Identity Management for Hyper-Linked Entities in reTHINK

Gil Alexandre Marques Dias

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Ricardo Jorge Fernandes Chaves
Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

November 2016

Acknowledgements

I would first like to thank my advisors, Professor Ricardo Chaves and Professor Nuno Santos, for their continuous support on my Master Thesis. Their guidance helped me reach this Thesis goals thanks to their motivation, good will and immense knowledge. They provided me space for my own ideas for the Thesis development, steering me in the right direction when needed.

This work was partially supported by the reTHINK project and I would like to give an appreciation to all teams for always being available to discuss ideas and providing valuable inputs. In particular, I would like to thank the team from PT-Inovação for their tireless support and splendid project leadership taking into account the magnitude of reTHINK. A special gratitude to Ana Caldeira for her cheerfulness, creating a pleasant work environment, and her willingness to help solve problems, that proved to be extremely helpful during the development of this Thesis.

Finally, I would like to express my sincere appreciation to my parents and my brother for their endless support and motivation, since the beginning of my academic course to the end of this chapter, the Master Thesis.

Lisboa, November 2016
Gil Alexandre Marques Dias

Dedicated to my family,

Resumo

As operadoras tradicionais de telecomunicações estão a perder terreno para os novos serviços de comunicação que têm resultado do rápido crescimento da Internet. De forma a tirar partido da vasta rede que a Internet oferece, estes operadores precisam de se reinventar. É esta a principal visão do projeto reTHINK, um projeto Europeu financiado pela Comissão Europeia: desenvolver uma nova plataforma normalizada para comunicações entre dispositivos que usem as tecnologias web atuais.

A gestão de identidades tem um papel importante no projeto reTHINK e é o principal foco desta Tese. Ter uma identidade associada ao dispositivo do utilizador é fundamental para que os utilizadores comuniquem entre si. O objetivo deste trabalho é desenhar e implementar duas componentes chave para suportar a gestão de identidades no reTHINK: o Identity Module e o Runtime Registry. O Identity Module é responsável por suportar o ciclo de vida das identidades dos utilizadores, dos dispositivos e de outros recursos do universo reTHINK. Esta componente é também responsável pelo estabelecimento de canais de comunicação seguros que oferece tanto garantias de confidencialidade como de integridade. Para auxiliar na associação de uma identidade ao dispositivo do utilizador, outro módulo é implementado para gerir esta informação, o Runtime Registry. Este módulo é principalmente responsável por criar essa associação e publicá-la num serviço público de procura. A avaliação dos dois módulos apresentam resultados adequados, e que os módulos conseguem suportar uma carga elevada de pedidos, como demonstrado pelos testes experimentais, resultando na integração bem sucedida da solução no reTHINK.

Abstract

The typical telecommunication operators are losing ground to new communication services that emerge from the rapid growth of the Internet. In order to take advantage of the wide network offered by the Internet, these operators must reinvent themselves. This is the main idea of the reTHINK project, a European project sponsored by the European Commission: to develop a new open and standardised platform for digital communication between user devices using modern web technologies.

Identity management plays an important role in the reTHINK project and is the main focus of this Thesis. Having an identity associated to the user device is crucial to provide the means for users to communicate with each other. The goal of this work is to design and implement two key components to support identity management in reTHINK: the Identity Module and the Runtime Registry. The Identity Module is responsible for supporting the lifecycle of end users identities, devices, and other resources of the reTHINK ecosystem. It is also responsible for the establishment of secure communication channels for users to communicate with both confidentiality and integrity guarantees. To assist in the association of an identity to the user device, another module is developed to manage this information, the Runtime Registry. This module is mainly responsible to create that association and publish it in a public discovery service. The evaluations of these two modules present adequate results, managing to withstand heavy workload as demonstrated by the experimental tests, resulting in the successful integration of the solution in reTHINK.

Palavras Chave
Keywords

Palavras Chave

Autenticação

Gestão de identidade

Ficha de identidade

Integridade e confidencialidade

OpenID Connect

reTHINK

Comunicação segura

Keywords

Authentication

Identity management

Identity token

Integrity and confidentiality

OpenID Connect

reTHINK

Secure communication

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CA Certification Authority

CSP Communication Service Provider

DHT Distributed Hash Table

idAssertion Identity Assertion

IdM Identity Module

IdP Identity Provider

IoT Internet of Things

IV Initial Vector

JWT JSON Web Token

GUI Graphical user interface

GUID Globally Unique Identifier

Hyperty Hyperlinked Entitity

M2M Machine to Machine

MAC Message Authentication Code

MIT Massachusetts Institute of Technology

OIDC OpenID Connect

OP OpenID Provider

OTP One-Time Password

OTT Over-The-Top

P2P Peer-to-peer

PE Policy Engine

PMS Pre-master secret

ProtoStub Protocol Stub

ProtoFly Protocol-on-the-fly

PRF Pseudo-Random Function

RBAC Role-Based Access Control

RP Relying Party

SSL Secure Sockets Layer

SSO Single sign-on

SIM Subscriber Identification Module

TGT Ticket Granting Ticket

TLS Transport Layer Security

URL Uniform Resource Locator

VoIP Voice over IP

W3C Wide Web Consortium

1 Introduction

In today's world, mobile communications are present worldwide and are brought to us as a centralised telecommunication service held by telephone service providers. These service providers are characterised for having absolute control over the communication environment, ranging from the subscriber identification module (SIM) card inside the user device, to the telephone infrastructure. Although this total control is the perfect scenario for the service provider it brings interoperability problems between service providers, resulting in a worse and more expensive experience for the user.

The rapid growth of the Internet introduced new communication services, such as, Over-the-Top content (OTT) delivery method. Communications using OTT services can be in the format of audio, video or other media content and are delivered over the Internet structure. With this approach, the Internet providers are aware of the Internet Providers packets that pass by but are not able to control the information that goes within the packets. Therefore, Internet Providers are not responsible for the content of the packets. The WebRTC standard [1] introduces a new OTT communication service, which allows browser-to-browser communication for audio calls, video calls and file sharing in a peer-to-peer (P2P) manner without the need of an intermediary delivery service. This WebRTC technology brings a major advantage for the development of new services, by removing the need of using traditional signalling methods and by allowing the developer to choose the preferred method for signalling in a P2P communication.

The introduction of Internet communication systems allows for novel approaches that can operate Worldwide without obstacles, contrary to what happens with the Telephone communication services, where physical barriers exist, such as country borders. Although some interoperability exists between these Telephone companies, this type of communication starts to become not only expensive but also outdated, comparatively to the growth of the Internet. The telecommunication providers are struggling to compete against the agility and innovation of web based services. Instant messaging services and Voice over IP (VoIP) via Internet has revolutionised the way people can communicate dethroning the business models of telecommunication services. With a need to adapt communications to modern technologies, the reTHINK project, a new communication framework, arises to introduce a new concept of a Web-centric P2P Service Architecture.

The reTHINK aims to provide a framework, to be run in the user devices, that introduces interoperability among Communication Service Providers (CSPs) and enables the communications between clients using services from different CSPs. It allows CSPs to create services compatible with reTHINK that can communicate with other services from other CSPs. For example, the reTHINK framework is foreseen to allow a communication service created by Skype, reTHINK compatible, to communicate with a communication service created by Google, also compatible with reTHINK. The framework does not restrict the type of information that can be conveyed, which can be voice, text or file sharing, for

example. If two services from different CSPs supports the same type of capabilities, then is possible to create a communication between those two services.

An in depth view on the reTHINK project is provided in the Chapter 2, Related Work, explaining in detail each framework component and the interconnection between them.

1.1 Motivation

Having explained the main purpose of reTHINK project, it is now possible to describe the focus of this Thesis work. Current telecommunication providers are in possession of the entire infrastructure that relays the communications, including the SIM cards in the users' mobile phones. This allows them to create their trustworthy network structure, but leads to others problem such as expandability and geographical extension, such as the borders of the countries. Because reTHINK architecture aims to operate over the Internet layer, new problems emerge, that current telecommunication networks do not possess.

To allow devices to be identifiable, a user identity is required. The standard telecommunication providers make use of the SIM cards to enable the user device to be identifiable and discoverable in the mobile phone network. With the use of the Internet as the platform for the operations, it is required to use other methods in order to allow for the users devices to be identifiable. Given this, identity management becomes an important component in the project, and the larger the population using the service, the more impact it will have. Given an individual identity, it is crucial to provide a trustful management for it, to allow a proper functioning of the architecture.

Considering the current state of traditional communications, it is easy to observe that the association of a person to an identity, a phone number in this case, is rather basic and restricted, whereas the number associated is limited to a single SIM card. Also, the act of carrying an identity to another telecommunication provider is laborious and time consuming, primarily because it is not of the interest of the company to facilitate this operation, for competition reasons. Although a communication between two users using the telecommunications' infrastructure can be classified as private, given the use of encryption, this communication is not completely confidential, mainly because telephony services have access to both identities and infrastructures used for the communication, gathering all the mechanisms required to restore a conversation that occurred in the past. So with this problem in mind, reTHINK aims to give a clear distinction and separation of operational services on the entities providing the identity and communication services.

With the separation of the CSPs and the Identity Provider (IdP) services, reTHINK creates the need for authentication of users at the beginning of a communication in a decentralised fashion. Since a centralised authority to authenticate the users who wish to initiate a communication does not exist, the user devices need to assume the responsibility for the authentication process, in order for a trustful communication to be established. Because the communication is intended to be in a P2P manner, this requires the user's device to ensure the properties of a secure communication. Therefore, the reTHINK framework running in the user's devices must be capable of providing the required mechanisms to establish a secure communication.

For a device to have an associated identity and to become identifiable, the reTHINK framework must provide mechanisms for the users to freely choose and obtain an identity from several IdPs services. This identity is then associated it to the user device, allowing the device to be discoverable.

1.2 Goals

To instantiate and execute the obtained Communication Service Providers (CSPs) communication services in an end user device, in order to provide a communication channel between two users, it is required to do the storage and identity management in the reTHINK framework. The Objective of this dissertation consists in the development of a front-end Identity Module component in the reTHINK framework which runs on an end-user device, providing the capabilities of identity acquisition, authentication, and access to resources using an identity selected by the user. An appropriate management of identities and access tokens is required, to trustfully manage communication between user devices including the association between those identities and the user devices.

The acquisition of an identity can be performed using different IdPs, for example Google or Microsoft. The Identity Module must allow users to register to a CSP service using the identity obtained through the authentication mechanism. This module is intended to prevent CSPs from monitoring the communications performed by the user devices, providing privacy of communications.

This Identity Module was developed in JavaScript, a high-level, dynamic programming language supported by the majority of web browsers, in order to be supported the maximum number of devices. Since the information about the identities and other components are required to be stored, until the user ends the application, a Registry module also needs to be developed in order to support the needed storage mechanism. This Registry module will handle not only the registration of Identity Module, but all the components required to run in the end-user application referring to the reTHINK framework. The Registry will handle the allocation of URL addresses for all these components, and ensure synchronisation with the public Back-end Service Provider Registry.

1.3 Requirements

This section lists the several requirements needed to achieve the objectives set for this project. Some of the requirements are already established by the reTHINK consortium, but others are established to ensure a strong and reliable identity management solution.

ReTHINK with the motivation to separate Communication and IdP service, must enable end-users to choose a communication service from a CSP and an identity from external IdPs. For example, the users should be able to select a communication service from Skype and an identity from Google. Also, the end-user should be able to register and authenticate to a CSP, using an identity from a separated Identity Service Provider. From the end-user point of view, to switch identities between domains should be easier than it currently is with the traditional phone numbers, and should as much as possible be automated and transparent to the user.

It is required for the identity component, running in the user device, to store information about the selected communication service and other relevant information for the service delivery, when the user selects a communication service from a CSP or an identity from an IdPs. This information must remain valid until the user exits the service. Another requirement comes from the user device compatibility, since this system is intended to run in multiple devices worldwide, namely the reTHINK framework should use tools that offer the best possible compatibility to be deployed in most devices.

Regarding the trust level of identities, when a user device receives a request for a communication from another user, it must verify the truthfulness of the identity. To do this, it must start a mutual authentication process, so that each user device is validated with the claimed identity. Additionally, because the communication works in a decentralised fashion, the user device should offer to the user the option to enable a secure communication, ensuring a communication between users with confidentiality and integrity, provided only to the participating parties.

1.4 Contributions

The work of this Dissertation resulted in several contributions for the European reTHINK project. Regarding the development, three major components were developed for the reTHINK framework, namely:

- Implementation of an Identity Module to allow the users to obtain an identity from Identity Provider and use it in reTHINK. For this component the necessary mechanisms to make this component more modular were also implemented, supporting several Identity Service Providers for the identity acquisition.
- Development of a mutual authentication protocol to be used by the Identity Module, allowing the creation of secure communication channels within reTHINK.
- Implementation of a Runtime Registry that manages several internal components from the reTHINK framework. This component is responsible for the association between a user identity with services running in the reTHINK framework, and for storing it in a public discovery service.

Part of this work was already published at the 19th International ICIN Conference - Innovation in Clouds, Internet and Networks [2].

1.5 Structure of the Document

The rest of this document is organised as follows. In Chapter 2 a description on the state of the art for the identity management and acquisition is provided. Chapter 3 introduces the reTHINK project, explains its goals, and provides the framework architecture. Following with the introduction of the Thesis work in Chapter 4. The proposed architecture is provided along with the mechanisms used to allow the acquisition of identities. The details on how the secure communications are implemented are also provided. In Chapter 5 implementation details for each component developed for this Dissertation are

presented, as well the decisions made for some problems. Chapter 6 shows the evaluation results for the components developed. To conclude, in Chapter 7 a summary of this Dissertation and a list of future work for this project is provided.

2 Related Work

As the necessity to manage identities over the Internet has increased, so does the number of solutions to help achieve that objective. Not only for the Internet, but also for large companies with a great number of workers exists the necessity to do a proper Identity management of all them. This digital identity, as will be explained below, is an identity created from the need to provide an authentication for someone or something, and is used in different ways by diverse systems that requires some sort of identity. Over time several solutions emerged to fulfil specific problems in projects, mainly because different approaches are required to overcome distinct obstacles. This Chater starts with the introduction of the concept of Identity, and how it can be used. Later authentication mechanisms are described along with their practical use. To conclude, several identity management systems and data structures for browsers are analysed.

2.1 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, where 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. Since identification binds a person identity with the respective individual attributes [3], an authentication of identity is required. Given this, and since the majority of the current transactions are performed digitally, we need a digital identity to prove who we are in remote communication.

This concept of Identity comprises two important information security mechanisms, the authentication and authorisation [3]. In a short description, the authentication is an identification followed by verification. In this identification process an entity supplies its identity, while in the verification process, the identity provided is checked before the system. Therefore, the correctness of an authentication strongly depends on the verification procedure employed. The authorisation is the decision to allow a given identity to execute or access a certain resource. Access control to a service or system, can be achieved based on authorisation mechanisms, where is possible to define the access rights or policies for each Identity, thus making it possible to decide to allow or deny a particular action based on an identifier or attribute. This is useful if the system requires having access restrictions.

Traditionally, the authentication is performed with something a user knows (like a PIN or a password) or holds (such as a key, or a magnetic card). But there is another method, biometrics, which can be used to authenticate users [4]. Biometrics are automated authentication methods based on measurable

human characteristics, such as voice samples, fingerprint, or facial features. There are two types of biometric identifiers, the physiological and behavioural characteristics. The physiological characteristics represent the physical aspects of a person, which can be used for biometric authentication, for example, fingerprints, iris pattern, ear features, among others. Behavioural characteristics are related to individual behaviour patterns, such as gestures, voice samples, typing rhythm, and others. However, biometric methods are not easily used for remote authentication, as such they will not be herein considered.

2.1.1 Access Token

In today's world, most of the information exists in digital format, where it is possible to find articles, films, music's, and many others. With adequate infrastructure, this information can be accessed anywhere, by someone with the right authorisation, without the need of a physical barrier. To facilitate this process of user authentication and authorisation towards a system, a new concept of access token has emerged [5]. This token, allows for a user to access a resource just by providing the access token, if still valid, will be accepted by the system. The fetching of this token can be done using the Oauth 2.0 protocol. An example of an access token can be seen in figure 2.1.

```
{
  'access_token':'2YotnFZFEjr1zCsicMWpAA',
  'token_type':'bearer',
  'expires_in':3600,
  'refresh_token':'tGzv3JOkF0XG5Qx2TlKWIA'
}
```

Figure 2.1: Access token example

Oauth 2.0 is an authorisation framework [6] which allows clients, also named a 'bearer', to access protected resources by obtaining an access token, instead of using the owner's credential directly. This token contains the information about the access authorisation issued to a client. To prevent eavesdropping attacks[1], the Transport Layer Security (TLS) protocol is used, when sending the token over HTTP. If the user possesses this bearer token, he does not need to prove possession of cryptographic key material (proof-of-possession).

Before a client is able to access a resource, he must first obtain an access token through a protocol flow. This flow typically requires four participants: the client, the resource owner, the authorisation server, and the resource server. The client is the application that wants to access the user's account, the resource owner is the user who authorizes the client, the authorisation server is the entity responsible for issuing access tokens, and the resource server is the repository hosting the protected resources. The Oauth2.0 protocol flow, illustrated in Figure 2.2, describes the interaction between all four participants. Step 1 and 2 describe a request for authorisation by an application to the user, and the response with the authorisation grant. In steps 3 and 4, the client requests an access token to the authorisation server, providing the authorisation grant obtained by the user. The authorisation server, after checking the

[1] Unauthorised interception of private communications

validity of the authorisation grant, replies with the access token. Steps 5 and 6 occur when the client is in possession of the access token and requests a resource from the resource server. The client sends together with the request, the access token with the authorisation for the resource. If the access token is valid, the resource server returns the information requested by the client. To send the access token from the client to the resource server, three methods are available. The access token can be sent in a header field in a HTTPS request, in a body parameter in a HTTPS request, or as a URI query parameter.

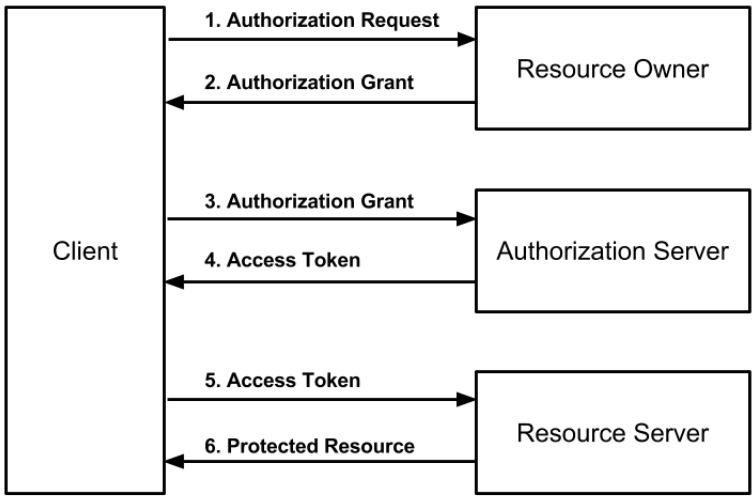


Figure 2.2: Abstract Oauth 2.0 Protocol Flow

The access token provides an abstraction by replacing different authorisation schemes, for a single token recognised by the resource server. With this abstraction, it is possible to issue access tokens with a validation for a short time period, as well as remove any kind of authorisation mechanisms on the resource server side.

2.1.2 One-time password Token

In a Web communication among two entities, replay attacks[2] are a constant threat, particularly if the attack is done to replay a non-authorised user authentication. To minimise this problem, a One-Time Password (OTP) authentication system can be put into practice[7]. This authentication system generates OTPs with the objective of being used only once, using a secret to generate them. With this method, the secret pass-phrase never needs to pass through the network.

The process of generating OTP requires two entities, a server and a generator. The server is responsible for sending a challenge for the user, and the validation of the OTP created by the user. The generator must create an appropriate one-time password using the challenge received by the system,

[2]Replay attack occurs when an attacker collects data between two parties and deliberately repeats or delays this same data to its own benefit.

and the user's secret. In the generation of OTPs, the seed received from the server is concatenated with the pass-phrase known by both, and a hash function is applied to the result of the concatenation. In the end, even if the seed provided to the user by the server is transmitted through the network in plain text, the attacker will not be able to generate the OTP.

One-time passwords are good alternatives to ordinary passwords, but not feasible for humans to memorise them, so auxiliary technology is required to aid in this task. Since the introduction of OTPs, many methods of delivering them exist, such as security tokens as dedicated hardware, with the sole purpose of providing an ease in the authentication step, or even mobile phones, containing applications with a OTP generation algorithm, or via text messaging, where the user receives a message with the OTP.

Considering the example of a solution using a mobile phone as a security token, as presented in [8], combines the user's computer and mobile phone to generate an OTP. Both user devices run a specific code to manage a simple challenge-response protocol between them, where the computer running a Java applet, communicates via Bluetooth with a MIDlet, running in the mobile phone. So whenever a client needs to authenticate towards a service, his personal computer, sends a challenge to the mobile phone, which in turn generates an OTP using the challenge as a seed, and send it to the server. In the end, the server checks whether the OTP is valid or not and in case of being valid, authenticates the user. This system provides a good solution for of an OTP. However, it is not without its problems, since the user is required to have both devices operational and close to each other.

2.2 Authentication Mechanisms

Remote user authentication can be performed, using different methods [9], e.g. using passwords, tokens, or even authentication involving trusted third parties. In this section, several token-based authentication protocols are presented.

2.2.1 BrowserID

The Mozilla Corporation developed a new single sign-in login system for web browsers, the BrowserID [10]. This protocol was built with the idea of using email addresses as an identity. The main motivation behind this protocol is to prevent users from having to memorize or store multiple passwords and usernames for each site and service, and facilitate the user authentication process.

The BrowserID protocol involves three actors to grant authentication, users, Relying Parties (RPs), and Identity Providers (IdPs). Users are the entities who want to sign into a website or a service, using web browsers as Relying Parties, to achieve their objective. The Identity Providers are the domains responsible for issuing identity certificates, compatible with BrowserID, to their users. This protocol requires three distinct steps for a successful authentication, namely the user certificate provisioning, the assertion generation, and the assertion verification.

The certificate provisioning is the process where a user proves ownership of their email address. The main piece of this proof is a cryptographically signed certificate provided by an IdP, certifying the

connection between a web browser and an identity, in this case the email address from a certain IdP domain. This certificate is signed by the IdP with a RP public key, generated before the request is made. Although a verifiable link between an identity and a public key can be established with the user certificate, this alone is not enough since the user is still required to prove ownership of his private key.

In the assertion generation step, the user's browser creates and signs a new document named 'identity assertion', containing information about the origin of the web browser and an expiration time for the assertion, signed with the user's private key. This identity assertion along with the certificate is then provided to the website the user wishes to sign in, by the browser.

In the end, in the assertion verification step, the RP validates the signature in the identity assertion with the public key inside the user certificate. Once verified the identity assertion, the RP requests the IdP public key, to check whether it matches with the signature in the user certificate. After performing all verifications, the RP authenticates the user in the system, with the identity provided in the certificate.

This BrowserID protocol is introduced with good concepts, however it presents some issues. The main problem is the complexity of the solution: BrowserID solution produces a large quantity of messages to authenticate a user towards a website.

2.2.2 WebID-TLS

The World Wide Web Consortium (W3C), have designed and developed the WebID [11], an open standard for identity and login. The WebID is a URI which refers to an identity and makes use of existing infrastructure implemented in browsers to enable a global and decentralised identity system, with which is possible to create a web of trust.

This WebID-TLS protocol allows users to authenticate into any site using a digital identity (WebID) linked to a certificate. The web browser usually is in possession of the associated private key, while the public key, along with the WebID are part of the certificate. This authentication protocol uses the Transport Layer Security (TLS) standard, to provide security during the communication.

This protocol requires the existence of a Key Store in the client device, to store client certificates and sign cryptographic tokens. A certificate with a user URI entry must be created and stored in the Key Store. Because users are allowed to be in possession of multiple certificates, it is important to choose a user friendly Common Name, in order to allow for the distinction between different certificates.

To briefly describe how the WebID-TSL works, let's assume a communication between the users Alice and Bob. When Bob tries to access Alice's resources, Bob starts a communication using a TLS, and provides his certificate to Alice Web server, which will extract the Bob WebID profile using the URI from the certificate. Upon confirming that Bob's WebID profile is related to the public key from the certificate, Alice knows she is interacting with a user referred by Bob's Profile. Alice server then checks Bob's identity in the graph of relations in order to determine the trust level according some criteria. In the end, depending on the established trust level, the access can be granted fully, partially, or even denied.

With the increase of a social component in the Web over the last decade, the WebID protocol appears with the interesting concept of Web of Trust, and decentralised authentication protocol. However it also entails the several problems of Web of Trust. Many single points of failure can emerge, such as

the lack of a certificate authority for the certification of the whole web of trust, or the necessity to store in the device of the user, information from all the nodes, where, for a large web of trust, the information size can become a burden to the device.

2.2.3 OpenID connect

OpenID connect 1.0 is a simple authentication layer [12] on top of the OAuth 2.0, an authorisation framework [6] described in section 2.1.1. This protocol allows clients to verify the identity of the End-User based on the authentication performed by an authorisation Server, and obtain the information about the End-User identity using RESTful services.

OpenID connect provides a framework for a wide range of third-party applications, such as mobile, web based, JavaScript client, to obtain and use Access Tokens to access resources and to request and receive information about authenticated sessions. When a request is made by a Client to an OpenID Provider (OPs) the information about the authentication performed is returned in a JSON Web Token (JWT)[3], called an ID Token. Also OAuth 2.0 Clients using OpenID Connect are referred to as Relying Parties (RPs).

The OpenID Connect protocol requires several steps to obtain the final ID token. As depicted in Fig.2.3. The relying party, that can be a mobile phone for example, sends, in step 1, a request to an OpenID Provider. In step 2, this Provider sends a request for the end-user to authenticate himself, so he can obtain the authorisation. The OpenID Provider after confirm the end-user identity, responds to the relying party with an ID Token, and an Access Token. In step 4, the user application, within possession of the access token, can make requests to the Userinfo endpoint for identity tokens or other information from the user, which will be returned in step 5.

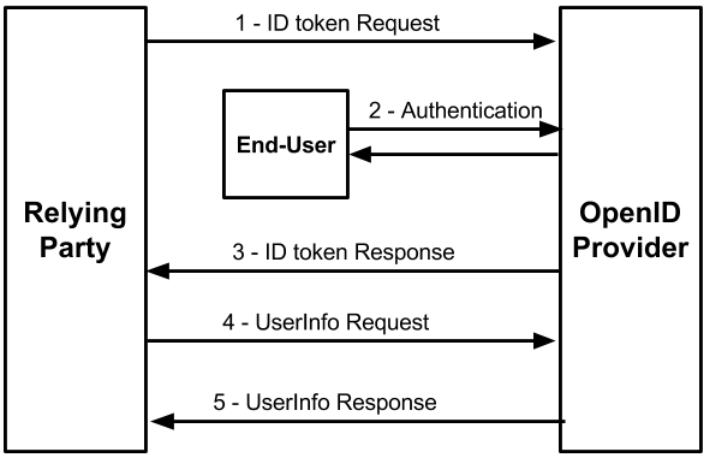


Figure 2.3: OpenID connect protocol flow

[3]A description of a JWT can be found at `https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32`

The ID Token represented as a JWT is a security token containing Claims about the Authentication of an End-User, signed by the OpenID Provider. The ID Token contains several fields, also known by claims, to be used for all OAuth 2.0 flows by OpenID connect. Although the claims inside ID Tokens may vary from different Providers, it typically contains the following fields:

- iss - Issue Identifier for the Issuer of the response.
- sub - Subject Identifier.
- aud - Audience(s) that this ID Token is intended for.
- exp - Expiration time of the ID Token.
- iat - Time at which the JWT was issued.
- auth_time - Time when the End-User authentication occurred.
- nonce - String value used to associate a Client session with an ID Token, and to mitigate replay attacks.

This ID Token is flexible and can be used in multiple ways. Can be stored in a browser cookie to implement lightweight stateless sessions, passed to other components of the application or to backend services.

The OpenID connect authentication by the End-User, can be performed by three different flows: authorisation Code Flow, Implicit Flow, or Hybrid Flow. The authorisation Code Flow is intended for traditional web and mobile applications. With this flow, the OP redirect the user to a page requesting the authentication and consent, which later returns the ID token via a second channel. The Implicit Flow is meant for JavaScript applications running in the browser that do not have a backend. The ID token is received directly from the OpenID Provider in the redirection response. The Hybrid Flow some tokens are returned to the application front-end and others are returned to the back-end.

This authentication protocol is widely implemented and used throughout websites, since it has the support of reliable and well-known Identity Providers, such as, Google or Microsoft.

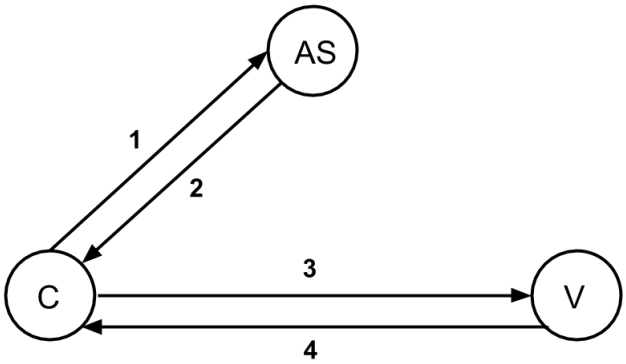
2.2.4 Kerberos

In the traditional system of authentication, the user is required to insert a password during the login, but password-based authentication has flaws, when used on a computer network. The passwords sent through the network can be intercepted and later used by eavesdroppers to impersonate the user. In Kerberos, a principal is the party needing his identity verified and the verifier is the party who validates the principal's identity. There are many authentication mechanisms with different type of assurances, as they differ in the number of verifiers, for example, some support a single verifier per message, others support multiple. It is important to highlight, that different implementations will have different performances. Also relevant is to know the requirements of an application, when choosing the method.

To mitigate this password authentication problem, Kerberos was developed in the mid-80s as part of Massachusetts Institute of Technology (MIT) Project Athena. Kerberos is a distributed authentication

service which allows a process (a client), running on behalf of a principal (a user), to prove its identity to a verifier (an application server) without sending information through the network, that might allow an attacker or the verifier to subsequently impersonate the principal [13]. For Kerberos authentication protocol to prove to a verifier that a client is running on behalf of a specific user, uses a chain of encrypted messages. This Kerberos protocol is based on the Needham and Schroeder authentication protocol [14] with some improvements to mitigate some limitations. Included in the changes is the requirement of timestamps to reduce the number of messages required for basic authentication and to prevent replay attacks.

For the Kerberos protocol to work properly, it requires an authentication server, which will maintain a database with all the passwords for all the users in the system. As such this server must be secure and adequately protected. When a client wants to authenticate towards other entities using the Kerberos protocol it will produce a flow of message between these 3 parties, the client (as the principal), the authentication server, and the destination entity which will have the role of verifier. The messages 1 and 2, depicted in Figure 2.4, represent the first step the principal requires to do, so he can obtain a Kerberos ticket. In message 1 the client sends to the authentication server his identity along with the name of the verifier and a requested expiration time for the ticket. The authentication server then replies to the principal, with message 2, containing the session key, the name of the verifier, the assigned expiration time, a random number from the request, and a Kerberos ticket to be used to distribute to the verifier, all encrypted with the key of the principal. This Kerberos ticket is a certificate issued by an authentication server, encrypted with the server key. Inside this ticket, a random session key is provided to be used between the client and the verifier, the name of the client to whom the session key was issued and an expiration time which represents the session key validity time. The server key used to encrypt the Kerberos ticket is assumed to be known only by the authentication server and intended verifier, so the client cannot tamper the ticket without detection.



1. as_request: c,v, $time_{exp}$, n
2. as_response: {$K_{c,v}$, v, $time_{exp}$, n, …}K_c, {$T_{c,v}$}K_v
3. ap_request: {ts, ck, $K_{subsession}$, …}$K_{c,v}$, {$T_{c,v}$}K_v
4. ap_response: {ts}$K_{c,v}$ (optional)
$T_{c,v}$ = $K_{c,v}$, c, $time_{exp}$, ...

Figure 2.4: Simplified Kerberos authentication protocol

After the client receives the response from the authentication server, it sends message 3 to the verifier, containing the ticket and an authenticator, including the current time, a checksum, and an optional

encryption key. Upon receiving the message the verifier will extract using the server key, the session key from the ticket to be used to decrypt the authenticator. If the key used to encrypt the authenticator is the same obtained from the ticket, which will be used to decrypt the authenticator, then the verifier can compare and see if the cryptographic hashes are the same. This allows attesting that the message was not tampered, along with the timestamp to check the freshness of the message within an interval of time. Message 4 is used when a mutual authentication is required between the client and the verifier, where the verifier extracts the client's time from the authenticator and returns it to the client along with other optional information, encrypting the message using the session key.

At the end of the exchange, messages 1 to 4, the client and the verifier can now exchange messages in a secure communication using the session key provided by Kerberos. It is relevant to highlight that the Kerberos protocol has small variants where different types of attacks are mitigated. Herein, we just detail the core protocol.

The Kerberos authentication protocol allows users with the right credentials to obtain a ticket and a session key, but if a user is required to go through these steps, every time he needs to obtain a session key, this can overload the Kerberos infrastructure. To resolve this problem, the Kerberos system can support a single sign-on (SSO) architecture [15]. The SSO represents the ability of a user to authenticate once towards a system and then access other resources without the need to re-authenticate. The SSO arises to unload the traffic on a system, reduce the risk of a password exposure and to remove the necessity of users having multiple password credentials, which may lead to a bad management of these passwords.

The solution implemented in Kerberos to support SSO is to cache only tickets and encryption keys, which can be used within a small time interval. The ticket granting service of the Kerberos protocol allows a user to request tickets and encryption keys using short-lived credentials, without the need to re-use the user's password. When a user first logs into a system, he receives a ticket, known as ticket granting ticket (TGT) with a relatively short life. Later when the user needs prove its identity to a new verifier, the user sends the TGT to the ticket-granting service to request a new ticket. When the identity is successfully validated, the system provides the ticket requested by the client, encrypted with the session key from the TGT. This system provides a good mechanism to minimize the exposure of a user password.

2.2.5 Smart Cards

A smart card is a device with the capability to store data and execute commands, embedded into a plastic card [16]. This smart card device is built to be tamper resistant [4], in order to protect the stored data and the computation against physical attacks. This device is like a portable computer with the benefits of being extremely secure and portable. An important feature of the smart card is that the information inside it cannot be duplicated, allowing to securely store secret keys, which can be used in multiple scenarios, where authentication is required. For example, the card can be used in a building access system, to contain the digital key required to open the door.

[4] Tampering consist in deliberate adulteration of a system, and in this case, in the information contained within. Tamper resistance means this smart card cannot be easily tampered.

Taking in consideration the Internet environment, many resources require authorisation to access them, and the password authentication has been embraced as one of the favourite solutions. These passwords, should be large and preferable with random characters, but this can be a problem for the humans which tend to choose, not only, passwords easy to remember, but also easy to guess through dictionary searches [17]. Given this, smart cards arise as a good solution to mitigate this problem, by allowing the use of smart cards for authentications, and providing the access in a client/server environment [18]. A common smart card authentication scheme, allows users to authenticate with the card as an identity token, by taking as input a PIN from the user, then creating a login message from the stored key to be delivered in a remote server, which will in the end confirm and validate the login message, so the user can access the system. Due to their low cost, tamper resistance and portability, the smart card is an alternative to authentication mechanisms which can be deployed widely in diverse systems.

2.2.6 Transport Layer Security

The Transport Layer Security (TLS) [19] is a protocol to operate on top of TCP layer [20], a reliable transport protocol. Although the TLS is not mainly an authentication protocol, it incorporates an authentication phase essential ensure that the protocol runs with the desired security properties, and is able to initialise and establish a secure channel.

The TLS protocol was designed to enable the creation of secure channels to provide secure communications between two parties over the Internet. The protocol allows a client and server applications to establish a secure communication channel that prevents eavesdropping, message forgery and tampering. The TLS is the successor of the Secure Sockets Layer (SSL) protocol, which shares an identical protocol flow with some improvements on the cryptographic methods used. This TLS protocol has in its constitution two layers: the TLS Handshake Protocol and the TLS Record Protocol.

The Handshake Protocol is used to authenticate the server by the client or mutual authentication in case the client provide his certificate, and negotiate a shared secret that will serve as input to the function that generates the keys to be used in the TLS Record Protocol phase. This Handshake Protocol requires several messages exchanged, in order to establish the final key to be used to in the TLS Record Protocol, which is illustrated in Figure 2.5.

The Record Protocol provides a secure communication with both confidentiality and integrity on the messages exchanged and uses the keys generated previously during the Handshake Protocol. The confidentiality is granted by encrypting the data using symmetric cryptography, while the integrity is granted by doing integrity check on the messages using a keyed MAC. The keys used in the encryption and hash functions are obtained in the end of the Handshake Protocol, which will be detailed below.

The flow from the Handshake Protocol requires several steps between two parties (i.e. a client and a server) in order to be able to negotiate a session. These steps are:

1. **Hello request**: This message is used to initiate a new Handshake and can be requested to the client, by the server, at any time.
2. **Client Hello**: This message contains a random value composed by the current time and date of the client and a 28 bytes generated by a secure random number generator and also a cipher suite

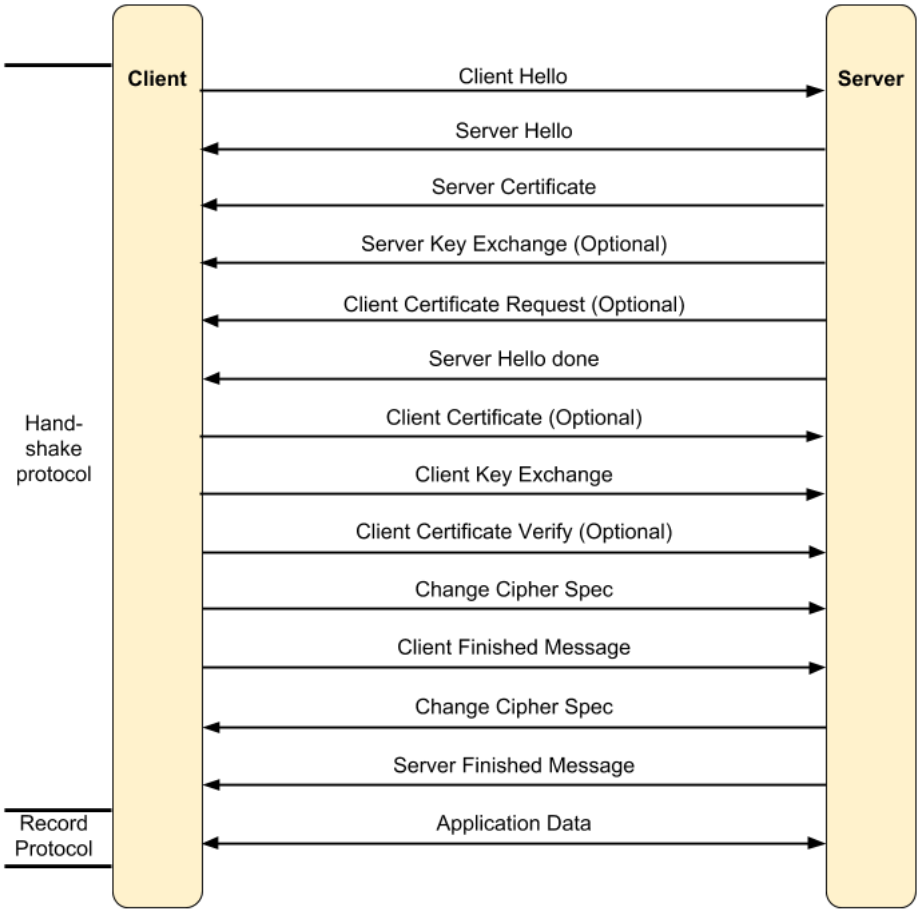


Figure 2.5: TLS flow

list with all the cryptographic combinations supported by the client.

3. **Server Hello**: This message is the response of the Client Hello. The server then verifies if there is any set of algorithms in the list of cryptographic algorithms sent by the client is supported by the server. If a match is found, the server creates a random number and informs the client of the set of algorithms chosen. Otherwise respond with a handshake failure alert.

4. **Server Certificate**: This message contains the server certificate used for authentication.

5. **Server Key Exchange**: This message is optional and is sent when the server certificate sent previously does not contain enough information to allow clients to exchange a premaster secret, i.e. a server public key. This message contains a Diffie-Hellman public key so that the client can complete a key exchange or a public key for others algorithms.

6. **Client Certificate Request**: This message is optional and contains a request for the client certificate, a list of hash/signature algorithms supported by the server and a list of acceptable Certificate Authorities.

7. **Server Hello Done**: This message is sent to indicate the server has ended sending messages and will be waiting for the client response.

8. **Client Certificate**: This message is optional, is only sent when the server requests the client certificate and contains the client certificate. If the client does not send the certificate or is not the appropriated certificate, the server may reject the Handshake.

9. **Client Key Exchange**: This message contains a premaster secret generated by the client or a Diffie-Hellman parameter that allows each side to agree on a premaster secret. In case the client generates a premaster secret it must be with 48-byte generated randomly and must be encrypted with the public key from the server certificate.

10. **Client Certificate Verify**: This message is optional and is only sent when the clients provides previously his certificate and serves to provide explicit verification of the client certificate. It contains all the handshake messages received and sent since the beginning of the Handshake protocol digitally signed with the private key corresponding to the client certificate.

    In the phase both client and server can generate a master key, using the premaster secret and both client and server random exchanged in the beginning. With this master key are computed the following keys: a client MAC key, a server MAC key, a client encryption key, and a server encryption key.

11. **Client Change Cipher Spec**: This message is sent to notify the server that the following messages will be protected under the newly cipher specifications and keys.

12. **Client Finished Message**: This message contains the hash resulted from the output of a pseudo random function that receives as input the master key, a finished label with the string ”client finished” and the hash of all the handshake messages exchanged in the handshake protocol. This hash is then encrypted with the client encryption key.

13. **Server Change Cipher Spec**: This message is sent to notify the client that the following messages will be protected under the newly cipher specifications and keys.

14. **Server Finished Message**: This message contains the hash resulted from a pseudo random function that receives as input the master key, a finished label with the string ”server finished” and the hash of all the handshake messages exchanged in the handshake protocol. This hash is then encrypted with the server encryption key.

    The Handshake Protocol ends, and starts the Record Protocol with the keys established in the Handshake Protocol.

15. **Application Data**: Communication between the client and the server using the cryptographic functions and keys negotiated during the Handshake Protocol. The message is encrypted with the sender encryption key and created a MAC of the entire message with the sender MAC key.

TLS supports several protocols in the handshake phase for it to be successfully accomplished, providing a range of configurable parameters for message integrity, data encryption and keys exchange. Additionally the property of perfect forward secrecy [5] can be assured, assuring the TLS is well configured and

[5] Perfect forward secrecy is a property that ensures that session keys obtained through a derivation of a long-term key are not compromised if this long-term key is discovered in the future.

uses protocols in the handshake phase that ensures the perfect forward secrecy, such as Diffie-Hellman for the key negotiation.

Given the robustness of the TLS protocol in providing a secure connection over the Internet, the last version of TLS, version 1.2, is currently defined as the standard protocol for secure communications between servers and web browsers, providing privacy and data integrity.

2.3 Management and Storage

Identity management systems were created to agglomerate a set of tools, in order to provide ease in the administration of these identities. Also, the process of authenticating a user requires sometimes the user device to store information regarding the identity. This section consider the identity management systems available in the market, along with existing web browser storage options.

2.3.1 Identity Management

Most organizations start having problems with the management of the registered users, when they start to grow, due to the lack of scalability [21]. Taking a university as example, every year thousands of students arrive to the institution, and all of them need to have access to resources, such as the student registration for classes or computing labs. On the other hand, former students that leave the institution should have access to some resources revoked, or, in case they opt to continue to work in the university, have their role changed to have different privileges.

Identity management systems were designed and created [21], to aid in the management of users identifications, as well as their respective lifecycle of roles inside the organization. These systems are strongly integrated with other systems, such as, single sign-on and authentication mechanisms, to provide protection and facilitation to a user which has the intention to access resources. Typically an Identity management system comprises the following points:

- Collection of technology - Identity management systems are not individual systems, but a conglomerate of several components to provide a complete solution for the organization.
- Business process and underlying policy - Organizations have their own business processes, which must be reflected in the Identity management system. For example a reliable process is required upon the registration of new users, to ensure that a proper assignment of systems resources is performed. The Policies refers to what a user should or should not have access to, and for how long.
- Networked systems - One of the focus in an Identity management system is to provide availability through the web network. The user should be able to login to any resource in the organization, i.e. a mail server, and a single sign-on system should be put in practice to facilitate the authentications towards other resources.

- Access and authorisation - The access management certifies whether the credentials provided by the user, matches with who they claim to be. While the authorisation refers to what the user is allowed to do. This access control usually is performed using a Role-Based Access Control (RBAC) [22], to provide a more appropriate and central for access control, based on policies defined to each user.

Regarding the Identity management systems, there are many offers by various companies, like OpenIDM from ForgeRock, Apache Syncope from Apache Foundation, among others. Despite diverging in some aspects, these systems share an identical architecture, as depicted in Figure2.6. This architecture is composed of the following modules:

- REST interface - To allow users to communicate over the HTTP, providing API for managing all functions of users and administrators.
- Centralised directory service - To prevent the users from having multiple credentials to different systems.
- Workflow Engine - Also including a User Role-based Provisioning, creates and manages roles assigned to users based on organisational needs and structure such as job function, title, etc.
- Logging, Auditing and Reporting - All internal activities as well as those within connected systems.
- Identity and access management - Also providing administrators with the ability to instantly view and change access rights.
- Connectors- Identity connectors providing some abstraction within the system.
- Identity Repository - This usually consists in a database, to store all the information about the users, roles and policies, in a persistent way.

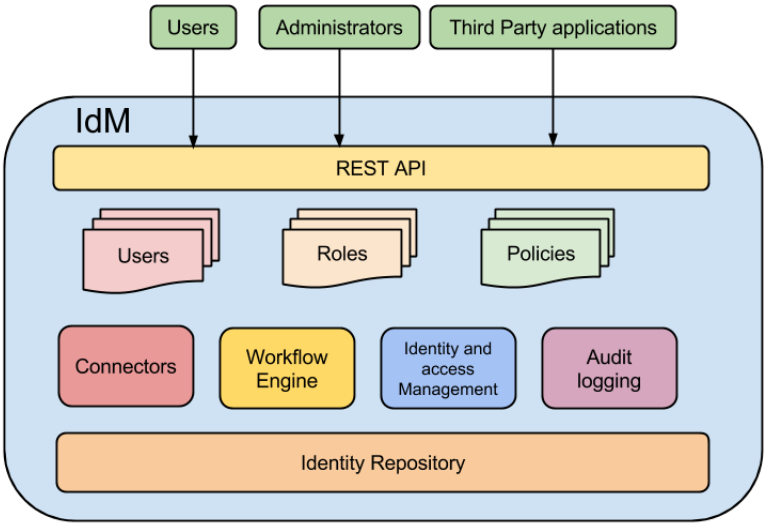


Figure 2.6: IdM system architecture

Given the open source requirement of reTHINK, an analysis is made of the main open source Identity Management Systems, namely: Apache Syncope [6], OpenIDM[7], and midPoint[8].

The Syncope Apache, this system is composed by two main subsystems, the core and the console. The core is a web application that implements the Identity management features. It offers a RESTful interface for caller applications, implements the workflow engine and its propagation layer, and manages data persistence. The console is a web management interface for configuring and administering Syncope core. This system stands out for being a completely free Open-Source system and subscription free, and has a fairly complete graphical user interface (GUI) for the user interface. The weak parts are the lack of customisation, basic authentication and authorisation mechanisms, and poor or sometimes nonexistent documentation.

The OpenIDM system has two important modules, the access layer and the core services. The core services are the vital part of the OpenIDM, where the objects are managed along with the workflow definition and the audit logs data. The access layer provides the REST Java API. The most favourable points in this system are the complete documentation and a great customisation capability, superior when compared to other systems. In contrast, it lacks of a proper GUI, and for the deployment of this system, a paid subscription is required.

The midPoint architecture consists of several subsystems, which can be grouped into User Interface and system Core. The system Core consists of a repository, provisioning and model subsystems, and represents the vital part of the system. The User Interface subsystem implements a web-based end-user interface and the administration interface. The system highlights are the, subscription free Open-Source model, a very good documentation, an RBAC that stand out from the other systems, and possess a quite complete and clean GUI for the user interface. The less favourable points are its basic REST interface and the lack of relevant customisations, since it does not allow for structure changes.

2.3.2 Browser Storage

Client applications running in web browsers that make use of authentication protocols like OpenID connect 1.0 [12] to authenticate a user, have the need to store the received user identity token. So, this implies a storage component on the web browser, which is taken in consideration in this section, so that the best option can be chosen.

It is important to separate the data structures that require data persistency and those who do not. Hash Tables [23], do not provide persistence, being the data structure implemented in the majority of the programming languages. It is also known for its speed on the data access. The more entries this table has, more visible is the impact gains, when compared to other data structures.

Considering data structures in web browser with persistence, there are three main solutions: HTML local storage[9], SQLite[10] and Indexed Database[11]. The HTML local storage, allows for client devices to

[6] https://syncope.apache.org/
[7] http://openidm.forgerock.org/
[8] https://evolveum.com/midpoint/
[9] http://www.html5rocks.com/it/features/storage
[10] https://www.sqlite.org/
[11] http://www.w3.org/TR/IndexedDB/

store small information, enough just to provide applications to work when the user is offline. However, since the information security is not guaranteed, it is not recommended to store critical data using HTML local storage. The SQLite is a relational database[12], but not a client/server SQL database, since the data is stored in the end-user device. This solution presents itself as a good alternative to the file system, and since it can be used as a cache for enterprise data, it helps reduce the access latency. The major downside of this solution is that it does not support concurrency, authentication, allowing the database to be updated/read by anyone, and is not supported by all web browsers. The last one, the Indexed Database, is a non-relational database that stores data in the user device. This solution provides the same benefits as SQLite, and mitigates some of the downsides. Indexed DB have an improved concurrency control, that implements read-only and read-write options, and restricts the access to the database objects, of scripts from domains that had not created the database. Since the indexed DB was adopted as a standard by the W3C, it is supported by the majority of web browsers.

2.4 Summary

This Chapter introduces the state-of-the-art on identities, authentication mechanisms and its management. The concept of identity tokens and several mechanisms to obtain this token are introduced. A description on several authentication mechanisms is also provided, namely the BrowserID, the WebID-TLS, the OpenID connect, the Kerberos and the smart card. The transport layer protocol was also studied, with more focus on the mutual authentication phase. An introduction is made on identity management systems, showing a list of complete open-source products available on the market and a comparison between them. Followed by a study on different storage technologies, highlighting their characteristics.

[12]This model organises data into tables of columns and rows.

3 reTHINK Background

ReTHINK is a European Project with the goal to develop a new communication framework with new solutions to real time communications over the Internet, while mitigating recurrent communication problems from the use of Internet environment.

3.1 Architecture

The reTHINK project introduces a new web service - Hyperlinked Entities (Hyperties) [2], explained below, which enables the creation of trusted relationships among these Hyperties and can be deployed in a wide variety of devices ranging from smartphones, laptops, cloud infrastructures, just to name a few. The goal of reTHINK is to provide a trusted worldwide service communication, where different services providers can be dynamically created to better meet consumers' needs, and thus overcome the geographical implications that affect the telephone companies. ReTHINK project intends to establish an open standardised platform for communication over web technologies, among digital devices which combine authentication and identity management to establish trusted communications among user devices. ReTHINK project aims to allow service providers to develop communication services, that can establish communications between the users devices running Hyperties in a decentralised fashion. This means that users can talk to each other without the need for a service to intermediate the established communication, using for that purpose the Internet structure.

The communication between Hyperties is based on a protocol-on-the-fly (ProtoFly) concept, which allows the use of standard network protocols without the need to modify them or to create new ones. The Hyperty communication can be carried out in two ways, through a message communication, providing an asynchronous communication among the Hyperties instances, or by a stream communication, where media streams are established between Hyperties, granting an audio and video communication. The ProtoFly concept supports the code on-demand facility on Web runtime, granting interoperability among distributed systems, and promoting loosely coupled service architectures. Since this project is designed to operate over the Internet infrastructure, it supports a variety of features, such as Machine to Machine (M2M), Internet of Things (IoT) or social communications. Using an example of the reTHINK application, Alice has a laptop running the reTHINK browser application with a Hyperty instantiated from the CSP A and Bob has a smartphone running the reTHINK native application with a Hyperty instance from the CSP B. Because both Alice and Bob have theirs Hyperty instances running in their devices, both of them are discoverable via Domain registry, a public service that stores the Hyperties instantiated in each user device. Alice wants to contact Bob, and for that she queries the Domain Registry for Bob device. After the query response, the device of Alice is able to start a call with the device of Bob. In the end, we have

an established connection between different CSPs and devices, which is in line with the so well desired interoperability together by this project.

Providing a better insight of the project architecture, its four main components are described, namely: Global Registry, CSP, Identity Provider (IdP) and the user device. Figure 3.1 depicts the interconnection between these four components and their details, also illustrating their internal components.

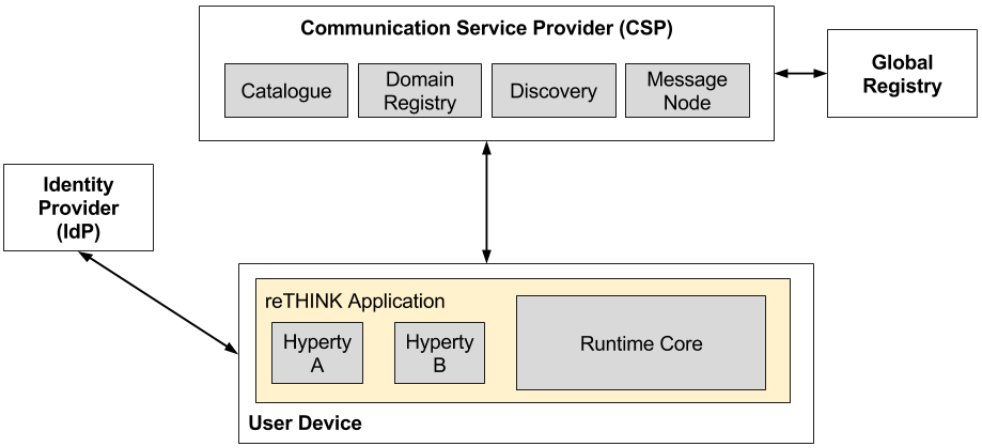


Figure 3.1: reTHINK components

Starting with the user device, this component is responsible for running the reTHINK client side application, which is composed of one or more Hyperties instances from the CSPs and the Runtime Core. The Hyperty is a module of software provided by a CSP which enable the creation of trusted relationships among these Hyperties and can be deployed in a wide variety of devices ranging from smartphones, laptops, cloud infrastructures, just to name a few. The Hyperty can be obtained from the CSP whenever a user pretends to access a new Hyperty and installed immediately on the user device, resulting in a Hyperty instance which will be used for identification. The reTHINK framework running in the user device has no limits regarding the number of Hyperties running at the same time. Runtime Core is the component responsible for ensuring the correct functioning of the reTHINK application, as well as managing the Hyperties running on it. It is responsible for installing Hyperties, ensure the security aspects of the application, and manage the identities obtained from the user and for handling all the messages entering and leaving the reTHINK application. The Runtime Core has a big emphasis in this Thesis as detailed in the architecture chapter.

The long and complex strings derived from Hyperty instances URLs can be difficult for humans to memorise and use as a contact number to provide to others. So, Hyperties registered in the user device need to have a user identity associated to it. With this association becoming publicly registered, it is possible to discover the user Hyperty instance by searching for a known user identity. For example, if Alice install a Hyperty and uses her identity obtained from an IdP, for example the Google email address ”alice@gmail.com” to associate with the Hyperty, Bob, when searching for this email, receives the Hyperty instance of Alice, which allows him to initiate a conversation with Alice. Because Bob also needs to register his Hyperty with an identity, when Alice receives the call, she can identify who is calling her. Note that, the user identity can be obtained from a service provider other than the CSP and because of that, IdPs are a separate service from CSPs.

A CSP is comprised by several internal components, namely the: Domain Registry, Discovery, Message Bus and Catalogue. The Domain Registry has the responsibility to register publicly the Hyperties upon their installation in the user device, as well as the identity associated to it. The Message Bus is the component responsible for intercepting and forwarding messages to be exchanged between users. These messages are usually signalling messages used to initiate a call. The Discovery is the public interface that enables the search for users registered in their own Domain Registry, as well as users registered in others CSPs Domain Registries. The catalogue is a repository provided by the CSP that hosts the web oriented software to be used by user devices, for example, the Hyperty code to be installed in the user device.

For the user device to be able to communicate with the CSP, the installation of a Protocol Stub (ProtoStub) in the user device is required. This ProtoStub implements the protocol used to communicate with the CSP. It can be dynamically installed onto the user device, being the CSP responsible to providing this ProtoStub through his catalogue. For instance, when a user's pretends to start a communication with a new CSP, the Runtime Core triggers the installation of the ProtoStub including downloading it from the Catalogue, to which he pretends to communicate.

To each user in reTHINK a Global User Identifier (GUID) is assigned, which is a unique, unchangeable and unrelated to CSP domains identifier. Whenever a user uses one of his identities to associate it with a Hyperty, this identity becomes associated to his GUID independently from the IdP used to obtain the identity. With this, the GUID can have multiples identities linked to it, ensuring that all those identities belong to the same user.

As explained previously, the CSP Discovery service functionality includes the search for identities registered in others CSPs. To achieve this feature another service is required, namely the Global Registry [2]. This public service deployed in a distributed hash table (DHT), stores the GUIDs and the associated identities used to register Hyperties in the CSPs. The Global Registry has the role of resolving a given user identifier to the CSP domain that is registered. For example, Alice is registered in a CSP A, and Bob in a CSP B. Since Alice and Bob are registered in different CSPs, when Alice tries to search for Bob, the Alice CSP discovery queries the Global Registry, which in return resolve the identifier domain and uses it to query the Bob CSP. And finally, the Domain Registry from Bob CSP, replies to Alice with the Hyperty instance URL running in his device.

To give an example of a complete operation of the reTHINK framework, we will demonstrate a use case where Alice initialises the reTHINK application from the start and then try to contact Bob, also running the reTHINK application. Figure 3.2 demonstrates the flow with the messages that need to be exchanged to achieve this objective.

When Alice runs the reTHINK application in her device, she is given the choice to install a Hyperty from several CSPs Hyperties list. Upon the choice of the desired Hyperty, on step 1 of Figure 3.2, she is prompted to obtain an identity from a preferred IdP. The acquisition of this identity requires a user authentication towards the IdP. In step 2 and 3, after the Runtime Core receives the identity Assertion (idAssertion), it starts downloading and installing the Hyperty from the CSP A (Catalogue chosen by Alice). Once the installation is done a Hyperty instance URL is generated, and in the step 4 the Runtime Core associates the initially obtained identity with the Hyperty instance URL and registers it in the Domain Registry of the Alice CSP A.

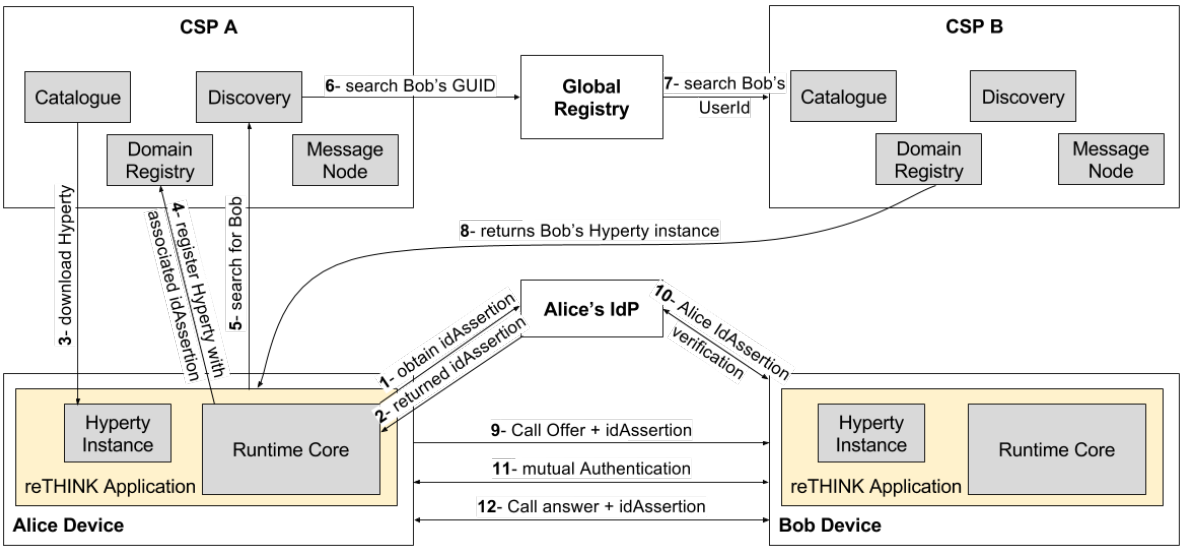


Figure 3.2: reTHINK communication between two users

In steps 5 to 8, Alice tries to search for Bob by inserting the email address of Bob she knows, for example, in a search bar in the reTHINK application. The Runtime core then uses the Discovery service from the CSP A, but since Bob is not registered in that CSP, the Discovery service queries the Global Registry using the Bob's GUID to resolve the current domain to which the identity is registered. After being contacted, the domain registry from Bob's CSP, replies to Alice's device with Bob's Hyperty instance URL, for which he can be contacted.

Steps 9 to 12, occurs after Alice receiving the address of Bob, represent the call initiation between Alice and Bob. The call initiated by Alice contains her idAssertion. The Runtime Core on Bob's device consequently requests to the IdP of Alice to verify the legitimacy of Alice identity. After receiving the response from the IdP chosen by Alice, Bob's Runtime Core initiates a mutual Authentication between Alice and Bob, to authenticate themselves mutually and to exchange the keys used to establish a secure communication. Finally, after the successful completion of the mutual authentication, Bob's device replies to the call offer with his idAssertion already verified by Alice and the call starts.

Most of these steps are transparent to the user, being the user only responsible to login into an IdP to obtain an identity, choose the Hyperty to install and search for another user with a known identity, in order to start a call.

3.2 Communication types

The communication between devices can be performed in several ways, particularly since reTHINK intends to support various types of communications, presenting no restrictions to the CSPs when developing their Hyperty services. The communications may be as diverse as text, video or audio chat for example, and can be transmitted in different manners, such as, direct communication or broadcast transmission. If two Hyperties from different CSPs support the same communications capabilities, then they are capable of communication with each other.

Even though CSPs may develop new protocols for the communication, the reTHINK offers native support for two types of communication, a direct Hyperty to Hyperty communication and a broadcast protocol. All messages sent using the reTHINK framework go through the public Message Node, a public message distributor service. Both these communications use messages with a JSON format and follow the following structure:

```
{
  'id':   '1',
  'type': 'create',
  'from': 'hyperty://domain.com/bob-hyperty-instance-n-1',
  'to':   'hyperty://domain.com/alice-hyperty-instance-n-2',
  'body': {
          'value': 'Hello Alice! :)',
          'identity': 'Bob identity assertion'
          }
}
```

The Hyperty to Hyperty communication consists of a direct communication between two user devices. The fields 'from' and 'to' will have the respective Hyperties URL and do not require any special action from the Message Node besides the routing of the message. This type of communication is not only good for direct communication but also useful for the communication of signalling. For example, using the WebRTC [1] technology to provide video and audio streaming is a viable solution in the reTHINK. However a P2P architecture a signalling message is required, so that a connection between the two devices can be negotiated and established. As such, the Hyperty to Hyperty message can be useful to provide this signalling option.

The group chat communication example is implemented using a new reporter-observer pattern [24], based on the Observer pattern[1]. The group chat communication is achieved using Data Objects, which is an object registered in a public Message Node that broadcasts to all subscribers from that object whenever it receives a message. To start a group chat communication, a Hyperty is required to create a Data Object, which will be the Hyperty reporter. After that, other Hyperties can subscribe that Data Object, by searching for the Data Object in a discovery public service, i.e. the Domain Registry, or by being invited by the Hyperty Reporter. After the Hyperties subscription to the Data Object, when a message is sent to the group chat, that message goes to the Data Object registered in the Message Node and is forwarded to all the subscribed Hyperties.

3.3 Summary

In this Chapter the reTHINK project is introduced, a new communication framework, and new concepts that come with it. It starts by introducing the project goals and how it is intended to operate. A description of the reTHINK internal components is provided, namely the Hyperty service, the role of the

[1] A design pattern which defines a dependency from one-to-many, where an object, named the subject, possess a list of dependents, named observers. When the object is modified, all the dependents are notified immediately about the state changes.

Communication Service Provider (CSP) and its components, the Protocol Stubs defining the communications' protocol and the role of the Identity Service Providers. Following with the description of its architecture and how these components works and the interoperation between them. An example of a communication between two users is provided with the interaction of all components. To conclude it provides a description on the communications that the reTHINK aims to support.

4 Proposed System

This chapter details the design of the developed modules for identity management in the reTHINK framework: the Identity Module and the Runtime Registry. Before diving into the description of their architecture, a brief overview of their role is presented.

In the reTHINK application the communication between users is achieved by using Hyperties. To do so, the user must instantiate one from a public service that exposes a list of the available Hyperties. To enable the discovery of Hyperties in a public discovery service, the instantiated Hyperty must have an associated identity. This way, when a user intends to reach another user, he can search for his identity in the discovery service, and that service is able to return the Hyperty associated with the searched identity. Since reTHINK is a P2P framework, there is no centralised authentication authority. Consequently, the user's identity must be authenticated by the user device. To do so, we created a mutual authentication protocol that uses identity assertions with the same capabilities as digital certificates to perform the authentication of the endpoints. When a user initiates a communication with another user the mutual authentication protocol is triggered. The two endpoints negotiate the cryptographic keys necessary to guarantee a secure communication channel that offers confidentiality and integrity guarantees.

The following details the proposed architecture, namely the system's structure and the protocols created to fulfil the requirements of the solution.

4.1 Architecture

The architecture of the system herein proposed consists of two major components in the reTHINK framework, namely the Identity Module with a Proxy and the Runtime Registry. These are two of the core components of the Runtime Core. Along with the remaining components developed by others partners in the reTHINK project, as depicted in Figure 4.1. Besides the Runtime Registry and Identity Module, the Runtime Core is also composed of the Message Bus, Policy Engine and other smaller components[2]. The Message Bus and the Policy Engine components are worth mentioning and describing because they play an active role in the interaction with the components developed in this Thesis.

As defined, all communications in the reTHINK framework are performed using messages with a JSON structure and the Message Bus is the component responsible to manage the reception and the dispatch of messages through the components, including internal and external messages. For example, all the messages sent and received in a communication between Hyperties instantiated in the user's devices passe through the Message Bus. The Message Bus works with listeners for the specific URLs, and when a message is received, it is dispatched to the listener with the destination URL.

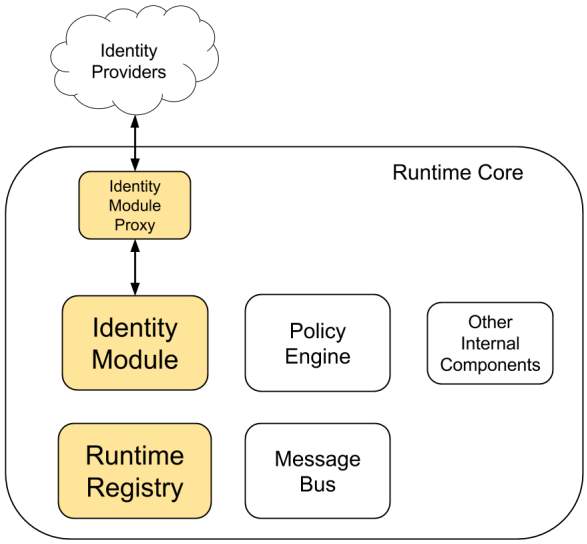


Figure 4.1: Runtime Core architecture

The Policy Engine has the role of enforcing user policies in the reTHINK framework. For example, if a user wants to block the communications with another user, he can do it by adding a rule to block that identity in the Policy Engine configuration page. For the Policy Engine to be effective, it needs to listen to all messages that go through the Message Bus. As such, both Policy Engine and Message Bus work together to intercept all messages and analyse each one, to check whether a given policy needs to to be applied.

With a deeper insight on the reTHINK framework flow, the Runtime Core communicates with external components, such as a Communication Service Provider, using messages that are sent by the Message Bus. To ensure the security and isolation of the internal components of the Runtime Core, the Message Bus along with the Identity Module Proxy are the only components that can communicate out of the Runtime Core, although the Proxy has a completely different purpose, as detailed below. The Policy Engine besides allowing the definition and enforcement of the user's policies it also allows filtering all the messages that passes through the Message Bus, which in the worst case can result in the blocking of the message. Being so, the Policy Engine plays an important role in the Identity Module by rerouting messages that require some kind of action by it, i.e. message encryption or decryption.

The identity management comprises not only the action of managing the user's identity but also the acquisition of these identities. To do so, the standard OpenID Connect (OIDC) Protocol [12] is used to allow for users to get an identity from the identity provider, which will be managed by the Identity Module. This identity token, provided by the IdP, is managed by the Identity Module, and used to associate an identity with the Hyperties instantiated in the user's device.

OIDC is a popular authentication protocol used in the authentication of a large number of users in several services, for instance, Google and Microsoft. This protocol has some features of great relevance for the reTHINK project. On the one hand, it offers a field that can be used to send the user's public key, which is necessary for the generation of an identity token with the capabilities of a digital certificate, and consequently to have the IdP as a certification authority. On the other hand, while Kerberos does not provide any tool that can be used for reTHINK, OIDC implementation provides an easily adaptable

implementation. Together, the widespread use and the flexible implementation make OIDC a good fit for user authentication in the reTHINK framework.

So that a Hyperty instance running in the user's device can be associated with an identity, a module is required to store the registration of Hyperties, being that module the Runtime Registry. The Runtime Registry component has the responsibility to register and manage several runtime components, including Hyperties along with the respective sandbox provided by a Service Provider [1]. The Runtime Registry is responsible for assigning an identity to a Hyperty instance, and publish that association to a Domain Registry, making it discoverable for other users. The identity used in the association is requested to the Identity Module, making these two components, namely the Runtime Registry and the Identity Module, essential to each other.

4.1.1 Mutual Authentication Protocol

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.

The 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. Using IdP's that supports OpenID Connect, it is 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 verify 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.

[1] Service provider sandbox allows the isolation of the Hyperties code obtained from a service provider from being accessed by others Hyperties running in the device.

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).

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.

Figure 4.2 illustrates all the messages exchanged to provide mutual authentication between the users Alice and Bob.

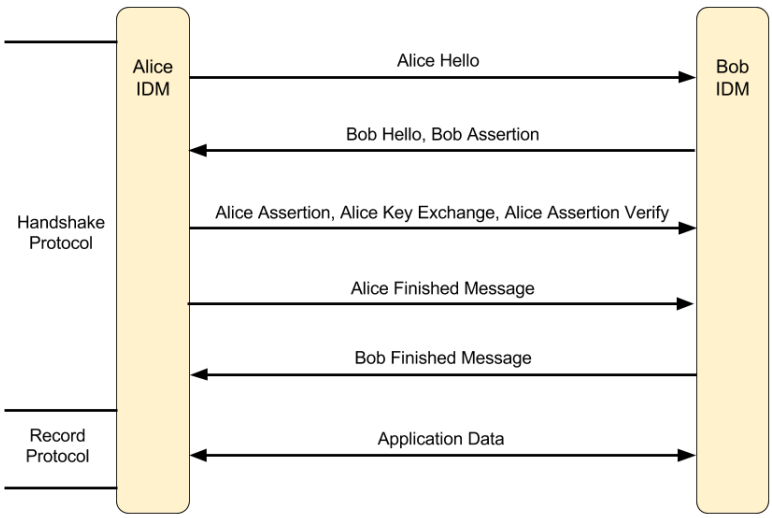


Figure 4.2: reTHINK mutual authentication flow

The following provides an explanation of the overview on the developed mutual authentication flow. A more detailed explanation can be found in the chapter 5.1.5:

1. **Alice Hello**: This message contains a number generated randomly by Alice.
2. **Bob Hello, Bob Assertion**: This message contains a random number generated randomly by Bob and his identity assertion with his public key.

3. **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.

4. **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.

5. **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.

6. **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.

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, Figure 4.3 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.

Taking Figure 4.3 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

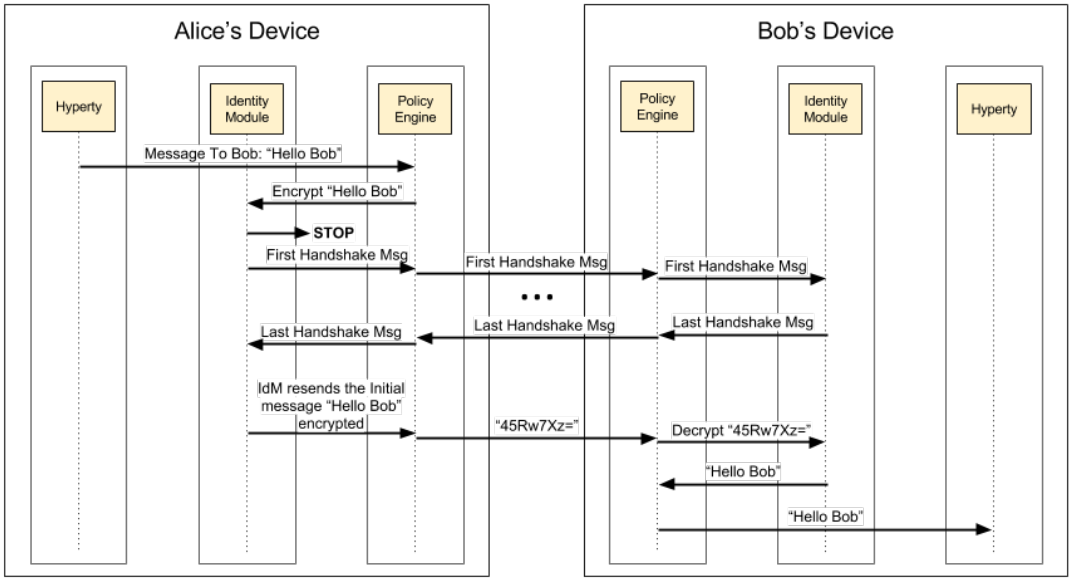


Figure 4.3: Example of Alice sending a message to Bob, for the first time.

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 MAC 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 MAC. The IdM then uses Alice's symmetric keys to decrypt and verify the MAC 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.

4.1.2 Identity Module

The Identity Module is the component responsible for handling all the user identities as well as ensuring confidentiality and integrity on the messages exchanged either in a direct communication between Hyperties or in chat groups where multiple Hyperties are connected.

The identity in the reTHINK framework is not fixed to a unique Identity Service Provider (IdP). It can be obtained from several IdP's, being the IdModule responsible for providing the user with the option to choose which IdP he prefers. To do so, a graphic user interface (GUI) component that communicates with the Identity Module is required in order to provide the best experience to the user. With this GUI component, the user is presented with the possibility to add new identities, select the preferred identity and remove identities previously registered, all this, in a simple and accessible interface.

For an identity token to be obtained by Identity Module from an IdP, a module needs to be acquired from the IdP, the IdP proxy. This proxy is the component responsible for the communication with the external IdP server. So, when an Identity Module asks the IdP for a user assertion, this request is made to the IdP proxy via the internal Message Bus, as presented in Figure 4.4. Then, the proxy communicates directly with the IdP requesting the user assertion. The decision to support proxies came up with the need to support multiple IdP's without the need to modify the identity module itself, being this IdP proxy downloaded and compiled on the fly. With a proxy, the Identity Module can be agnostic to the available IdP's. In order for the Identity Module supports the load of the proxy dynamically, the proxy should be standardised, and for that, the WebRTC IdP proxy standard [25] was chosen. With this API well defined, any Identity Provider following the WebRTC standard for proxies can provide their proxy, compatible with reTHINK.

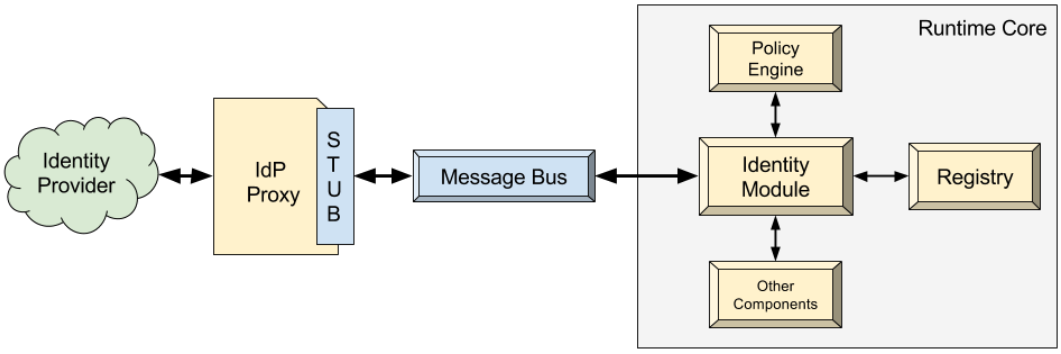


Figure 4.4: IdP proxy communication

The communication between users is one of the major characteristics in reTHINK. As such, the authentication of each user takes a big role to ensure that no personification attack 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 native support for two types of communication, a direct communication between two users through their respective Hyperties and a group chat communication where the messages are exchanged between all participants in the group. 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 4.5 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 passes 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

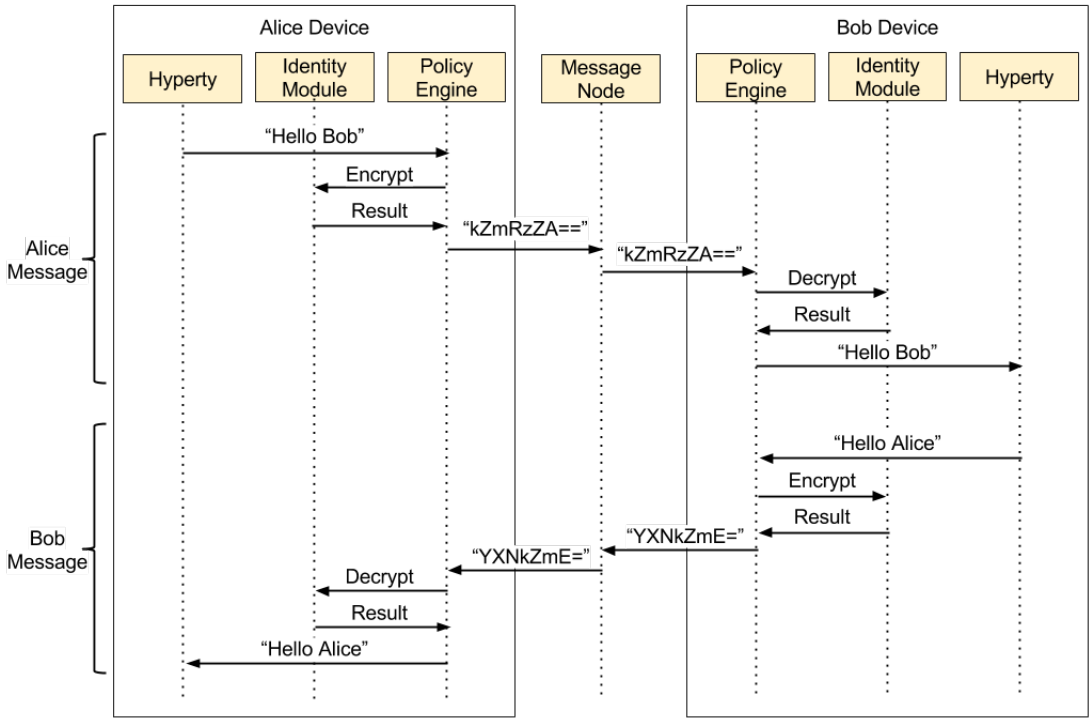


Figure 4.5: Hyperty to Hyperty communication

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.

In a group chat communication, the messages exchanged follow a different architecture, in this case based on the reporter-observer pattern, a broadcast communication approach [26]. Observing Figure 4.6, that illustrates a communication in a group chat, 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.

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 Figure 4.7, 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

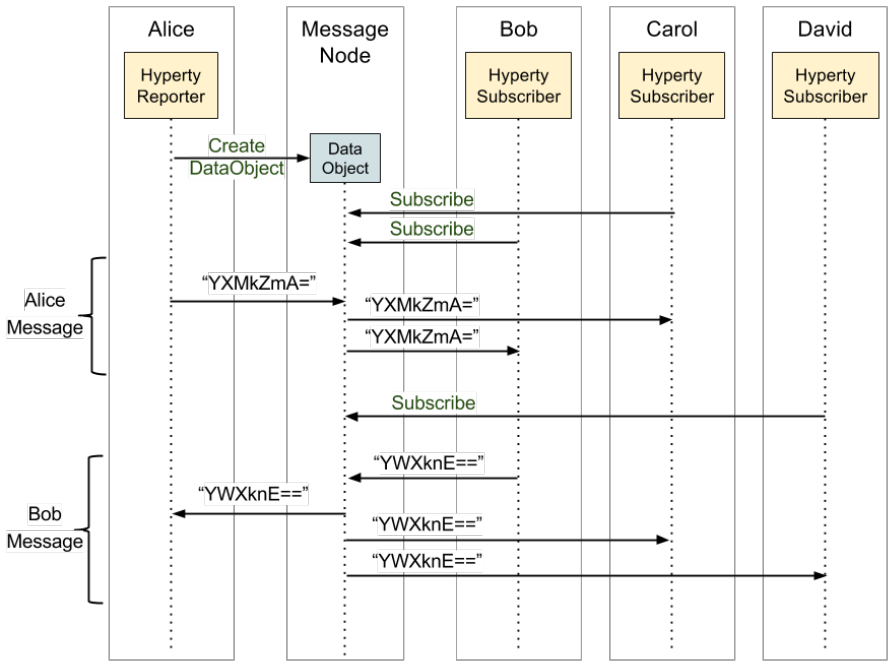


Figure 4.6: Group chat communication

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.

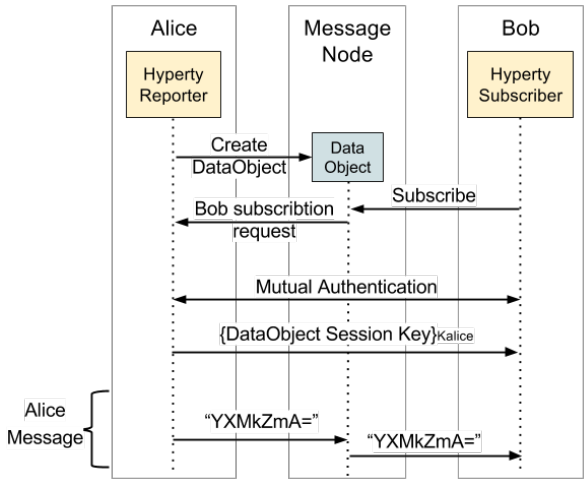


Figure 4.7: Hyperty subscription flow

The Identity Module is responsible for obtaining, managing the identities in reTHINK, and providing 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 to register 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 clicks 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.

4.1.3 Runtime Registry

The Runtime Registry has the responsibility to register and unregister several internal components. Among these components, we can find Hyperties supplied by Service Providers, Protocol stubs (Protostubs), which implements the protocol to be used to communicate with the Service Providers and Data Objects, a component that allows establishing group communication among Hyperties. Besides the storage of those components, the Runtime Registry allows for internal components running in Runtime Core to request information about the registered components, such as, return all the Hyperties registered in the device.

The registration of Hyperties requires a bit more than just storing the component internally. As depicted in Figure 4.8, when a request is made to register an Hyperty, the Runtime Registry receives information regarding the Hyperty and the respective sandbox with the executable code. Then a request for a user identity is made to the Identity Module, which promptly asks the user, to select or add a new identity using the GUI. When the identity is received, the Runtime Registry requests the public Domain Registry to generate a unique Hyperty URL and return it to the Runtime Registry. After receiving the unique URL, it is then associated with the Hyperty sandbox received in the Runtime Registry, allowing this Hyperty to be identifiable and addressable. This URL is important to allow the Hyperty sandbox to have a listener for that address. So, incoming messages to that specific URL are rerouted to the Hyperty sandbox. After the Hyperty has been associated to the Hyperty URL and to an identity, the Runtime Registry sends a request to register an association of the Hyperty URL and the identity within the Domain Registry, to allow this association to be public. With this information, users can query the Domain Registry for the Hyperties registered with a given identity, receiving as result the Hyperty URL, which can be used to start a communication. After successfully registering the information in the Domain Registry, it returns a confirmation message, which is propagated until reaches the Runtime Registry. In the end, the Hyperty URL is returned, to whom made the request to register the Hyperty.

For the Data Objects, the registration follows an identical flow to the one used for the Hyperties, with some minor differences. When a request to register a Data Object is made, the Hyperty reporter URL is passed as input, and information with characteristics regarding the Data Object. The difference regarding the Hyperties registrations is that instead of associating the Data Object with an identity, it must be associated with the reporter URL received as input. The Runtime Registry then asks for a unique Data Object URL, to allow it to be identifiable, and subsequently requests the Domain Registry to store all this information. After the confirmation by the Domains registry, it returns the Data Object URL that was generated and associated to the reporter URL.

The registration of Protostubs has a different purpose on the Runtime Registry than for Hyperties and Data Objects. When a request to register a Protostub is made, a domain name is passed as well

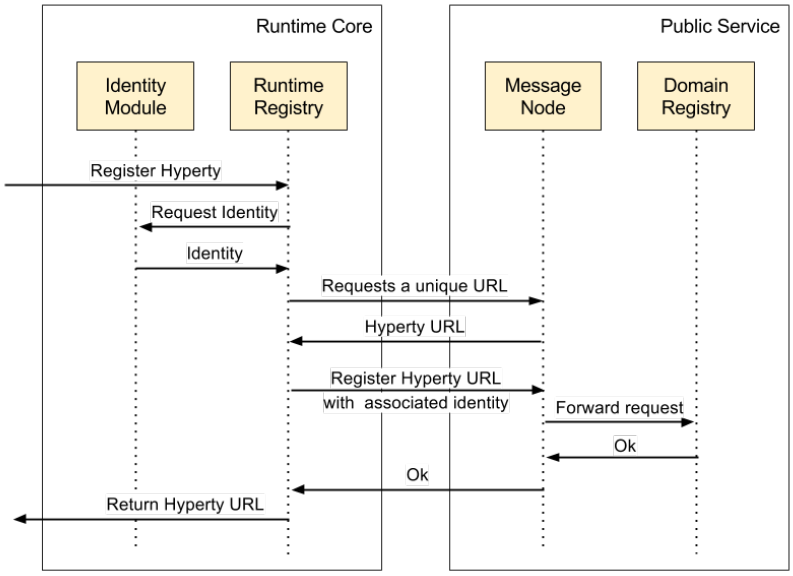


Figure 4.8: Hyperty registration

as a sandbox with the Protostub code. The Runtime Registry generates a URL and associates it to the received domain. At the end, the generated URL is returned. With this association, the Runtime Registry has the function to resolve the Protostub URL associated to a given domain. For example, considering that a request to register a Protostub was made previously for the domain 'mydomain.com' and the returned URL was 'mydomain.com/uniqueID-123', when a request is made to resolve the domain 'mydomain.com' it should return the 'mydomain.com/uniqueID-123'. This is useful because Hyperties that wish to communicate with a domain are not required to know the specific URL and can just send messages to the domain.

The Runtime Registry in addition to registering the Hyperties and Data Objects also has the task to register them in the Domain Registry to make them public. The query for Hyperties or Data Objects registered in Domain Registry can be made by anyone, but it requires the knowledge on the message structure used to query the Domain Registry. In order to facilitate this process, the Discovery library is created with a well-defined API, having the methods for searching in the Domain Registry. For example, if a Hyperty wants to find a Data Object from a user in the Domain Registry, it just needs to instantiate the Discovery library, and call a single method that sends a query for Data Objects from a given user.

Another library was also created so that Hyperties can query the Runtime Registry for the identity they got associated with. Since the association of a Hyperty and identity is made by the Runtime Registry, the Hyperties do not have this information. So, Hyperties can instantiate the IdentityManager library that allows them to query for the identity that was associated with.

4.2 Summary

This Chapter introduces the work done for this Thesis. A description on the components Runtime Registry and Identity Module is provided, as well as for the mutual authentication protocol implemented for reTHINK. The features supported by the components developed are presented including examples demonstrating how these features work. For the Identity Module a description is provided on how identities can be obtained, when the mutual authentication is triggered and process to ensure the creation of a secure communication channel. For the Runtime Registry it is explained what components it stores and the process of the association of an identity to a Hyperty. The motivation behind certain decisions is described, including decisions made to support some characteristics and scenarios of the reTHINK project.

5 Implementation

In this chapter, the implementation of several components in this project is described. The work developed is focused mostly in two components: the Runtime Registry and the Identity Module. There are other smaller aspects that are covered in this implementation chapter as well, such as the Discovery libraries.

The implementation of all the modules was done in JavaScript, a high-level programming language, using the ECMAScript 6 which is the latest standardised language specification. The code was developed to run in a browser. The Chrome web browser was chosen as the main platform for the development and test of the reTHINK framework.

Starting with the Identity Module, this section describes the internal organisation of the component and how it is structured. Others aspects concerning the Identity Module are also explained, starting with the identity acquisition, continuing with the identity management and establishment of trustful communications, up to execution of the secure communication. This chapter also details the used technology and some implementation decisions regarding the technology used. To conclude, an explanation on the Runtime Registry implementation is provided, and how it was implemented along with the design choices.

5.1 Identity Module

In order to obtain an identity, the Identity Module component is created within the Runtime Core from the reTHINK framework. With these identities, the Identity Module is able to establish trust among users using the reTHINK application and to create secure communications assuring confidentiality and integrity. To obtain identity tokens, the OpenID connect authentication protocol is used.

The reTHINK project aims to bring interoperability among different service providers and, for it to be successful, some components are required to be modular in order to support different implementations of the same component, from different service providers. To do so, it is essential that the IdP Proxy complies with the requirements, explained in the previous chapter. So, the Identity Module must provide the support for all IdP Proxies develop using the WebRTC IdP proxy [25] implementation.

Trustful communications between users can be assured by trigger the mutual authentication protocol. This mutual authentication protocol follows the message flow similar to the TLS protocol [19], in order to provide the same security properties. During this protocol, symmetric cryptographic keys are generated. These keys are useful for the secure communication that occurs after authentication of users. Because several cryptographic methods are used by the Identity Module during the establishment of secure communications, a cryptographic library was created exposing an API, to ease the tasks.

All these features are provided in a well-defined API by the Identity Module, so it can be used by other components belonging in the Runtime Core. Since latency in the network is a constant presence, the main application should not stall, waiting for the response that may never come. Given this, most of the Identity Module methods use Promises, which are a useful tool for applications that make requests over the Internet. Promises work like callbacks and are intended to be used for asynchronous computations in order to allow the code to resume its execution, even if the method is still waiting for the response.

5.1.1 Identity acquisition

In order for the IdM to be able to manage identities, these need to be acquired in the first place. This is achieved by supporting and using the OpenID connect (OIDC) authentication protocol, as described in the previous chapter.

The OIDC protocol has the advantage of using REST calls to make requests, providing in return identity assertions in a JSON format, which is an asset for the reTHINK framework. This allows the IdM running in JavaScript to make HTTP requests and easily handle the received JSON token. Additionally, because OIDC offers a standardised API, all Identity Service Providers implementing the OIDC, will follow the same standard REST API for the user's requests

Providing an example of an OIDC REST request, we have the following URL:

```
https://login.windows.net/common/oauth2/authorize?response_type=
    id_token&client_id=7e2f3589-4b38-4b1c-a321-c9251de00ef2&scope=
    openid&nonce=7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7&response_mode=
    fragment&redirect_uri=https://localhost
```

Figure 5.1: Example of the URL used to access the Microsoft OpenID connect

Analysing this URL, it is possible to see that the construction of the URL is composed of several fields. It starts with the IdP's endpoint URL, followed by the response_type indicating whether an id token and access token is requested, the client_id which is the application client identifier for the IdP authorisation server, the field nonce, that can contain any value to be asserted, the response_mode which informs the IdP's authorisation server of the mechanism to be used for the returning parameters and the redirect_uri the URI to witch the response will be sent.

Identity Service Providers that intends to provide authentication solutions with the OIDC protocol must follow the OIDC specification. The specification states that the fields in the URL request must be the same, independently of the IdP, where only the IdP Authentication EndPoint can vary. With this standard, all the requests will be similar, which is an advantage by allowing the request URLs from different IdPs to have the same fields and format.

For example the 'nonce' field in the URL can be used to assert any string sent in that field. Since the 'nonce' field is standard for all IdPs using OIDC for authentication, the Identity Module mutual authentication solution sends an authentication request with the public key, to be asserted by the IdP, in the

‘nonce’ field. With this approach, the reTHINK framework is able to work with any IdPs using the OIDC authentication protocol.

The URL for the IdP’s authentication requests supports additional parameters, but since the majority of them are not required by the IdM, they are not taken in consideration in the construction of the URL. Given this, the construction of the URL includes the fields depicted in Figure 5.2, as an example of a request to the Microsoft OIDC EndPoint:

```
let microsoftInfo = {
  clientID:            '7e2f3589-4b38-4b1c-a321-c9251de00ef2',
  redirectURI:         location.origin,
  tokenEndpoint:       'https://login.windows.net/common/oauth2/
     authorize?',
  type:                'id_token',
  scope:               'openid',
  mode:                'fragment'
};

let m = microsoftInfo;

let requestUrl = m.tokenEndpoint +
  'response_type=' + m.type +
  '&client_id=' + m.clientID +
  '&scope=' + m.scope +
  '&nonce=' +  contents +
  '&response_mode=' + m.mode +
  '&redirect_uri=' +  m.redirectURI;
```

Figure 5.2: Construction of the OpenID connect URL

The URL depicted in Figure 5.1 is the result of URL construction depicted in Figure 5.2. The contents provided to the nonce field can be any string value, and the location.origin provides the domain of the page that opens the request URL, i.e. ‘https://localhost’

After obtaining the constructed URL the IdM needs to open the received URL in the web browser, in order to allow the user to authenticate himself towards the IdP. This authentication requires the users to insert their username and password to validate the ownership of the account.

There are two different solutions to open the new URL with the OIDC request, in order to obtain the identity token. The first one is to refresh the main web page with the new URL. Because the OIDC request URL contains the URL to be redirected, after the user successfully authenticates to the IdP web page, the IdP refreshes the authentication page with the redirected URL which includes the user ID token. The second solution consists in the opening of a new child window with the request URL. After the user authentication towards the IdP, this child window receives the id token, in the redirected URL which can be accessed by the parent window.

The first solution brings some problems is regard to the Runtime Core running in the web page. This is due to the fact that each time the reTHINK web page is loaded, the Runtime Core running in it is downloaded and instantiated. Due to the large and complex reTHINK framework, the web page loading time is not the major concern. The main problem is in the loss of the non permanent data and Hyperties

instantiated. With the first solution, after each request to obtain an identity the reload of the reTHINK application page would occur, since the OIDC protocol reloads the current web page to the redirect URL provided, after each user authentication. So with those problems listed, this option was discarded.

Although with some issues, the second solution was adopted as the solution to load the OIDC URL. With this solution the reTHINK framework only needs to be downloaded and instantiated for the first time the web page with the reTHINK application is opened. Further identity acquisitions do not trigger the reload of the web page. The main obstacle for this solution, is the management of the new child window. For JavaScript, libraries already exist to handle OIDC authentication requests for multiple IdPs, providing the solution with the automatic management for opening new child windows. However, these libraries lack of advanced customisations, doing only the standard requests. The most important feature that is missing is the option to add a value in the nonce field, which is crucial to the mutual authentication protocol herein considered.

With the lack of library support to include the 'nonce' field, a custom implementation was developed. This implementation receives the OIDC URL and opens it in a new child window, pop-up like. For example, using the Microsoft OIDC authentication endpoint the new window opens a page as illustrated in the Figure 5.3. Following this, the IdM solution adds a listener in the new child window for any changes that may occur after the user insert his authentication credentials. Since the IdP redirects the web page, to the URL provided in the request, after a successful authentication by the user, the child window, that displayed the authentication page, sends a signal indicating it has been changed, allowing the IdM to listen to that change. After listening to the changes, the IdM extracts the identity token and closes the child window.

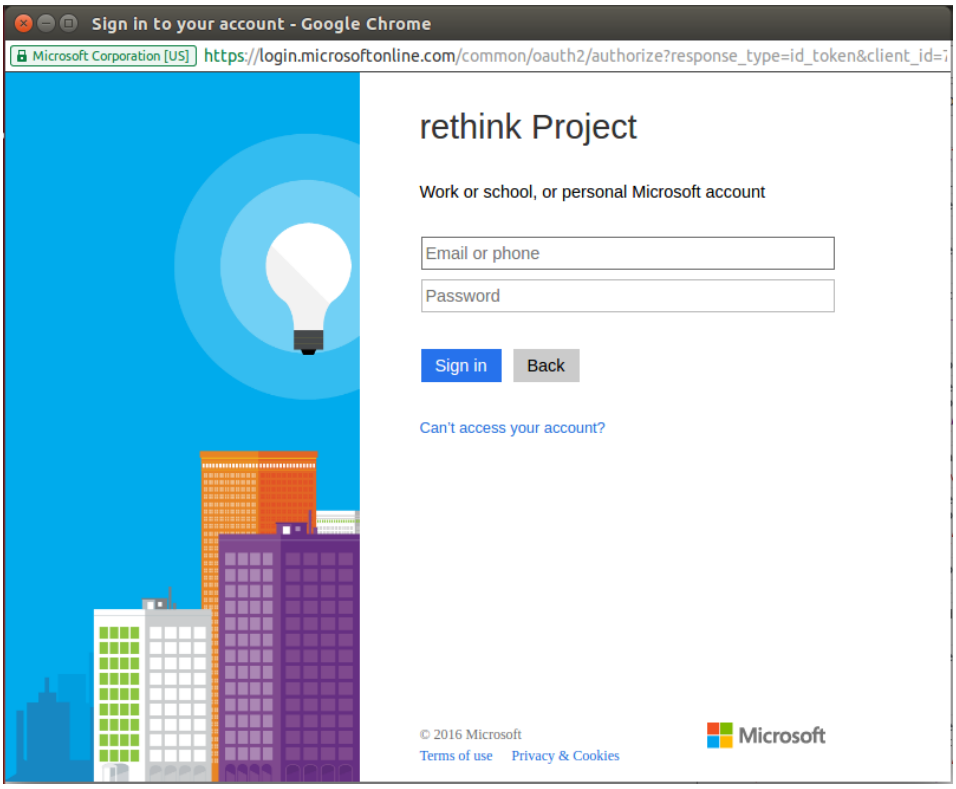


Figure 5.3: Microsoft authentication page

The identity token received in the IdM after the OIDC request, contains several mandatory fields, such as: the issue number and the subject identifiers, as illustrated in Figure 5.4 for the account 'openidtest10@outlook.com'. But there are many other optional fields that can be added, during the REST request. For the mutual authentication the field 'nonce' is mandatory, so that the protocol can work. As stated before, this field is intended to carry a public key.


```
{
 "aud":"7e2f3589-4b38-4b1c-a321-c9251de00ef2",
 "iss":"https://sts.windows.net/3fa4042c-7c4d-4382-aba8-fc8ec61103a4
    /",
 "iat":1462125871,"nbf":1462125871,"exp":1462129771,"amr":["pwd"],
 "email":"openidtest10@outlook.com",
 "family_name":"Test",
 "given_name":"OpenID",
 "idp":"live.com",
 "ipaddr":"188.37.184.74",
 "name":"Test OpenID",
 "nonce":"7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7",
 "oid":"53dd6b54-8e88-4485-9773-a0e29856b6c9",
 "sub":"Ihg61tEOcHjXr_tMCw5z8TU3psPJ76vd4SAx3HaqNAM",
 "tid":"3fa4042c-7c4d-4382-aba8-fc8ec61103a4",
 "unique_name":"live.com#openidtest10@outlook.com",
 "ver":"1.0"
}
```


Figure 5.4: Example of an identity token received from Microsoft OIDC

Whenever the IdM receives a request to obtain a new identity, it uses a cryptographic library, which is detailed below, to generate a public/private key pair. Because the generated keys are in the format of a byte Array, and the public key is intended to be added in the OIDC request, it must be transformed by a serialization process[1]. To do so, the IdM converts the byte array into a Base64 encode, and add the resulting string into the 'nonce' field.

Providing the example of a URL for a OIDC request depicted in Figure 5.1 with the string '7362CAEA-9CA5-4B43-9BA3-34D7C303EBA7' in the nonce field, the received identity token from that URL request is depicted in Figure 5.4 and comes with that string in the nonce field, asserted by Microsoft. This solution is valid for every IdP supporting authentication with OpenID Connect.

For the OpenID connect protocol, the identity tokens validation can be performed in two ways. The first is by contacting the IdP endpoint and providing the received token to validate the token. The other method is by validating the token locally. The identity token received is encoded in base64 and contains three parts. The first part contains information regarding the protocols used, the second part contains the identity token with the information and the last part contains the hash of the identity token.

The identity token validation consists in additional smaller steps in order to verify if the values received in some fields matches to the ones in the request URL, for example if the client id is the same as in the 'aud' field. The most critical step is to verify the signature received. To do that, the cryptographic protocols received in the first part of the identity token are used to validate the signature received.

[1]Serialization is the process of converting data structures or objects in a format that can be stored or sent via network.

5.1.2 Identity GUI

To allow users to easily select the identity they want to associate with a Hyperty, the identity GUI component was developed. This identity GUI allows the user to: add new identities, select from existing ones and remove previously added identities. This GUI was developed using the materialize framework[2], a CSS framework that combines both: CSS, HTML and JavaScript language into a easy to use framework.

The final page for the identity GUI is illustrated in Figure 5.5. The page demonstrates a case where three identities are already registered by two different IdP, two from Google and other from Microsoft. These identities can be removed using the remove button on the right of each identity. The bottom left button 'add an identity' allow adding new identities by showing a drop-down list with the IdPs available to authenticate the users.

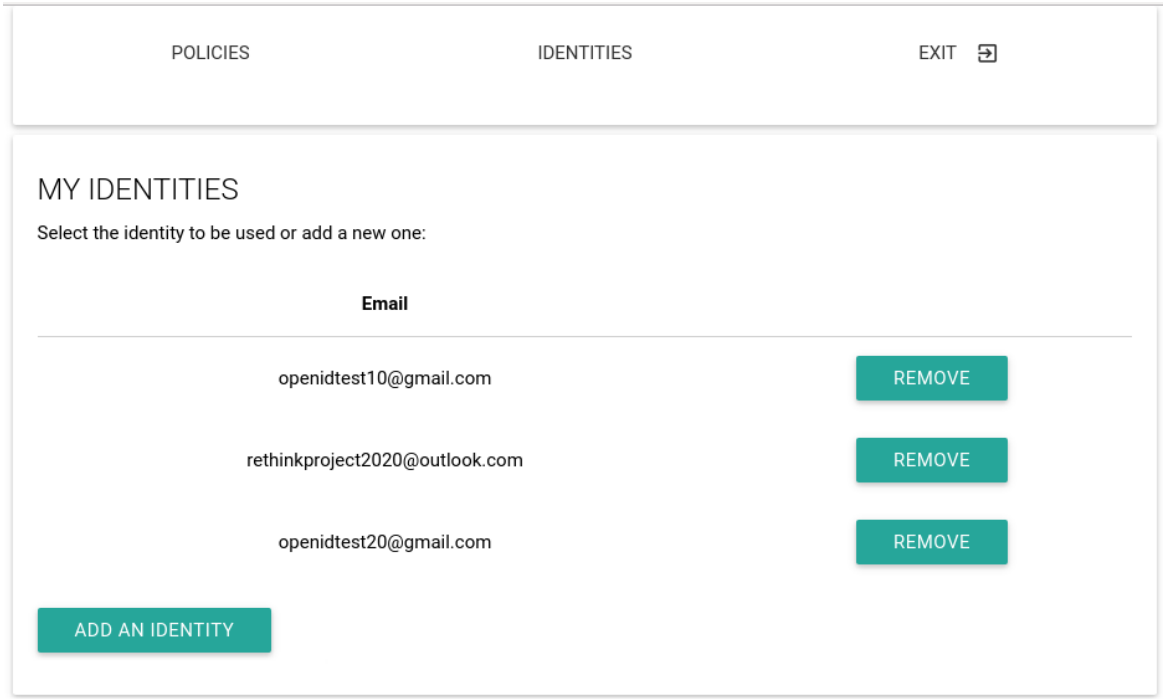


Figure 5.5: Screenshot of the Identity GUI

The identity GUI is available in two occasions: when the user clicks the configuration button and the identities button, and when the users register a Hyperty. The first one stays open until the user hits the exit button. In the second one, when the user selects a Hyperty to be registered, the identity GUI opens immediately waiting for the user to select a previously registered identity or to add a new one. When the user chooses an identity or registers a new one, the identity GUI closes and the Runtime Core proceeds with the Hyperty installation. The decision to show and close the identity GUI when registering a new Hyperty, came from the need to provide a better experience for the user. With this solution the user is not required to open and close the identity GUI whenever an identity is required when installing a Hyperty.

[2]http://materializecss.com/

To allow for the removal of identities from the list, the Identity Module needs to keep track of two lists containing identities: the first one containing the list of all Hyperties previously registered and the respective public/private key pair, and the second one containing the list of identities to be displayed in the identity GUI. So, whenever a new identity is acquired, it is added in both lists, but when the users removes an identity in the identity GUI, this is only removed in the second list containing the identities to be displayed. The identities from the first list cannot be removed, because it maybe has been already assigned to a Hyperty and if other users contact that Hyperty, the Identity Module is required to be in possession of the public/private key pair to execute the mutual authentication protocol.

In order to be able to add news identities and remove them, the identity GUI communicates with the Identity Module, given that the Identity Module object pointer is received in the constructor from the identity GUI, mainly since it is the last component of those two, to be deployed. However, the Identity Module also needs to contact the identity GUI. Thus, in order for the Identity Module to communicate with the identity GUI, the Message Bus is used being the communication done with messages. When the identity GUI is instantiated it adds a listener in the Message Bus for a specific URL, and when the Identity Module intends to communicate with the identity GUI, it sends a message for that specific URL waiting for the message response to continue its operations.

Figure 5.6 depicts the messages flow required when an identity is requested to the Identity Module. When a request for an identity is made to the Identity Module, it sends a message for the Identity GUI, with a list of all available identities previously registered and a list of supported IdPs. After the user selects the identity or wishes to register a new identity, the identity GUI replies to the Identity Module with the choice of the user. In case the received message contains an identity, the Identity Module returns that identity to who made the request, if the message contains information about an IdP, the Identity Module proceeds to contact the respective IdP proxy to obtain a new identity, as detailed ahead. After receive the identity assertion by the IdP proxy, the Identity Module stores it and returns that identity assertion.

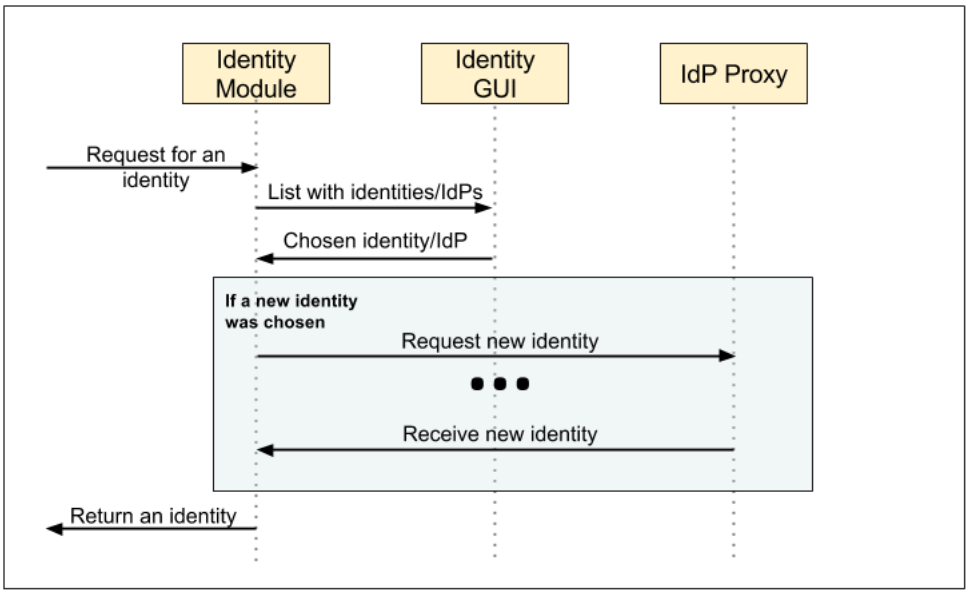


Figure 5.6: Message flow for the identity GUI

5.1.3 IdP proxy

Given the defined requirements to support several Identity Providers, the IdP proxy as detailed ahead was defined. The IdP proxy component is provided by the Identity Service Providers, and is downloaded an instantiated by the Runtime Core in the user device whenever a request to obtain an identity from a IdP is made for the first time.

The Runtime Core security is a major concern and the instantiation of a component downloaded from a possible untrusted IdP endpoint can lead to security flaws, such as, code injection attacks. To mitigate this security issue, when instantiating the downloaded IdP proxy, the Runtime Core put the IdP proxy code into a private sandbox, isolated from the Runtime Core and Hyperties components. In the reTHINK framework the IdP proxy is put on a web worker in a private thread that runs isolated from the rest of the code. The code isolation provided by the web worker, prevents the code running in it, from accessing the default methods and properties of the window object.

With the isolation of the Runtime Core and the IdP Proxy, the communication between these two components cannot be achieved by the direct invocation of the methods. This communication must be done through a Message Bus, a component instantiated by the Runtime Core but accessible by every component running the reTHINK framework. This includes Hyperties, IdP Proxies, and the Runtime Core. The IdP proxy communication with the IdM is performed using the Message Bus as illustrated in Figure 5.7, but for the IdP proxy to use the Message Bus, a small component need to be installed, namely the Protocol Stub.

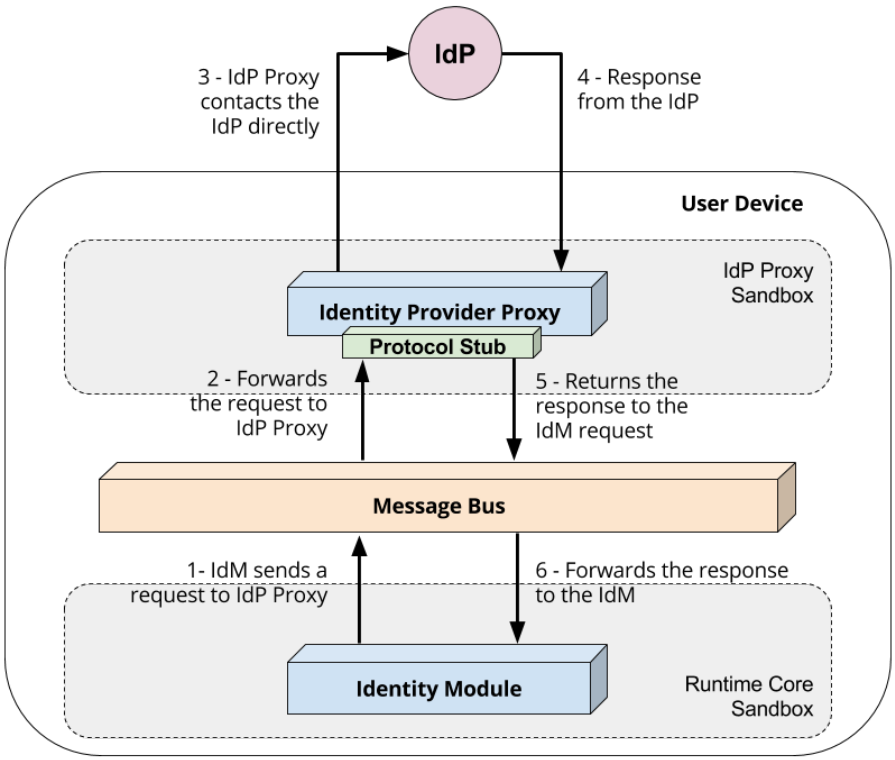


Figure 5.7: IdP proxy messages flow

The IdP proxy Protocol Stub component is installed along with the IdP Proxy in the isolated sandbox

and enables the use of the Message Bus. The Protocol Stub adds a listener for a specific URL, for example 'domain-idp://microsoft.com', so, messages for that URL are intercepted and depending on the message request, different IdP Proxy methods are directly called by the Protocol Stub. When it receives the result of the method, the Protocol Stub creates a response message and sends the result to the component that sent the message with the request.

The Figure 5.7 depicts the message flow between the Identity Module and the IdP proxy when requesting an identity assertion. Because the communication is done through the Message Bus, when the IdM needs to call an IdP proxy method, it sends a message that is forwarded to the IdP proxy by the Message Bus. The Protocol Stub intercepts the message and calls the IdP proxy method directly. The IdP Proxy can then contact the IdP directly to request or validate information regarding the user authenticated. When the IdP proxy receives the IdP response, it returns the value to the Protocol Stub. The Protocol Stub then creates the response message and posts it via the Message Bus, so that the IdM can receive the response. With this solution the IdM does not need to contact directly with the IdP, being the IdM unaware of the IdP used.

The solution found to implement the Protocol Stub along with the IdP proxy, was to add the code from the Protocol Stub into the IdP proxy code, both in the same file, so that when the Runtime Core deploys the IdP Proxy the Protocol Stub is also deployed. It is required for the Protocol Stub and the IdP proxy to be in the same file, since the installation process by the Runtime Core can only be done to one file at the same time. This stub consists in a code with four methods:

- ***activate(url, messageBus)*** : This method is required for the deployment of the Protocol Stub and IdP proxy within the reTHINK framework. Receives an assigned internal URL and the Message Bus.
- ***constructor(url, messageBus)*** : This is the default constructor for the Protocol Stub, and is called by the activate method, during the deployment. The constructor also adds a listener for a specific address in the Message Bus.
- ***requestToIdp(message)*** : This is the method called in the message listener. Receives the message and interprets the message request. Then calls a IdP proxy method depending on the type of request.
- ***replyMessage(message, value)*** : This method is called after the method requestToIdP receives the response. It receives the originating message and the resulting value from the IdP proxy method. With these, it creates a response message with the value received and post it to the Message Bus.

Another requirement for the development of the IdP Proxy is to follow the WebRTC standard [25] for IdP Proxies. The IdP proxies created by the IdPs that follow the WebRTC standard are automatically compatible with reTHINK. This is possible because the WebRTC defines two methods for the IdP proxy implementation:

- ***generateAssertion(contents, origin, hint)***: This method receives as input the contents to be asserted, the origin that triggered the request and an optional hint parameter which can be used

for the authentication request. It returns a Promise containing a JSON with the identity assertion in this format:

```
{assertion: 'identityAssertion', idp: 'IdP info'}
```

- ***validateAssertion: (assertion, origin)*** : This method receives the identity assertion to validate the origin that triggered the request. If the identity assertion is successfully validated, it returns a Promise with the identity information in the format:

```
{identity: 'user identifier', contents: 'identity information'}
```

When the *generateAssertion* method listed above is called, the IdP proxies using the OpenID connect protocol will try to open the OIDC URL so that the user can authenticate towards the IdP. However, this will fail since the IdP proxy is running in a web worker and cannot access the libraries that manage the opening of new child windows. This problem is considered by the WebRTC standard. Whenever the opening of the new child window fails, the *generateAssertion* method returns the OIDC URL so that others components are able to open the URL for the user to authenticate.

Figure 5.8 depicts the message sequence used for the *generateAssertion* method. The IdM starts by sending a request for the *generateAssertion* method, providing in the content field the public key to be asserted. When the IdP proxy tries to open the URL and fails, since it is running in an isolated web worker sandbox, it sends a JSON with a description of the error and a login URL to be received by the IdM, in order to open the new child window. After the user's authentication, the IdM closes the child window and collects any value received in the window. After this, the IdM calls the *generateAssertion* method again, but now providing the value received by the child window in the hint field. The IdP proxy then returns the identity assertion, which the IdM can use to establish the mutual authentication and establish the needed secure communications.

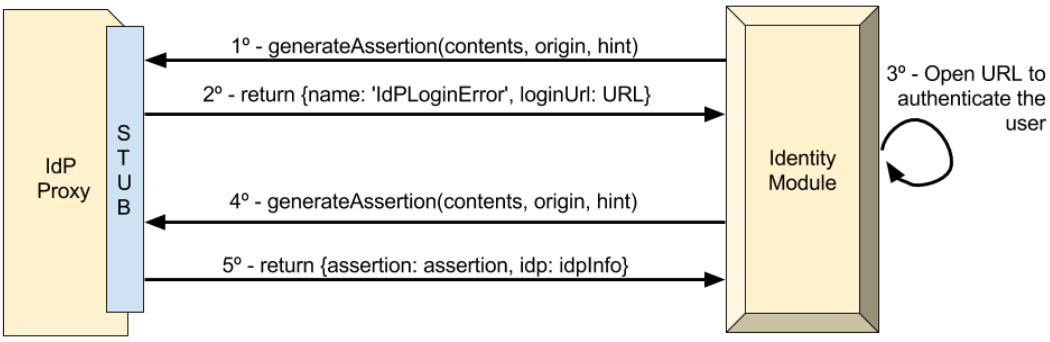


Figure 5.8: Messages sequence for the generateAssertion method

There are two distinct options for the IdP proxy to receive the response from the user authentication in the child window. These options are defined during the construction of the request URL, using the field 'mode'. One option is by returning the information explicitly in a JSON format to the window, to be retrieved by the IdM. The other method returns the information implicitly, by storing the information in cookies from the IdP domain. Afterwords, to IdP Proxy access the information stored in the cookies, instead of receiving it explicitly in the hint field.

The solution that uses cookies has some disadvantages compared to the information provided explicitly. The first problem comes from attempting to access the information stored in the cookies, because the code executing in web worker has constraints on what it can access. No simple solution was found to overcome this problem. The second problem, is in the limitation on the number of accounts that can be active at the same time, since the cookies only allow to access information on the last authenticated user. Thus, the cookie solution limits the number of possible active accounts, at most one per different IdP domain.

With the solution implemented in reTHINK framework, using the explicit method, the IdP proxy holds no limitation on the number of accounts for the same IdP domain that can exist at the same time, being in this case the responsibility on the IDP proxy side to manage the identities life-cycle. For the *validateAssertion* method, the process of validating the identity assertion is simpler. The Figure 5.9 illustrate the message flow, when the IdM wants to validate an assertion. The IdM sends the request with the identity assertion, to be verified and validated by the IdP proxy. The proxy in the end responds with the user information in case it is successfully validated, or with an error, in case it has not been successfully validated. To validate the identity assertion, the OIDC validation methods are used on the identity token.

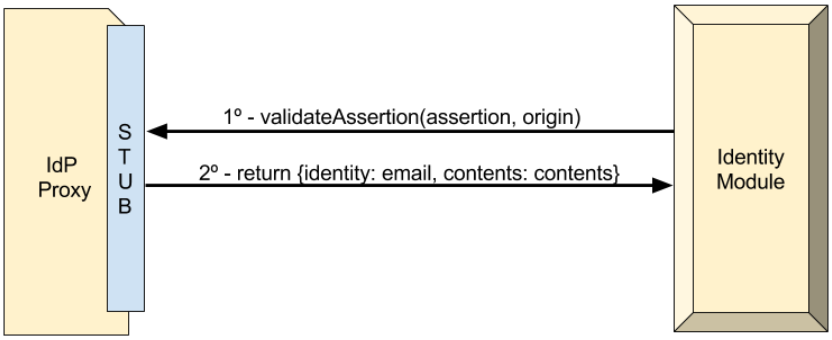


Figure 5.9: Messages sequence for the validateAssertion method

For the reTHINK project, two IdP proxies were developed. One for the Google IdP and another for the Microsoft IdP, both using OpenID connect protocol. The identity acquisition in the Identity Module is not limited to Google and Microsoft. These two IdP proxies were created as a proof of concept, to demonstrate that any IdP can create his IdP proxy compatible with reTHINK, if the WebRTC standards are followed.

5.1.4 Cryptographic Library

The Identity module is responsible for obtaining identities and for managing the secure communications. To do so, the IdM makes use of cryptographic methods, for example, to protect the message content. To assist with the management of the cryptographic methods in the IdM, a cryptographic library was created in this work.

The function of this library is not to implement the cryptographic protocol, but to help to abstract the implementation details of the cryptographic services in the IdM. This library defines what protocol should be used for a specific task and the key sizes required for each protocol. For example, if the IdM intends

to use a symmetric encryption to protect the message, it just needs to call the method responsible for that in the cryptographic library and does not need to defined anything else.

JavaScript does not yet have an official library for these cryptographic methods, but there is one recommended candidate from the World Wide Web Consortium (W3C), the Web Cryptography API [27]. The cryptographic library implemented for reTHINK make use of the cryptographic methods created and defined by W3C, because it is the most stable version and has native support in the most used web browsers, being the Chrome web browser in this list. The implemented cryptographic library does not implement the cryptographic functions, it presents only an API which makes easier the use of cryptographic methods implemented for the W3C.

Cryptographic algorithm choices used by the IdM are defined in this cryptographic library. For asymmetric encryption and decryption, the RSA cryptosystem, more specifically the RSA-OAEP with SHA-256, was chosen with a key size of 2048 bits. The signature and validation methods uses the RSASSA-PKCS1-v1_5 with SHA-256 cryptosystem, also with a key size of 2048 bits. For the symmetric encryption, the AES cryptosystem with Cipher Block Chaining (AES-CBC) was selected using a key size of 256 bits. To create keyed-hashes and validate them, the HMAC SHA-256 cryptosystem was chosen, using a 256 bits key size. The suggestions from the RSA security team[3] were taken into consideration for the size of the keys.

The implemented cryptographic library exposes an API with the following methods:

```
encryptRSA(pubKey, data);
decryptRSA(privKey, data);
signRSA(privKey, data);
verifyRSA(pubKey, data, signature);
encryptAES(key, data, iv);
decryptAES(key, data, iv);
hashHMAC(key, data);
verifyHMAC(key, data, signature);
generateRSAKeyPair();
generateIV();
generateRandom();
generatePMS();
generateMasterSecret(pms, data);
generateKeys(key, data);
encode(value);
decode(value);
```

The encrypt and decrypt methods are straightforward functions. They receive as a byte array both the key and the data to encrypt or decrypt, encrypting or decrypting the data and returning the end the result in a byte array. The same applies for the sign, hash and verify methods, they receive the keys and the data in a byte array format, and return a byte array, with the signature or the hash in case of the sign and hash methods respectively, and a Boolean in case of the verify method.

[3] http://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm

The generate methods, with the exception of the *generateMasterSecret* and the*generateKeys* return randomly generate values in a byte array format. The *generateRSAKeyPair* method returns a public/private key pair with 2048 bit length. Both the *generateRandom* and *generateIV* returns a random with 256 bits. The *generatePMS* returns a 384 bits random value. All of these random values are generated using the Web Cryptography API.

The *generateMasterSecret* and the *generateKeys* methods are similar, both receive a key and additional data, and return a byte array with 384 bits and 1024 bits (4 * 32 bytes), respectively. A particularity on these functions is that the information is not generated randomly, unlike the other generate methods. These methods make use on a pseudo random function (PRF), a similar method to the one used by the TLS to create the keys for the mutual authentication process.

Starting with the description of the PRF, the PRF(secret, label, seed) = P_hash(secret, label + seed), with P_hash defined as:

```
P_hash( secret, seed) = HMAC_hash( secret, A(1) + seed) +
                        HMAC_hash( secret, A(2) + seed) +
                        HMAC_hash( secret, A(3) + seed) + ...
```

where + indicates concatenation. The HMAC uses the hash algorithm SHA-256 and A() is defined as:

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

The P_hash can be iterated several times until it reaches the required quantity of data necessary to obtain all the required keys. The surplus data is discarded. For example, if only need 80 bytes are needed for the master key, a single SHA-256 iteration generate 32 bytes of data and 3 SHA-256 iterations generates 96 bytes of data. Therefore, discarding the last 16 bytes.

Using the PRF to compute the keys based on the protocol described above, the master key can be obtained from the result of PRF(premaster key, 'master secret', Alice random + Bob random) where only the first 48 bytes of the PRF output are used, as:

```
master_key = PRF(premaster_key, 'master secret', Alice_random + Bob_random)
```

For the *generateKeys* method, the keys are extracted from a key block that is the result of PRF(master key, 'key expansion', Alice random + Bob random), until there is enough data for the keys. Using this, the keys are for example:

```
key_block = PRF(master_key, 'key expansion', Alice_random + Bob_random)

Alice MAC key          = key_block [0, 31]
Bob MAC key            = key_block [32, 63]
Alice encryption key   = key_block [64, 95]
Bob encryption key     = key_block [96, 127]
```

JavaScript does not provide native support to serialise byte arrays, which is a problem since there are some values in byte array format that are needed to be exchanged, for example the random values. To

solve this, the methods *encode* and *decode* were created. The *encode* method converts a byte array into base64 format, and the *decode* method converts a base64 value into a byte array.

The separation of this library function from the IdM brings many benefits. It make it easier to manage small details about the cryptographic protocols used, since the IdM can abstract from these details. All the cryptographic methods are centralised in this library, and when another implementation needs to be used, this file can be swapped with a new cryptographic library, maintaining the compatibility with the IdM, making this library modular.

5.1.5 Mutual Authentication

The mutual authentication protocol used in reTHINK is based on the TLS protocol. Because the TLS is typically a protocol for the transport layer in OSI model[4], the mutual authentication protocol had to be develop from scratch, in order to be used in the application layer. During the protocol development, the TLS protocol was taken into consideration.

Analysing the mutual authentication protocol with a more in-depth perspective, depicted in Figure 5.10, the content exchanged in each message between Alice and Bob Identity Module are:

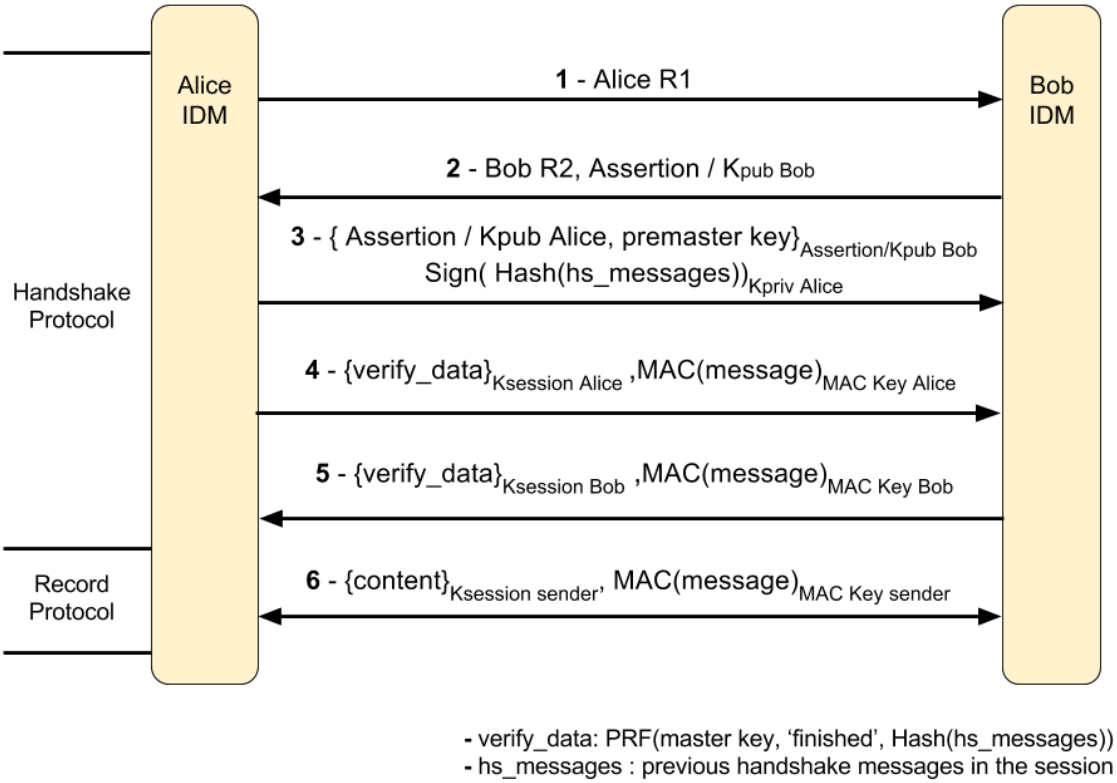


Figure 5.10: In-depth view of the mutual authentication protocol

[4]Open Systems Interconnection model (OSI model), is a reference model that standardise and characterise the communication functions, in order to facilitate clear comparisons among communications tools.

1. Alice sends a randomly generated value with 256 bits length.

2. Bob replies to Alice with is his Identity Assertion containing his public key with a 2048 bits. Receiving a randomly generated value by Bob with 256 bits length is sent to Alice.

   After receiving Bob's message, Alice verifies his identity assertion by requesting his IdP to validate it. If the IdP responds as valid, Alice then extracts the public key from Bob's identity assertion.

3. After receiving the public key of Bob, Alice generates a random pre-master secret(PMS) with 48 bytes (384 bits) and encrypts it, along with her identity assertion containing her public key, using the RSA encryption protocol. Alice also generates a signature using her private key, from the hash of all previous exchanged messages.

   In this phase, after generating the 48 bytes using PMS, Alice's Identity Module computes a pseudo random function (PRF) with the PMS and the two 256 bits random values exchanged previously, to obtain the master key with 48 bytes (384 bits) size as an output. Again, the PRF is computed, but this time receiving as input the master key and the two random values exchanged, to receive as output 4 blocks of 32 bytes (256 bits) each. Each block consists of a 256 bit symmetric key, and represents Alice's MAC Key, Bob's MAC Key, Alice's encryption Key and Bob's encryption Key.

4. Alice then sends in the message a block with 256 bits resulting from a PRF that receives as input the master key and the hash of all previous handshake messages. This block is encrypted with the AES protocol, using Alice's symmetric encryption key and a 256 bits hash. This is generated using a Hash-based MAC (HMAC) receiving as input the MAC key and the message.

   After receiving the message, Bob decrypts the block containing the identity assertion of Alice and the pre-master key using his private key. Bob's IdM proceeds to verify the identity assertion of Alice by requesting her IdP to validate it. After the validation and after receiving the public key of Alice, he uses her public key to verify the provided signature. If the message is validated, Bob's IdM proceeds to generate the master key with the same input as Alice's IdM, obtaining the 4 blocks of 32 bytes in the same manner as Alice, resulting in the same Alice's MAC Key, Bob's MAC Key, Alice's encryption Key and Bob's encryption Key.

   Now Bob is in possession of the symmetric keys to decrypt the message block, encrypted with Alice's symmetric encryption key, and generates a HMAC of the message to compare and validate with the one sent by Alice. If it matches, Bob's IdM proceeds to compute the PRF using the master key and the hash of all the previous handshake messages to compare the PRF result with the received decrypted data. If the data is equal, then Bob's IdM asserts that the information is valid and that the person on the other side of the communication is the same as the person identified in the identity assertion., i.e. Alice.

5. After generating the symmetric keys, Bob proceeds to prove to Alice his identity. To do so, Bob's IdM computes the PRF using the master key and the hash of all previous handshake messages and encrypts it with his symmetric encryption key. Then creates a HMAC from the message, with his MAC key and sends it to Alice.

   After the message reception, Alice's IdM uses the same procedures used by Bob to validate his identity. Starting by decrypting the message content with Bob's encryption key, the validation of the message integrity is performed by generating a HMAC from the message received and comparing

it with the HMAC received in the message. Then Alice's IdM computes the PRF using the master key and the hash of all previous handshake messages and compares it with the decrypted value to validate the content. After successfully validating the received message, Alice's IdM sets the handshake protocol as finished and marks the communication with Bob as authenticated.

At this phase, the handshake protocol is completed with both parts mutually authenticated. Additionally, the keys for a secure communication are created and shared by both IdMs.

6. The messages exchanged from now on, make use of the keys established in the handshake protocol. When Alice's IdM sends a message, its value is encrypted with Alice's 256 bits encryption key using the AES protocol and a HMAC of the message is generated using Alice's 256 bits MAC key. When Bob's IdM receives the message, it uses the Alice's keys to decrypt the content and to verify the integrity by generating a HMAC of the message and comparing it with the HMAC received. When Bob sends a message, the same steps are taken but this time with Bob's encryption and MAC key. When Alice's IdM receives the message, it also uses Bob's keys to decrypt and validate the message.

To store the resulting cryptographic information from the mutual authentication process, for example the session keys, the IdM make use of the hash tables. Using this key/value data storage, all cryptographic information is stored in the value field and a string containing the association of the two Hyperties to which the keys belong is stored in the key field, for example by storing the string 'hyperty://localhost/123-asd<->hyperty://localhost/45-rty'. This association is simpler to search for the cryptographic keys, since the field 'to' and 'from' from the message received contains the required information.

The mutual authentication can be triggered in two ways, either by sending a initial message to the other user, or by calling specifically the method to start the mutual authentication between two Hyperties. Usually this method is called when there is a request to subscribe a Data Object that enables group chat communications. So, when a the Data Object subscription triggered this method, the necessary keys for a secure communication are exchanged in the end of the mutual authentication protocol.

The Data Object encryption and hash keys are created when a Hyperty creates a Data Object, becoming the Hyperty reporter. These keys are then distributed to all Hyperties that get subscribed to that Data Object. These keys are exchanged from the Hyperty reporter to the Hyperty subscriber using the symmetric keys obtained in the mutual authentication protocol. This key is encrypted with the reporter's encryption key and authenticated with the reporter's MAC key. After decrypting and verifying the message, the subscriber sends a message confirming the reception of the Data Object key, encrypted and authenticated with his encrypt and MAC keys.

The decision to use another set of keys for the group chat communication comes from the need to simplify the message broadcast and not overload the user's device by sending individual messages for all Hyperty subscribed. For example, if a Data Object had 20 subscribers, this would require for the device to encrypt a message 20 times when it was sent, all with different set of keys. With this solution, the user device only encrypts each message for the Data Object once.

The management of the key associated to a Data Object is done by the Hyperty reporter. This set of keys is only shared with a Hyperty subscriber if the mutual authentication protocol is concluded successfully. This prevents malicious Hyperties to subscribe to a Data Object, preventing them from

reading the data exchanged in the messages. The group chat communication does not guarantee the same secure properties as the Hyperty to Hyperty communication, simply due to the fact that the same cryptographic keys are shared by all participants in the conversation. Thus, the message authenticity in a group chat cannot be granted.

5.1.6 Secure communication

The secure communications in reTHINK are established after a successful mutual authentication between two users IdMs. It supports two types of communication, a direct communication between users in this case a Hyperty to Hyperty communication, or a group chat communication where one Hyperty can communicate with many Hyperties subscribed in a data object.

To enforce the secure communications, two methods were created within the Identity Module, the 'secureSend(message)' and the 'secureReceive(message)'. The 'secureSend' method receives as input a message, creates a HMAC of the message, encrypts the message sensitive content, swaps the message value with the encrypted one and returns the message encrypted with the HMAC. The 'secureReceive' method does the exact opposite of 'secureSend' does. This method receives a protected message, decrypts it, verifies the integrity and returns the original message, with the data in plain text.

To accomplish their objectives the 'secureReceive' and 'secureSend' methods use the reTHINK cryptographic library. Namely, they use the methods encrypt and decrypt supported by AES, the create and verify HMAC methods, and the method to generate an initialisation vector(IV).

As mentioned in the previous chapter, all the messages exchanged between users are intercepted by the Policy Engine. This component is then responsible for calling the 'secureReceive' or 'secureSend' method from the IdM whether it is an incoming or outgoing message, respectively. The Policy Engine has the important role of filtering all the messages relevant for the IdM, so that the IdM only receives messages to be secured.

When the IdM receives a message to be secured, it makes an analysis based on a number of factors. First, it verifies if it is a Hyperty to Hyperty communication, or if it is a group chat communication. The distinction from the type of communication is made by analysing the source and the destination of the message. Using the Figure 5.11 to demonstrate the structure of a message used in reTHINK, if both fields 'to' and 'from' contain a Hyperty URL, then it is a Hyperty to Hyperty communication. Otherwise, if the field 'from' contains a Hyperty URL and the field 'to' has a Data Object URL, then it is a group chat communication. Given this the IdM checks if the keys for the Data Object or the Hyperty to Hyperty exists, and if not triggers the mutual authentication protocol.

Towards a secure communications in reTHINK, confidentiality is achieved by encrypting the content of the message using AES with a key size of 256 bits, integrity is assured by creating an HMAC with a 256 bits key of the entire message. This is valid for both Hyperty to Hyperty and group chat communication, being the only difference in the keys used for the encryption and the HMAC. When encrypting the message, only a subset of this message is encrypted. Taking as example the message structure depicted in Figure 5.11, the information to be encrypted is inside the 'message.body.value' field, while the HMAC is obtained from the entire message. Given that other reTHINK components need to see certain message fields, the encryption on the entire message cannot be applied.

```
{
  type: "create",
  from: "hyperty://localhost/499966e1-84e1-4539",
  to: "comm://localhost/b537fce7-672b-49b7-a0e5/children/chatmessages",
  body: {
    resource: "hyperty://localhost/499966e1-84e1-4539#1",
    value: {
      message: "Hello Alice!"
    },
    identity: {
      userProfile: {
        username: "openidtest10@gmail.com",
        userURL: "user://gmail.com/openidtest10"
      },
      idp: "google.com",
      "assertion": "eyJ0b2tlbklEIj...wIn19"
    }
  },
  id: 6
}
```

Figure 5.11: Structure of a message to be encrypted

The used cryptographic methods, return the result in a byte arrays type. However, to send this data through the network, it is required to serialize the data. Because JavaScript does not provide native libraries to serialize the data, the problem was studied to find the best solution for the serialization. The solution had two candidates, the first one was turning the byte array into a string with the JSON stringify method, the second one was encoding the byte array with the Base64 encode tool.

Taking as example the serialisation tests made and depicted in Figure 5.12, it is possible to see that a byte array of 5 bytes, results in a 40 bytes string with the JSON stringify and in 24 bytes using the Base 64 encode. Although these values may vary slightly, it is a good indicator of the cost it takes to serialize a byte array. Given the best results of Base 64 encode, this was the method chosen as the serialization method to be used in the secure communications.

```
//considering an byte array of 5 bytes using new Uint8Array(5);
encryptedContent = [143, 111, 102, 231, 13];

//the resulting string from JSON.stringify(encryptedContent):
-> "{"0":143,"1":111,"2":102,"3":231,"4":13}"
//with a size of 40 bytes

//the resulting string from encoding with btoa(encryptedContent):
-> "MTQzLDExMSwxMDIsMjMxLDEz"
//with a size of 24 bytes
```

Figure 5.12: Encoding examples of a byte array

5.2 Runtime Registry

The Runtime Registry is the component responsible for registering the multiple data from the several internal components of the reTHINK framework. To do so, it provides a specific API so other components from the Runtime Core can use it. This API is illustrated in Figure 5.13. All these methods are imple-

```
getAppSandbox();
registerHyperty(sandbox, descriptorURL, descriptor);
updateHypertyInstance(resource, value);
unregisterHyperty(url);
unregisterAllHyperties();
getHypertyOwner(hypertyURL);
getHypertyName(url);
registerDataObject(identifier, dataObjectSchema, dataObjectUrl,
   dataObjectReporter, resources, authorise);
registerSubscriber(dataObjectURL, subscriberURL);
getDataObjectSubscribers(dataObjectURL);
getReporterURL(dataObjectURL);
unregisterDataObject(name);
registerStub(sandbox, domainURL);
discoverProtostub(url);
unregisterStub(url);
registerIdpProxy(sandbox, domainURL);
discoverIdpProxy(url);
resolve(url);
```

Figure 5.13: Runtime Registry API

mented using the Promise API to turn them asynchronous. This is required for some methods since they have dependencies on others asynchronous requests. For example, the method 'registerHyperty' sends a request to the public Domain Registry to register the Hyperty instance and waits for the response in a callback to resume its logic.

With the option to return Promises even if the initial method is synchronous, in case the methods from this module are later required to implement persistence on the data stored, or to call asynchronous methods in its logic, with all methods returning promises, other modules calling these methods do not need to restructure their code to handle the Promise returns. Another reason for implementing the Promises is that they can be used for symmetric and asymmetric cipher methods without sacrificing performance.

Information within the Runtime Registry is not required to be persistently stored, where another component of reTHINK is responsible on the information persistence. Given this, the information in the Runtime Registry can be stored in volatile data structures, so, the hash tables data structure were chosen. The hash table is a key value data storage, where the key can map to the respective value, using hash functions to compute a key that will be assigned to an array of buckets to where the information will be stored.

For every component registered in the Runtime Registry a new runtime URL is assigned to identify it. As such, when using the hash tables to store information, the URL of the object is used as the key to

map the respective bucket which will contain all the informations about the components being registered. For example, when the method 'registerHyperty' is called, the Runtime Registry requests the Domain Registry for a URL to associate it with the sandbox received. This URL is used as the key element, and all the information about the sandbox, the meta-data, and the identity associated with it will be stored in the entry of the hash table.

5.2.1 Discovery Library

One of the tasks of the Runtime Registry is to register the information regarding the association of identities to components registered locally, on a public discovery service. For example when the 'registerHyperty' method is called, the Runtime Registry sends the information about the association of the user identity to the Hyperty being registered in the device, to a public Domain Registry. To facilitate the search for these public associations by other components, the Discovery Library at the Runtime Core was created.

The Discovery library is implemented using the JavaScript Class object, allowing it to be instantiated in any reTHINK component. It exposes the API with the methods depicted in Figure 5.14.

```
constructor(hypertyURL, msgBus);
discoverDataObjectPerName(name, domain);
discoverDataObjectPerURL(url, domain);
discoverDataObjectPerReporter(reporter, domain);
discoverDataObject(name, schema, resources, domain);
discoverHyperty(user, schema, resources, domain);
discoverHypertyPerUser(email, domain);
discoverHypertiesPerUser(email, domain);
```

Figure 5.14: Discovery Library API

Excluding the constructor, all these methods return Promises, since a information query to the public Domain Registry is required. These messages are sent via the Runtime Core internal Message Bus, which is received in the constructor. The discovery library allows querying the Domain Registry from any domain, by specifying the intended Domain in the 'domain' field.

5.2.2 Identity Manager library

The Identity Manager library has a similar task as the Discovery library, with the difference being in the component that the Identity Manager has to contact to request the information. This library contacts the Runtime Registry in order to obtain the desired information. It shares the same implementations methods as the Discovery library and provides the API depicted in Figure 5.15.

```
constructor(hypertyURL, runtimeURL, msgBus);
discoverUserRegistered(type);
```

Figure 5.15: Identity Manager Library API

This library offers a method for the Hyperties to know the identity that was assigned to them, and allows selecting the type of information of the user required. It supports requests for specific information such as, the email, user full name, and the user URL. By default, if no specific data is requested is sent all the information about the associated identity.

On the Runtime Registry side, a listener is added in the Message Bus to enable this Identity Manager library. This listener checks which information is being requested from the Hyperty that made the request. The Runtime Registry searches the information about the identity associated to the Hyperty, creating a response message and sending it back via the Message Bus, to be catch by the Identity Manager library. This library, after receiving the response message, extracts the information from the message received and sends it as the return value.

5.3 Summary

This chapter described the implementation details for the Identity Module and Runtime Registry. It provided a description on the main decisions made each for specific part of the developed components, as well as solutions created to assist the components. For the Identity Module it describes the identity acquisition process, the GUI to manage identities, the IdP proxy to request identity assertions to IdPs, the Cryptographic library, created to ease the manipulation of the cryptographic methods; the mutual authentication protocol, created for reTHINK and the secure communication channel. For Runtime Registry its architecture and the two libraries used by the Hyperties is described, the Discovery, and the Identity Manager Library. It also lists the resulting API of the several components and details how they work.

6 Evaluation

This work presents a solution to manage the users authentication towards the IdP and the mutual authentication between users, in order to ensure the establishment of the secure communication, within the reTHINK framework. In order to evaluate the performance of the proposed and developed solution, several tests were performed to evaluate the performance cost to the reTHINK framework in several aspects.

This evaluation covers the Identity Module and the Runtime Registry components. For this evaluation, 1000 samples were taken for each test and averages of these samples were considered. The tests were performed on a computer with the following characteristics: CPU with an intel core i5-3210M running at 2.5 Ghz, 8GB of memory RAM and the Ubuntu 14.04 LTS operating system.

The tests were performed on the components developed and integrated in the reTHINK framework. To test them, scripts were made in JavaScript language to allow them to run in the web browser and to allow data to be retrieved. Since some scenarios requires two users to evaluate the component, for example the mutual authentication protocol, two web browsers were used, namely Chromium and Chrome. Each browser represents a single user running an instance of the reTHINK application. The decision to separate the users in different browsers comes from the need to ensure that each instance of the reTHINK application running in the browser has its own resources and is running isolated from each other. The Chromium and the Chrome web browsers were chosen given the similarity of their structure.

To measure the times of each method or sub parts of the methods, the performance tool[1] which offers an API to obtain the current time in sub-millisecond resolution was used. To calculate the time each part takes, the 'performance.now()' function was called at the beginning of the method under test and called again at the end of the method to be tested. The measured time is the difference from the time obtained in the end minus the time obtained at the beginning.

6.1 Identity Module

This section introduces the evaluation performed for the Identity Module. It covers the following features: identity acquisition from a user, the mutual authentication protocol and the secure communication using messages. Tests that required two users in different web browsers are demonstrated using the fictional users Alice and Bob, which are represented with the letter 'A' and 'B', respectively.

[1] https://www.w3.org/TR/hr-time-2/

6.1.1 User Authentication

The process of authenticating a user and obtaining his identity comprises several steps. Starting by the generation of a public/private key pair for the RSA protocol, followed by the call to the IdP to generate the assertion. In the first call a URL to authenticate is obtained. A second call for the generate assertion method to actually obtain the identity assertion followed by the storage of this identity. The times for these steps were measured to find the performance cost added to the reTHINK framework by using the developed solution which is based on the WebRTC standard. For those tests, Google IdP proxy, developed in this Thesis was used to obtain the identity. This test ignores the time the user takes to select the identity via the GUI and the time the user takes to insert his credentials.

Figure 6.1 illustrates the time each component takes for the whole process of identity acquisition, showing an average of 1000 measures. These results suggest that, the methods with more impact on performance are the generation of the public/private key pair, the opening of a new window and the second call of the generate assertion method. The key generation time, taking about 200 ms, is relevant. However the main impact is in the opening of a new window, followed by the second call to the generate assertion. This second call includes the time the IdP takes to generate the assertion itself.

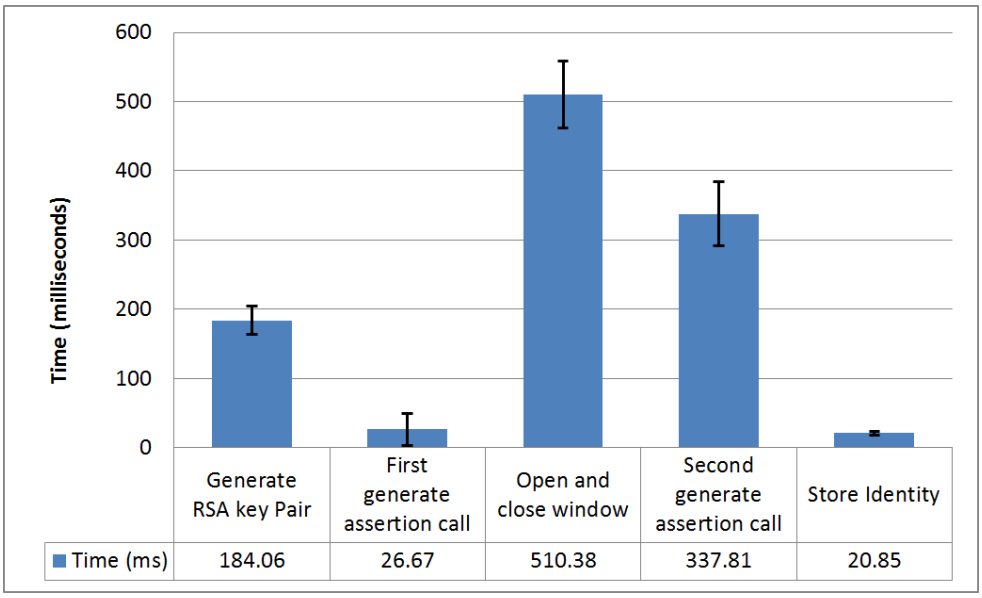


Figure 6.1: Chart representing the Averate time for each method during the identity acquisition

The considered mutual authentication solution herein proposed requires the Identity Module to generate a public key to be asserted by the identity module. Taking into account that the average time needed for the identity acquisition is approximately 1080 milliseconds, and that the RSA key pair generation only corresponds to 17% of this time, it can be considered that the actual local processing is low. It can also be concluded that the methods causing the highest performance degradation on the user authentication, is the actual interaction with the IdP server, particularly by the Google endpoints. Further optimisations on the user authentication could be made by removing the generation of the keys and have a key pair previously generated, when no activity is being made on the reTHINK application, however this may have an impact on privacy issues.

6.1.2 Mutual authentication

The mutual authentication is triggered whenever a user starts a communication with another user for the first time. This is essential so that both users are able to mutually authenticate each other before starting to exchange messages, thus preventing attacks, such as man-in-the-middle attack.

For the mutual authentication protocol, the time each phase of the protocol takes was tested, in order to evaluate the more demanding phases and its performance impact on the reTHINK framework. The execution of several mutual authentications simultaneously in order to determine the responsiveness of the Identity Module to more extreme cases. These tests were performed using two different browsers, so they do not share the same resources provided by the browser. The tested protocol starts with a message sent from Alice to Bob and is divided in 6 phases, being the phase 1, 3, 4 and 6 on Alice's device, and the phase 2 and 5 on Bob's device.

The Figure 6.2 illustrates the average times that each phase of the mutual authentication protocol described in section 4.1.1 takes. As expected, the most demanding phases are phase 3 + 4 and phase 5. It is in these phases that the operations with asymmetric encryption used to validate the identities and to encrypt the secrets occur.

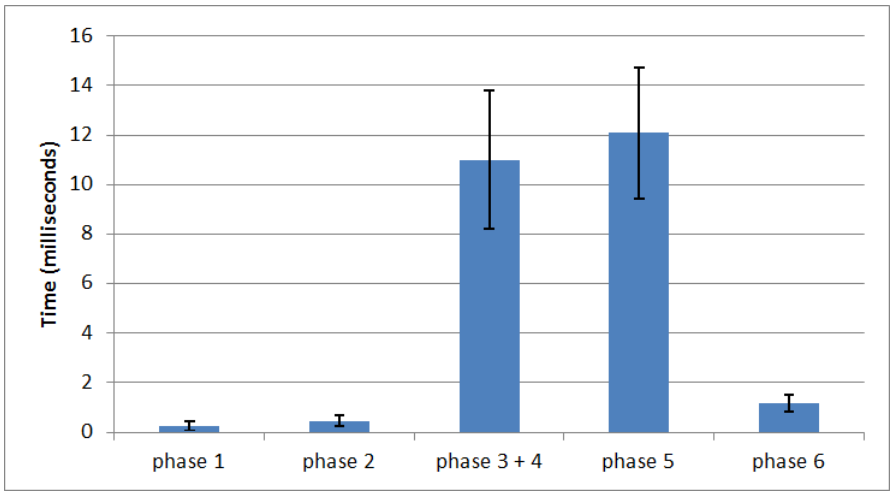


Figure 6.2: Chart representing the average time of each phase for a single mutual authentication process

Table 6.1 illustrates the total time for the mutual authentication execution and the percentage in regard to the total time. In phase 1 and 2 and 6, since there is no asymmetric cryptography the time they take are minimal. It is in the phase 3 + 4 and phase 5 that the largest time-consuming phases can be found, since it is where the heaviest asymmetric operations are made, such as asymmetric encryption and digital signature. Those times can be disregarded when compared to the total time including the round trip time (of about 48 milliseconds using a localhost Message Node), as described below.

phase 1	phase 2	phase 3 + 4	phase 5	phase 6	Total
0.24938	0.45186	10.99686	12.08462	1.16074	24.94346
0.24%	0.61%	44.43%	49.71%	5.02%	100.00%

Table 6.1: Table representing the percentage of each phase

To evaluate more demanding usage scenarios for this protocol, the time required to perform multiple mutual authentication processes simultaneously was evaluated, as depicted in Figure 6.3. From the values obtained it is possible to observe that the values grow linearly and do not introduce a bottleneck to the reTHINK framework.

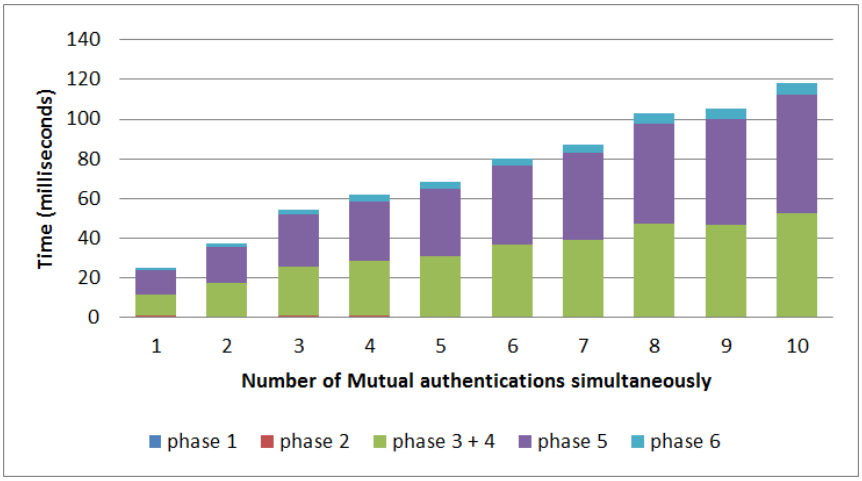


Figure 6.3: Chart representing the average times of several mutual authentications running simultaneously

The above evaluation does not contemplate the round trip time. Since a public Message Node is required, its time must also be taken into account. To evaluate the impact of this node in the reTHINK framework a Message Node was defined to run in a localhost environment. The Figure 6.2 illustrates the total time of the mutual authentication execution including the round trip time imposed by the messages exchanged itself.

Number of mutual authentications running simultaneously	Total mutual authentication execution time (ms)	Round trip time (ms)	Total time (ms)
1	24.94	23.20	48.15
2	37.54	28.40	65.93
3	54.51	39.09	93.60
4	61.71	39.32	101.03
5	68.14	40.23	108.37
6	80.30	44.74	125.04
7	87.33	46.50	133.83
8	102.76	55.13	157.89
9	105.47	55.45	160.92
10	118.26	59.32	177.57

Table 6.2: Table representing time of the mutual authentication several times and its round trip time

From this analysis, it is possible to conclude that the mutual authentication, in an ideal communication network, presents values almost unnoticeable to the common user (<200 milliseconds). Even in rare scenarios when 10 mutual authentication processes are executed. Since the execution time of the protocol is independent from the round trip time, the biggest bottleneck of this mutual authentication is in

the communication time itself, where the communication latency can introduce a significant time to the conclusion of the mutual authentication process.

6.1.3 Secure communications

Within the scope of this Thesis a method to provide secure communications in the messages exchanged between Hyperties was developed, being the Identity Module the component responsible for the encryption, decryption and integrity verification of those messages.

To perform the evaluations of the messages exchanged over the secure communication several tests were created that tried to replicate the sending of messages in different scenarios, from one Web Browser running an instance of the reTHINK application to another Web Browser also running instances of reTHINK, replicating the communication between two users. Both of reTHINK instances have a Hyperty running, representing a real case of two Hyperties communicating between them. The tests were performed by sending messages with different sizes, multiple messages simultaneously, and with different Message Nodes, one running in the test machine and the other running in the Altice Labs servers, one of the partners in the reTHINK project.

For each sent message the following operations were measured:

- **Runtime Core A**: Time it takes for the Alice's Runtime Core to process the message created by a Hyperty and send it, excluding the time taken by the IdM.
- **Runtime Core B**: Time it takes for the Bob's Runtime Core to receive the message, process it and delivery it to the Hyperty destination, excluding the time taken by the IdM.
- **IdM A**: Time Alice's IdM takes to encrypt and authenticate the message.
- **IdM B**: Time Bob's IdM takes to decrypt and to validate the received message.
- **Round trip time**: Travel time of the message, including the passage through the Message Node.

The performed evaluations consider the transmissions of several messages ranging from 16 Bytes to 8192 Bytes. It was not possible to send messages with more than 8192 Bytes of information due to message size limitations of the Domain Registry. Figure6.4 depicts the obtained results, excluding the round trip time of the messages. It is possible to observe that both IdM A and B suffer a rather small increment to the time required to process the messages with the increase of the message size. The obtained values suggest that the IdM does not introduce a bottleneck or a considerable decrease in performance, taking into consideration the total time taken from both Runtime Core's A and B to process the messages.

To test the responsiveness of the IdM to several message transmission requests at the same time, two tests were performed that send multiples messages simultaneously. One test ranging from 1 up to 10 messages sequentially and another test testing multiples of 10 messages up to 50 messages. Each message contains a 32 Bytes value to be handled by the IdM, simulating a conversation with the exchange of short messages between users. The results obtained for both tests are depicted in Figures 6.5 and 6.6. These Figures depicts the average time in milliseconds that each component takes to send

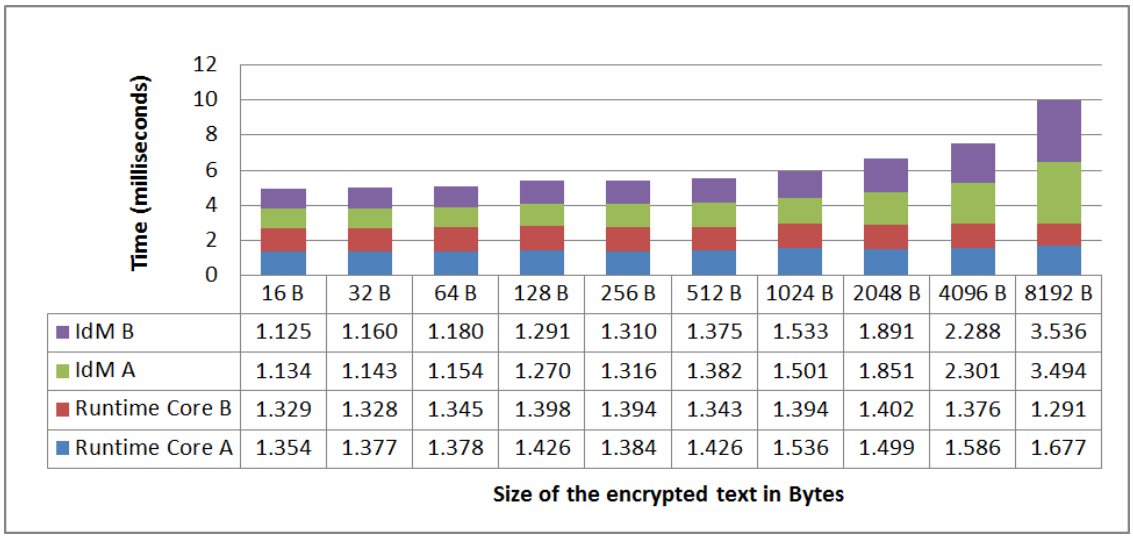


Figure 6.4: Chart representing the average time for sending a single message with different sizes

a single message when multiple messages are sent, including the round trip time. From these results it is visible that the A and B IdM delay increases with a linear proportion, for up to 50 small messages sent simultaneously. This linear increase is due to the fact that browsers do not possess concurrency, meaning they only use a single thread to run JavaScript code. Because the IdM methods are asynchronous, it is possible for the IdM method to loses its thread processing time to another method, only regarding the processor later on to conclude the task. The time the method is interrupted is accounted in the total time the method takes.

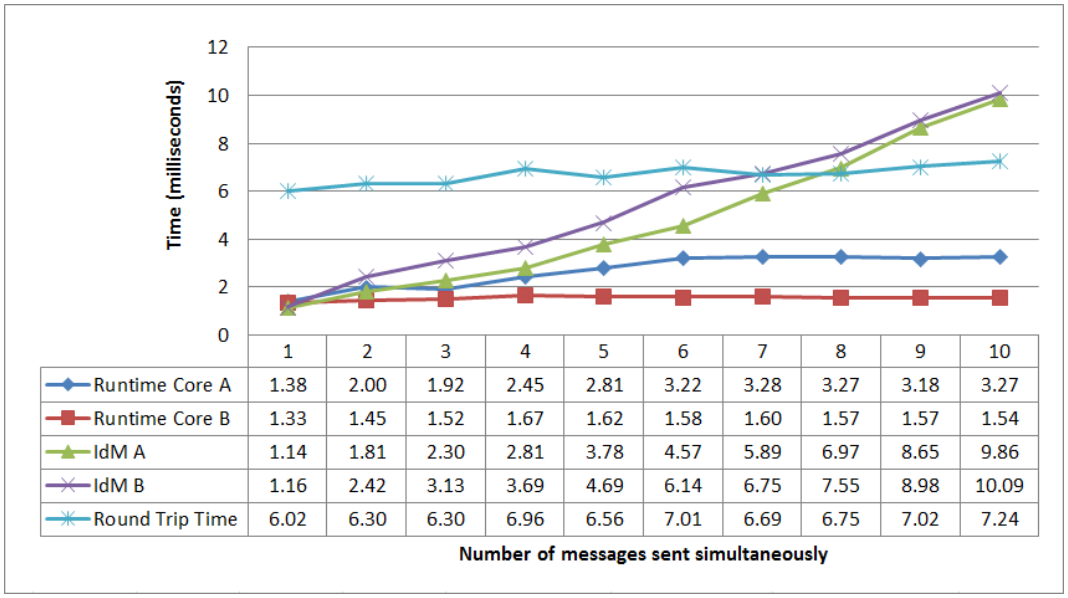


Figure 6.5: Chart showing the average time each component takes to send each message, by sending multiple messages simultaneously (1 to 10 messages)

When sending multiple messages simultaneously there is a difference between the latency and the throughput, being the latency in this case the total time to send a single message takes and the

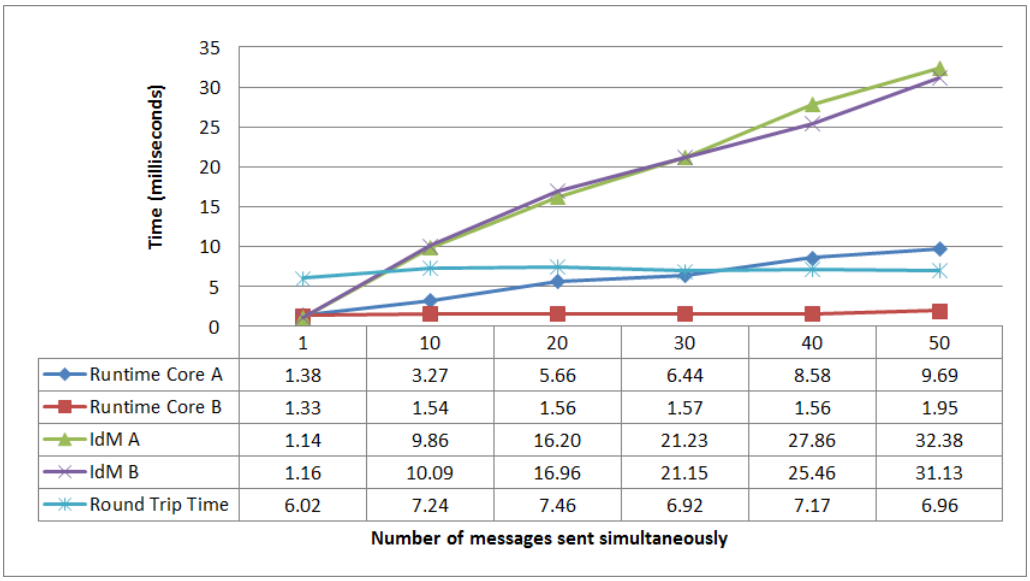


Figure 6.6: Chart showing the average time each component takes to send each message, by sending multiple messages simultaneously (1 to 50 messages)

throughput the total time to send all the messages. So, the obtained results are depicted in Table 6.3, showing the individual time for each component, the total time for each message, the latency and the total time of all messages. For this test a localhost Message Node was used, leading to similar round trip times, independently of the number of messages sent. From the results it is possible to observe that the latency increases with the increase of the messages sent simultaneously, resulting in a sub linear increase. Despite this increase with number of messages sent simultaneously, the total delay is relatively low, even for multiple messages.

Number of messages	Runtime Core A	Runtime Core B	IdM A	IdM B	Round Trip Time	Total Time of each message	Total time of all messages
1	1.38	1.33	1.14	1.16	6.02	11.03	11.03
2	2.00	1.45	1.81	2.42	6.30	13.97	14.62
3	1.92	1.52	2.30	3.13	6.30	15.17	16.24
4	2.45	1.67	2.81	3.69	6.96	17.58	19.93
5	2.81	1.62	3.78	4.69	6.56	19.46	21.79
6	3.22	1.58	4.57	6.14	7.01	22.52	24.36
7	3.28	1.60	5.89	6.75	6.69	24.21	27.90
8	3.27	1.57	6.97	7.55	6.75	26.11	29.76
9	3.18	1.57	8.65	8.98	7.02	29.40	32.15
10	3.27	1.54	9.86	10.09	7.24	32.00	34.41
20	5.66	1.56	16.20	16.96	7.46	47.85	65.10
30	6.44	1.57	21.23	21.15	6.92	57.30	88.66
40	8.58	1.56	27.86	25.46	7.17	70.63	112.77
50	9.69	1.95	32.38	31.13	6.96	82.11	134.81

Table 6.3: Table representing the total time each message takes to send, individually and all together, using the localhost Message Node (Time in milliseconds)

To evaluate the impact of the Message Node, the same tests were performed, but this time using a public Message Node located in the Altice Labs servers. The values are depicted in Table 6.4. These values suggest that the times for both Runtime Cores and A and B IdMs remains similar independently of the Message Node used, where the biggest difference is in the round trip time, where in this last test

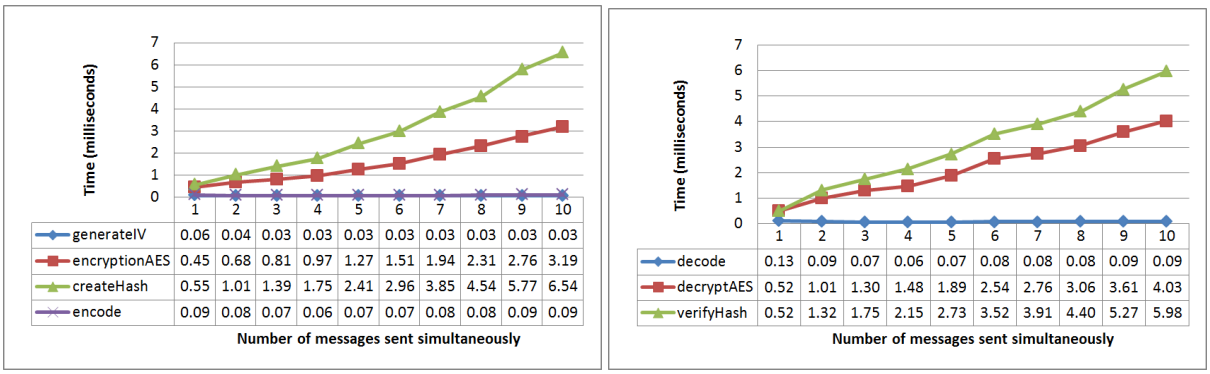
it expectedly takes more time.

Number of messages	Runtime Core A	Runtime Core B	IdM A	IdM B	Round Trip Time	Total Time of each message	Total time of all messages
1	1.35	1.40	1.13	1.14	302.35	307.37	307.37
10	3.29	1.52	9.89	10.13	261.14	285.97	459.69
20	5.62	1.58	15.17	15.91	482.21	520.49	635.80
30	6.89	1.58	20.28	21.21	399.85	449.82	613.57
40	9.56	1.55	25.93	26.33	408.41	471.78	815.46
50	10.12	1.98	31.34	32.36	456.56	532.36	974.32

Table 6.4: Table representing the total time each message takes to send, individually and all together, using the Altice Labs Message Node (Time in milliseconds)

From these results it is possible to conclude that the biggest bottleneck in the transmission of messages is the network latency, and that the performance of the IdM is only slightly affected when sending multiple messages at the same time.

Making a deeper analysis of the IdM times, the IdM times were decomposed on the following aspects: initial vector (IV) generation, encryption using AES, message authentication and encoding, for the IdM A and the opposite, for the IdM B. The values depicted in Figure 6.7 suggest that when sending multiple messages simultaneously the values that tend to increase are related to the encryption and hash computation. The IV generation and the encode/decode steps are barely noticeable. It is also possible to observe that both A and B IdM take approximately the same to process a message, for the same number of messages sent simultaneously.



(a) Chart representing the average time of each step used by IdM A to process a message

(b) Chart representing the average time of each step used by IdM B to process a message

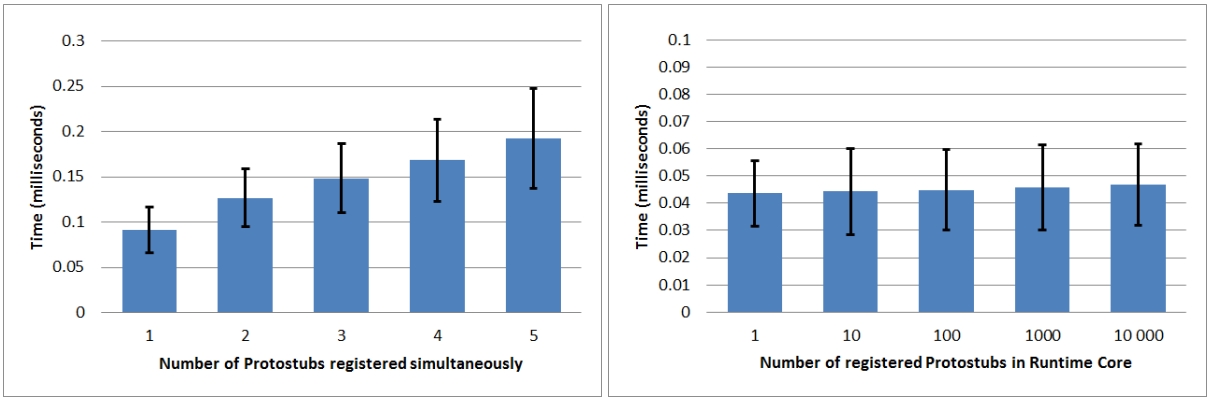
Figure 6.7: Methods times used for A and B IdM's

To provide a overall conclusion on secure communication using messages, it is possible to conclude that the performance required for the IdM methods do not increase exponentially, not even in the most extreme cases. This allows to the conclusion that the system is able to withstand the performance with adequate metrics, independently of the number of messages exchanged or the size of the messages. The biggest factor in the performance is the network latency.

6.2 Runtime Registry

The components that can be searched are and registered in the Runtime Registry are stored in hash tables. To evaluate its performance, the storage and load times for the Protostub component in Runtime Registry were tested.

To perform the test of the Protostub registration the total time it takes to perform the registration of 1 and up to 5 Protostubs at the same time were measured. The values depicted in Figure 6.8(a) shows the resulting average values for 1000 test samples. It is possible to observe that the resulting times are relatively low and the simultaneous registration of several Protostubs are almost negligible.



(a) Chart representing the average time for the Protostub registration message

(b) Chart representing the average time for the Protostub search

Figure 6.8: Times for Protostub registration and search

To test the search of the Protostub component stored in hash tables, the time it takes to search for one Protostub in a Runtime Registry with 1, 10, 100, 1000, 10 000 Protostubs registered was measured. The obtained results, depicted in Figure 6.8(b), shows that the search times is practically the same independently of the number of Protostubs registered in the Runtime Registry. This results from the efficiency of the hash tables, which scale very well.

The tests performed on the Runtime Registry shows very good scalability, justifying the decision to use hash tables to store the reTHINK internal components.

6.3 Summary

In this chapter several experimental evaluations were performed on different components developed in this work. The methodology for each test as well as the test results was provided with the results demonstrated in charts and tables. In the end of each test an explanation on the obtained tests was provided. For the Identity Module, tests were performed to evaluate the performance of the user authentication process, the mutual authentication protocol and the secure communications, from the most common cases to the more extreme ones. For the Runtime Registry, tests were performed to evaluate

the storage and search capacity of the reTHINK components using hash tables. These results allow concluding that the developed modules can achieve a good performance and scalability.

The next chapter concludes this Thesis by providing the conclusions regarding the work developed, also providing some opinions and ideas in terms of the future work developments.

7 Conclusions

7.1 Conclusions

This Thesis presents and details the proposed solution for two important modules of the reTHINK framework: the Identity Module and the Runtime Registry. The Identity Module is the component in reTHINK responsible for user authentication, identity management and deployment of secure communications. The Runtime Registry is responsible for the management and storage of the Runtime Core components, and, more importantly, for associating an identity to the user's device.

To design and develop the Identity Module, a survey on the related work was carried out. Among other subjects, several authentication mechanisms we presented from which we chose one to be implemented, the OpenID Connect authentication protocol. With the chosen authentication protocol, the best solution to implement it while following the reTHINK project requirements was studied. As a P2P framework, reTHINK requires that the implementation of the authentication protocol supports mutual authentication between two users. The proposed and implemented solution introduces a new mutual authentication protocol, similar to the well-known TLS, using the received identity token as a digital certificate. The mutual authentication protocol is achieved using Identity Providers with the OpenID Connect protocol, which can be used to assert a given public key, sent in the identity request. Since the identity token received contains a signature created by the IdP. This IdP behaves as a certification authority, where the identity token has a similar role as a certificate. This protocol allows generating session keys shared by the authenticated users, providing secure communications between them.

The Runtime Registry has the functionality to store internal components from the Runtime Core, such as Protostubs, Hyperties and Data Objects. Several storage solutions for browsers were studied, being the hash tables the chosen data structure to store the internal components. To enable the identification and discovery of the user device, the Runtime Registry associates an identity to a Hyperty running in the device, which is obtained from one of the two supported identity providers, Google and Microsoft. This association is then published in the public Domain Registry, so that it can be consulted by other users.

Finally, the evaluation of both modules was carried out, where the performance of the proposed solutions for the common load cases and for the heaviest load cases were tested. For the Identity Module the performance of the user authentication, the mutual authentication protocol, and the secure communication of the messages exchanged were evaluated. The obtained tests showed the capacity of the Identity Module to endure the tests with the heaviest load. For the Runtime Registry, tests were made to evaluate the storage and search capacities of the registered components. The obtained results validate the decision to use hash tables, showing good performances for both the storage and the search of components. To conclude, the objectives and requirements set for this Thesis were achieved: both

components are fully operational and integrated in the reTHINK project, and currently in use by the reTHINK partners.

7.2 Future Work

The solutions developed for this Thesis have plenty of room for improvements in both the Identity Module and Runtime Registry. For the Runtime Registry, a cache system can be implemented to store the information requested to the Domain Registry. With this cache it is possible to prevents the Runtime Registry from sending repeated requests to the Domain Registry. For the Identity Module, it would be advantageous for the user to add more authentication mechanisms to obtain identities, so that the user can have more freedom of choice. On a more practical aspect it would be beneficial to implement a method to ensure the message freshness, in order to avoid replay attacks, to check the stored identity tokens validity, and proceed to renewing them when necessary. With these improvements, the final solution will be more complete and flexible and will provide a better experience to the final user.

Bibliography

1. Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012.
2. reTHINK Consortium. Global Identity and Reachability Framework for Interoperable P2P Communication Services. *19th International ICIN Conference*, 2016.
3. J L Camp. Digital identity. *IEEE Technology And Society Magazine*, 23:34–41, 2004.
4. Jr. Matyas, V. and Z. Riha. Toward reliable user authentication through biometrics. *IEEE Security Privacy*, 1(3):45 – 49, 2003.
5. Lars Erik Holmquist, Johan Redström, and Peter Ljungstrand. Token-Based Acces to Digital Information. *Lecture Notes In Computer Science*, 1707:234–245, 1999.
6. Mike Jones and Dick Hardt. The oauth 2.0 authorization framework: Bearer token usage. Technical report, Internet Engineering Task Force (IETF), 2012.
7. Neil Haller, Craig Metz, Phil Nesser, and Mike Straw. A one-time password system. Technical report, Network Working Group, 1998.
8. Steffen Hallsteinsen, Ivar Jorstad, et al. Using the mobile phone as a security token for unified authentication. In *2007 Second International Conference on Systems and Networks Communications (ICSNC 2007)*, pages 68–68. IEEE, 2007.
9. Wenbo Mao. *Modern cryptography: theory and practice*. Prentice Hall Professional Technical Reference, 2003.
10. Mozilla. Persona - Mozilla — MDN, https://developer.mozilla.org/en-US/Persona, December 2015.
11. Henry Story. WebID Specifications, http://www.w3.org/2005/Incubator/webid/spec/, December 2015.
12. N. Sakimura. OpenID Connect Core 1.0, http://openid.net/specs/openid-connect-core-1_0.html, December 2015.
13. B. C. Neuman and T. Ts'o. Kerberos. An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, 1994.
14. Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
15. Jan De Clercq. Single sign-on architectures. In *Infrastructure Security*, pages 40–58. Springer, 2002.

16. Uwe Hansmann, Martin S Nicklous, Thomas Schäck, Achim Schneider, and Frank Seliger. *Smart card application development using Java*. Springer Science & Business Media, 2012.

17. Jeff Yan, Blackwell Alan, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2:25–31, 2004.

18. Jing Xu, Wen Tao Zhu, and Deng Guo Feng. An improved smart card based password authentication scheme with provable security. *Computer Standards and Interfaces*, 31(4):723–728, 2009.

19. Tim Dierks. The transport layer security (tls) protocol version 1.2. *RFC 5246*, 2008.

20. Jon Postel. Transmission control protocol. *RFC 793*, 1981.

21. K Tracy. Identity management systems. *Potentials, IEEE*, 27(6):34–37, 2008.

22. David F. Ferraiolo and D. Richard Kuhn. Role-Base Access Controls. *ACM Transactions on Information and System Security*, 2(1):34–64, 1992.

23. W D Maurer and T G Lewis. Hash Table Methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

24. reTHINK Consortium. reTHINK D2.2 - Data Models and Interface Specification of the Framework, 2015.

25. Eric Rescorla. Webrtc security architecture. *draft-ietf-rtcweb-security-arch-12*, June 2016.

26. reTHINK Consortium. Hyperty Runtime and Hyperty Messaging Node Specification, ReTHINK Deliverable 3.1, 2015.

27. World Wide Web Consortium. Web cryptography API, https://www.w3.org/tr/webcryptoapi/, 11 December 2014.