# Introduction

## Objectives and Overview

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

This report provides a detailed specification of reTHINK Core Framework components comprised by the runtime environment where Hyperties are executed and the messaging nodes used to support messages exchange between Hyperties. This report complements deliverable D4.1 (Management and Security features specifications)[109], which specifies reTHINK Support Services, namely: Policy Management, Governance, Identity Management, Graph Connector, and Hyperty Directory services (Catalogue and Registry). Thus, and according to reTHINK Architecture [38], the scope of this report includes the specification of the Messaging Node providing reTHINK Messaging Services and the specification of the Hyperty Runtime that will be included in User Devices and Application Servers to deliver User Hyperties and Network Side Hyperties (See Figure 1).

Figure 1 - Specification Scope

Figure 1 - Specification Scope

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

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

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

The specification of the Hyperty Runtime and the Messaging Node is sustained by a very comprehensive work in terms of state of the art research and procurement of existing open source that will be used to demonstrate the feasibility of the radical reTHINK concepts.

An exhaustive study of relevant IETF, W3C standards and others that facilitate the fulfillment of previously analysed requirements, is reported. Special attention was given to the research on security in Web Runtime. In parallel, existing open source solutions to be used to develop Hyperty Runtime and Messaging Nodes was researched, experimented and selected.

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

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

The Runtime design enables reuse of most of the core runtime components through different platforms including Browsers, Standalone Mobile Application, Network Side Application Servers and more constrained M2M/IoT standalone devices. The Hyperty Runtime architecture follows a security by design approach where different types of components are executed in isolated sandboxes. Communication between different sandboxes is only possible through a Message Bus and is subject to access control. Communication with remote Hyperties is provided by protocol stubs executed in isolated sandboxes.

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

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

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

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

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

## Structure

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

## User Manuals

How to use eg Msg Nodes and associated protostubs.

## Development Manuals

*give a short summary for each document and the reading order for each type of developer (Hyperty, App, Msg Node or Runtime developer)*

* [Getting Started](start.md)
* [Hyperty Concept Overview](hyperty.md)
* [Hyperty Messaging Framework Overview](hyperty-messaging-framework.md)
* [Hyperty Data Synchronisation: Reporter - Observer communication model](p2p-data-sync.md)
* [Hyperty Trust and Security Model](hyperty-trust.md)
* [Hyperty Development](development-of-hyperties.md)
* [Application Development](development-of-apps.md)
* [Message Nodes and Protostubs Development](development-of-protostubs-and-msg-nodes.md)
* [Runtime Development](development-of-runtime.md)

## Introduction

### Introduction to the ReThink Framework

*Describe concepts*  
*Big picture with Devices/CSP (including support services + IdPs)*

### My First Service

*Hello World like application A calls B with a communication Hyperty already provided*  
*Second application with a sensor emulation (Get the room temperature)*

### 

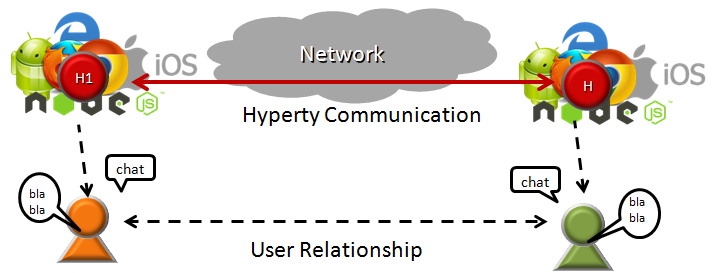
## Hyper-linked Entities - Hyperties

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

* the [Hyperty Messaging Framework overview](hyperty-messaging-framework.md)
* the [Reporter - Observer Data Synchronisation model](p2p-data-sync.md)

Hyperties are cooperative [Microservices](http://martinfowler.com/articles/microservices.html) that are executed in devices on behalf of users through simple but sophisticated Identity Management techniques. This means, Hyperties are independently deployable components each one providing a small set of business capabilities, using the *smart endpoints and dumb pipes* philosophy i.e. Hyperties don't depend on complex and sophisticated communication middleware like Enterprise Service BUS (ESB). Instead, Hyperties rely on a very light but powerful [Messaging Framework](hyperty-messaging-framework.md) concept).

On the other side, Hyperties follow emerging [Edge](https://en.wikipedia.org/wiki/Edge_computing) and [Fog](https://en.wikipedia.org/wiki/Fog_computing) computing paradigms as opposed to more popular Cloud Computing. This means, when compared with Cloud Computing, Hyperties promotes a more effective usage of computing and network resources, decreases communication latency, improves security and extends scalability.



Hyperty Concept and Edge Computing

However, Hyperties can also be executed in Network Servers for specific Business Capabilities (e.g. Media Servers) or when End-user devices don't have enough capabilities in terms of computing resources and/or power.

In addition, Hyperties have some unique characteristics including:

* Hyperties are programmed in Javascript ECMA5/6, i.e. any existing device featuring a Browser or a NodeJS can be used today to execute Hyperties without requiring the installation of any new software. This means, **billions of devices** are already Hyperty enabled and ready to make part of reTHINK ecossystem.
* The User Identity associated to an Hyperty is decoupled from the Hyperty Service Provider. Ie Identity Management is handled under the scene and the Developer does not have to care about it and just have to focus on the development of Business Capabilities. This also means, the end-user has the power to decide which is the Identity to be securely associated to a certain Hyperty instance. *put link*
* Hyperties cooperate and communicate each other via P2P Synchronisation of Hyperty JSON Data Objects supported by the novel [Reporter - Observer communication pattern](p2p-data-sync.md).



Reporter-Observer Communication Pattern

The API to handle the Synchronisation of Hyperty Data Objects is extremely simple and fun to use. The Developer of the Hyperty Reporter just has to create the Data Sync object with the Syncher API, and write on the object every time there is data to be updated and shared with Hyperty Observers.

....  
  
 console.info('---------------- Syncher Create Reporter Hyperty Data ---------------------- \n');  
 syncher.create({}, [hypertyURL], {}).then(function(dataObjectReporter) {  
 console.info('1. Return Create Data Object Reporter', dataObjectReporter);  
  
 })  
 console.info('--------------- END Create Reporter Hyperty Data------------------ \n');  
 })  
 .catch(function(reason) {  
 console.error(reason);  
 reject(reason);  
 });  
  
 // missing snippet for updates and delete  
  
 ...

On the Hyperty Observer side, Data Objects are also created with the Syncher API and the emerging [Object.observer() Javascript method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/observe) is used to receive the stream of data changes coming from the Reporter Hyperty.

onNotification() {  
 console.info('---------------- Syncher Subscribe ---------------- \n');  
 syncher.subscribe(objectUrl).then(function(dataObjectObserver) {  
 console.info('1. Return Subscribe Data Object Observer', dataObjectObserver);  
  
 // TODO: put source code to add listeners to updates by using Object.observer()  
  
  
 console.info('------------------------ END ---------------------- \n');  
 }).catch(function(reason) {  
 console.error(reason);  
 });  
 }  
  
 ...  
  
 // missing snippet for updates and delete

* Hyperties can easily cooperate with Hyperties from other domains with no federation required or the standardisation of Protocols thanks to the [Protocol On-the Fly powered Messaging Framework](hyperty-messaging-framework.md). Hyperties only have to agree on a common json-schema for one or more Hyperty Data Objects, in order to be able to cooperate each other.
* Hyperties can be used on any Application Domain, but they are specially suitable for Real Time Communication Apps (eg Video Conference and Chat) as well as IoT Apps.

## Hyperty Messaging Framework powered by Protofly (Adhoc MOM)

This document gives an overview on the Messaging Framework (*note to be removed: I guess Messaging Framework is a more friendly term for web developers than Messaging Middleware*) technical solution used to support Hyperty's interaction through the higher level [Data Synchronisation Reporter - Observer communication mechanism](p2p-data-sync.md). Details about how to develop Hyperties is provided in [this](development-of-hyperties.md) document.

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. It should be noted, that [Hyperty Service Development Framework](development-of-hyperties.md) to be used to create new Hyperties, abstracts Developers from the Hyperty Messaging Framework (*note to be removed: too many Frameworks?*) described in this document including lower level Hyperty Messaging APIs.

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.



Hyperty Messaging Delivery Network

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



Hyperty Message Routing Management

The Messaging Framework works at three layers:

At the Runtime Sandbox level where 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 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. Details about how to develop a new Message Node and associated Protostub is provided in [this](development-of-protostubs-and-msg-nodes.md) document.

It is also possible to have P2P communication between Message BUS from different Hyperty Runtime without using any Message Node server (planned for phase 2). P2P Communication between Message BUS will also be based on the protofly mechanism.



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. A detailed description on how to deploy a protostub is provided **here**.

*do we need a more detailed description?*



Protocol on-the-fly and Protostubs

### Message Delivery between different Hyperty Runtimes

Communication between the Message BUS and Message Nodes are provided by a Protostub that implements the protocol stack used to interact with the Message Node e.g. JSON over Websockets or a Restfull API Client. Listeners of protostubs are registered in the MessageBUS for a set of Message recipient addresses, usually a Hyperty Domain like domain://example.com.

When the MessageBUS is processing a new message and looking up routing paths for an address that is not local (eg hyperty://example.com/alice-hyperty), it won't find any registered listeners. In this case, the MessageBUS will ask the Runtime Registry (*point link here*) to resolve the "Message.to" header field, which will look for registered Protostubs that are able to deliver messages to such non-local address. If successful, the Registry will return the Hyperty Runtime protostub address and the MessageBUS will look up again for the protostub listener registered for its address.

## P2P Data Synchronisation: Reporter - Observer Model

This document gives an overview on how Hyperties cooperate each other through a Data Synchronisation model called Reporter - Observer. Details about how to develop Hyperties based on this model is provided in [this](development-of-hyperties.md) document.

The usage of Data synchronisation models in [Web Frameworks](https://www.meteor.com/ddp) looks very promising and is becoming very popular. The usage of the emerging [object.observe](https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Reference/Global_Objects/Object/observe) javascript API is making it even more appealing. However, current solutions require server-side databases that has an impact on performance and scalability.

Hyperty Reporter - Observer communication pattern goes beyond current solutions by using a P2P 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**.



Reporter-Observer Communication Pattern

The API to handle Hyperty Data Objects is extremely simple and fun to use. The Developer of the Hyperty Reporter just has to create the Data Sync object with the Syncher API, and write on the object every time there is data to be updated and shared with Hyperty Observers.

....  
  
 console.info('---------------- Syncher Create Reporter Hyperty Data ---------------------- \n');  
 syncher.create({}, [hypertyURL], {}).then(function(dataObjectReporter) {  
 console.info('1. Return Create Data Object Reporter', dataObjectReporter);  
  
 })  
 console.info('--------------- END Create Reporter Hyperty Data------------------ \n');  
 })  
 .catch(function(reason) {  
 console.error(reason);  
 reject(reason);  
 });  
  
 // missing snippet for updates and delete  
  
 ...

On the Hyperty Observer side, Data Objects are also created with the Syncher API and the emerging [Object.observer() Javascript method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/observe) is used to receive the stream of data changes coming from the Reporter Hyperty.

onNotification() {  
 console.info('---------------- Syncher Subscribe ---------------- \n');  
 syncher.subscribe(objectUrl).then(function(dataObjectObserver) {  
 console.info('1. Return Subscribe Data Object Observer', dataObjectObserver);  
  
 // TODO: put source code to add listeners to updates by using Object.observer()  
  
  
 console.info('------------------------ END ---------------------- \n');  
 }).catch(function(reason) {  
 console.error(reason);  
 });  
 }  
  
 ...  
  
 // missing snippet for updates and delete

### 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.



Parent - Child Sync

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

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

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

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

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

### Syncher and Sync Manager

This section, gives an overview on how the Hyperty Data Object synchronisation transparently works on top of the [Hyperty Messaging Middleware](hyperty-messaging-middleware.md). However, Hyperty developers don't have to know the technical details of this solution and can directly move to the [Hyperty Development Manual](development-of-hyperties.md).

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

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

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

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



Routing Management for Hyperty Data Syncronisation

A detailed description of the Hyperty Data Synchronisation procedures are provided [here](https://github.com/reTHINK-project/core-framework/blob/master/docs/specs/runtime/dynamic-view/data-sync/readme.md)

## Hyperty Trust and Security Model

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

It should be noted, that [Hyperty Service Development Framework](development-of-hyperties.md) to be used to create new Hyperties, abstracts Developers from the Hyperty Trust and Security Model described in this document including lower level Identity Management APIs. Details about how to develop Hyperties is provided in [this](development-of-hyperties.md) document.

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

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



Hyperty Trust Management

### User Identity

### Identity Module and IdP Proxy

*picture illustrating usage of IDP proxy as a special protostub in the Messaging Framework*

### Runtime Sandbox

## Hyperty Development

**Introduce the Criteria to use or not to use Hyperties, the APIs to be used and code snippets**

### Criteria do use the Hyperty Concept

### APIs

### Examples

## Application Development

\*\*An Overview of the Application vs Hyperty and an How To with some examples \*\*

### Application vs Hyperty

### How to use Hyperties

### How to adapt existing Applications

## Message Node and Protostubs Development

**Introduce an Overview of the Messaging Architecture, the Protocol on the fly Concept, the Message Model, Protostub APIs, Message Node functionalities and main Procedures**

### Overview

### Protocol on-the-fly

### Messaging Model

### APIs

### Message Node functionalities

### Messaging Procedures

### Protostub Source Code Examples

# Hyperty Runtime Specification

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

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

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

Four main types of Runtime procedures are described:

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

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

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

## Runtime Architecture

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



Figure (**???**) High Level Runtime Architecture with trusted Hyperties

The Application and the Hyperty can be delivered by the same Service Provider or by different Service Providers, i.e. Hyperty is delivered by an (Hyperty) Service Provider and the Application is delivered by an Application Service Provider. These two different situations impacts the level of trust between the Application and the Hyperty, that should be handled by the Hyperty Runtime accordingly.

In Figure above, the Application and the Hyperty Instances it consumes, are downloaded from the same Service Provider. Thus, it is assumed they trust each other and that they can be executed in the same sandbox with no impact on how the Application consumes the Hyperty Application API. In Figure below, it is depicted the Runtime Architecture where the Application and the Hyperty Instances it consumes, don't trust each other, for example, they are downloaded from different service providers. In such situation, Hyperties and the Application are isolated from each other and they are executed in different sandboxes. In this case, the Hyperty Application API is no longer local and the application is only able to reach the Hyperty Instance through the Message BUS. It is desirable to abstract the Application developer from these situations and to let the Application developer call the Hyperty Application API as if they are always local. This implies that the Core Runtime and the Sandbox implementation, is able to support a Remote Procedure Call (RPC) communication when the Application and the Hyperty Instance are in different sandboxes.

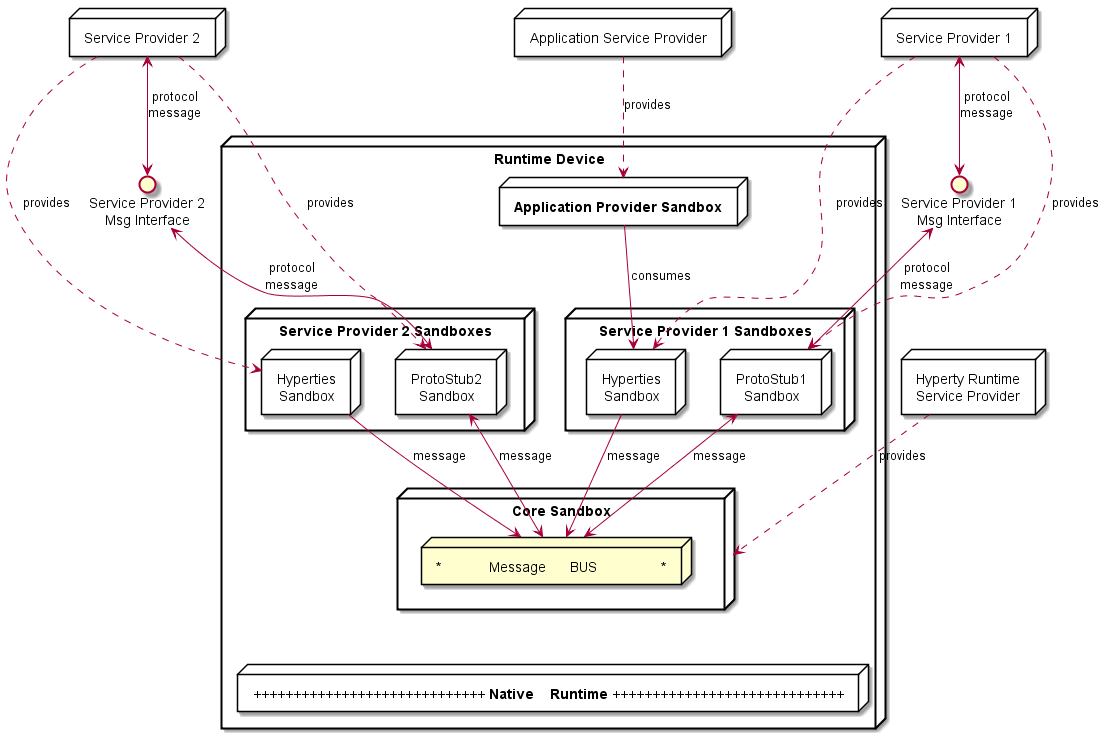


Figure (**???**) High Level Runtime Architecture with untrusted Hyperties

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

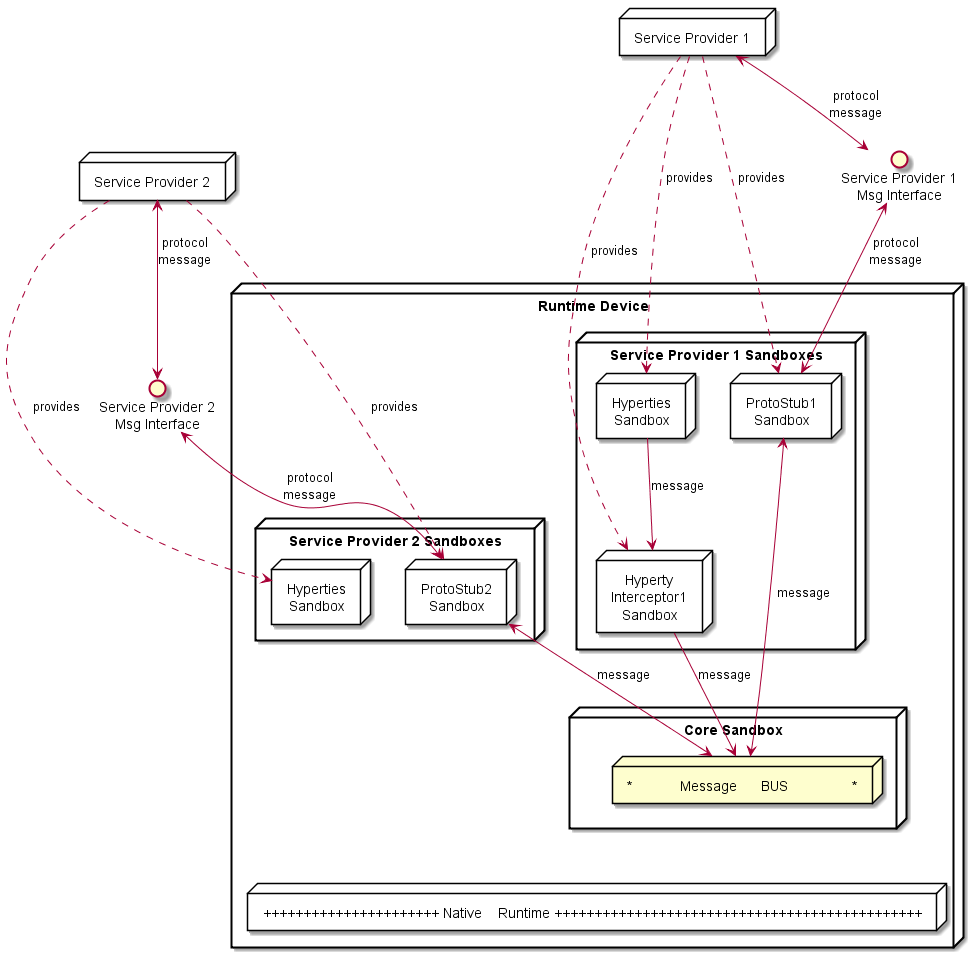


Figure (**???**) High Level Runtime Architecture with domain specific Policy Enforcer

In addition, Core Policy Engine should enforce general access control policies that are agnostic of sender and target domains, or specific to the domain managing the device runtime (Core Runtime Provider). The policies used to control the access to [Hyperty Data Objects](https://github.com/reTHINK-project/dev-service-framework/blob/master/docs/manuals/p2p-data-sync.md) (see below) , are a good example of such policies.

Some more details are provided in the following sections.

### Service Provider Sandboxes

#### Hyperty

As [previously defined, Hyperties](https://github.com/reTHINK-project/dev-service-framework/blob/master/docs/manuals/hyperty.md) cooperate each other via P2P Synchronisation of Hyperty JSON Data Objects supported by the novel [Reporter - Observer communication pattern](https://github.com/reTHINK-project/dev-service-framework/blob/master/docs/manuals/p2p-data-sync.md) and on top of the [Hyperty Messaging Framework](https://github.com/reTHINK-project/dev-service-framework/blob/develop/docs/manuals/hyperty-messaging-framework.md).

#### Hyperty Interceptor

Hyperty Interceptor complements the Core Policy Engine functionality enabling the enforcement of proprietary or closed Policies in the Hyperty Runtime for a specific Hyperty instance.

#### Protocol Stub

The Protocol Stub implements a protocol stack to be used to communicate with the Service Provider's backend servers (including Messaging Server or other functionalities like IdM) according to [Protocol on the Fly](https://github.com/reTHINK-project/dev-service-framework/blob/develop/docs/manuals/hyperty-messaging-framework.md#protocol-on-the-fly-protofly-and-protostubs) concept.

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

### Core Runtime

The Core Runtime components are depicted in Figure below.

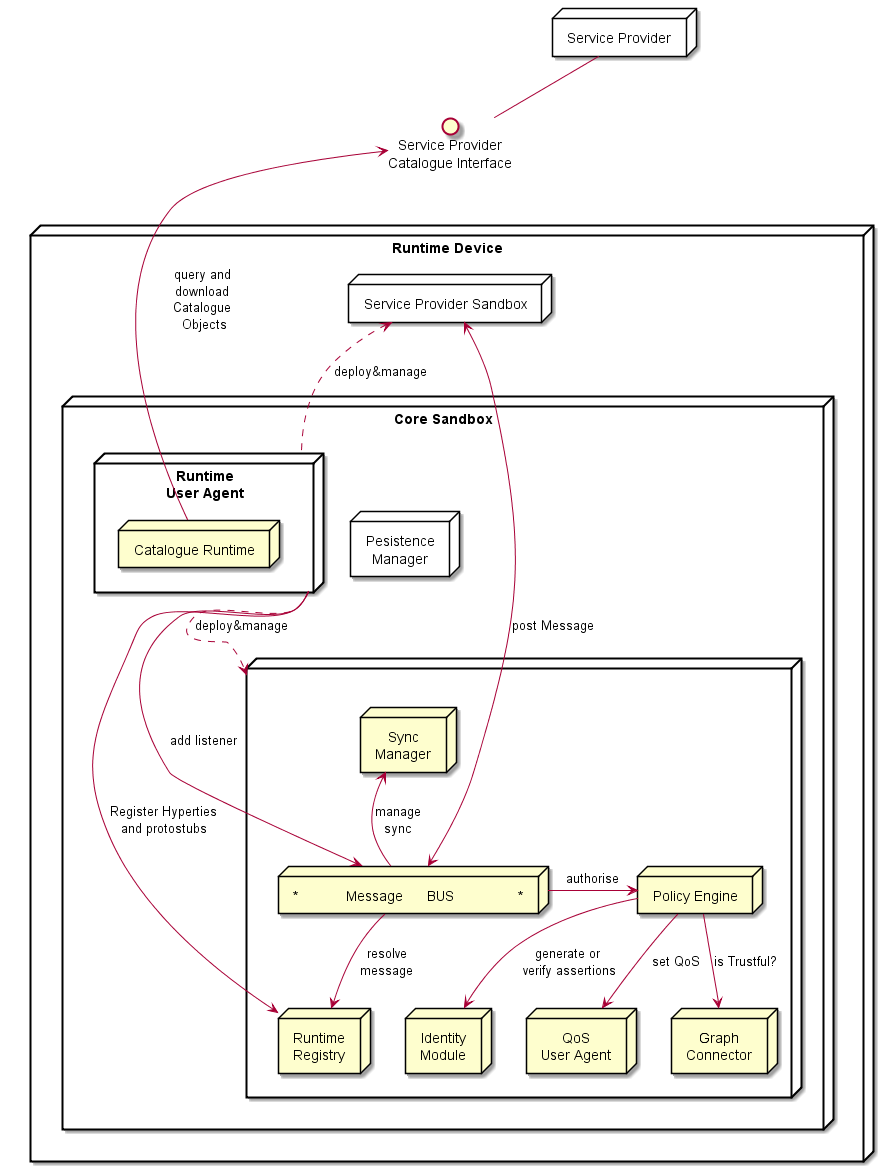


Figure (**???**) Runtime Core Architecture

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

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

#### Message BUS

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

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

#### Core Policy Engine

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

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

#### Runtime Registry

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

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

In addition, the Runtime Registry ensures synchronisation with Back-end Service Provider's Domain Registry.

The Runtime Registry must have listeners to receive messages at:

hyperty-runtime://<runtime-instance-identifier>/registry

#### Identity Module

The Runtime Identity Module manages ID and Access Tokens required to trustfully manage Hyperty Instances communication including trustful association between Hyperty Instances with Users. In addition, it also supports the generation and validation of Identity assertions. Identity module is an extension of [WebRTC Identity](http://w3c.github.io/WebRTC-pc/#identity) and interacts with Identity Providers via IDP Proxy protostubs.

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

The Runtime Identity Module must have listeners to receive messages at:

hyperty-runtime://<runtime-instance-identifier>/idm

#### Runtime User Agent

The Runtime User Agent, manages Core Sandbox components including its download, deployment and update from Core Runtime Provider. It also handles Device bootstrap and the deployment and update of Service Provider sandboxes including Hyperties, Protocol Stubs and Policy Enforcers, via the Runtime Catalogue.

#### Runtime Catalogue

The Runtime Catalogue manages the descriptors of deployable components and Hyperty Data Object schemas that are downloaded from the Service Provider Catalogue via the [Catalogue Service interface](https://github.com/reTHINK-project/architecture/blob/master/docs/interface-design/Interface-Design.md#73-catalogue-interface). The Runtime Catalogue ensures synchronisation with Back-end Catalogue servers.

The Runtime Catalogue must have listeners to receive messages at:

hyperty-runtime://<runtime-instance-identifier>/catalogue

#### Persistence Manager

The Persistence Manager provides data storage functionalities (write and read) to Core Runtime Components including Runtime Catalogue, Runtime Registry, Policy Engine and Graph Connector.

#### Sync Manager

The Sync Manager is in charge of handling authorisation requests to create Sync Data Objects and subscription requests to Sync Data Objects. As soon as authorisation is granted the Sync Manager handles all required MessageBUS listeners in order to setup the Data Sync Stream routing path among Hyperties. The Sync Manager must have listeners to receive messages at:

hyperty-runtime://<runtime-instance-identifier>/sm

#### QoS User Agent

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

#### Graph Connector

The Graph Connector is a local address book maintaining a list of trustful communication users. This functionality is further detailed in deliverable D4.2.

The Graph Connector must have listeners to receive messages at:

hyperty-runtime://<runtime-instance-identifier>/graph

### Native Runtime

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

# Messaging Node Specification

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

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

## Messaging Node Architecture

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

Figure (**???**): Messaging Node Architecture

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

The 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

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

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

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

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

### Subscription Manager

The Message Node Subscription Manager is in charge of handling Subscription and Unsubscription requests from Runtime Sync Managers in order to manage the Data Sync Stream routing path in the Message Node.

The Subscription Manager functionality must have listeners to receive messages for the following addresses:

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

### Protocol Stub

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

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

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

### Connectors

Connectors implements protocol stacks used to interoperate with external elements from the domains. 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.

#### Registry Connector

Registry Connector to interact with remote Registry functionalities. It must have listeners to receive messages for the following addresses:

domain://registry.<sp-domain>

#### IdM Connector

IdM Connector interacts with remote Identity Management functionalities. It must have listeners to receive messages for the following addresses:

domain://idm.<sp-domain>

#### End-User Device Connector

End-User Device Connector to interact with Hyperty Instances running in the end-user device

#### Network Server Connector

Network Server Connector to interact with Hyperty Instances running in a Network Server

# Documentation

bla vla

# Conclusions

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

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

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

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

The reTHINK Core Framework specification is sustained by a comprehensive state of the art research on web runtime and real-time messaging with special attention given to security as well as by an exhaustive work in terms of procurement of existing open source solutions to be used to prototype reTHINK Core Framework components. Taking as input the procurement report, some solutions were selected and some implementation considerations were made. This approach, positions reTHINK prototypes at the forefront of technology with its new functionalities. At the same time it also promotes a rapid and iterative prototyping of reTHINK Core Framework with optimised usage of resources, in order to provide in time, the required components to start the implementation of scenarios in WP5.

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

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

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