework will be given in the dedicated QoS deliverable.

## Trust Management and Security

This section 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.



Figure - 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:

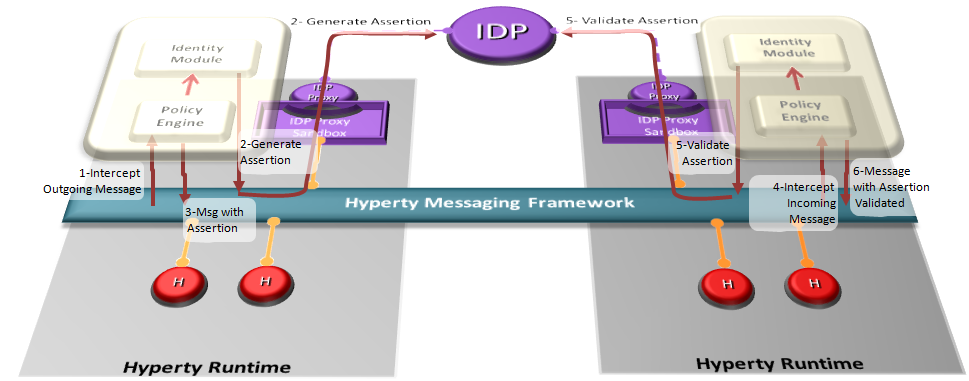


Figure - Interaction between the Identity Module and the Identity Provider

*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) *specification [15].*

#### 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).

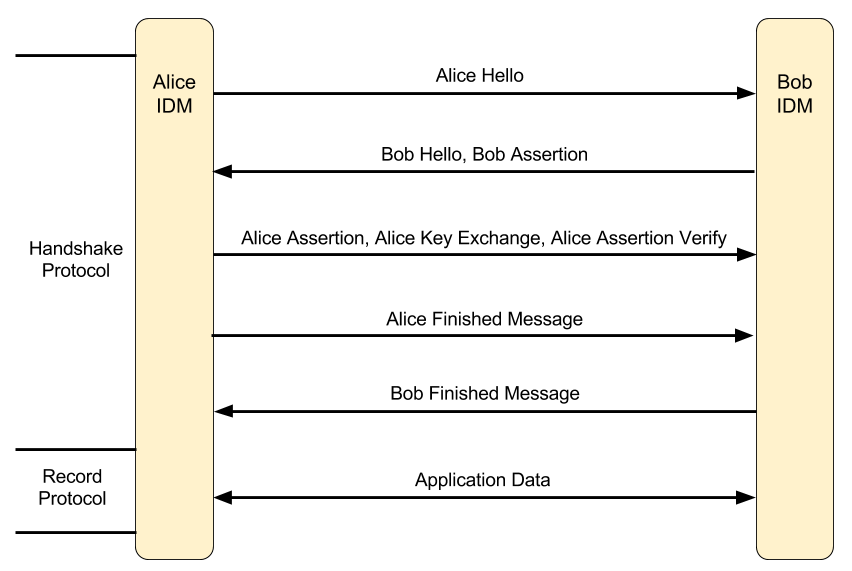


Figure - reTHINK mutual authentication flow

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.

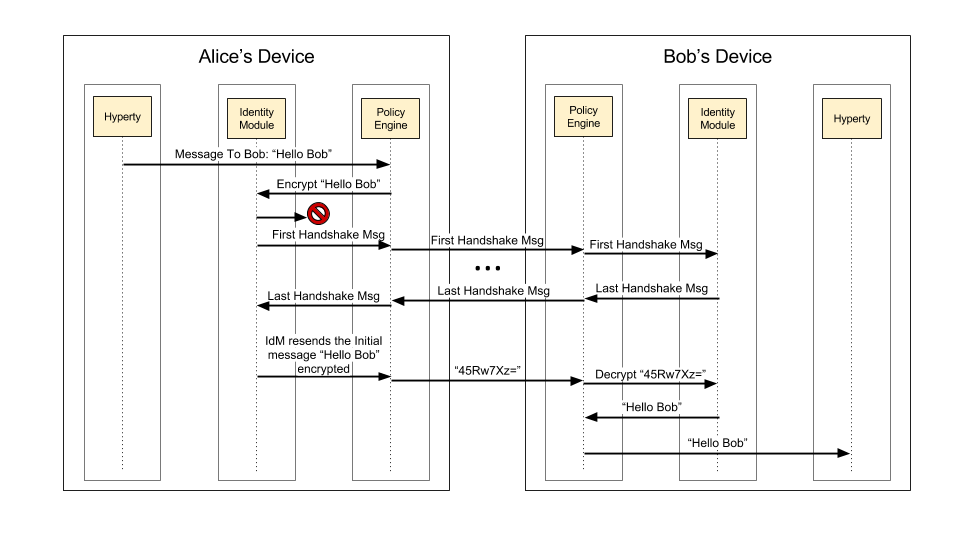


Figure - Example of Alice sending a message to Bob, for the first time

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.

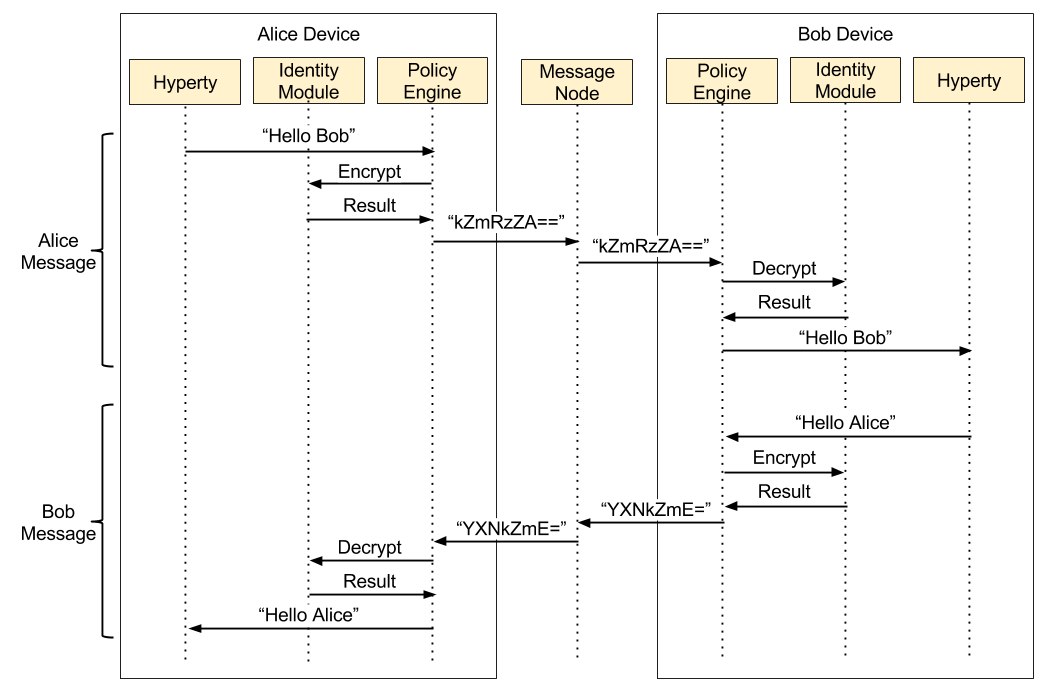


Figure - Hyperty to Hyperty communication

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.

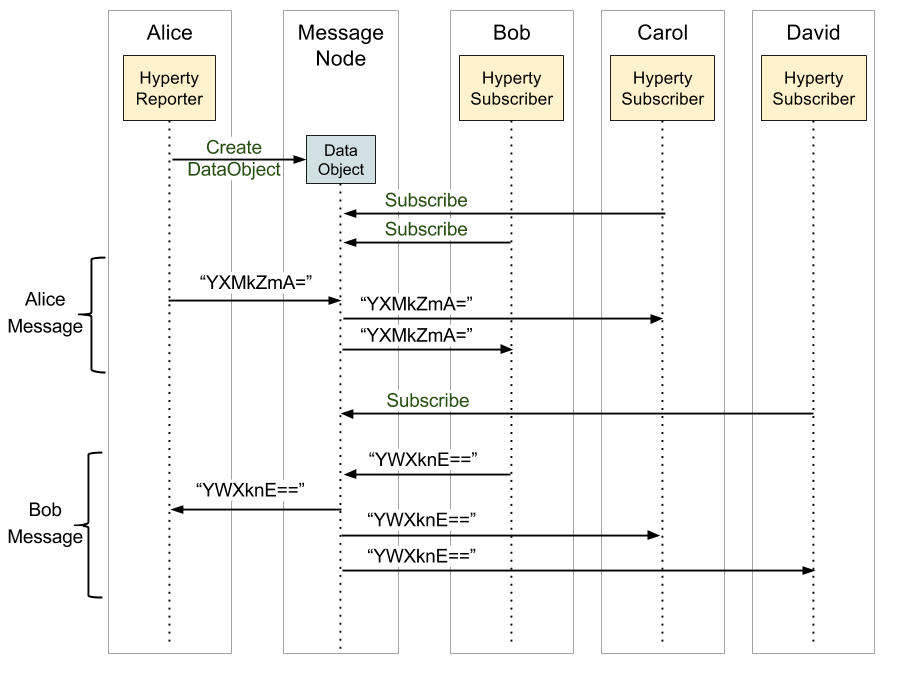


Figure - Group chat communication

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.

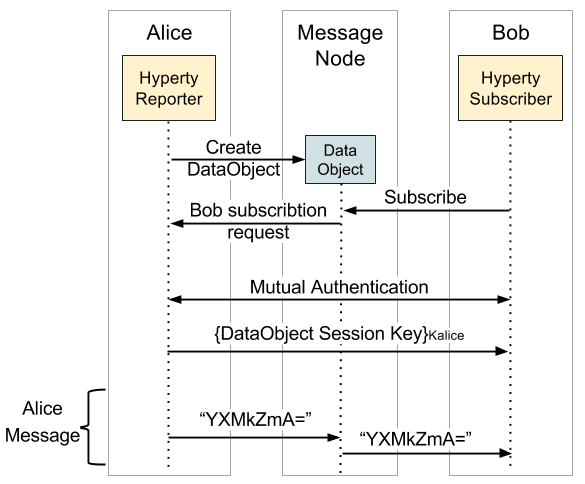


Figure - Hyperty subscription flow

#### 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.

# New Features specification

This Chapter introduces the specification of main new features planned for phase 2 namely:

* Peer-to-peer Message Delivery;
* Interworking with Legacy domains;
* Multiparty Communication with WebRTC;
* Quality of Service Control

## P2P Message Delivery

Communication between different Hyperty Runtimes can be supported with peer to peer connections or via a Message Node. In both options a Protostub implementing the most appropriate network protocol stack is used. For example, JSON over Websockets and Restfull API Client are good options for Protostubs used to interface with a Message Node, while WebRTC Datachannel is a good option for a Protostub used to directly interface with another Hyperty Runtime. The usage of P2P Protostubs are favored since less resources are spent and Network latency should be better.

Protostubs are registered in the Runtime Registry with its own Hyperty Runtime URL e.g. hyperty-runtime://example.com/runtime-123/protostub-3 and have listeners in the MessageBUS to receive messages targeting its URL.

When the MessageBUS is processing a new message and looking up routing paths for an address, which is not local (eg hyperty://example.com/alice-hyperty), it won't find any registered listeners. In this case, the Message BUS will ask the Runtime Registry to [resolve the "Message.to" header field](../dynamic-view/basics/resolve-routing-address.md) and it should return a registered Protostubs that is able to deliver messages to such non-local address. In this process, in case the required Protostub is not yet available, [its deployment is performed](../dynamic-view/basics/deploy-protostub.md).

There are two types of P2P Protostubs:

**P2P Handler Stub:** it is deployed as soon as the Runtime is instantiated in order to be ready to receive requests for P2P Connections setup. This Stub playes the [observer](p2p-data-sync.md) role of [Connection Data objects](../datamodel/data-objects/connection) that are created by remote P2P Requesters Stub. The P2P Handler Stub can observe more than one connection data objects ie it can handle several p2p connections to remote runtimes. Each p2p connection would be managed by a connection controller (see Connector hyperty design). As soon as the Runtime is instantiated, the P2P Handler Stub is deployed and the path to receive P2P Data Connection creation requests from P2P Requester Stubs is set.

**P2P Requester Stub:** it is deployed to setup a p2p connection with a remote runtime P2P Handler Stub. It plays the [reporter](%5Bobserver%5D(p2p-data-sync.md)) role for a single [Connection Data object](../datamodel/data-objects/connection) object instance.

These P2P Protostubs are provisioned in the catalogue.

Hyperties that are deployed in a P2P enabled runtime, are registered in the Domain Registration with its Hyperty Runtime URL and its P2P Handler Stub instance URL. P2P Requester Protostubs in other Runtimes can setup a P2P connection exchanging messages with the P2P Handler Protostub through the Message Node ie using Message Nodes Protosubs (see picture below).

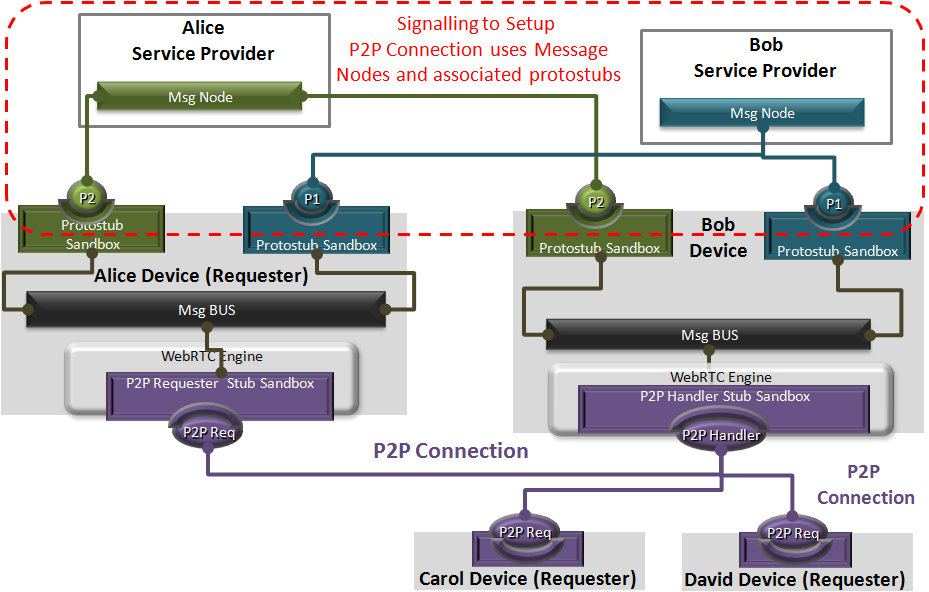


Figure - P2P Protostubs

Detailed description of P2P Message Delivery procedures can be found at:

* <https://github.com/reTHINK-project/specs/blob/master/dynamic-view/basics/resolve-routing-address.md>
* https://github.com/reTHINK-project/specs/blob/master/dynamic-view/basics/p2p-setup.md

## Quality of Service Control

### QoS in reTHINK

The reTHINK architecture enables activating QoS and policy as selectable options, via APIs to the service providers. While OTT services have no such choice, and Mobile services automatically provide managed QoS over managed packet network, the reTHINK architecture can deliver QoS ‘on-demand’ over the Internet, selected only where necessary, according to network conditions, user preference and service requirements. Several QoS enforcing points and technologies have been envisioned. One of the solutions is based on providing QoS on CPE Broadband and mobile access. The other one is a solution based on network selection (LHCB) in which a client provides information about available "uplinks" (i.e. alternative wired or wireless connections) and associated quality parameters, and in which the client may be requested to switch its connectivity over to an indicated network (interface) providing a certain QoS level.

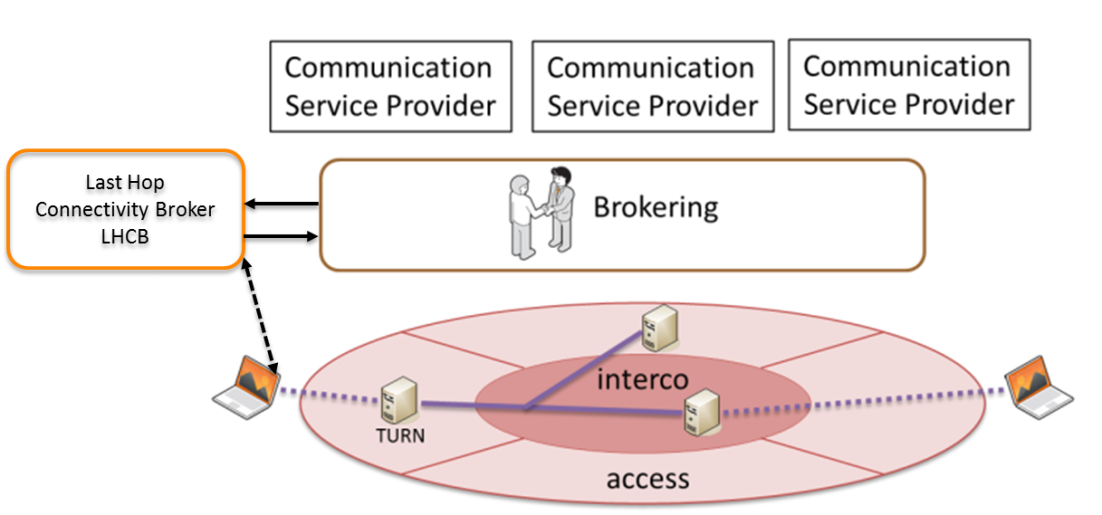


Figure – QoS in reTHINK

*Note: the final spec should be included (refined) in the dedicated QoS deliverable*

### reTHINK TURN services

On the first solution, design of network traffic control has been implemented in the CPE. The general mechanism is the following one:

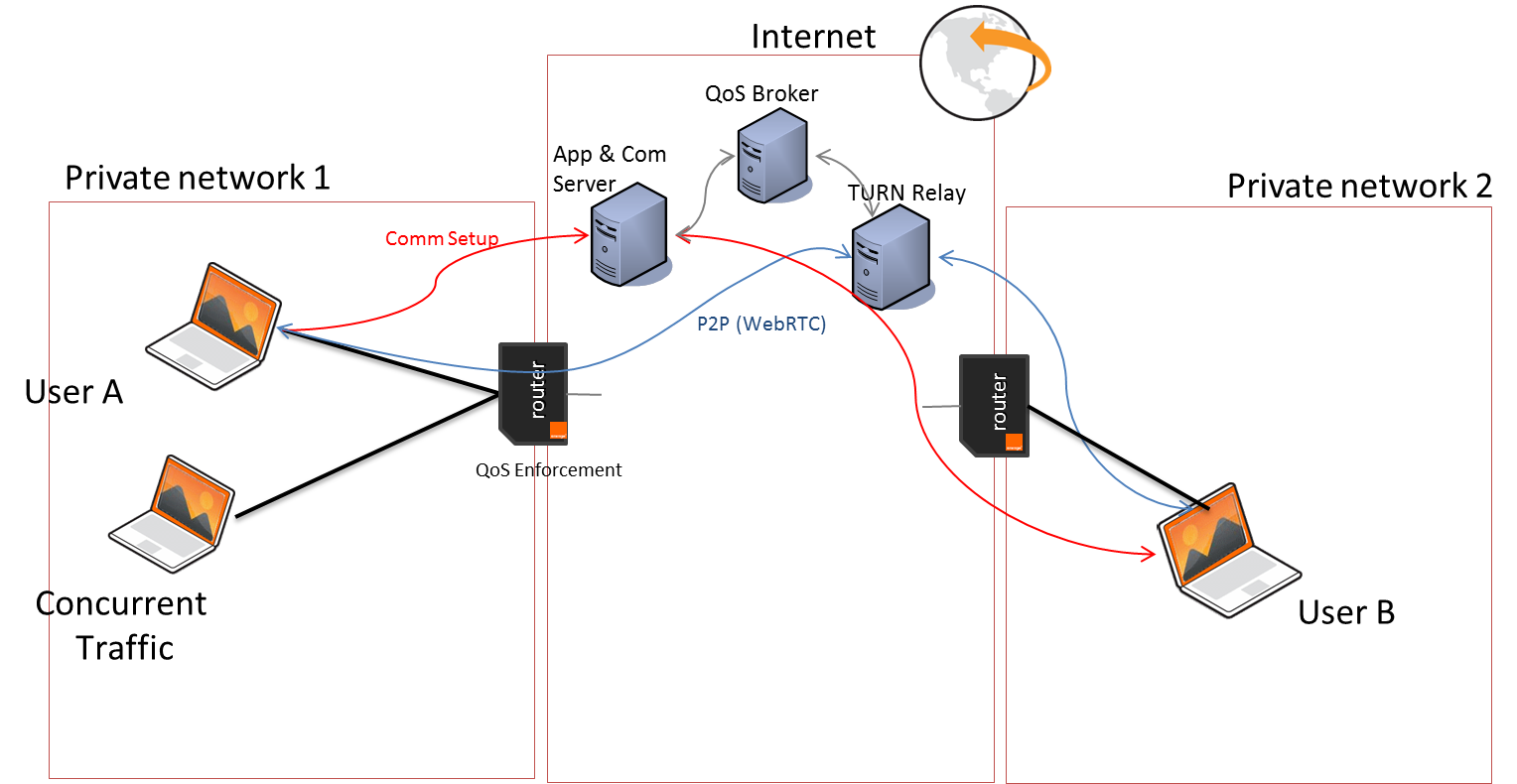


Figure – QoS based on TURN Services

To use the right TURN server, the general flow is the following:

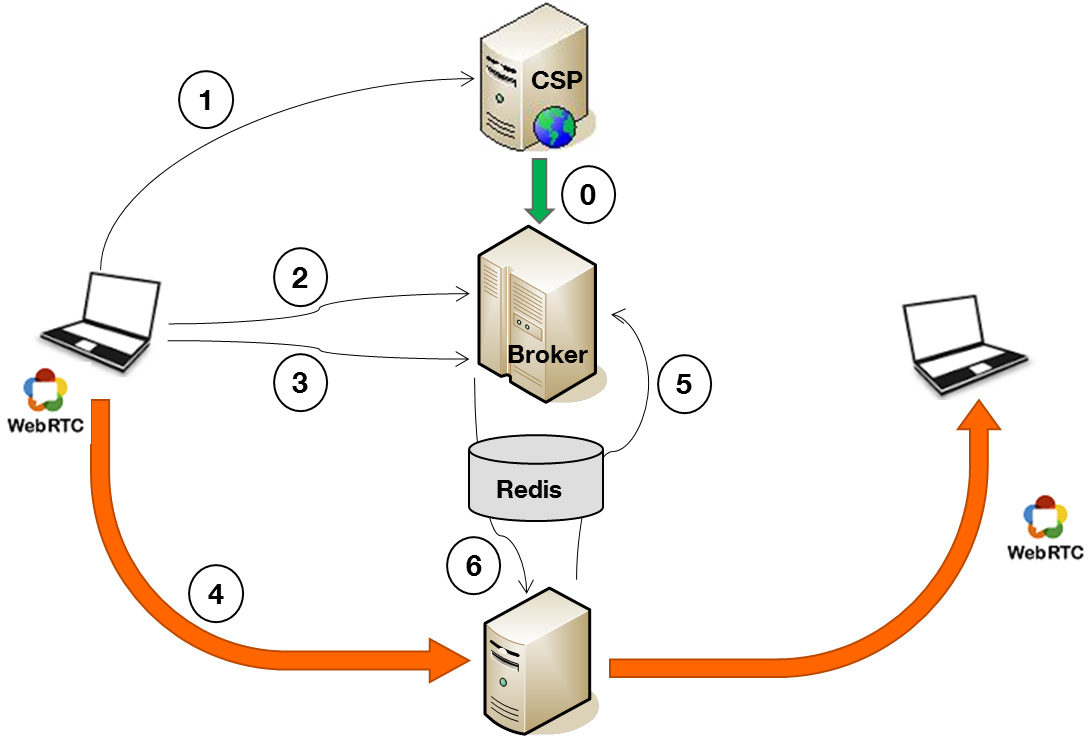


Figure – Main flows for TURN based QoS

* Step 0: Offline CSP’s provisioning for a certain amount of a data : audio, video, throughput datachannel. The subscriber (CSP) is assigned a unique identifier cspID, it will provide to its clients so they can later be associated to the subscriber by the broker.
* Step 1: The client retrieves from its CSP a communication Web App (including cspID).
* Step 2: The Web App asks the broker information about the TURN server to use (IP address). If the data quota reserved is exceeded, the broker returns an error. clientID = getTurnAddr(cspID, mediaType, clientName);
* Step 3: The Web App asks the broker the login information (credentials, username / password ...) to provide to the TURN server. credentials = getTurnCredentials(clientID);
* Step 4: The client initiates the communication WebRTC via the TURN server.
* Step 5: The TURN server sends periodically to the broker, information on data volume consumed through Redis Database (pub/sub mechanism).
* Step 6: If the data quota reserved by the subscriber is exceeded, the broker sends the server TURN a command to interrupt the current communication.

The application is using a Quality of Service Broker, that manages a fleet of TURN servers available. The application has to register to the QoS Broker to be able to get the best TURN server dedicated to a user, regarding parameters (this registration is done ofline). Then, when a user is setting up a call, the runtime gets, from the broker, the best TURN candidate, with the good credentials to authorize the usage of TURN. If the operation is ok, it will benefit from quality of service, and the router will manage traffic regarding the traffic management implemented.

### Last Hop Connectivity Broker

The Last Hop Connectivity Broker (LHCB) aims at providing information about available, alternative (wireless) uplinks of an end-device and the associated current quality of service. Such information is provided within the reTHINK framework to both, the global QoS reTHINK mechenism provided by the TURN services on an end-to-end scale, and to Hyperties and applications running on the client and wishing to access link information about their current connectivity. As such, the LHCB provides a supporting means to ensure and retrieve information about the current quality of service.

Besides providing only information about a single, i.e. the local end device, the LHCB's architecture allows for monitoring QoS indicators for a set of devices. The following figure illustrates the overall LHCB architecture as implemented in reThINK.

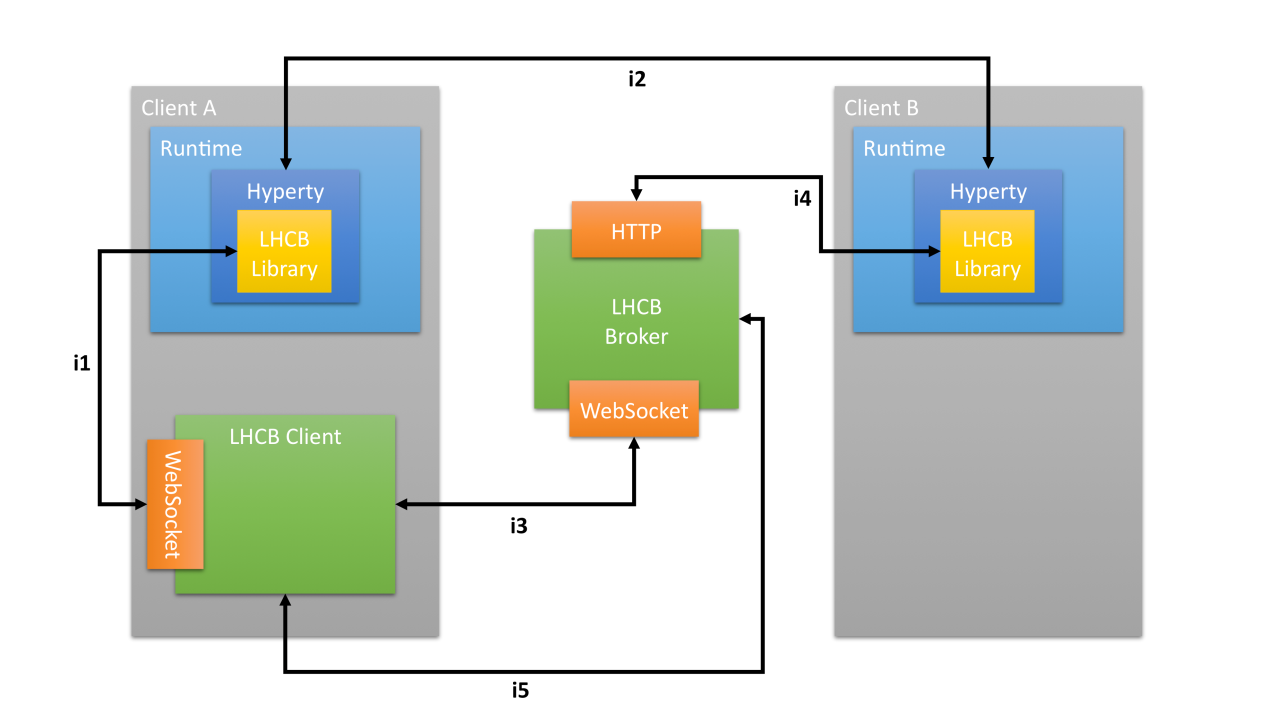


Figure – LHCB based QoS

The LHCB Client is an entity running at the end-device and providing low-system-level access to network interface drivers. Due to the complexity and close integration with the hosting operating system, the LHCB Client is provided in parallel to the client's runtime environment.

Within the runtime, a LHCB library is provided that allows to transparently access the LHCB Client -- which is itself operating system depending -- via the web-socket-based interface i1. The latter interface provides means to retrieve and set information about the QoS status of all available (local) interfaces.

In addition, the LHCB arichitecture allows to provide information about a client's last hop connectivity QoS to other clients (i.e. communication partners using the reTHINK hyperty framework) or components in the network's backbonde, such as the reTHINK TURN service. For that, every LHCB Client registers itself at a LHCB Broker service via interface *i3*, which is also used to regularly update the clients QoS and link information at the broker. Note that the LHCB Broker may manage a set of clients. The LHCB Broker exposes a public http-based interface *i4* that allow for retrieving information about the QoS status about each client registered at the LHCB Broker. In order to guarantee privacy, a randomized globally unique identifier that maps to a client's identity has to be known to retrieve these information from the LHCB Broker; this identifier has to be retireved directly from the client via interface *i2* thus leaving the control at the user about which information about available links and QoS to share with whom. Hence, in the figure above, "Client B" may be exchanges with the reTHINK TURN service if the latter requests LHC QoS information via interfaces *i4* and *i2*.

The auxiliary interface *i3* is only required as generalized deployment of the LHCB service allow to deploy the LHCB Broker "as a service" in data centers which put the running LHCB instance behind firewalls and IP proxies. For such a deployment, the public IP address under which the LHCB Broker is accessible has to be know, e.g. by Client B, to contact the LHCB Broker. This information is provided by the client -- here Client A -- about which Client B want to retrieve link QoS information via interface *i2*. The local LHCB Library retrieves this information via *i1* from the LHCB Client which in turn retrieves it once via *i3* directly from the broker. This generalized approach also supports a set-up in which several, different LHCB broker handle diffent users, i.e., a globally / domain-specific unique LHCB broker is not required.

An example dynamic view on the interaction of components is provided at <https://github.com/reTHINK-project/specs/blob/master/dynamic-view/qos/readme.md>.

## Multiparty WebRTC Communication

The goal of this section is to provide specifications for enabling WebRTC group communication in reTHINK (aka WebRTC Multiparty).

### Multiparty Topologies

We can distinguish two topologies:

* Full mesh topology
* Start topology

In full mesh WebRTC conference, every peer establishes a connection with the rest of the participant peers. Thus, n\*(n-1) number of connections are needed, where n is the number of participant peers. For examples, a full mesh topology of 4 users has 12 connections.

This topology performs well for a small number for participants. Easy to deploy without extra resources(no server is needed).

However, it is inefficient for scalable multiparty systems. As the number of participants increases, bandwidth and CPU processing consumption becomes more excessive on peer side, which is not scalable, especially for mobile Peers.



Figure - Full Mesh Topology for group communication

Alternatively, in a star topology, a relay server in the middle will be in charge of establishing peer connections and distributing streams among peers. As a result, each peer establishes only one peer connection to the media server independently of the increasing number of peers. Which is very scalable approach. In such way, all the burden and processing is left to the middle server. However, an extra latency will be observed due to the presence of the intermediary server.

This star topology, describes H2H WebRTC group communication between reTHINK users. Therefore, peer conference hyperties running on runtime browsers will exchange signaling descriptions between each other and the media server through reTHINK edge server(Runtime Node).Particularly, reTHINK Runtime Node [dev-runtime-nodejs](https://github.com/reTHINK-project/dev-runtime-nodejs) is a justified choice for exchanging WebRTC signaling. In addition, Runtime Node is a fully conform with reTHINK specs in term of reliability and security.



Figure - Star Topology for group communication

### Solution for Star Topologies

According to the reasons presented above, star topology seems a good candidate for scalable rethink WebRTC group communication.

Three main components are required in this topology : **Peer conference Hyperty** running on **Runtime browser**, **Conference Hyperty** running on **Runtime Node**, and the **Media Server**. While, the messaging node is just relay point in reTHINK framework. Thus, it's transparent for the different communication messages.

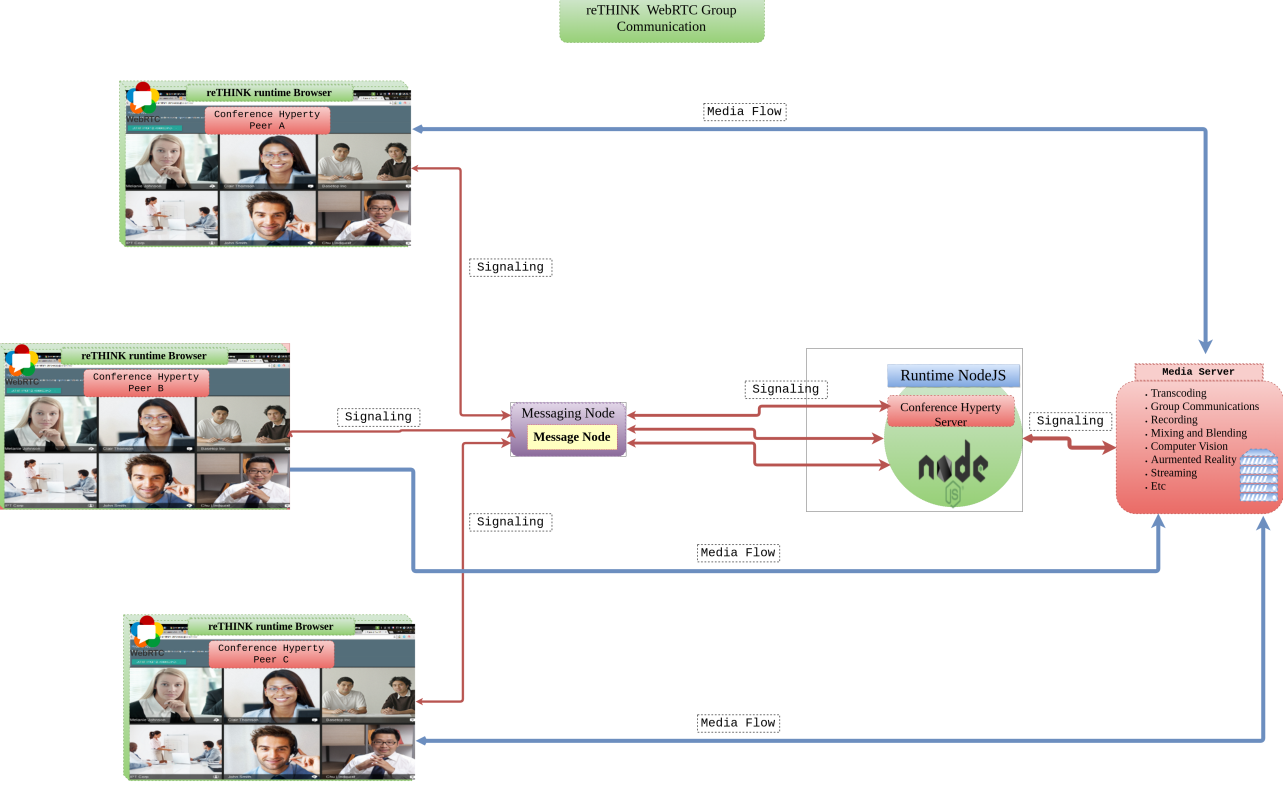


Figure - reTHINK Group cummunication overall architecture

The detailed call flows between the components of this topology are provided at <https://github.com/reTHINK-project/specs/blob/master/dynamic-view/group-communication/readme.md>.

## Interworking with Legacy Services

### Introduction

reTHINK framework provides a mechanism to interact with legacy networks. This allows. for example. to setup calls with an IMS system from a Hyperty running in a browser, or exchanging Slack messages from a Hyperty. These scenarios are realized through the implementation of an InterWorking protostub - the "IWStub" - which will interact with the legacy service. Since protostubs also have to be created to interact with different Message Nodes, it does not add any relevant changes to reTHINK architecture. It may be also necessary to associate the Hyperty to more than one Identity, at least one identity used by the application which uses the Hyperty and also an identity valid for the Legacy domain. Both identities could be the same, however this would not be a common case.

### Applications

Integration with existing services is a critical requirement in order to make possible a soft migration from the existing services to reTHINK framework by making reTHINK application inter-operable with potentially any service.

One example is to integrate reTHINK with existing telephony networks. Being able to make and receive calls, and to use other advanced services provided by operator's IMS platform allows to open the operator services in a flexible and secure way to all the devices and platforms where reTHINK runtime can be executed.

The same is applicable to other popular services such as Facebook, Slack, Salesforce and any other social network or messaging system which expose public APIs. For example, this inter-working mechanism will allow to build an application which can receive calls from IMS in the public Identity of the user (normally an e.164 number) and also to send and receive Slack messages in the same web interface. These features can be combined with any reTHINK-based service.

### Interworking strategy proposal

#### Who provides the IWstub?

The IWStub must be provided by the legacy domain and it must make reTHINK interoperable with the API or GW deployed in the legacy service to expose service to third parties. For example, in the case of IMS the IWstub must implement the protocol needed to interact with gateway element which translates web-based signaling protocol and WebRTC media profile in SIP and media profiles compatible with IMS.

Ideally the IWstub should also be downloaded from a back-end service of the Legacy Domain. If the Legacy Domain does not allow to download it, then it could be loaded from the default domain.

#### Protocol implemented by the protostub

IF the runtime is being executed in a browser runtime, it must be taken into consideration that only HTTP or Websocket based protocols can be used (those are the only protocols that a browser can use without adding any additional plug-in). If the runtime is being executed in a Node.js runtime (or in any other runtime that can be created in the future) this limitation may not exist.

#### High level diagram

The diagram below shows a high level architecture of the integration of reTHINK with an external service.



Figure - Legacy domain interworking diagram

As stated above, the Hyperty will need to be associated to two identities. The Identity Module will handle the authentication against the Identity Provider of the Legacy domain. After a successful authentication normally a token will be provided. This token has to be used from the Protostub to authenticate itself during the registration/login process to the legacy domain. Depending on the Legacy Domain this process may be different, however it should be compatible with the most scenarios.

Once the Identity Module has finished the authentication process, the Hyperty is ready to instruct the Protostub to register into the legacy domain and start the exchange of messages in order to give service to the application using the Hyperty.

The Hyperty will be able to interact with the legacy domain sending messages to the Protostub as it is done for a regular Message Node. The same way the Hyperty will be able to receive messages from it. The messages received by the Protostub from the legacy domain will also be translated into reTHINK messages.

### Technical implementation

The diagram below shows the deployment process of an IWstub.

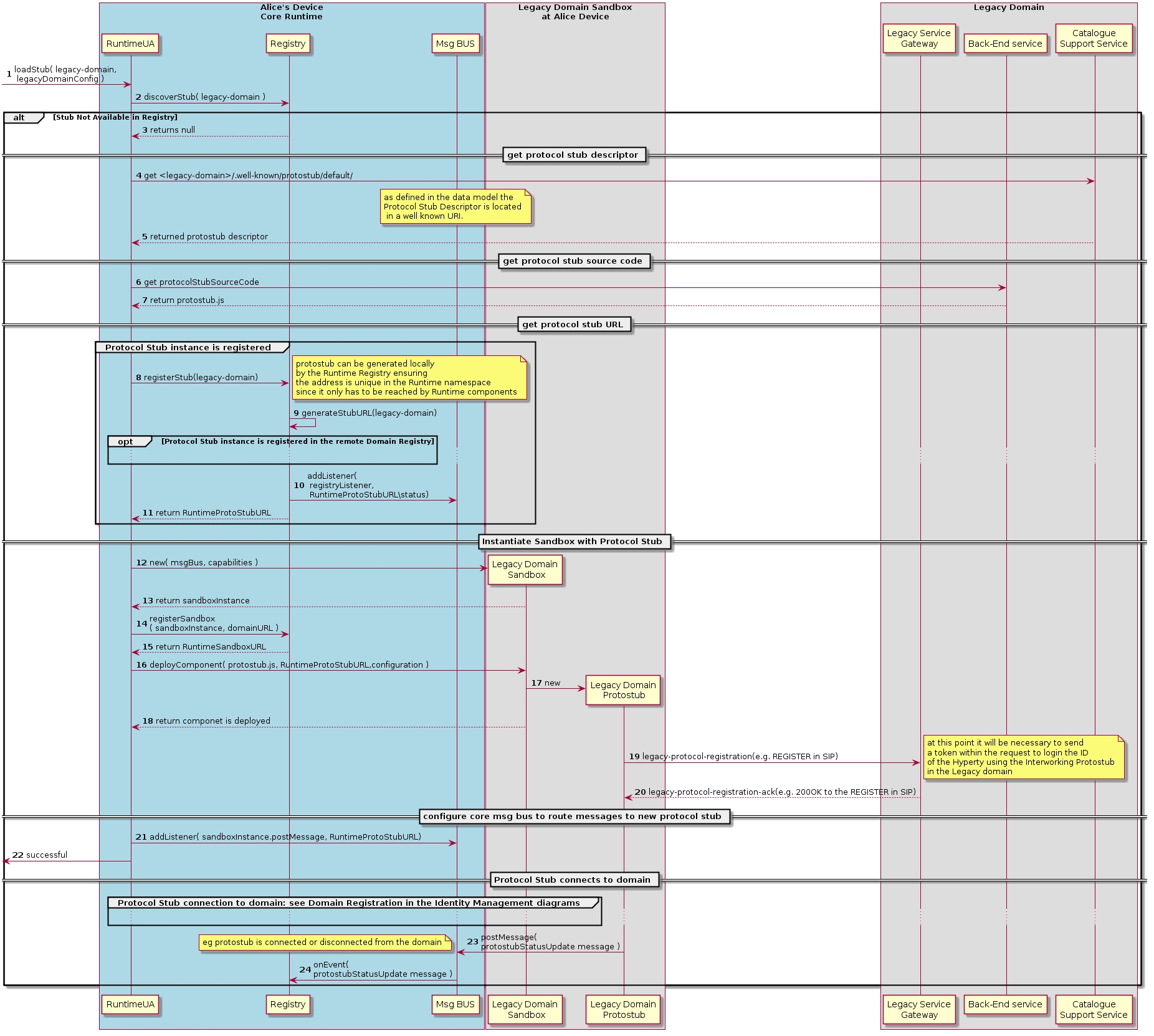


Figure - Legacy domain IWstub deployment diagram

The main difference is that the IWstub code is downloaded from a back-end service of the Legacy Domain and the IWstub descriptor is also downloaded from a Catalogue at the Legacy Domain. Alternatively both of them could be downloaded from the Default Domain if it has some kind of agreement with the Legacy Domain.

Another big difference respect to a regular protostub is that normally there will a login or registration process once the protostub is deployed. This will depend on the protocol of the Legacy Domain, for example in the case of SIP networks there will be a registration process.

A complete description of the diagram has been included at <https://github.com/reTHINK-project/specs/blob/master/dynamic-view/legacy-interworking/readme.md>.

### IWstub implementation

The data model of the Protostub which has been used from it conception in reTHINK, has been adapted to be compatible with the IWstub so in terms of data model it is like any other protostub.

Several attributes were included to accomodate the same data model to the IWstub:

* interworking: if this boolean is true it indicates that the protostub is used to connect with a legacy domain that is not compliant with reTHINK.
* idpProxy: this boolean indicates if protostub also provides Idp Proxy features. This may be needed to support interact with legacy domains.
* HypertyDataObjects: It defines the HypertyDataObjects supported by peers belonging to the domain served by this protostub. To parameter may be useful to interact with a legacy domain which supports several Data Objects to implement different functionality.

A complete list of attributes can be consulted at <https://github.com/reTHINK-project/specs/tree/master/datamodel/core/hyperty-catalogue#protocol-stub-descriptor>.

### Token based authentication techniques

Many Internet-based services expose APIs to be accessed from third-party services. Many of these APIs use token-based mechanisms to authenticate the request coming from authorized users.

The emergence of WebRTC support by most important browser vendors motivated 3GPP to defined token-based strategies to access the IMS network from Web applications. This will allow to use potentially any web browser with WebRTc support to behave as a user Equipment which has been restricted to native SIP clients. In the case of IMS, the authentication provider enabled by the operator (which can be the operator itself) provides a registration token after a correct login. This token based authentication has been designed to open IMS services to Web browser. reTHINK will leverage this token-based authentication feature by using a Interworking IDP Proxy separated from the IWStub.

### IMS interworking

3GPP has released a [draft specification of TS 24.371 [17]](https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1087) to define WebRTC access to IMS systems. The proposed legacy interconnection scheme for reTHINK is compliant with this specification and it will be shown in T.6.3.

The diagram below shows the interconnection diagram which is very similar to the generic one previously shown. The IMS gateway will perform a validation of the token obtained from the Identity provider and it will be passed to the gateway element of the IMS network in order to validate the token and register the identity associated to the reTHINK hyperty in the IMS network through the IMS protostub (which is an example of IWstub). From this point on, the Hyperty will be able to interact with the IMS network through the protostub. From the IMS point of view the registered Hyperty is just another User Equipement. To enable voice and video calls, the WebRTC gateway would perform the translate between the media with WebRTC profile to a media profile compatible with the IMS network.



Figure - IMS interworking diagram

### IWstub Extensibility Considerations

Extending reTHINK to make it inter-operable with different services may require to support scenarios and use cases which has not been considered at design time. So the IWstub has been enriched with attributes not used in regular protostub which enable future extensions.

#### Data Object adaptation to meet new scenarios

reTHINK internal communication is based on data object synchronization so two hyperties can "talk" to each other if they use a common data object. In order to implement complex message flows (e.g. a SIP call flow needed to implement call transfer) may not be implementable with the current connection Data Object so it may need to be extended in short term. That is why a the dataObjects attribute has been added to the protostub descriptor which includes all the HypertyDataObjects supported by peers belonging to the domain served by this protostub. New HypertyDataObjects or extensions to HypertyDataObjects may be needed to support new scenarios in order to be flexible enough to meet future requirements.

#### IdP Proxy

Additionally to the dataObjects a new boolean attribute called idpProxy has been defined to specify if the IWstub is also a proxy for the Identity Provider exposed by the Legacy Domain. Allowing the IWstub to act as an IdPProxy gives an extra flexibility which will help to accommodate future Identity management mechanisms.

# References

1. Deliverable D6.3 – “Assessment Report”, 30-06-2016
2. Deliverable D2.3 “Final design of the Architecture”, ?
3. Deliverable D4.4 “Management and Security features specifications” – Revision”, 31-11-2016
4. Deliverable D3.1 “Hyperty Runtime and Hyperty Messaging Node Specification”, 31-09-2015
5. reTHINK Specification Github repository, <https://github.com/reTHINK-project/specs>
6. Deliverable D3.2 “Hyperty Runtime and Hyperty Messaging Node Phase 1”, 29-02-2016
7. Hyperty Runtime Specification, <https://github.com/reTHINK-project/specs/blob/master/runtime/readme.md>
8. Hyperty Messaging Framework Specification, <https://github.com/reTHINK-project/specs/tree/master/messaging-framework>
9. Data Model Specification, <https://github.com/reTHINK-project/specs/tree/master/datamodel>
10. Messages Specification, <https://github.com/reTHINK-project/specs/tree/master/messages>
11. Dynamic View Specificarion, <https://github.com/reTHINK-project/specs/tree/master/dynamic-view>
12. Service Framework Specification, <https://github.com/reTHINK-project/specs/tree/master/dynamic-view>
13. Deliverable D2.2 “Data Models and Interface Specification of the Framework ”, 30-08-2015
14. WHATWG URL standard, <https://url.spec.whatwg.org/>
15. Policy Engine Specification - <https://github.com/reTHINK-project/specs/tree/master/policy-management>
16. Quality of Service specification - <https://github.com/reTHINK-project/specs/blob/master/dynamic-view/qos/readme.md>
17. 3GPP TS 24.371 - Web Real-Time Communications (WebRTC) access to the IP Multimedia (IM) Core Network (CN) subsystem (IMS); Stage 3; Protocol specification, <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1087>
18. Deliverable D2.1 “Framework Architecture Definition”, 31-07-2015.
19. TURN rfc, <https://tools.ietf.org/html/rfc5766>
20. https://whatwg.org/