Security Policies Support for the reTHINK Web Service Architecture

Ana Sofia Lameira Caldeira

Thesis to obtain the Master of Science Degree in

Telecommunications and Informatics Engineering

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

Examination Committee

Chairperson: Prof. Fernando Henrique Côrte-Real Mira da Silva
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

October 2016
Acknowledgements

I would like to thank my advisors for their guidance in this final adventure of my academic life. I would like to express my sincere gratitude for giving me the right mindset on how to structure my wording during the writing of this document - I believe it will stay with me throughout my career and in general in the way I expose my ideas.

This work was partially supported by the reTHINK partners, who had always shown to be available to solve the problems that inevitably occur in projects as big as reTHINK. I would like to address my special appreciation to Gil Dias for his tireless support when my doubts and obstacles were bigger than I thought I could handle.

Last but certainly not least, I would like to thank my family for encouraging me to pursue my Master’s degree. A special thank you to Pedro Raminhas for being sure that I would be successful in this journey and for always being there for me when times have gotten rough. Words cannot express how lucky I am to have such motivational and inspiring person in my life.

Lisboa, October 2016
Ana Sofia Lameira Caldeira
For my advisors and my family,
Resumo

A popularidade dos serviços oferecidos pelas operadoras de telecomunicações tradicionais tem vindo a diminuir devido ao aparecimento de novos serviços de Internet, tais como serviços de mensagens instantâneas e as redes sociais. Contudo, muitas vezes esses serviços são incompatíveis, o que torna mais difícil conseguir interoperabilidade entre eles. Para ultrapassar este problema de incompatibilidade, a Comissão Europeia financiou um projeto de investigação de nome reTHINK, que tem como objetivo desenvolver uma arquitetura de serviços P2P centrada na web.

Para o desenho e desenvolvimento do projeto reTHINK, a existência de mecanismos para especificar e aplicar políticas de segurança é fundamental. Em particular, são necessárias para garantir que o acesso aos recursos é feito de forma autorizada e segura. Considere-se o cenário em que é possível gerir o contexto de um utilizador. A Alice pode estar interessada em apenas permitir que o seu amigo Bob tenha acesso à sua localização, mas não quer que isso aconteça com a Trudy, uma pessoa que perturba constantemente a sua privacidade.

O objetivo desta tese é o desenvolvimento de um subsistema de apoio à especificação e aplicação de políticas de segurança em domínios de aplicação heterogêneos, onde as políticas devem ser geridas facilmente e a sua avaliação deve ser eficiente. Concretamente, esta tese apresenta o PoliTHINK, um subsistema responsável pela especificação, gestão e aplicação de políticas de segurança no projeto reTHINK. O PoliTHINK oferece baixa complexidade na especificação de políticas, e ao mesmo tempo proporciona bom desempenho no seu carregamento e avaliação.
Abstract

The popularity of traditional operator-enabled services is decreasing as they are being replaced by Internet-based services, such as chat and social networks services. However, these services are often incompatible, which makes interoperability across them more difficult. To overcome this interoperability problem, the European Commission sponsored a research project named reTHINK, which aims to develop a web-centric P2P service architecture.

For the development of the reTHINK project, the existence of mechanisms to specify and enforce security policies is fundamental. In particular, security policies are necessary to ensure authorized and secure access to resources. Considering a user scenario where it is possible to manage human context. Alice may find it interesting to allow her friend Bob to know her location at all times, whereas she does not want that with Trudy, someone who constantly disrupts her privacy.

The goal of this thesis is the development of a subsystem to support the specification and enforcement of security policies in heterogeneous application domains, where policies must be easy to manage and whose evaluation should be efficient. Concretely, this thesis presents PoliTHINK, a subsystem responsible for the specification, management and enforcement of security policies in the reTHINK framework. PoliTHINK offers low complexity in policies specification and good performance in their loading and evaluation.
Keywords

reTHINK Project
Security Policies Enforcement
Security Policies Specification
Service Providers Interoperability
## Contents

1 Introduction ......................................................... 2
   1.1 Motivation .................................................. 2
   1.2 Goals and Requirements ..................................... 3
   1.3 Contributions ............................................... 4
   1.4 Structure of the Document ................................... 4

2 Background ......................................................... 6
   2.1 Usage Scenario ............................................... 6
   2.2 The reTHINK Framework ....................................... 7
      2.2.1 Core Runtime ............................................. 9
      2.2.2 Messaging Node .......................................... 10
   2.3 Data Flow in reTHINK ......................................... 11
      2.3.1 Data Objects Creation ................................... 11
      2.3.2 Data Objects Subscription ............................... 12
      2.3.3 Data Objects Update ..................................... 13
   2.4 Summary ....................................................... 14

3 Related Work .................................................... 15
   3.1 Access Control Models ....................................... 15
      3.1.1 Discretionary Access Control (DAC) .................. 15
      3.1.2 Mandatory Access Control (MAC) ..................... 16
      3.1.3 Role-Based Access Control (RBAC) ................... 16
      3.1.4 Attribute-Based Access Control (ABAC) ............. 16
   3.2 Security Policies in Browser and Mobile Platforms ....... 17
      3.2.1 Policies in the Browser Environment ................. 17
      3.2.2 Policies in the Mobile Environment ................. 18
### Table of Contents

#### 3.3 Policy Specification Languages
- **3.3.1 Content Security Policy (CSP)** ......................................................... 20
- **3.3.2 WS-Policy** ......................................................................................... 20
- **3.3.3 XACML Language** .......................................................................... 21
- **3.3.4 Ponder** ............................................................................................ 24
- **3.3.5 Discussion** ......................................................................................... 25

#### 3.4 Policy Enforcement Mechanisms
- **3.4.1 JSand** ............................................................................................ 26
- **3.4.2 ConScript** ......................................................................................... 27
- **3.4.3 WebJail** ............................................................................................ 27
- **3.4.4 XACML Enforcement** ..................................................................... 28
- **3.4.5 Discussion** ......................................................................................... 29

#### 3.5 Summary ........................................................................................................ 29

#### 4 Architecture .................................................................................................... 31

##### 4.1 PoliTHINK Overview .............................................................................. 31

##### 4.2 Policy Specification Language
- **4.2.1 Decision** ............................................................................................ 33
- **4.2.2 Scope** ................................................................................................. 33
- **4.2.3 Applicability** ...................................................................................... 34
- **4.2.4 Combining Rules Results** ................................................................. 34
- **4.2.5 Actions Service** ................................................................................ 35
- **4.2.6 Policy Syntax** ...................................................................................... 35
- **4.2.7 Combining Attributes** ...................................................................... 36

##### 4.3 Policy Engine
- **4.3.1 Policy Engine Architecture** .............................................................. 38
- **4.3.2 Handling Context** .............................................................................. 39
- **4.3.3 Storage** ............................................................................................... 42

##### 4.4 Administration Page .................................................................................. 42

##### 4.5 Summary ........................................................................................................ 45
5 Implementation

5.1 Implementation Overview ........................................... 46

5.2 Policy Specification Language ..................................... 48

5.2.1 Policy .......................................................... 49

5.2.2 Rule ................................................................... 50

5.2.2.1 Scope and Target .............................................. 50

5.2.2.2 Condition and Decision .................................... 52

5.2.2.3 Combining Algorithms and Priority ..................... 54

5.3 Policy Engine .......................................................... 55

5.3.1 Policy Enforcement Point (PEP) ................................. 56

5.3.2 Policy Decision Point (PDP) ..................................... 58

5.3.3 Context .................................................................. 59

5.3.3.1 Acquisition of attributes values from message fields .... 59

5.3.3.2 Acquisition of attributes values from JavaScript native methods ........................................... 59

5.3.3.3 Core Runtime .................................................... 60

5.3.3.4 Vertx Messaging Node ......................................... 62

5.3.4 Actions Service ....................................................... 64

5.4 Graphical User Interface ................................................. 65

5.4.1 Policies management ............................................... 67

5.4.2 Groups management ............................................... 70

5.5 Summary .................................................................... 71

6 Evaluation .................................................................. 72

6.1 Methodology .............................................................. 72

6.2 Memory Usage ........................................................... 73

6.3 Policy Loading Time .................................................... 75

6.3.1 Loading Time from Persistent Memory ....................... 75

6.3.2 Loading Time from Local Hash Table ......................... 76

6.4 Policy Evaluation Time .................................................. 77

6.5 Summary .................................................................... 78
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Simplified representation of interdomain communication in reTHINK</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>reTHINK impact on today's services</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>reTHINK framework (adapted from [1])</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Core Runtime of the reTHINK framework (adapted from [2])</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Messaging Node of the reTHINK framework (adapted from [2])</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Message sequence in a data object creation</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>Message sequence in a data object subscription</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>Message sequence in a data object update</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>General authorization architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Simplified version of Mozilla’s security architecture (adapted from [3])</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Simplified version of Android’s applications security model (adapted from [4])</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>CSP example (from [5])</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>WS-SecurityPolicy example (from [6])</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>XACML policy example</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>Ponder group policy example</td>
<td>24</td>
</tr>
<tr>
<td>3.8</td>
<td>Ponder policies class diagram (adapted from [7])</td>
<td>25</td>
</tr>
<tr>
<td>3.9</td>
<td>JSand operation (adapted from [8])</td>
<td>27</td>
</tr>
<tr>
<td>3.10</td>
<td>WebJail architecture (adapted from [9])</td>
<td>28</td>
</tr>
<tr>
<td>3.11</td>
<td>XACML data-flow model (adapted from [10])</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>reTHINK Policy Specification Overview</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>The reTHINK Policy Syntax in BNF notation</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Advanced condition's syntax for the and logical operator (a), for the or logical operator (b), for the not logical operator (c) and for a combination of the three (d)</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Example policy using not logical operator</td>
<td>37</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Policy specification languages comparison .................................. 26

4.1 Policy Engine conditions supported by the Core Runtime and the Messaging Node .... 40

5.1 Attributes of a condition and corresponding keywords ....................... 52
ABAC  Attribute-Based Access Control
ALFA  Axiomatics Language for Authorization
API   Application Programming Interface
CSP   Content Security Policy
CSS   Cascading Style Sheets
DAC   Discretionary Access Control
DOM   Document Object Model
HMAC  keyed-Hash Message Authentication Code
HTML  HyperText Markup Language
HTTP  HyperText Transfer Protocol
HTTPS HyperText Transfer Protocol Secure
Hyperty Hyperlinked Entity
IOT   Internet of Things
JVM   Java Virtual Machine
JSON  JavaScript Object Notation
MAC   Mandatory Access Control
OASIS Organization for the Advancement of Structured Information Standards
PAP   Policy Administration Point
PDP   Policy Decision Point
PIP   Policy Information Point
P2P   Peer-To-Peer
QoS   Quality of Service
RBAC  Role-Based Access Control
Runtime UA Runtime User Agent
SELinux Security-Enhanced Linux
SOP   Same-Origin Policy
URI   Uniform Resource Identifier
URL  Uniform Resource Locator
URN  Uniform Resource Name
UTF  Unicode Transformation Format
VoIP  Voice Over Internet Protocol
WS-Policy  Web Services Policy
XACML  Extensible Access Control Markup Language
XML  Extensible Markup Language
XSS  Cross Site Scripting
This thesis presents a security policy subsystem developed under the reTHINK project, a European project aimed at improving the interoperability of Internet services. The rapid pace of Internet development allowed the emergence of new communication services to meet user needs. Standardization of such services facilitates the communication between clients of different service providers, but standardization is a complex process and tends to withhold the community of efficiently taking advantage of the Internet innovation potential. Consequently, the absence of standards causes a lack of interoperability, which is especially troublesome in applications with the same core functionality, for instance, messaging applications like Skype and WhatsApp, where both offer instant messaging services, but a WhatsApp user can only send messages within WhatsApp, not to a Skype user.

1.1 Motivation

To overcome the announced interoperability problem, the reTHINK consortium [11] aims to develop a service architecture which enables authenticated and secure communication between applications from different service providers [12].

In reTHINK, the key to enable cross-service interoperability is based on hyperlinked entities (hyperties). Hyperties are reusable JavaScript code that implement the service logic of communication services, for instance, chat and audio. A hyperty works as a dynamically-loaded plugin that allows a user to communicate from any service provider he has subscribed, while guaranteeing interdomain communication by the means of a stub downloaded from the destination domain.

The concept of hyperty is better illustrated in Figure 1.1, where Alice and Bob are two clients of chat applications sending a message to one another. Alice only has a Skype account and Bob only has a WhatsApp account. Nowadays, if Alice wants to send an instant message to Bob, Alice will have to create a WhatsApp account. Using the reTHINK Application, designed to support instant messaging communication between users from different service providers, the new hyperty concept eliminates that burden: the Chat Hyperty in Alice’s device will coordinate the download of the software that contacts WhatsApp, hypothetically allowing her to send the message from her Skype account to Bob’s WhatsApp account. When Alice sends a message to Bob from the reTHINK Application using her Skype identity, it is forwarded to the Chat Hyperty (step 1). The Chat Hyperty, which is responsible for mediating the communication between the two service providers, forwards the message to the WhatsApp server (step 2), which by its turn relays it to the Chat Hyperty in Bob’s device (step 3). The message is then delivered to the reTHINK Application to be displayed (step 4). When Bob sends a response to Alice from the reTHINK Application using his WhatsApp identity, it follows the same sequence as stated before: the
message is forwarded to the Chat Hyperty (step 5), which does the low level interaction with Skype’s specific protocols, and forwards the message to their server (step 6). Finally, the message arrives to the Alice’s Chat Hyperty (step 7), and is then delivered to the reTHINK Application to be displayed (step 8). The communication between the service providers and the Chat Hyperty is done over the reTHINK Platform, designed to support the execution of hyperties like the one just described.

The web architecture proposed by the reTHINK consortium consists of several components, which will be further detailed in Section 2. A particularly important component is the security policy subsystem, which is responsible for supporting the specification and enforcement of policies. Considering the aforementioned example, if for some reason Alice does not want Bob to reach her, the system must have a means for Alice to specify it and a mechanism to enforce it, i.e., block or redirect messages sent to her from Bob. This behavior, among others specified by the user or the service provider, requires a subsystem capable of interpreting and enforcing preferences and rules that manage access to resources.

1.2 Goals and Requirements

The goal of this thesis is the design, implementation, and evaluation of a security policy subsystem for the reTHINK project. The security policies subsystem includes three fundamental components: a policy specification language, an engine to enforce the specified policies, and a graphical user interface to manage them. These components must be designed according to the following requirements:

- **Expressiveness of the policy specification language**, so that the language allows the representation of the concepts it is expected to (from characteristics of a message to user’s events on his personal calendar),

- **Low complexity** on policy specification, so that the learning curve is as smooth as possible for the user. This is accomplished by using intuitive keywords in the policies specification that directly correspond to the functionality they represent, and by providing a friendly user interface for policy management,

- **Efficiency** when evaluating policies, so that the introduced memory and processing time overhead is small,
• Portability, to allow its easy deployment on different environments, namely the browser and the message overlay system.

1.3 Contributions

This thesis presents PoliTHINK, a subsystem for the specification, enforcement and management of policies and makes four important contributions:

• Design of the PoliTHINK subsystem, which comprises several components: a formal language to be used in the specification of policies, an engine for their enforcement and a graphical user interface for their management. The design of the formal language and the engine meets the requirements presented in Section 1.2 as it provides an expressive language composed of a closed vocabulary to be combined in a way that covers all concepts the reTHINK project requires. Policy specification is simplified by the possibility of using a simple graphical interface. Efficiency on the evaluation is attained by the use of asynchronous calls where it is possible to have parallel computing, and portability is attained by eliminating all context-related dependencies of the core engine.

• Implementation of the formal language and the engine in JavaScript, where policies are JavaScript objects that hold the representation of the behavior that is expected from the system. As the engine is independent from the context, it is deployable on both target platforms: the browser and the mobile environment.

• Evaluation of the implemented subsystem using quantitative tests.

• Integration of the subsystem on the reTHINK project in two different environments, the browser and the message overlay system. The integration in both components was possible due to the modular architecture of the subsystem, which allowed the code reuse.

During the development of both the design and implementation phases, the requirements previously defined were met, leading to an efficient solution completely integrated in two components of the reTHINK framework. Also, this work resulted in a contribution to a conference paper [13] for the 19th International ICIN Conference - Innovation in Clouds, Internet and Networks.

1.4 Structure of the Document

The remainder of this thesis is organized as follows: in order to better understand the role of the security policy subsystem intended for reTHINK, Section 2 provides an overview of the reTHINK framework. In Section 3 it can be found a summary of the related work, with particular emphasis on the relevant policy specification languages and enforcement mechanisms. The architecture of the proposed solution is presented in Section 4 where the proposed syntax for security policies and the enforcement...
mechanism design are described. Section 5 provides implementation details of PoliTHINK. The evaluation of the solution can be found in Section 6 and conclusions and considerations for future work in Section 7.
2.1 Usage Scenario

The main goal of the reTHINK consortium is to improve interoperability between services in the Internet of the future. In a world being increasingly flooded by interconnected devices, services such as instant messaging and Voice over Internet Protocol (VoIP) revolutionized the traditional business models of service providers [12]. The shifting in the business model led to the emergence of different services, more and more sophisticated, but in a fragmented fashion, where communication is restricted to the service provider network and migration to another implies the loss of data and contacts.

The left side of Figure 2.1 illustrates an overview on how current services are clustered, from vehicular networks to sensor networks, telephony services or instant messaging solutions. To overcome the interoperability problem between such heterogeneous services, the reTHINK consortium proposes a platform based on today's web technologies for personal devices, sensors and, in general, services, that enable the use of heterogeneous services through a platform responsible for doing the low level interactions between them. This platform will, for instance, make possible to exchange context information between users, vehicles and sensors in a seamless manner, taking further advantage of the rapidly evolving Internet of Things (IoT).

The right side of Figure 2.1 shows how the interoperability between these services takes advantage of the potential of the already deployed and well accepted services. Interoperation between these services enables the emergence of applications that take great advantage of these services potential. Use cases A, B and C, on the right side of Figure 2.1 are three examples of how reTHINK adds value to today’s services by making them interoperable.

- **Use case A**: Alice, Bob and Carol met on a conference and they need to gather later on that day. Each of them uses a different audio service - Alice has no smartphone, so she is using Skype on her laptop, Bob only has WhatsApp on his smartphone, and Carol only has a VoIP phone in her hotel room. In order for Alice to inform Bob and Carol about the time and place of the gathering,
Figure 2.1: reTHINK impact on today's services

Currently she had no way of doing it as she has no telephone number to create a WhatsApp account, nor has a VoIP application to call Carol. Using the reTHINK platform, Alice can use her Skype identity to contact Bob's WhatsApp identity and Carol's telephone number, making it possible that they can agree on a time and place for the gathering.

- **Use case B**: Dave subscribed a service that consists on receiving notifications from a sensor placed in a bus stop. This sensor detects when each bus arrives and sends notifications to users that subscribe the service. Dave subscribed with his VoIP telephone to receive notifications about his mom's bus, so he is notified via call when she arrives to the bus stop and starts cooking dinner to have it ready by the time his mom gets home.

- **Use case C**: the Smith family had a motion sensor in their house that automatically calls Eve's VoIP telephone when the alarm is triggered. Their house was robbed because the telephone ran out of battery while she was working, giving the thieves time to steal their valuable belongings. They upgraded to a reTHINK motion sensor that contacts any device they choose. Now, the motion sensor not only calls Eve, but also triggers nonstop notifications on her husband's WhatsApp account, where he is online at all times.

In order to make possible the interoperability between such heterogeneous services, reTHINK aims to create a trusted worldwide cooperative model for service delivery. A description of the platform that makes this possible will be presented next.

### 2.2 The reTHINK Framework

To allow for interoperability between different service providers, reTHINK relies on two new concepts: the **Hyperlinked Entity** and **Protocol-on-the-fly** [13]. Both these concepts represent software components that are loaded in a dynamic and transparent way for the user device, and their role is to mediate access to existing services. Hyperlinked Entities (hyperties) are pieces of JavaScript code maintained by service providers to be instantiated in end user's devices. Since different service providers may use different communication protocols, in order to enable communication between them there are
adaptations to be made. This is where the Protocol-on-the-fly is fundamental to resolve the interoperability issue: it dynamically selects and loads the implementation of the protocol stack of the destination service, which is then used to reach that service. It has a fundamental role on solving the interoperability among different services, as it promotes loosely coupled service architectures and at the same time minimizes standardization efforts [2].

The reTHINK project is built around hyperties, as they are the building block of the framework for the previously announced reasons. There are three major reTHINK components that support their execution and management: the Directory Services, the Messaging Node and the Core Runtime. Directory Services are composed of the Catalogue and the Domain Registry, and are used to list the hyperties available for instantiation. The Directory Services are co-located with the Messaging Node, the reTHINK messaging overlay service to enable communication among hyperty instances running in different user devices.

To clarify these concepts, consider the following example, where Alice is using Skype as her service provider, and Bob is using WhatsApp. In Figure 2.2, Bob is trying to start a call with Alice. To achieve this, both needs to have an instance of a hyperty that provides a call service, available on the Catalogue of their service provider. The figure illustrates Alice's browser, Bob's browser, and Alice's service provider, Skype. In order to simplify the figure, it is only presented Alice's hyperty instantiation, which she does by contacting the Catalogue of her service provider (step 1), but the same process occurred for Bob by contacting WhatsApp. Then, a unique hyperty URL is generated by the Messaging Node, and that URL is registered along with the user identity in the Domain Registry (step 2), which enables Bob to be able to discover her by asking the Domain Registry which hyperty instances Alice has available (step 3). Having the hyperty instance address, Bob can publish a message through the Core Runtime to that address, which will be relayed by the Messaging Node (step 4) to Alice. The Core Runtime is comprised
by a set of subcomponents responsible for the management of the hyperty instance execution in the user device, as it was introduced with this simple example. At the client side, both illustrated runtimes execute in the browser environment, in separate sandboxes. At the service provider side, the presented components are exposed at different ports of the same machine. Next, we describe the details of the two components where security policies are enforced in the reTHINK framework: the Core Runtime and the Messaging Node.

### 2.2.1 Core Runtime

The Core Runtime is a JavaScript middleware that runs in the browser and provides a secure runtime environment for hyperty and protocol stub code. The Core Runtime is also responsible for enabling local reTHINK applications to access the functionality delivered by co-located hyperties.

The Core Runtime is composed of a set of subcomponents, of which the most relevant for this overview are the Message Bus, Identity Module, the Policy Engine, the Runtime Registry and the Synch Manager, illustrated in Figure 2.3. The Runtime User Agent (Runtime UA) does the installation and management of these subcomponents. The Message Bus of the Core Runtime supports the communication attempt between the two hyperty instances illustrated in Figure 2.2, working as a pipe for messages exchanged between the service providers and the hyperty instance. The Identity Provider supplies the identity of the user that triggers the deployment of a hyperty instance in the Core Runtime, which is then stored in the Identity Module. For every message sent by a hyperty instance, the Identity Module is queried about which identity owns the hyperty instance, and this information is sent in the message to the destination so that the recipient can identify the sender. The Runtime Registry stores other relevant information about the hyperty instance and the Core Runtime subcomponents, and is available to be consulted by the remaining subcomponents. Communication between hyperty instances is done through the synchronization of objects distributed in the several runtimes. The Sync Manager subcomponent is responsible for creating those objects and for managing subscriptions that allow other hyperty instances to read it. The protocol used to achieve this is further described in Section 2.3. Even though
hyperties are controlled and coordinated by the user’s runtime, we need to ensure that they behave according to the service provider specifications. Their correct behavior must be ensured by the Policy Engine, as a subcomponent that forwards or drops messages according to an authorization decision obtained from policies evaluation. Policies defined by the service provider are added to the Policy Engine by the Runtime UA when a hyperty is instantiated in the user’s device. There is also the possibility to have user defined policies that concern user privacy and reachability preferences. The design and implementation of the Policy Engine is the focus of this thesis, and will be one of the components of PoliTHINK.

To enable a hyperty instance to contact another hyperty instance in a different device, we need some way of relaying the message to it. As illustrated in Figure 2.2, this is one of the responsibilities of the service provider, which is performed by Skype in the given example. Next, we explain in more detail how it works and why it is also relevant to the enforcement of security policies.

### 2.2.2 Messaging Node

The Messaging Node is a component of the reTHINK project that enables communication between hyperty instances running in different devices. It is composed of a set of subcomponents, of which the most relevant for this overview are the Message Bus, the Address Allocation Manager, the Policy Engine and the Subscription Manager, illustrated in Figure 2.4. Similarly to the Message Bus from the Core Runtime, the Message Bus of the Messaging Node works as a pipe for messages between hyperty instances and for messages used in their configuration. For instance, in Section 2.2 it has been introduced that the deployment of a hyperty instance implies the generation of a URL in the Message Node. The Message Bus is the subcomponent responsible for forwarding this generation request to the Address Allocation Manager, which creates a unique URL that identifies the hyperty instance in the reTHINK framework. For a hyperty instance to receive messages that are destined to it, the Subscription Manager adds and removes listeners for that address. The listeners intercept them and they are forwarded to the corresponding hyperty instance after validation in the Policy Engine, where service provider policies are enforced.

The Policy Engine in the Messaging Node is a fundamental component for the proper functioning of the framework for three main reasons: it ensures the enforcement of policies that may restrict the service provided to the end user, and allows the enforcement of Quality of Service (QoS) mechanisms and charging policies. Since a user has access to the policies stored in his device, a malicious user...
would be able to delete them and therefore occur in situations that are prohibited by the service provider. For instance, if a service provider detects that there is a new domain being used for the wrong reasons, it is not enough to install a policy that blocks the evil domain in all end user devices: users could simply delete that policy and they would be able to use the domain again to communicate. Adding to this, the enforcement of QoS and charging policies have to be enforced in the service provider, where the communication flows can be shaped and the charging is done according to what is contracted - which is not possible to control at the end user device. The Messaging Node's Policy Engine is one of the Policy Engines developed for PoliTHINK, along with the Core Runtime's Policy Engine described in Section 2.2.1.

2.3 Data Flow in reTHINK

In order to be able to design the subsystem for the enforcement of security policies, it is necessary to understand in depth the data flow in reTHINK.

Each hyperty instance uses data objects, software pieces that enable the execution of the implemented service. These objects can be created, changed and deleted by the hyperty instance, and may be subscribed and unsubscribed by other hyperty instances. To prevent concurrent changes to cause inconsistencies in the objects (or data objects), the reTHINK consortium proposes a synchronization model, the Reporter-Observer model [14]. This model grants write access only to the object creator (the Reporter Hyperty), and read-only access to other hyperties that successfully subscribe it (the Observer Hyperties). The correct understanding of this model is fundamental for policies enforcement, as we need to identify the characteristics of the messages to intercept and validate them. The enforcement of policies guarantees that data object creation, subscription and update are valid both from the point of view of the user and service provider. Next, we clarify the data flow in each of these three operations.

2.3.1 Data Objects Creation

Policies enforcement in the creation of data objects allows the filtering of the entities that have access to the framework or to other users, making it possible to block communication from untrusted domains or when a user does not want to be contacted, for instance.

More than a window to restrict access control, the message sequence presented in Figure 2.5 registers the data object and the associated information, including the reporter hyperty, in the Runtime Registry. This is a required step in order to enable the validation of a data object update, as will be discussed ahead.

The creation of a data object is triggered by a hyperty through a message of the create type, as presented in the message sequence of Figure 2.5. When this message is received by the Sync Manager (step 1), it sends a request to the Message Node to assign to the new data object (step 2). Then, the Sync Manager creates a Reporter Data Object locally and subscribes it by requesting the Message Node to add a listener to intercept events concerning the newly created data object (step 3). The Message
Node then registers the data object in the Domain Registry, where it can be discovered and subscribed by other hyperties.

To guarantee that the enforcement of policies is done at the appropriate time, i.e., before the creation of the data object, the Policy Engine verifies if the intercepted message is of the create type and has the Sync Manager as destination. Those two verifications are mandatory since they reflect univocally the creation of a data object.

### 2.3.2 Data Objects Subscription

The enforcement of policies in the subscription phase allows not only the verification of the user preferences, similarly to the creation phase, but also allows the specification of policies to restrict subscription preferences. By default, all hyperties are allowed to subscribe a data object, but this behavior may be adapted to what the user needs. Adding to the default behavior, it is possible to specify that no hyperties are allowed to subscribe a given data object or that only previously authorized hyperties can subscribe. For instance, a user may create a chat room, which in the reTHINK framework is a data object, and only want to allow his closest friends to join it. Or maybe this user wants to use the chat room hyperty to keep his personal diary and disallow any other user to join and see his logs. The specification of the policies that allow this will be further discussed in Section 5.

Hyperties may request read permission to data objects through a message of the subscribe type. As shown in Figure 2.6 when the Sync Manager receives this message, it request authorization for the subscription to the corresponding reporter hyperty (step 1), where policies are applied. If the request is accepted, the Sync Manager requests the addition of a listener for the data object in the Message Node (step 2) and the subscriber becomes an observer hyperty.

To guarantee that the subscription to a data object follows the data object reporter’s policies, the Policy Engine verifies if the intercepted message is of the subscribe type and has a remote Sync Manager as source. These two characteristics are the minimum necessary to guarantee that the intercepted message refers to a subscription to a data object.
2.3.3 Data Objects Update

For the update of a data object, adding to the enforcement of applicable policies, it is necessary to ensure that the change was issued by the data object's reporter hyperty. This is done by requesting the Runtime Registry about which is the reporter hyperty of the data object being updated, and checking if that was the hyperty that issued the update.

When the state of a data object is changed, for instance, after a new subscriber joins the data object, the change is advertised to all observer hyperties following the sequence shown in Figure 2.7. The advertisement is triggered by the Sync Manager when a change to a data object is detected (step 1). The Sync Manager then sends a message of the update type to the Message Node (step 2), where it is broadcasted to the observer hyperties.

To guarantee that the update of a data object was issued by its reporter, the Policy Engine verifies if the intercepted message is of the update type, and if the message.body.source field corresponds to
the data object reporter of the *message.from* field. This information is stored in the Runtime Registry when the data object was created, and is retrieved to the Policy Engine when these two characteristics are met, guaranteeing that the update indeed comes from the data object’s reporter.

### 2.4 Summary

This chapter provided an overview on the reTHINK environment, with particular emphasis on the Core Runtime and the Messaging Node. Since both are frequently mentioned throughout this thesis, it is important to have the correct understanding about their role in the reTHINK architecture and this thesis. For a better visualization of the framework applicability, we presented use cases A, B and C, which greatly benefit from the singular features offered by reTHINK. Also, we described the characteristics of the data flow, necessary for the interception of key messages where policies must be enforced.

The presentation of the reTHINK operation helped to clarify the importance of a system to specify and enforce security policies. To assist on the design and implementation of such system, we carried out a research on policy specification languages and policy enforcement mechanisms, which will be presented in the next chapter.
This chapter presents the related work on the specification and enforcement of security policies. This topic is related with the goal of this thesis, which is the design and development of PoliTHINK, a system for the specification and enforcement of security policies. In order to get some foundations about the subject, a survey on the existing related work was carried out and presented in this chapter.

This chapter is organized as follows: Section 3.1 presents the theoretical models that provide the base for the implementation of access control mechanisms. The correct understanding of those models is important for the design of the security policies’ module. Thereafter, in Section 3.2 an overview of existing security policy mechanisms for the target platforms of reTHINK: the browser and the mobile environment. Finally, Section 3.3 and Section 3.4 present the most relevant policy languages and enforcement mechanisms, respectively, and their suitability within the reTHINK project will be discussed.

3.1 Access Control Models

Security policies can be defined as rules that determine how authorization decisions are carried out by permitting only authorized entities (subjects) to access target resources (objects) [15]. Starting with Lampson’s access matrices in 1971, a number of access models was proposed to deal with authorization decisions on a system. Some of those are widely used: Discretionary Access Control (DAC), Mandatory Access Control (MAC), Role-Based Access Control (RBAC) and, more recently, Attribute-Based Access Control (ABAC).

3.1.1 Discretionary Access Control (DAC)

In DAC, policies specify explicit access modes (read, write and execute) a subject is allowed to perform on an object [16]. Incoming requests are checked against those policies and if there is an authorization allowing the subject to access the object in that specific mode, the access is allowed; otherwise, it is denied. For instance, this class of policies is implemented by nearly all operating systems in terms of file permissions - a directory listing in Linux shows which of the three access modes that the owner, the group and the world can perform on each file. The privileges associated with an object are controlled by its owner, but he has no control on the copies that other subjects may do. This leads to a major drawback: it is possible to access the information that object carries after it has been copied to another file, even if the owner of the original object does not grant access to it.
3.1.2 Mandatory Access Control (MAC)

MAC deals with DAC’s drawback by centralizing the control of the policies in the policy administrator instead of the object owner [17]. A security class is assigned to each object, and each subject is attached to a clearance; a subject can access an object if some relationship is satisfied between the security levels associated with both subject and object. The Bell-Lapadula model is an example of a MAC model developed to formalize the multilevel security in the United States Department of Defense. The system levels go from ‘Unclassified’ to ‘Top Secret’ and is characterized by the phrase ‘no read up, no write down’ implemented by the following two mandatory policies: a subject can read an object if the clearance of the subject is equal or higher than that of the object (guaranteeing that a subject with a given clearance cannot read objects with a higher security level) and a subject can write an object only if the security level of the subject is equal or lower than that of the object (guaranteeing that a subject with a given clearance cannot be led by malicious software to write to lower security levels where attackers could access the information).

3.1.3 Role-Based Access Control (RBAC)

RBAC is a higher level access control model which groups authorization policies related to a position rather than to individuals [18]. For example, in a given company where the developers are allowed to access Java files, instead of creating one policy per developer stating the same permission, using this model only requires one policy stating that developers can access Java files, and every employee that is assigned to the developer role is subject to this policy. This model also offers reuse of the specifications by allowing inheritance between roles. By decoupling the user and the policy through the role abstraction, this access control model is especially adequate for organizations where many users are driven by the same policies and the quick revocation and granting of privileges is important.

3.1.4 Attribute-Based Access Control (ABAC)

ABAC is an access control model that evolved from RBAC to consider additional attributes. Authorization is granted by evaluating attributes related to the subject, object or environment against policies that allow actions given a set of satisfied conditions [19]. ABAC has the means to cover the features from the previously introduced models: for instance, to get the same functionality as DAC, ABAC may access the ‘identity’ attribute; for MAC, access the ‘security class’ attribute and for RBAC, access the ‘role’ attribute. Languages for specifying policies based on attributes’ value and on attributes’ relationship provide more flexibility and are more expressive. The ABAC model is an interesting approach for the reTHINK framework, since reTHINK is a framework that consists of many entities sometimes managed by several security domains, e.g., a service provider may impose routing policies or policies that block untrusted domains, whereas a user may impose privacy policies or access control policies.
3.2 Security Policies in Browser and Mobile Platforms

There are a number of platforms that implement mechanisms for the enforcement of security policies, e.g., operative systems, web applications, servers, or file systems. In this section, we focus on the existing mechanisms of the platforms where reTHINK will be executed: the browser and the mobile environment. A brief overview of security policies architecture in two platforms targeted in the reTHINK project - the browser and the mobile environment - is presented in this section.

Both the browser and the mobile environment share a general high-level authorization architecture. In this architecture, the authorization process is done in two main points, the decision point and the enforcement point, illustrated in Figure 3.1. An incoming access request is received (step 1) and forwarded to the decision point for authorization (step 2). The policies associated with the access request are loaded from a policy repository (steps 3 and 4) and an authorization decision is made based on them - for instance, allowing the request to access the object or block it. The authorization decision is communicated to the enforcement point (step 5), that concretely carries it by forwarding the request to the object (step 6a) or by returning an error to the requester (step 6b).

3.2.1 Policies in the Browser Environment

The web browser is a client-side application used to retrieve and present information to the user from multiple sources of the web. Such information normally consists of HTML pages, media files and JavaScript code transported over HTTP [20] or HTTPS [21]. Taking into account the general authorization architecture presented before, the web pages are the subjects, and the information that composes them are the objects in the authorization system.

The ability that web pages have to combine HTML and JavaScript code from different servers makes them very powerful, hence the importance of implementing the security policies module in JavaScript. JavaScript allows the maintenance of compatibility with the web's typical languages (JavaScript, HTML, XML, etc.). Even though JavaScript makes webpages very powerful, including untrusted JavaScript code exposes them to attacks. Two of the most exploited browser security vulnerabilities that malicious users can use to damage others are malicious JavaScript code and cross site scripting (XSS). A typical example of XSS is when a user is tricked into clicking on a link that appears to point to a trusted site
but instead leads to a malicious host. Several defense mechanisms emerged from the need to secure the web, and among the most commonly used are sandboxing, same-origin policy, and signed scripting policy driven mechanisms [22].

Sandboxing consists of isolating programs, preventing them from damaging or snooping outside of its environment by restricting the resources the program can access. One of the several sandbox implementations available is JSand [8], which consists of placing wrappers around security-sensitive objects. The wrappers intercept all attempted operations on the object and grant permission or not according to the defined security policies.

The same-origin policy (SOP) restricts JavaScript code to only interact with resources that belong to the same origin, (origin being a \langle protocol, domain name, port \rangle). Under this policy, a browser allows that scripts contained in web page 1 can only access web page 2 if both pages have the same origin, despite the origin of the JavaScript code.

The signed script policy is an alternative to the SOP that gives more functionality to the scripts by taking into account their signature. When the browser downloads a digitally signed script, the browser verifies if the signature is valid. If the script’s signature is valid, the script can ask for extended permission to access specific targets and a prompt appears to ask for the user’s permission [23].

As an example of a security architecture, Figure 3.2 illustrates a simplified version of the Mozilla’s security model. As shown, the execution environment of web content can only access the Web API after checking with the Permission Manager if the access attempt is valid. In reTHINK, there will be a runtime similar to the ones offered by browsers, but that will implement security policies specific to the reTHINK project.

### 3.2.2 Policies in the Mobile Environment

Along with the browser, the mobile platform is currently very popular and a primary target for the reTHINK project. The increasing amount of sensitive information a user may have stored or processed on his mobile device leads to a greater concern on access control and information flow control.

Google’s Android, the most popular smartphone operating system, deals with this responsibility
by having applications running in isolated sandboxes and guaranteeing privilege separation between applications through a permission system [24]. Permissions are granted by the user to the application (which is the subject in the authorization architecture) at install-time and are enforced at two different points: when accessing the file system and at the system services or applications.

Android implements DAC mechanisms inherited from the traditional Linux, along with specific MAC mechanisms implemented at middleware and kernel levels, which are responsible for implementing the authorization model when applications access to resources [24]. Android's latest versions also include SELinux's MAC model. A simplified version of Android's application security model is illustrated in Figure 3.3, which shows the positioning of the permission check for the protection of three types of sensitive information: personal information (for instance, calls log), sensitive input devices (for instance, GPS) and device metadata (for instance, the phone number).

Similarly to Android, the reTHINK project will implement the policy enforcement mechanism at the middleware level, allowing to control application access attempts to sensitive information.

### 3.3 Policy Specification Languages

To represent policies, a specification language is normally used. A policy specification language is the means of expressing the intended behavior of a system in a way that can be interpreted by a computer. To specify policies in reTHINK, it will be necessary to define a specification language. However, before starting to design a language from scratch, we will study the existing languages and which of those may be adopted in the reTHINK project.

There are several important aspects to take into consideration when assessing a policy specification language. First, the language must be flexible enough to adapt to the various security domains that need to be taken into account. Second, the specification complexity of a policy must be reduced, so the burden of learning the language is not too cumbersome for the user to specify. Third, its expressiveness is important so that it is possible to specify the expected behavior from the system. Fourth, the ability to detect conflicts between policies avoids that authorization decisions go against the administrator's intended behavior. Finally, extensibility points are also an important feature by providing the possibility of adding unforeseen functionality.
The reTHINK project requires that every aspect is taken into account, but with special attention to expressiveness. It is a fundamental requirement of the project that the Policy Engine performs well so it does not add too much overhead to the information flow, and an expressive language means the policy specification occupies less memory. Furthermore, it is necessary to ensure compatibility with today’s web standards (JavaScript, HTML, XML, etc.).

Next, policy specification languages that were found relevant due to their acceptability by the community and consequent wide deployment will be described in detail.

### 3.3.1 Content Security Policy (CSP)

Content Security Policy [5] is a security layer designed to mitigate attacks like cross site scripting. It allows web applications to restrict the location and type of resources that can be loaded into the browser.

A client’s browser can activate CSP by providing the X-Content-SecurityPolicy HTTP header in an HTTP response and the policy in the contents of the header, which specifies directives that state where the content may be loaded from - CSP directives apply to fonts, iframes, images, media, objects, scripts and style sheets. Furthermore, activating CSP in a website results in two base restrictions: inline scripts are disabled and strings may not become code (from the eval() function, for instance). The example in Figure 3.4 is a CSP directive which specifies that a website allows all content from its own domain, images from anywhere, plug-ins from media1.com, media2.com and all the subdomains of a content distribution network, and scripts only from trustedscripts.example.com.

![Figure 3.4: CSP example (from [5])](image)

It is possible to specify the sources by scheme, hostname, URI or subdomain, and also four key-words are allowed in the source list: none to represent that no source is allowed, self for allowing the current origin, unsafe-inline to allow inline JavaScript and CSS and unsafe-eval to allow mechanisms like eval() [25].

Even though CSP has the advantage of being a low complexity language, its expressiveness is insufficient for the reTHINK project needs. For instance, it is not possible to specify a policy that blocks messages from Bob in a chat application.

### 3.3.2 WS-Policy

WS-Policy is a framework to specify capabilities and requisites of web service endpoints. The web service provider exposes a policy specifying the conditions under which a given service is provided and the requester checks it against his capabilities through a policy intersection mechanism [6].

The WS-Policy language model comprises three main components: policy assertion, policy alternative and policy. A policy assertion identifies a capability or a requisite of a subject, a policy alternative is
a set of policy assertions and a policy is a set of policy alternatives. Policy assertions are combined by one of the following operators: \texttt{wsp:All} requires all policy assertions to be satisfied, \texttt{wsp:ExactlyOne} requires one policy assertion to be satisfied and \texttt{wsp:OneOrMore} requires at least one policy assertion to be satisfied \cite{26}. When more than one alternative is applicable, the attribute \texttt{wsp:preference} establishes an order between them by using a higher integer to represent higher preference.

WS-SecurityPolicy is a specification of WS-Policy for the definition of security properties for web services. The assertions can be used to define security constraints and requirements, for instance, parts of the message to secure, security mechanism to provide the security or token types to provide additional claims. The example policy in Figure \ref{fig:ws-security-ex} states that the requester must satisfy exactly one of the two alternatives; if the first one is selected, the message body must be signed, if the second one is selected, the message body must be encrypted.

Even though WS-SecurityPolicy is a powerful tool that allows automatic configuration of clients of a service, this language is overkill to preserve confidentiality and integrity, especially when the communication is to be done by clients and the service does all the processing directly rather than through other servers \cite{27}. Similarly to CSP, WS-Policy has the advantage of being flexible on combining policies, but has client-server negotiation focus - finding a configuration that suites both - rather than a fine grained permission description. As a significant disadvantage, its closed vocabulary does not have the expressiveness the reTHINK project demands.

\subsection{3.3.3 XACML Language}

Unlike CSP or WS-Policy, XACML is a very expressive language. It is a standard that describes a policy language used to express general access control constraints through an extension language of XML. It is used to define requirements that must be matched in order to give access to a resource.

The policy in Figure \ref{fig:xacml-ex} specifies two rules: the first one, from line 17 to 33, specifies that any message from a subject that belongs to the blacklist will be unauthorized. The second rule, from line 34 to 59, specifies that any message regarding a chat communication with Alice as destination will be authorized. Both rules will be verified when a given message is used to reach a user (lines 4 to 16). In order to improve the example readability, the URNs that identify the XML namespaces were omitted.

The main components of the model are the rule, the policy and the policy set. A rule is the elementary unit of a policy and can be evaluated on the basis of its contents. The main components of

\begin{figure}[h]
\centering
\begin{verbatim}
<wsp:Policy>
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SignedParts>
        <sp:Body/>
      </sp:SignedParts>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
\end{verbatim}
\caption{WS-SecurityPolicy example (from \cite{6})}
\end{figure}
a rule are a target, a condition, an effect, obligation expressions and advice expressions. The element
<Condition> may further refine the applicability established by the target and the <Effect> element
indicates the rule intended consequence of a True evaluation for the rule, which can be Permit or Deny.
When a rule containing obligation expressions or advice expressions is evaluated, those expressions
are taken into account when enforcing the policy, usually resulting in logging messages or other form
of notification - the difference between the two is that advice expressions can be safely ignored. For
instance, the example from Figure 3.6 could have an obligation expression associated, for instance, to
return a message to the requester saying the user he tried to reach is not available.

A policy comprises a target, a rule-combining-algorithm, a set of rules, obligation expressions and
advice expressions. The target may be declared or it may be calculated from the
elements of the rules it contains. The rule-combining-algorithm specifies the procedure by which the results of
evaluating the rules are combined to obtain an authorization decision.

The applicability of a policy depends on a <Target> element: if the target matches the request,
the evaluation of the policy is carried out. The evaluation results in one out of four possibilities: Permit,
Deny, Indeterminate or NotApplicable.
A policy set comprises a target, a policy-combining-algorithm, a set of `<Policy>` or `<PolicySet>` elements, obligation expressions and advice expressions.

The XACML standard defines a number of language primitives to obtain an authorization decision from individual results. The existing combining algorithms may be extended by the creation of new ones with different semantics. The main standard combining algorithms are 

- **deny-overrides** - if any decision is `Deny`, the result is `Deny`,
- **permit-overrides** - if any decision is `Permit`, the result is `Permit`,
- **first-applicable** - the result is given by the evaluation of the first applicable rule or policy in the list, and
- **only-one-applicable** - if no policy or policy set applies, the result is `NotApplicable`; if there are more than one policy or policy set applicable, the result is `Indeterminate`; if there is exactly one policy or policy set applicable, the result is given by its evaluation. This algorithm only applies to policies, not rules.

When the need to deal with more than one subject arises, different attribute categories are used to differentiate between them - some standard values are specified (subject, resource, action and environment), and users may define new ones.

Also, because it is common to base an authorization decision on some characteristic of the subject or the resource, XACML supports the identification of their attributes in the `<AttributeDesignator>` element. For the situations where policies are based on the information resource, the element may identify data in the information resource to be used in policy evaluations. In order to obtain an authorization decision, attributes of many types may have to be processed. XACML provides a number of built-in functions and a method of adding new ones through the `<Apply>` element and its `FunctionId` attribute that identifies the function to be applied. XACML is a generic language that provides good coverage in not only access control to a given type of resource, but to any environment. As a language to be used in distributed systems, it allows a policy to be written taking into account policies in other locations. This feature allows policy administrators to manage sub-policies separately, while XACML deals with combining the various results in a final decision.

Regarding extensibility, there are a set of XML attributes that have values that are URIs, and consequently can be extended by the creation of new URIs associated with new semantics: `Category`, `AttributeId`, `DataType`, `FunctionId`, `ObligationId`, `AdviceId`, `PolicyCombiningAlgId`, `RuleCombiningAlgId`, `StatusCode`, and `SubjectCategory`. Adding to that, the combining algorithms are used to build policies as complex as needed, making XACML a very flexible policy specification language.

Even though XACML is a very flexible language, it comes at the cost of complex specification and verbosity: its use requires an extensive knowledge about the general concepts and policy authoring requires a steep learning curve. In order to mitigate this, a language for a high-level description of XACML policies was developed, ALFA [28], used to build the example from Figure 3.6. ALFA allows the specification of XACML policies in a compact and easier to read way. After the specification in ALFA, an XACML policy is automatically generated, but even though the specification complexity has been mitigated, the verbosity remains as a disadvantage and the corresponding performance overhead it implies is expected to be significant taking into account the requirements stated.
Contrary to XACML, Ponder allows the specification of policies in a very concise way. Ponder is a declarative, object-oriented language for specifying security and management policies for distributed systems [29]. The language includes domains to group the objects to which policies apply, which is useful to meet the needs of human managers. Also, an object or sub-domain may be a member of multiple parent domains, which allows the objects to be added and removed from the groups without having to change the policies associated to them.

This policy specification language provides five types of policies: authorization policies, information filtering policies, delegation policies, refrain policies and obligation policies. Authorization policies define what activities a member of the subject domain can perform on the set of objects in the target domain. Information filtering policies are positive authorization policies that include filters to transform input or output parameters associated with their actions, based on attributes of the subject, target or system parameters. Delegation policies specify an authorization to delegate to others to perform actions on their behalf, namely servers or third-parties. Refrain policies act as restraints on the actions that subjects are permitted to perform and obligation policies are event-condition-action rules that define which operations need to be performed when a certain event occurs.

The composite policies of Ponder facilitate the task of policy administrators by providing the ability to group policies and structure them to reflect the organizational structure and by reusing common definitions. Ponder allows specialization of policy types through the mechanism of inheritance; role and organizational hierarchies can be specified using specialization.

Adding to that, the language allows the declaration of sets of policies as groups, having roles, relationships and management structures as special cases of a group - roles provide a semantic grouping with a common subject, relationships define the rights and duties of roles towards each other and management structures may contain roles, relationships and other management structures.

As previously stated, Ponder allows a concise specification of policies. For instance, the policy previously specified in XACML (Figure 3.6), when specified in Ponder entails a lot less verbosity, as shown in Figure 3.7. The example reflects a situation to be catered for in the reTHINK framework: the first policy states that a given message will not be authorized if the source is in the blacklist; the second one states that a given message will be authorized if it is related to a communication via chat.

It is also possible to specify meta-policies, which are policies about which policies can coexist in the system. For instance, they can be used to check all pairs of policies in its scope for possible conflicts or
what are permitted attribute values for a policy to be a valid policy. Ponder policies previously described can be visualized as base classes forming an inheritance hierarchy, as shown in Figure 3.8. Extending the language to cater for new kinds of policies is simplified using an underlying object-oriented implementation - it can be extended by adding new base sub-classes to the existing ones, or by adding new attributes to existing base classes. Adding to that, the language supports the use of scripts (externally-defined code object that can be imported into a policy specification) to cover more complex sets of actions that Ponder may not account for, making it also very flexible. Ponder provides reuse by supporting the definition of policy types to which any policy element can be passed as formal parameter. It is also possible to create event definitions to take into account more complex event sequences, which contributes to the flexibility of the language. Regarding specification complexity, the fact that Ponder is a declarative language and that actions, conditions or obligations are methods of the target objects, makes the specification of policies an easy task. Also, the use of roles and relationships reflects the organization structure of the system, which facilitates the logic behind writing the policies.

The analysis on the Ponder policy specification language reveals that the characteristics of the language make it a good fit for the reTHINK framework. On the other hand, Ponder is implemented in Java, which means it is not compatible with the web standards. Furthermore, the language is no longer supported, so the Ponder implementation is not available.

### 3.3.5 Discussion

Up until this point, a survey on the most relevant policy specification languages was made, namely CSP, WS-Policy, XACML and Ponder. A table summarizing their expressiveness, specification complexity, verbosity and web compatibility - the policy specification language requirements for the subsystem to be developed - is presented in Table 3.1.

CSP was shown to be a web-compatible and concise policy specification language, but its expressiveness is not enough for the reTHINK project needs. CSP only allows filtering sources of web page components (e.g. images and scripts), and the reTHINK framework requires a finer granularity to deal with the user’s context and preferences.
WS-SecurityPolicy provides a more expressive syntax than CSP, but as a language designed to establish security constraints and requirements, it does not allow the management of the previously mentioned framework’s resources.

Contrary to both CSP and WS-SecurityPolicy, XACML is a very expressive language. It supports the specification of reTHINK’s policies, but at the cost of a high specification complexity and verbosity. Once the security policy subsystem efficiency is a fundamental requirement to fulfill, XACML’s high verbosity (and consequent high number of bytes to be transferred, loaded and evaluated) makes it an unsuitable language to use in reTHINK. Furthermore, because it is a very general language, the specification of policies is a complex task that requires a specialized policy administrator, which also is not compliant with the reTHINK project, where the end user will have the opportunity to specify his preferences as policies to be enforced by the subsystem.

Similarly to XACML, Ponder is a very expressive language. Its advantage is related to the verbosity: it allows very concise specifications. Along with the low specification complexity, Ponder is shown to be the perfect fit for the reTHINK framework but, unfortunately, its implementation is no longer available.

### 3.4 Policy Enforcement Mechanisms

To enforce security policies over subjects attempting to perform operations on objects, accesses are generally controlled by reference monitors [30]. The implementation of this abstract model has is characterized by properties that make the policy enforcement secure: it is non-bypassable, tamper-proof and evaluable. Next, it will be described the most relevant implementations of reference monitors for web content security.

#### 3.4.1 JSand

As introduced in Section 3.2.1, JSand is a sandboxing mechanism that operates through wrappers around security-sensitive objects [8]. Figure 3.9 shows how JSand operates when third-party scripts are used. As illustrated, JSand confines each script to a sandbox, isolating it from the remaining components of the browser, namely the JavaScript environment. To control access attempts from the sandboxed script to the JavaScript environment, JSand uses policy enforcing wrappers around the script, which ensures that those scripts do not have unauthorized access to security sensitive information. This is
achieved by using an API that intercepts all accesses on a target object and applies the policy given as parameter to determine if an access attempt is allowed.

3.4.2 ConScript

ConScript [31] was the first general browser-based policy enforcement mechanisms for JavaScript. It is a fine-grained mechanism that allows a web page to specify and enforce policies by allowing the application to constrain the code it executes. The policy related to the execution of a script is stated in the `<SCRIPT>` tag and enables the web page to change the behavior of the code. An example of ConScript use is disabling the `eval` construct, except when de-serializing JSON strings. In this case, the reference `window.eval` points to an advised function, which executes instead of the original `eval`.

The built-in function *around* has the objective to enable a deep approach, i.e., it is able to apply policies to any aliases of the target function. For instance, `window.eval` and `window.parent.eval` will both execute the defined advice, as they are aliases and this enforcement mechanisms follows a deep approach.

Other applications of the ConScript policies are, for instance, disallowing dynamic or inline scripts, access to cookies or dynamic iframe creation, provide whitelist for scripts, limit the number of popups or prevent resource abuse.

3.4.3 WebJail

Similarly to ConScript, WebJail [9] is a security architecture for the browser that forbids the invocation of APIs that contradict the specified security policies through a deep approach. The enhancement WebJail provides over ConScript is that the policies are related to each component of a web mashup in-
WebJail allows a mashup integrator to express policies to be enforced within the browser, which means that it can impose restrictions on the behavior of untrusted components of the mashup. Every policy defines a set of operations that the component is allowed, disallowed or restricted to invoke.

The architecture of WebJail is illustrated in Figure 3.10 consisting of three abstract layers. The policy layer associates the policy with a mashup component, the advice construction layer maps those policies to low-level operations, and the deep weaving layer enables support for the deep approach in the JavaScript engine.

### 3.4.4 XACML Enforcement

More than a policy specification language, XACML also defines a policy enforcement mechanism [10]. The components that comprise are the Policy Enforcement Point (PEP), the Policy Decision Point (PDP), the Policy Administration Point (PAP), the Policy Information Point (PIP), the context handler and the obligations service. The PEP and the PDP correspond to the components presented in the general authorization architecture (Section 3.2), the PAP is the entity that manages the policies, and the PIP is responsible for providing attribute values for the evaluation of policies.

A simplified version of the model is illustrated in Figure 3.11 and operates in the following manner: after an access request is intercepted by the PEP (step 1), it is sent to the context handler that deals with the translation to an XACML request (step 2). The XACML request is forwarded to the PDP (step 3), which loads the applicable policies (step 4) and asks the context handler for the attribute values necessary to make a decision (step 5). When the PDP obtains the necessary information from the PIP, it makes an authorization decision and communicates it to the context handler (step 6). The context
3.4.5 Discussion

So far, a brief overview of the most relevant policy enforcement mechanisms was provided, namely JSand, ConScript, WebJail and XACML. Both ConScript and WebJail are characterized by a deep approach, which demands client-side support in the JavaScript engine. The difference between the two is the granularity at which they operate: ConScript offers enforcement at web page level whereas WebJail offers the possibility of restricting the behavior of each component of the web mashup. JSand allows the website administrator to specify policies and have them associated with script providers at the client browser, whereas XACML allows for a complete decentralization of the creation of policies, the authorization decision and the enforcement of the decision. The security policies enforcement in the reTHINK project will take as reference the XACML model.

3.5 Summary

In this chapter, we present the related work on security policies. First, we visited the most traditional access control models, then we took an overview on access control mechanisms on the reTHINK target platforms - the browser and mobile environments. After, we presented the most relevant policy specification languages and enforcement mechanisms, and finally we evaluated their suitability in the reTHINK project. As none of the presented solutions completely fits the requirements of the system to develop, namely the expressiveness and the low complexity of policy specification, the next chapter will present
PoliTHINK, a solution tailored for the needs of the reTHINK project based on the related work presented in this chapter.
This chapter presents PoliTHINK, a subsystem for the specification and enforcement of policies in the reTHINK framework. Even though policies may be specified by the user or by the service provider, their enforcement is done in both Core Runtime and Messaging Node components. Section 4.1 gives an overview of the proposed solution, Section 4.2 presents the formal language for policies specification, Section 4.3 describes the policy enforcement mechanism, and Section 4.4 the User Interface that was created to facilitate their specification.

4.1 PoliTHINK Overview

Before diving in the details of PoliTHINK, an overview of the policy specification and enforcement points will be provided in this section. Policies enable both the service provider and the end user to specify discovery and reachability preferences while avoiding the system restart. In reTHINK, there are two entities involved in the specification of policies: the user, who uses hyperties to communicate with other users, and the service provider, who provides the hyperties and facilitates the communication signaling between the users. The user should have the opportunity to control his reachability preferences: who can reach him, when he can be reached, and what are the characteristics of the service being used to reach him. This is achieved by enforcing user policies in the Core Runtime of the user’s device. On the other hand, the service provider needs to ensure the correct execution of hyperties in the user’s device. To do so, the service provider needs, for instance, to guarantee that communication attempts from untrusted domains are denied, and to control the subscription configurations of hyperties by allowing or denying them to be subscribed by other hyperties. This is achieved by enforcing service provider policies in both the Core Runtime and the Messaging Node.

Consider Figure 4.1, which illustrates an overview of security policies in reTHINK. Since the service provider is expected to have enough programming knowledge to understand a policy specification language, adding policies to the system can be done by directly using the language for their specification. In the Messaging Node (situation A) this is done by feeding the Policy Engine with a policy file. These policies are independent of user IDs and hyperties, i.e., they are to be applied to all the intercepted messages. If a service provider wants to have policy enforcement for communications associated with a given hyperty in the user endpoint, he sends those policies in the hyperty descriptor (step 1 of situation B) to be enforced in the Core Runtime. When a hyperty is downloaded to a device (step 2), the policies retrieved from the hyperty descriptor are added to the Policy Engine. These policies are to be applied to messages regarding communications where the downloaded hyperty intervenes.

The user can specify his preferences by defining his own policies. Since it is not expected that the
common user understands how to use a policy specification language to set his preferences in reTHINK, a graphical user interface is provided for this purpose. By the means of input fields, dropdowns and buttons, the user states what is the behavior he expects from the system, for instance, reject all incoming calls from 11 p.m. to 8 a.m. (step 1 of situation C). This information is then forwarded to the Policies Manager (step 2), which serves as translator from the provided input to the policy specification language by creating a policy specifying the restriction. The policy is then added to the Policy Engine in the Core Runtime (step 3) to be enforced from then on. Adding to this simple approach, the administration page offers the option to import a policy file, which enables more advanced users to take full advantage of the policy specification language potential.

At any given time, there may be several policies stored in the Policy Engine from the service provider and the user. For each message intercepted in a Policy Engine, there may be zero, one or two policies to be evaluated, depending if the service provider and the user had specified any policy - at most, one service provider policy and one user policy is evaluated. Conflicts between user policies and service provider policies do not occur because the Policy Engine gives priority to the evaluation of service provider policies, so if its evaluation results on a negative authorization decision, the user policies are not evaluated.

The policies that reflect the behavior that both the user and service provider expect from the system must be specified in a formal language that enable their automatic validation. This language is presented in Section 4.2.
4.2 Policy Specification Language

The reTHINK Policy Specification Language is a language tailored to the needs of the reTHINK framework. Policies following this language use a closed vocabulary that gives the ability to set user or service provider configurations in an expressive and flexible way. In Section 4.1, the entities involved on the specification of policies were introduced. To meet their needs, our policy specification language must allow the representation of several operations: it needs to represent if an attribute matches a given parameter, if an attribute is in a set of parameters, and if an attribute is greater or less than a given parameter. In its core, a policy specifies if someone can do a certain action and under which circumstances. To support the validation of various circumstances, a policy must be able to represent various rules and a way to combine the output of those rules’ evaluation. Adding to this, more than specifying the authorization decision when a certain condition is met, it must also be supported to output an authorization decision when a condition is not met, or when more than one condition are mandatorily met.

In reTHINK, as presented in Section 2.3, the data flow is carried through messages exchanged between hyperties and data objects attempting to create, subscribe or update hyperties on behalf of users. To ensure the correct execution of hyperties, these attempts must be validated by applying the service provider’s and user’s policies to obtain an authorization decision. Often, more than consulting who is attempting to do a given action, it is needed to verify other attributes of the system. It is useful to filter action attempts not only because it is a given subject attempting it, but because he is attempting it under circumstances that are not allowed, for instance, accessing an object on a time slot reserved for that object’s maintenance. In short, a policy allows to express when the policy is to be applied, what is the system attribute to be verified, and the decision to make if a given condition is met.

4.2.1 Decision

The most important information that a policy states is an authorization decision, i.e., if a message is to be allowed or rejected. In a reTHINK policy, this information can be found in the decision field. Given several authorization decisions provided by different rules of a policy, the individual authorization decisions are then combined to obtain the final authorization decision for a given message.

4.2.2 Scope

Generally, we do not want to apply the same authorization decision independently of the event characteristics. The reTHINK environment is especially diverse in this matter as it is possible to have a wide variety of hyperties being executed, and several identities being used. While it is useful to specify policies to be applied to all messages, it is also interesting to be able to specify policies to only apply on messages that concern a given hyperty or an identity. To take into account this feature, a reTHINK policy allows the specification of its rules target in the target field. In this field it is possible to find the global string, a user URL, or a hyperty name. The first option, the global string, is used when the rule is to be applied to all messages exchanged between hyperties. The second option, a user URL,
is used when the rule is to be applied to messages where the owner of the source/destination hyperty or data object is the given user URL. The third option, a hyperty name, is used when the rule is to be applied to messages where the name of the source/destination hyperty is the said hyperty name. To prevent misinterpretations on what domain a given target belongs to, the scope field is used. The scope field may be one out of three possible keywords: ‘global’, ‘identity’ and ‘hyperty’, consequently solving eventual ambiguities in the target value.

### 4.2.3 Applicability

Even though the target field already allows to specify to which messages a policy applies, it may be also useful to further restrict the policy applicability by stating under which conditions the policy is to be consulted. In a reTHINK policy, this is done in the condition field, where it is specified the system attribute to verify, the parameter it is limited to, and the operation used to do the verification.

A survey on useful policies in the reTHINK framework was carried, and the contribution of the reTHINK partners revealed that it would be useful if the system allowed the specification of different behaviors according to:

- group membership
- user presence in blacklists
- domain of the user identity
- resources used in the communication attempt
- scheme of the data objects involved
- time sensitive preferences (time of the day, weekday or date)
- previously authorized users that can subscribe his hyperties

Such system attributes are always associated with one parameter that describes or restricts its expected value. Taking into account the system attributes previously presented, this parameter can be a string representing a group name, a user identity domain, a resource type (for instance, chat or video), the time of the day, a weekday or a date. The type of operation to do to verify an attribute value against a given parameter is done through the operation field. This field of the condition states how the evaluation of the policy must deal with the system attribute and its parameter independently of what the system attribute is. For instance, the operation may verify if the system attribute is equal to the parameter, if it is part of a collection, or if its value is greater than or less than the parameter.

### 4.2.4 Combining Rules Results

When a policy is composed of more than one rule, it may happen that the evaluation of the rules does not output the same authorization result and there may be conflicts. In order to solve those conflicts, each policy must state how the Policy Engine must combine the individual results of the rules. For each
message, the Policy Engine evaluates the policy and checks if any of the rules failed the verification. By default, if at least one rule failed, then the message in not authorized and it is discarded - this behavior is denominated as the block-overrides algorithm. On the other hand, there are situations where it is desirable that the authorization decision is the result of other combinations of the individual results of the rules. For instance, if a policy contains a set of rules and the user only needs that one of them is applied to the message, he may choose to have an algorithm that outputs the authorization decision given by the first applicable rule, allowing to shortcut policy evaluation. This approach is denominated as the first-applicable algorithm, and takes advantage of the existence of the priority field, where it is possible to control the order in which the rules are evaluated, being 0 the highest priority. Another option for the combination of the rules’ evaluation result is to require only one rule to authorize the message, and independently of the results of other rules, the message is authorized - this behavior is denominated allow-overrides.

4.2.5 Actions Service

As the survey revealed, it is useful to be able to optionally trigger additional actions. For instance, if a user chooses to have all communications to all his email accounts forwarded to his new email, he can specify that behavior in the actions field. Other interesting applications of this field are, for instance, returning an automatic message or forwarding the message being evaluated to other hyperty. This optional field allows for further expressiveness, opening the specification of policies to broader applications. The execution of the action is independent from the authorization decision, which allows them to be specified independently.

4.2.6 Policy Syntax

In the previous sections, we presented the policy fields as an answer to the increasing demands of the reTHINK project. Figure 4.2 presents the fields of a policy and the syntax of each of them. For user policies, the policy key is an identifier chosen by the user when the policy is created. More than giving a hint to the user about what the created policy concerns, the key is a unique string that prevents the creation of multiple policies with the same identifier. For service provider policies, the key is given by the hyperty name, which identifies the deployed policy at hyperty instantiation time. A policy has a set of rules, where each rule is composed of several fields with enough information to obtain an authorization decision: the scope and target of the rule, the condition under which the rule applies, the decision to represent the authorization decision in case the condition applies, and the priority, a relevant field for the first-applicable combining algorithm used to manage the evaluation order of policies. Each policy also has the actions field, where the policy administrator may represent any additional action to be executed after the policy evaluation, independently of the authorization decision. Lastly, the combining-algorithm field allows the specification of the algorithm that solves possible conflicts in the individual rules’ results.
**4.2.7 Combining Attributes**

Up until this point, it was discussed the specification of policies that only support the verification of one system attribute. This syntax is not enough to specify a policy that expresses an authorization decision based on multiple system attributes, hence the introduction of advanced conditions. This feature is achieved by using the *and*, *or* and *not* logical operators to combine a set of attribute verifications. Figure 4.3 presents the syntax of the condition field used for the specification of an advanced condition using the three logical operators and a combination of them.

(a) condition: ['and', subcondition1, subcondition2]
(b) condition: ['or', subcondition1, subcondition2]
(c) condition: ['not', subcondition]
(d) condition: ['and', ['or', subcondition1, subcondition2], ['not', subcondition3]]

The use of these logical operators allows the language to have more expressive rules. The *not* logical operator allows the policy administrator to specify the authorization decision when the condition is not met, which is a significant improvement on what the language can represent. Assuming the default authorization decision is to authorize a message, consider this example: Alice decides that her smartphone is to be used exclusively to be contacted by her family. In order to accomplish this, she created a user group, and added all her family contacts to it. Now she needs to tell the reTHINK application that she only wants to receive communication attempts if they come from users of the family group. To do so, she will use the graphical user interface of the application to create this policy, which generates the policy shown in Figure 4.4 where the condition is that the message source is not in the family group. If the *not* logical operator was not used, the alternative would be to create a rule to block every existing contact, which is not feasible given the universe of existing contacts.

The *and* logical operator allows the policy administrator to represent a situation where more than
one condition must be met for the policy to be applicable, which also represents an improvement on the expressiveness of the language. Consider the example previously described where Alice had a new smartphone to be exclusively contacted by her family. Alice started participating on a volunteering activity and decided that she also wants to allow her colleagues to reach her. To do so, she will use the PoliTHINK graphical user interface to create a group for the volunteers contacts, and will generate the policy shown in Figure 4.5 where the condition is that the message source is not in the family group and is not in the volunteers group. Without the and logical operator, it would not be possible to achieve this behavior - the universe of possible contacts would still be too big.

The or logical operator enables the representation of rule applicability when any of the subconditions is applicable. Considering Alice’s example: she decides to be even more selective about who can reach her and when, and from now on she will not allow anyone to contact her before 9 a.m., not even her family or her volunteer colleagues. To do so, she will generate the policy shown in Figure 4.6 where the condition is either the source is not in her family and volunteers group, or the time is less than 9 a.m..

Combining several conditions using logical operators greatly improves the expressiveness of the policy specification language, enabling the policy administrator to cover more combinations of real-life situations that the user and service provider may use to specify their preferences in the reTHINK framework.
4.3 Policy Engine

The Policy Engine is the reTHINK component responsible for validating the messages described in Section 2.3. These policies can be specified by the user or by the service provider, and are enforced in two components of the reTHINK framework: the Core Runtime and the Messaging Node. In order to achieve portability of the Policy Engine, one of the requirements of PoliTHINK, the context-specific functionalities are handled in a separate module, making the core of the Policy Engine a reusable module for any context in reTHINK. Since the evaluation of policies on the Core Runtime and the Vertx contexts only depends on the message content and on native JavaScript methods, both contexts can obtain all attributes that are necessary to enforce service provider policies. The main difference between them is on the additional responsibilities that the Policy Engine of Core Runtime has in the reTHINK framework: while the Policy Engine of Messaging Node exclusively validates messages against service provider’s policies, the Policy Engine of Core Runtime has a much more important role in the management of hyperty execution in the endpoints. The Policy Engine of Core Runtime, more than an enforcer of user and service provider policies, also stamps the message with the identity of the sender to allow its identification at the destination, triggers the registration of hyperty subscribers in the Runtime Registry, and triggers the necessary procedures to ensure trustful communication between hyperties. The characteristics of the core Policy Engine and the different contexts where it was implemented will be further detailed in this section.

4.3.1 Policy Engine Architecture

In order to enforce an authorization decision on a message, there are a few steps to perform when the message is intercepted. Consider Figure 4.7 which illustrates the Policy Engine architecture and represents the interactions between its subcomponents through arrows. When a message is posted in the Message Bus, it is intercepted by the Policy Enforcement Point (PEP) before reaching its target component or leaving the runtime (step 1). Then, an authorization decision is requested to the Policy
Figure 4.7: Policy Engine architecture

Decision Point (PDP) (step 2). The PDP generates the authorization decision by loading the existing user and service provider policies from the Policies Repository (step 3), which are verified by consulting the system attributes requested to the Context handler (step 4). The PDP returns the authorization decision generated by the policies evaluation to the PEP (step 5), and then the additional actions are executed (step 6) independently of the authorization decision. Lastly, if the message’s evaluation results in a positive authorization decision, the message is forwarded to its destination; otherwise, the message is discarded. This architecture corresponds to the general Policy Engine, which consists of the components that are common to the different contexts it may be integrated in. The necessary adaptation to those contexts is done through the Context handler, represented in Figure 4.7 as the provider of the system attributes, and will be further detailed next.

4.3.2 Handling Context

As previously introduced, the two environments where the Policy Engine is present are the Core Runtime and the Messaging Node. Even though there are no restrictions to the attributes that both contexts can obtain, the Messaging Node is agnostic to the identity of the users and the hyperty behind the hyperty addresses, which means that policies are applied equally to all messages exchanged that concern hyperties. Since the Messaging Node does not store which hyperty corresponds to a given address, it is not possible to verify subscription preferences for a given hyperty, hence this policy is only supported by the Core Runtime. Also, since the Messaging Node does not store the user groups, which can only be created by the user in the endpoint, conditions that require the verification of user presence in user groups are not supported. To summarize the supported conditions in a clearer way, Table 4.1 lists the conditions supported by each of the contexts where the Policy Engine was integrated, the Core Runtime and the Messaging Node.

The Policy Engine of Core Runtime is more than an enforcement point: it also interacts with the Identity Module, the component responsible for the management of identities and trustful communication between hyperties in the reTHINK framework. To do so, the Policy Engine triggers the Identity
Module’s procedures that ensure confidentiality, message integrity and authenticity when key-messages are intercepted. To accomplish these tasks, there are some steps to be performed before and after policy enforcement, which are different for the incoming and outgoing messages generated by hyperties. An outgoing message is a message created by a hyperty using the same Core Runtime as the Policy Engine, whereas an incoming message is a message created by a hyperty using a different Core Runtime.

The authorization flow is illustrated in the pseudo algorithm of Figure 4.8. When an outgoing message is intercepted by the Policy Engine, before validating it against existing policies, it must be stamped with the identity that corresponds to the hyperty that created it, provided by the Identity Module. Only then the message has the identity attached and the policies of that scope can be consulted. After policy evaluation, the Actions Service is invoked, and if the message is approved, then it is necessary to forward it to the Identity Module for the generation of the keyed-Hash Message Authentication Code (HMAC) and the encryption of sensitive fields. When an incoming message is intercepted by the Policy Engine, the sensitive fields are encrypted, and the message integrity and authenticity have not been validated. The decryption of the message and its integrity and authenticity validation is requested to the Identity Module, and only then it is safe to assume that the message can be trusted. The applicable policies are then enforced, and after their validation the message is forwarded to its destination.
for each message do
    if source or destination is hyperty then
        if is incoming message then
            decrypt, verify integrity and authenticate
        end
        else
            stamp message with identity
        end
    end
    load applicable policies
    evaluate policy
    get authorization decision
    invoke Actions Service
    if is authorized message then
        if is outgoing message then
            generate HMAC, encrypt
        end
        forward message to destination
    end
    else
        drop message
    end
end

Figure 4.8: Pseudocode for the algorithm of the Policy Engine of Core Runtime

Since the majority of the hyperty management is done in the endpoint, namely in the Core Runtime, it is expected that its Policy Engine is more complex than in the Messaging Node, which mostly works as a message distributor. The pseudo algorithm of the Policy Engine in the Messaging Node is presented in Figure 4.9. Since the Messaging Node is agnostic to the identity of the users and the hyperty behind the hyperty addresses, the rules it can verify are to be applied to all messages concerning hyperties. Therefore, the only steps to be done are the loading of policies, their evaluation, obtaining an authorization decision and enforce it, and finally execute additional actions in the Actions Service.

Next section will further detail how the portability of the Policy Engine was designed to support the Core Runtime and the Messaging Node environments.
for each message do
    if source or destination is hyperty then
        load applicable policies
        evaluate policies
        get authorization decision
        invoke Actions Service
        if is authorized message then
            forward message to destination
        else
            drop message
        end
    end
    else
        forward message to destination
    end
end

Figure 4.9: Pseudocode for the algorithm of the Policy Engine of Messaging Node

4.3.3 Storage

When a policy is added to reTHINK, it can be either stored in volatile or persistent memory. In the Messaging Node all the necessary configurations are loaded when the server is started. The same goes for the policies stored there, which are added to the Policy Engine when it is initialized. On the other hand, policies can be stored in the Core Runtime, which is a component of the reTHINK framework deployed in the user device to be accessed by the browser and mobile applications. It is common that sessions on those platforms are short-lived. For example, Alice opens a browser application to chat with Bob, maybe she stays online for 1 hour, and then she closes it. By having a persistent way of storing information she introduced in the application, that information will not be lost after she closes the application, which is an advantage for the user, as policies can be associated with events that will happen in the far future.

4.4 Administration Page

As previously introduced, the user is not expected to understand a policy specification language. Taking this into account, PoliTHINK offers a friendly interface for the browser where the user can specify his preferences. The interface is divided in two main sections: the first section provides the means to create, list, and delete existing policies and rules; the second section allows to create, list, and delete groups of users. Consider the following example: Bob had a huge fight with his sisters, Alice and Carol, and does not want to hear from them any time soon. To block all their contact attempts, she uses the
reTHINK administration page to create the family user group, which has Alice and Carol contacts added, and creates a rule to block that group. To do so, the following steps must be performed:

1. Open the policy administration page (Figure 4.10) and click on the ‘Groups management’ header;

![Figure 4.10: Policy administration page](image)

2. Create a new group named ‘family’ and add Alice and Carol emails to it (Figure 4.11).

![Figure 4.11: Groups management section](image)

3. Create a new policy by clicking on the ‘New policy’ button of the policies administration header (Figure 4.10) and select ‘Group of users’ from the dropdown to create a rule;
4. In the rule configuration section, ‘Group name’ field, pick the *family* group, apply this setting to all identities and hyperties, and select the ‘Block’ authorization decision (Figure 4.12);

![Rule configuration section](image)

<table>
<thead>
<tr>
<th>FILTER COMMUNICATIONS BY:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updating groups configurations</td>
</tr>
<tr>
<td>Incoming communications from a user whose identity is in the introduced group will be allowed or blocked according to your configurations, which can be changed in the preferences page.</td>
</tr>
</tbody>
</table>

Group name: family

To which of your identities or hyperties do you want to apply this setting?

All identities and hyperties

Authorisation decision:

- Block
- Allow

Figure 4.12: Rule configuration section

5. The policy administration page now should have a new policy with one rule stating that the *family* group is blocked for all identities and hyperties (Figure 4.13).

The specification of other types of rules follows a similar process, only differing in the input fields of the rule configuration section. Adding to the presented one, it is possible to specify rules that restrict communication based on all attributes supported by the Policy Engine: the time of the day, date, weekday, source email, source domain, communication type, data object scheme and subscription preferences. After submitting the configurations in the user interface, they are translated by the Policies Manager to the policy specification language. The Policies Manager creates a policy using our formal syntax and uses the Policy Engine API to add the new policy for its enforcement.
4.5 Summary

This chapter presented the design of the Policy Engine, the reTHINK component responsible for the enforcement of policies in the Core Runtime and in the Messaging Node. The specification of these policies follows a fixed syntax created to fit the needs of the reTHINK framework. It is intended to be a simple language, non-verbose, which allows flexible use of its fixed vocabulary, and the possibility of using a graphical user interface further decreases the specification complexity. In the next chapter it will be further described the implementation details of this module.
Chapter 4 presented the architecture and provided a general overview of PoliTHINK. In this chapter, we will provide the implementation details of the three subcomponents of our system: the Policy Specification Language (Section 5.2), the Policy Engine (Section 5.3) and the Graphical User Interface (Section 5.4). Before further describing these subjects, we will present an overview of the implementation.

5.1 Implementation Overview

As a project for the web, the reTHINK framework is written in a scripting language based on JavaScript, ECMAScript 2015 \[^1\]. This scripting language offers a powerful syntax which is very useful for the development of complex applications like reTHINK, for instance, objects classes and the Promise object. To be fully compatible with the reTHINK framework, PoliTHINK is itself implemented using ECMAScript 2015. The Policy Specification Language uses several classes that store the information about policies, rules and conditions. The enforcement of policies in the Policy Engine is managed by three main classes, one to obtain the attributes to be verified in the policy evaluation, another to generate an authorization decision based on the attributes obtained from the first, and another to effectively enforce that decision to a given message. The Graphical User Interface of PoliTHINK provides an administration webpage for the reTHINK application for policies, and is composed of two classes, one to handle the user interactions with the page, and another to translate the information provided by the user in the interface to our Policy Specification Language.

Figure 5.1 presents the directory tree of PoliTHINK, where the classes that implement the logic of each component of the system are listed. The Policy Specification Language allows the specification of policies, where each policy is comprised of a set of rules and a combining algorithm to resolve conflicts that may arise in the rules’ evaluation. The supported combining algorithms are Allow Overrides, Block Overrides and First Applicable. The first gives priority to a positive authorization decision, the second to a negative authorization decision, and the third will prioritize whichever decision the evaluation of the first rule of the set decides (positive or negative). Among other information, each rule contains a Condition, which can be extended to a Subscription Condition or an Advanced Condition. In general, simple conditions verify one attribute of the system, subscription conditions verify the subscription preferences, and advanced conditions are able to combine several simple conditions through the \textit{and}, \textit{or} and \textit{not} logical operators, whose logic is implemented in the Operators class. The Policy Engine is responsible for the enforcement of the policies specified using our policy specification language. The enforcement

\[^1\]ECMAScript 2015 (or ES6) is standardized by Ecma International
of authorization decisions is done in the PEP, which are generated by the PDP by consulting the attributes retrieved from the ReThinkCtx; policies can specify additional actions to be executed, which are executed by the Actions Manager. To reuse the Policy Engine code, classes that work as plug-ins are implemented separately according to the specific environment the Policy Engine may be inserted in. We integrated PoliTHINK in both the Core Runtime component and the Vertx Messaging Node, to be further described in Section [5.3.3.4]. These two environments have different characteristics, which led to the implementation of separate classes to make the integration possible: the Core Runtime Context class and the Vertx Context class, respectively. For the Policy Engine in the Vertx Messaging Node, which is implemented in Java, the Policy Engine Verticle class was created to work as interceptor of the Vertx internal bus, and the Policies Connector class as stub between the Java language and the JavaScript language, the programming language in which the Policy Engine is implemented. To be able to enforce policies in that environment, the Policies Handler class was created to intercept the messages circulating in the Message Bus. Finally, the User Interface is the graphical component of PoliTHINK that provides a simple way to specify policies for the average user. To build this interface, the index.html was created to hold the HTML elements, and two classes were created: the Policies GUI handles the user interactions with the elements of the webpage, and the Policies Manager is responsible by translating those interactions to a policy in our policy specification language.
As introduced in Section 1.2, one of the requirements of this thesis is to guarantee the portability of the Policy Engine, enabling code reuse in the several environments of the reTHINK framework. In PoliTHINK, this is achieved by implementing the logic that is common to all environments independently from the one that is specific to the context the Policy Engine is in. The difference between the various contexts is mainly on how to load and store policies. Therefore, we implemented the Rethink Context class to provide the features that all reTHINK contexts can provide (namely time and fields of the messages circulating in the framework), and the particularities of policies management are handled by the context-specific classes, thereby enabling the portability of the component.

Next, the implementation details of each of the components of PoliTHINK are described: Section 5.2 illustrates how the Policy Specification Language classes are organized to hold the necessary information for the specification of policies, Section 5.3 provides further details on the policy enforcement flow, and lastly Section 5.4 presents the user interface, created to facilitate the specification of policies to the user.

### 5.2 Policy Specification Language

For the specification of policies in reTHINK, it is fundamental to follow a fixed syntax across all environments to enable the correct management of policies. Therefore, a policy in reTHINK is organized in a fixed structure, similar to what is commonly found in the state of the art presented in Section 3. This structure is illustrated in Figure 5.2, each policy may be composed of zero or more rules, and in its turn each rule is composed of one or more conditions. It is possible to create a policy without rules when it is only needed to specify actions to be carried out without examining the system attributes. Advanced conditions support any combination of several conditions through the use of the logical operators and, or and not, which provides more expressiveness to the policy specification language. The conditions’ applicability is verified using functions or operators by comparing the value of a given system attribute with a given parameter. To avoid conflicts in the generation of an authorization decision from the evaluation of several rules, a policy combines the individual authorization decisions obtained from each rule by using one of the three implemented combining algorithms - Allow Overrides, Block Overrides and First Applicable. More than a conflict solver, this feature also gives the policy administrator more options on how to use rules to obtain an authorization decision. In the next sections we will provide more implementation details of each class and their role in the reTHINK framework.
5.2.1 Policy

The Policy class holds the necessary information for the generation of an authorization decision. The enforcement of the user’s or service provider’s preferences and configurations is achieved by specifying what are the attributes of the system to examine and how to proceed in case that examination verifies some parameter. To store those preferences and configurations in an organized way, a Policy instance holds an array of rules in the rules property, each of them specifying how to proceed given a condition or a combination of conditions. The evaluation of each of the rules generates an authorization decision, which may or may not conflict with other rules’ authorization decision. Therefore, a Policy instance also allows specifying how to combine those individual results into a single one through an algorithm that generates one final authorization decision. The chosen algorithm is stored in the combiningAlgorithm property of the Policy instance. Adding to that, policies may request the execution of additional actions independently from the rules’ applicability or the final authorization decision. Therefore, a Policy instance also has the actions property, which is of the array type, and holds an array of JSON objects with two members, the method and the parameter. The former holds the name of the function to be executed when the policy is applied, and the latter holds the parameter to be used by that method. Lastly, we introduce the key property. In the reTHINK framework there are two entities involved in the specification policies: the service provider and the user. On one hand, the service provider specifies policies to be enforced for communications regarding a given hyperty, where the key corresponds to the hyperty name. On the other hand, the user may specify policies for different occasions and use the key property to identify that occasion. For instance, it is useful to have a policy that manages communications when a user is on vacations, and another that manages communications when the user is working. For the first situation, the key of the policy could be ‘Vacations’, and for the second situation could be ‘Work’. In both the service provider and the user situations, it is fundamental to have an identifier that unequivocally identifies each policy. This is achieved by verifying if the key of a policy being instantiated already exists before completing the instantiation process.

The management of the previously presented properties requires the implementation of several methods, namely for the assignment of a combining algorithm instance, for the addition, deletion, or-dination, and evaluation of rules, and to trigger the execution of actions. A description of each of the relevant methods is presented below:

**constructor (key, rules, actions, combiningAlgorithm):**
When the Policy class is instantiated, sets the key, rules, actions and combiningAlgorithm properties with the information received as parameter. The Rule instances are created if an array of JSON rules is received instead of an array of Rule instances, and a Combining Algorithm is instantiated if a string representing the combining algorithm is received instead of the corresponding class.

**evaluateRules (context, message):**
Receives the context the policy’s rules are to be evaluated in, and the message requesting an authorization decision. Triggers the evaluation of the policy’s rules and combines the various authorization decisions in a single authorization result.
executeActions (context, message):
 Receives the context the policy’s actions are to be executed in, and the message that triggered the execution of the policy actions. For each entry on the actions array of the policy instance, the Actions Service is invoked asynchronously to execute the action.

createRule (decision, condition, scope, target, priority):
 Creates a Rule instance with the decision, condition, scope, target and priority parameters and adds it to the rules array of the policy instance. An instance of the Condition class is created if a JSON is received instead of a Condition instance. The subclass of the Condition class to instantiate is determined by the type parameter. If the priority parameter is not defined, verifies which is the value of the last priority value assigned and increments it for the rule being created.

deleteRule (rule):
 Removes the rule instance received as parameter from the rules array of the policy instance.

sortRules ():
 Returns the rules array of the policy instance sorted by priority.

The listed methods are used by different components of the system: one the one hand, the evaluateRules and executeActions methods are used by the PEP in the enforcement of the policies; on the other hand, the createRule, deleteRule and sortRules are used by the user interface. The former two enable the management of one rule at a time as they are created and deleted through the user interface, whereas the latter is invoked to sort the rules by priority before presenting them at the user interface.

As suggested by the description of the evaluateRules method, the evaluation of a policy reflects the individual authorization decisions generated by the individual evaluation of its rules. Next, details on the Rule class are presented.

5.2.2 Rule

The Rule class holds the necessary information about which characteristic(s) of the system the policy administrator wants to examine, and what to decide in case the given condition is verified. As introduced in Section 4.2, a rule is composed of a scope, a target, a condition, a decision, and a priority. The role of each of them is described next.

5.2.2.1 Scope and Target

To specify a target for a given rule instead of applying it to all messages, the scope property can be set to one out of three possible keywords that represent domains of applicability - ‘global’, ‘identity’, or ‘hyperty’. This property is useful for the correct interpretation of the target property value, which may hold the ‘global’ string, a user email or a hyperty name. Without the scope property, it would be possible
policy: {
  key: ‘No work after 10pm’,
  rules: [{
    scope: ‘identity’,
    target: ‘alice@bestcompany.com’,
    condition: {
      attribute: ‘time’,
      operator: ‘greaterThan’,
      parameter: ‘2200’
    },
    decision: false,
    priority: 0
  }],
  actions: [],
  combining-algorithm: block-overrides
}

Figure 5.3: Example policy to apply to a given identity

to misinterpret the target of the rule: the identification of the type of the target is not always unequivocal.
At first sight, targets with the format of an email are user emails, and the ‘global’ string always reflects
that the rule targets all communications. Taking a closer look, this assumption is not always true: there
is no impediment in the point of view of a hyperty developer, that the hyperty name does not have the
characteristics of an email, for instance, ‘awes@me.hyperty’, or that the hyperty name is not ‘global’.
Adding to this, it significantly facilitates the verification of rule applicability, as it is not necessary to
consider the structure of the target property value - it is explicitly stated in the scope property. Consider
this example: Alice has two emails, alice@bestcompany.com is her work email, and alice@gmail.com is
her general email. She wishes to ignore all incoming communications through her work email after 10
p.m.. The policy that represents this definition has the format shown in Figure [5.3] The policy has one
rule, which is applicable if the message destination is alice@bestcompany.com and if the current time of
the day is greater than 22 p.m.. In case it is, the rule is applicable and the decision assigned to that rule,
in this case false, gives the final authorization, which is to reject the message.

If Alice decides that instead of filtering the communications destined to her work email, she decides
that she does not wish to be contacted through a hyperty that manages calls named HypertyCall after
22 p.m., independently of the identity it is destined to. The policy that represents this definition is similar
to the one shown in Figure [5.3] except for the scope and target properties - the scope must be ‘hyperty’
and the target must be ‘HypertyCall’. Using these two fields to restrict policy applicability in the identity
domain and in the hyperties domain is a feature that greatly improves the expressiveness of the policy
specification language. Since the reTHINK framework is a particularly diverse environment to the extent
that there are several identities and hyperties being executed in the same runtime, these fields reveal
itself a valuable improvement when comparing to rules that are applied to all messages equally.

The management of the Rule properties is handled by the Policy class, which deals with the creation
and deletion of rules. The only relevant method of the Rule class is evaluate(): it receives the context
the rule is to be evaluated in, and the message requesting an authorization decision. Verifies if the target
of the message is the one specified in the target property of the rule and if the condition is verified. If
the scope holds the ‘hyperty’ string, then the hyperty name of the source or destination (depending if
the message is incoming or outgoing) is retrieved from the Runtime Registry to be compared with the target value. If the scope holds the ‘identity’ string, then the identity used in the hyperty instantiation of the hyperty source or destination is retrieved from the Runtime Registry and is compared to the target value. If the scope holds the ‘global’ string, then the target does not verification, since the rule is to apply to all messages. Then, the condition property is examined. If the condition is applicable, then the decision property of the rule is returned; otherwise, the ‘Not Applicable’ rule is returned.

Next, a more detailed description of the condition property is provided.

5.2.2.2 Condition and Decision

The condition of a rule holds the system attribute to be examined, which is the reference value for that attribute and which operator to use to compare its value with the given parameter. The condition property of a rule is an instance of the Condition class, which itself has three properties, attribute, operator and parameter. The attribute is a string that represents the system attribute to be verified. When evaluating a rule, the Policy Engine requests the value of the system attribute that is specified in this property and compares it with the parameter using the function specified in the operator. As previously listed in Section 4.3, there are several attributes of the system that can be examined for rules’ evaluation. Those attributes are represented in the attribute property of the Condition instance by a keyword that identifies it. The list of the keywords that correspond to each attribute is presented in Table 5.1. These keywords were chosen because they represent the attribute they concern in a concise way, while enabling the policy administrator to intuitively understand which attribute they refer to. The operator property holds the function used to compare the current value of the system attribute specified in the attribute property with the parameter chosen. The implemented operators and corresponding descriptions are the following:

- **equals**: checks if the attribute value is equal to the string specified in the parameter,
- **in**: checks if the attribute value is present on the array specified in the parameter,
• **greaterThan**: checks if the attribute value is greater than the integer specified in the parameter,

• **lessThan**: checks if the attribute value is less than the integer specified in the parameter,

• **between**: operator to be used for timeslots only; checks if the received time is within a timeslot, for instance, checks that 5 a.m. is between 11 p.m. and 8 a.m..

The **parameter** property of a condition is the reference value specified by the user. This reference value may be a string, an integer or an array, and is used as one of the parameters of the functions that evaluate the condition applicability. These three properties of a condition, the attribute, the operator and the parameter, hold enough information regarding what the policy administrator expects from an attribute. If the value of the specified attribute is in accordance with the given parameter, the rule is said to be applicable; otherwise, the rule is not applicable. Consider the following use case: Alice wants to block incoming communications at Sundays. She does this by creating a policy and specifying a rule with the condition ‘weekday equals Sunday’. If Alice receives an incoming call on her smartphone, the policy she specified is examined, namely the rule regarding the weekday. The PDP verifies which is the current day of the week, which is Friday, hence the rule is not applicable and the decision associated with the rule is not taken into account; if the day of the week were Sunday, then the rule is applicable and the associated decision is relevant for the generation of the message authorization. After verifying rule applicability by the examination of the condition property, the value of the decision property is collected; otherwise, the ‘Not Applicable’ string is returned. When a message is intercepted, there are two possible options: either the message is authorized and forwarded to its destination, or the message is not authorized and consequently not delivered to its destination. Since there are only these two options, one affirmative and one negative, the authorization decision is specified through a boolean. When the intent is to authorize the message if a given condition applies, then the value of the decision property must be true; on the other hand, when the intent is to reject the message, the value must be false.

In Section 4.2 we presented advanced conditions, i.e., conditions that combine the examination of several conditions by the means of the **and**, **or** and **not** logical operators. An Advanced Condition instance is built from one or more Condition instances placed on an array along with a string representing the logical operator to combine them. The logical operator is placed in the leftmost position of the array, and depending on which operator it is, the array may have one or two more entries: if the **not** operator is used, then it is expected only one Condition instance; if the **or** or **and** operators are used, then two Condition instances are expected. Each array entry that is destined to the Condition instances may be recursively filled with more combinations. For the evaluation of advanced conditions, the array is swept from left to right; for each array in the advanced condition, we verify if the condition(s) are applicable and then the logical operator is applied to return if the rule is applicable or not. The logical operators are implemented in the Operators class, where the operators that are part of the Condition class are defined. The implementation of the logical operators basically consists on translating the string **and**, **or**, and **not** to the corresponding logical operator in JavaScript:

• **and**: applies the && logical operator to two booleans,

• **or**: applies the || logical operator to two booleans,
• **not**: applies the / logical operator to one boolean

A particular case of the *Condition* class is the enforcement of subscription preferences. A subscription preference, as the name implies, states how the user wants the system to proceed in case a subscription attempt occurs. Therefore, the subscription preference is only to be checked if two characteristics are present in the message: first, the message must be of the *subscribe* type; second, the source of the message must be a remote Sync Manager. To facilitate the understanding on the need to check for the remote Sync Manager, consider Figure 2.6 where the message sequence of a data object subscription is illustrated. When a hyperty in device B subscribes a hyperty in device A, the subscription message is propagated from one B to A. Since the subscription preference is to be applied in device A (which has the hyperty being subscribed), and not on device B (which has the subscriber hyperty), messages from the Sync Manager of the same runtime as the subscriber hyperty are not to be validated by the subscription condition. This peculiarity led to the creation of the Subscription Condition class, which inherits from the Condition class and verifies the condition applicability in the same manner after the extra inspections.

### 5.2.2.3 Combining Algorithms and Priority

When a policy has more than one rule, it is common that the authorization decision of one rule conflicts with the authorization decision of other rules. In order to prevent that these conflicts compromise the generation of an authorization decision, a policy allows the specification of the way to combine them. This feature of the language is represented in the *combiningAlgorithm* property. When the evaluation of a policy is triggered, the individual results of the rules’ evaluation are collected to an array, and the final authorization decision is the result of one out of three possible combinations. The first option is to reject the message if one of the rules evaluates to *false*; the second option is to authorize the message if one of the rules evaluates to *true*; the third option is to assign a priority to the rules and to evaluate them by their order. These three approaches are implemented in different classes, respectively named **Block Overrides**, **Allow Overrides** and **First Applicable**, which implement a method to combine the individual results collected from the rules’ evaluation. To enable the **First Applicable** algorithm, each rule must have an indicator of its importance compared to others. This is achieved by assigning an integer to the *priority* property of the rule, where a smaller integer means a higher priority, having the rule with priority 0 as the most important.

- **Block Overrides**: If there is one negative authorization decision in the collection, the final authorization decision is *false*. If there is no negative authorization decisions in the collection, but there is a positive one, the final authorization decision is *true*.

- **Allow Overrides**: If there is one positive authorization decision in the collection, the final authorization decision is *true*. If there is no positive authorization decisions in the collection, but there is a negative one, the final authorization decision is *false*.

- **First Applicable**: Since the collection of results is ordered by priority, the final authorization decision is given by the first entry different from ‘Not Applicable’, either *true* or *false*. 54
When the collection being examined by any combining algorithm only has ‘Not Applicable’ entries, the evaluation done in the combining algorithm is also ‘Not Applicable’. If this happens, the default authorization decision hardcoded in the Policy Engine is used. Given that the goal of the reTHINK project is to leverage today’s communication services and not the contrary, the default behavior is to authorize messages. If the default behavior were to reject messages, contexts with zero policies would have all messages blocked, which is not compatible with the reTHINK goal.

Together, the previously presented classes implement the methods for the management of policies in the reTHINK framework. In the next section we will describe how the specified policies are enforced in the Policy Engine.

### 5.3 Policy Engine

One of the goals of this thesis is the development of a policy enforcement mechanism, the Policy Engine. The implementation details of this component of PoliTHINK are described in this section. As illustrated in Section 4.3, the Policy Engine is responsible for validating the intercepted messages against the policies defined by the user or by the service provider. To guarantee its portability among the several components of the reTHINK framework, the Policy Engine is a module that contains the implementation of what is common in the different environments. Then, the instantiation of the Policy Engine receives the specific features of the environment where it is being instantiated as a parameter of its constructor, which works as a plug-in for the core Policy Engine.

The class diagram illustrated in Figure 5.4 describes the internal structure of the Policy Engine, which is composed of the PEP (Policy Enforcement Point), the ReThink Context, the PDP (Policy Decision Point), and the Actions Service. The PEP is the central point of the Policy Engine: it triggers the generation of an authorization decision in the PDP and the execution of actions in the Actions Service. The Context classes retrieve the system attributes values and handle the loading of configurations, which are necessary for the operation of the system. A PEP is composed of exactly one ReThink Context, one PDP, and one Actions Service. The ReThink Context class may be extended to provide context-specific features depending on the component of the reTHINK framework where the Policy Engine is used. In order to prove this thesis requirement regarding portability, the Policy Engine was integrated in two differ-
ent environments of the reTHINK framework: the Core Runtime and the Vertx Messaging Node. Before describing how the adaptation of the Policy Engine to different contexts is done, we will present how the components of the core Policy Engine operate.

5.3.1 Policy Enforcement Point (PEP)

As illustrated in Figure 4.7, the PEP is the component that intercepts the messages for validation against policies. The first step that is required for the enforcement of policies is to create an instance of a PEP. In that procedure, the PEP instantiates the necessary internal components for the enforcement of policies specified through the Policy Specification Language presented in Section 5.2, namely the PDP and the Actions Service. The next step for policy enforcement is to supply policies to the Policy Engine, which may be specified by two different entities - the service provider or the user. The service provider defines policies to be applied to messages whose source or destination (depending if it is an incoming or outgoing message) is the hyperty associated with the policy. The user defines policies according to his reachability preferences through the administration page; at a given moment, at most one user policy is active, which is a setting that can be configured through the administration page. Policies added to the Policy Engine must be able to be removed, for instance when the behavior that they represent is no longer desired. Finally, there must be a way to trigger their evaluation when a message is intercepted. The methods that implement these four features correspond to the API to be used by the reTHINK components that require the enforcement of policies, and are detailed below.

**constructor (context):** Creates an instance of a PEP. Receives an instance of the ReThink Context class as parameter to be used for policies management. Creates a PDP instance and an Actions Service instance, and loads the necessary configurations for its operation.

**addPolicy (source, key, policy, combiningAlgorithm):** Adds a policy to the Policy Engine. The source parameter can have the values ‘USER’ or ‘SERVICE_PROVIDER’; the key identifies the policy unequivocally; the policy is an instance of the Policy class; the combiningAlgorithm is a string representing the combining algorithm to be assigned to the policy.

**removePolicy (source, key):** Removes a policy from the Policy Engine whose source and key are the ones received as parameters.

**authorise (message):** Generates an authorization decision for the message received as parameter by examining the defined policies.

The instantiation of the PEP requires the specification of the ReThink Context it is operating in. The context will be used throughout the PEP operation flow to handle the configurations loading and to retrieve the value of the system attributes, necessary for policy evaluation. After its instantiation, the PEP is ready to receive the policies either from the two entities involved in policy specification in reTHINK: the service provider and the user. The specification of the policy source allows assigning the policy to different hashtables, which are consulted in different phases of the policy evaluation. Whichever
the source, the PEP may receive a policy in JSON format in the policy parameter or it may receive the identifier and the combining algorithm in the key and combiningAlgorithm parameter to create a Policy instance without rules or actions to be executed. When the behavior that policies represent is no longer desired, they can be removed from the PEP by providing the source and key of the policy to be deleted. The implementation of this method consists on consulting the hashtable that correspond to the given source, and then deleting the entry with the given key.

The PEP authorization flow is a bit more complex than consulting the policies stored in the hashtables. As described in Section 4.3, only messages exchanged between hyperties should be verified against the defined policies. To do so, the first step on the PEP authorization flow consists on verifying if the source and destination of the message is a hyperty. As described in Section 2, hyperties communicate through data object synchronization. Therefore, messages between hyperties may have hyperty URLs as the source or destination of the message, as well as data object URLs. Since the scheme of hyperty URLs is always hyperty, they are easy to identify, but the same simplification cannot be used when it comes to data objects: data object URLs are not associated with a set of possible schemes from which the services providers choose, they can be freely created. This feature of reTHINK implies that the identification of data objects URLs must be done by verifying if the scheme of the message source and destination is not one of schemes used in communications between the internal components of the reTHINK framework, i.e., is not domain, domain-idp, global, hyperty-runtime or runtime. The domain scheme is used to contact the service provider, domain-idp to contact an Identity Provider, global to contact the Global Registry, and runtime to contact the components of the device runtime where hyperty instances are running. If the scheme of the message source or destination is not any of these, then it corresponds to a hyperty URL or a data object URL and will be validated by the PEP; otherwise, the message is authorized and is forwarded to its destination. After verifying that a given message concerns a hyperty, the PEP verifies if the message is incoming or outgoing to enable the loading of the policies of the correct service provider and user. This information is obtained by consulting the identity field: if it does not exist, then it is an outgoing message was created in the current runtime and will be stamped with the identity before leaving the runtime; otherwise, the message was created in another runtime, and is an incoming message. This information, along with the message, is sent to the PDP and to the Actions Service in the boolean format: true if it is an incoming message, false if it is an outgoing message. The policy evaluation is then triggered in the PDP, where the appropriate policies are loaded and the generation of an authorization decision is carried out. The decision is then returned to be PEP, which requests the execution of the actions specified in the policy to the Actions Service. When this task is finished, the authorization decision retrieved from the PDP is enforced: if the authorization decision is positive or if no policy was applicable, the message is forwarded to its destination; otherwise, if the authorization decision is negative, this message is dropped and the error message with code 403 (Forbidden) of Figure 5.5 is returned to the sender.

The presented steps of policy evaluation constitute the main flow of the PEP. Next, we will further detail how the PDP operates to obtain an authorization decision for the messages exchanged between hyperties.
errorMessage: {
    from: originalMessage.to,
    to: originalMessage.from,
    body: {
        code: 403,
        description: ‘Blocked by policy’
    },
    id: originalMessage.id,
    type: ‘response’
}

Figure 5.5: Error message returned to the sender if the original message is blocked by a policy

5.3.2 Policy Decision Point (PDP)

When a message concerning a hyperty is intercepted by the PEP, an authorization decision is requested to the PDP. Since it is an internal component of the Policy Engine, this class does not expose any method to other components. The request for the generation an authorization decision in the PDP is triggered by the PEP through the `evaluatePolicies()` method. This method has two parameters, the message over which to obtain an authorization decision, and a boolean indicating if it is an incoming or an outgoing message. First, the appropriate policies are loaded. The service provider policy is obtained by verifying the boolean representing if the message is incoming or outgoing. This step would not be necessary if the communication between hyperty instances was restricted to the same hyperty, but this is not true: instances from different hyperties may communicate if the resource type they use is supported by both hyperties. If it is an incoming message, then the policy to be enforced corresponds to the hyperty in the message destination; otherwise, the policy to be enforced corresponds to the hyperty in the message source. Since service provider policies has the hyperty name as key, it is necessary to translate the hyperty URL or data object URL to the name of the hyperty it corresponds to. This is achieved by requesting the hyperty name associated to the hyperty URL to the Runtime Registry, which holds information about hyperties that is relevant for the management of the runtime. The PDP then fetches the entry of the service provider policies hashtable whose key is the hyperty name retrieved by the Runtime Registry. Then, if user policies are supported in the environment the PDP is operating in, the user policy is loaded. To do so, the key that corresponds to the active user policy is used to fetch it from the user policies hashtable. After loading both policies, the service provider policy’s rules are evaluated. If its evaluation returns a negative authorization decision, the user policy is not evaluated and the message is rejected; otherwise, if supported, the user policy’s rules are evaluated, and the resulting authorization decision is returned to the PEP to be enforced.

Evaluation of rules and the loading of policies or other configurations depends on the context where the Policy Engine is operating. To preserve the portability of this component, the context-specific features are implemented in a separate class that works as a plug-in to the Policy Engine, which will be further detailed in the next section.
5.3.3 Context

One of the requirements of PoliTHINK is the portability of the Policy Engine. This feature enables the developers to reuse the Policy Engine code in the reTHINK components that require policy enforcement, taking advantage of having a single maintenance point of the code. In order to build a Policy Engine with no dependencies of the environment it is in, any code that is common to all environments of the reTHINK framework is structured in a self-contained module. The context-specific features are implemented on a separate class that works as a plug-in to the core Policy Engine. Consider Figure [5.4] where the ReThink Context class is represented. This class holds the information regarding the context attributes that are common across the reTHINK framework and relevant for the Policy Engine operation. This class is responsible for retrieving the value of the system attributes requested for policy evaluation. Basically, the evaluation of policies examines two types of attributes: the first type concerns attributes that can be obtained directly or indirectly from the fields of a message; the second type concerns attributes that can be obtained from native JavaScript methods. Consider the attributes supported by the Policy Engine listed in Table [5.1]. Apart from the hyperties subscription preferences, which can only be obtained in the Core Runtime by querying the Runtime Registry, all attributes may be obtained directly from the message fields or from JavaScript native methods.

5.3.3.1 Acquisition of attributes values from message fields

- **Source email**: this attribute is found in the `params.message.body.identity.userProfile.username` field (e.g.: alice@gmail.com)
- **Source domain**: the email of the previous item is parsed to obtain the domain field (e.g.: gmail.com)
- **Communication type**: this attribute is found in the `message.body.value.resourceType` (e.g.: audio)
- **Data object scheme**: the data object URL found in `message.from` is parsed to obtain the scheme (e.g.: comm)

5.3.3.2 Acquisition of attributes values from JavaScript native methods

- **Date**: the JavaScript Date object offers methods to obtain the day, month and year from a Date instance, respectively `getDate()`, `getMonth()` and `getFullYear()`, which are then separated by a slash (e.g.: 08/01/2017)
- **Weekday**: the JavaScript Date object offers a method to obtain the weekday, the `getDay()` method, which returns an integer that represents the day of the week. This value is then converted to the name that corresponds to the weekday (e.g.: Sunday)
- **Time of the day**: the JavaScript Date objects offers the `getMinutes()` and `getHours()` methods, which are then concatenated (e.g.: 2200)
The above listed attributes are obtained in the same manner independently of the context the Policy Engine is integrated in. Unlike the acquisition of attributes’ values, the available policies in each environment and the way they are retrieved and updated is different as they can be managed through persistent or volatile memory. These context-specific features are implemented in a separate context class that inherits from the ReThink Context class. Since the most of the reTHINK framework computation is done in the endpoint, the Core Runtime is the most important component wherein the Policy Engine must be present. To prove that the portability requirement is met, the Policy Engine was also deployed in the Vertx Messaging Node, a component of the reTHINK framework that handles the routing of messages between different devices. The two environments have very different peculiarities that were tackled for the integration of the Policy Engine in each of them. The implementation of each environment particularities is represented in Figure 5.4 i.e., the Core Runtime Context and the Vertx Context. Next, we describe how the context-specific adaptations were implemented in both environments, starting with the Core Runtime.

5.3.3.3 Core Runtime

The Core Runtime is the component where most of the tasks are performed, making it the most complex environment of the reTHINK framework. Consequently, and as presented in Section 4.3.1, the Policy Engine has increased responsibilities with regard to the framework management. In the Core Runtime, an authorization decision is requested to all messages circulating in the Message Bus. To achieve this, the Runtime UA manages a handler that invokes the `authorise()` method of the Policy Engine, which is executed asynchronously. Since all messages pass through the Policy Engine by means of that handler, it becomes a central point in the Core Runtime, and consequently is in a good position to have a more active role on the management of the framework.

One of the tasks of the Policy Engine in the Core Runtime is the assignment of an identity to a message. More than enabling the destination to identify the user that sent the message, policies that examine the `identity` field for the enforcement of reachability preferences require that this field exists and is correct. When a message is sent by a hyperty running in the current runtime, this field is not set, and the Policy Engine uses the fact that the field is present or not to detect if the message is incoming or outgoing, respectively. In the case of an outgoing message, the field does not exist and the identity that corresponds to the message is requested by the Policy Engine to the Identity Module. Given the hyperty URL present in the `message.from` field, the Identity Module returns the identity that was used when the corresponding hyperty was deployed. Then, the Policy Engine creates the `identity` field in the message with the information received from the Identity Module. To ensure the message integrity and confidentiality at the destination, before forwarding it, the Policy Engine triggers a function of the Identity Module for the creation of the HMAC and the encryption of the message, where the `identity` field is included. At the destination, the integrity of the message is verified, and then the fields of the received message can be trusted and safely used for policy enforcement. The generation of the HMAC requires that the Identity Module’s authentication protocol is executed; this protocol is triggered by the Policy Engine when a hyperty subscription is intercepted and allows the generation of the necessary keys for guaranteeing the integrity of the messages. Another task of the Policy Engine, as described in Section 2.3.3, is to verify if the messages of the `update` type are valid from the point of view of the Reporter-Observer model. Since data objects can only be changed by the corresponding reporter hyperty, the
Policy Engine verifies if the intercepted update messages were issued by the reporter of the data object. To do so, the Policy Engine questions the Runtime Registry about which is the hyperty reporter of the source data object, and compares it with the message.body.source field, which holds the URL of the hyperty that issued the update. If they match, the update is valid.

From the Policy Engine point of view, the processing of a message since after its creation until its delivery follows the steps listed below, assuming that it is not rejected by any policy:

1. Message received in the Policy Engine of the source runtime
2. Stamp the message with the corresponding identity
3. Enforcement of the policies specified by the user and service provider of the message source
4. Trigger HMAC generation and message encryption
5. Forward the message to the destination runtime
6. Message received in the Policy Engine of the destination runtime
7. Message decryption and verification of its integrity
8. If the message is of the update type, verify if it follows the Reporter-Observer model
9. Enforcement of the policies specified by the user and service provider of the message destination
10. Trigger the mutual authentication protocol if it is a hyperty subscription
11. Message leaves the Policy Engine of the destination runtime

Summing up, the steps to be performed by the Policy Engine can be grouped in two stages: the first stage is carried out before the enforcement of policies, and the second stage after the enforcement of policies. This lead to the creation of two methods that are only implemented in the Core Runtime: the first method, prepareForEvaluation(), handles the identity acquisition in the source runtime, and the message decryption, integrity verification and validation of update messages in the destination runtime; the second method, prepareToForward(), handles the generation of the HMAC in the source runtime, and triggers the mutual authentication protocol in the destination runtime.

In Section 4.3.1 the Policy Engine of Core Runtime support for examination of the message source presence in groups of users was introduced. This is achieved by requesting the Core Runtime Context class which are the members of a group given its name. To be able to verify this type of policies, the Policy Engine must expose several methods that allow the management of groups, namely its creation and deletion, and a means to add and remove members to the created groups. The Core Runtime Context uses a hashtable to store the groups, where the key for each entry is given by the group name, and the corresponding value is an array of the user emails that are part of that group. To manage this feature, the methods that need to be implemented on the context-specific class are the following:
**createGroup** *(groupName)*:
Creates a group with the given *groupName*.

**deleteGroup** *(groupName)*:
Deletes the group with the given *groupName*.

**addToGroup** *(groupName, userEmail)*:
If the group with the given *groupName* exists, adds the given *userEmail* to it.

**removeFromGroup** *(groupName, userEmail)*:
If the group with the given *groupName* exists, removes the given *userEmail* from it.

The management of this feature is carried out by the User Interface component of PoliTHINK, and will be further detailed in Section 5.4.

In order not to lose the information about the user when the user session ends, the groups that were constituted and the created policies are stored persistently using local storage, which stores data locally within the user’s browser. This feature is handled by the Persistence Manager of reTHINK, developed to be used for the reTHINK components that require the persistence of information, such as the Policy Engine. The Persistence Manager stores the information as a key-value pair, where the key identifies the information to be stored, and the information to be stored in the JSON format is the value. To retrieve the stored information, the Persistence Manager can be queried through the `get (key)` method, which receives as parameter the *key* used to store the information, and returns the information stored in the JSON format. The Policy Engine uses the Persistence Manager to store four types of information: the user policies, the active user policy, the service provider policies and the groups, which were assigned the keys `rethink:userPolicies`, `rethink:activePolicy`, `rethink:spPolicies` and `rethink:groups`, respectively. Transforming object instances to JSON objects causes the loss of information regarding the object instances when the fields are also object instances. The only situation where this causes the loss of relevant information is when handling the `combiningAlgorithm` attribute. If this fact is not handled, transforming an instance of a policy in a JSON will cause the information regarding the combining algorithm to be replaced by an empty object. To avoid this, before storing this information persistently, the combining algorithm specified in the policy is translated to a string that identifies it and only then is stored. When the combining algorithm is an instance of the Deny Overrides class, that field is replaced by the ‘denyOverrides’ string; for an instance of the Allow Overrides class, the field is replaced by the ‘allowOverrides’ string; for an instance of the First Applicable class, the field is replaced by the ‘firstApplicable’ string. This way it is possible to store the configurations of the Policy Engine persistently without losing any information.

### 5.3.3.4 Vertx Messaging Node

The Vertx Messaging Node is one of the reTHINK messaging nodes developed for message relay between hyperty instances running in different devices. Even though the Vertx Messaging Node source
code is implemented in Java, it also enables the deployment of JavaScript code through Verticles. The Vertx source code and Verticle components can communicate with each other through an internal bus. Adding to this, Vertx implements a pipeline with several handlers that intercept the external messages for the validation the message fields and forwards them to a stub that handles the invocation of the Policy Engine methods, the Policies Connector. Together, these features allow reusing the existing Policy Engine, which is implemented in JavaScript.

The first step for the implementation of policy enforcement on Vertx is to add a handler to the pipeline. The messages intercepted by the handler must be forwarded to the Policy Engine for policy enforcement. To do so, the internal bus is used to communicate with a Verticle installed for the interaction with the Policy Engine: the message is posted in the internal bus, which is implemented in Java, and received in the Policy Engine Verticle, which is implemented in JavaScript. The Policy Engine Verticle uses the Policies Connector to trigger the `authorize()` method, which will enforce the policies stored in the Policy Engine. When an authorization decision is obtained, it is sent back to the policy handler as a reply, which results in the message being forwarded or rejected according to the received decision.

As hinted in Section 4.1, the policy administrator on the service provider side is expected to understand a policy specification language. Since the Messaging Node is a backend server which does not include an administration page, the most convenient way of testing that the Policy Engine operates correctly is to load the service provider policy from a JSON file. Each Messaging Node holds one policy to be enforced on all messages exchanged between hyperties. This policy is read from the policy file and added to the Policy Engine once, through the `addPolicy()` method, when the first message is intercepted by the policies’ handler. Summarizing, the processing of a message follows the steps listed below:

1. Message received in the Vertx Policy Engine
2. The policies’ handler intercepts it and uses the internal bus to send it to the Policy Engine Verticle
3. The Policy Engine Verticle forwards the request to the Policies Connector, which relays it to the Policy Engine
4. Enforcement of the service provider policy
5. Authorization decision is sent back to the policies’ handler
6. Message is forwarded to the next handler or rejected

Message integrity is not yet possible to validate in Vertx, as there is no component to perform this task. Since it is planned that a new module to do this task will be included in the Messaging Node, the features supported by the Policy Engine are maintained with the reservation that there are no guarantees about the fields’ integrity. The fact that this guarantee does not exist is more relevant when the policy to enforce requires the examination of the `identity` field or the data object scheme present on the `message.from` field, which may have been replaced by a fake one to avoid being blocked. Another limitation of the Policy Engine in the Messaging Node is that it is not possible to apply rules that examine the type of the communication being used, as this field is encrypted and can only be decrypted by the message destination. Finally, it is also not possible to apply rules regarding hyperties subscription preferences, as the necessary information to apply the rule does not exist in the Messaging Node.
automaticMessage: {
  from: originalMessage.to,
  to: originalMessage.from,
  body: {
    value: 'I am currently on vacations, I’ll be back around the 1st of September!
  },
  type: 'create'
}

Figure 5.6: Message created by the automatic reply feature

in the Core Runtime, this information is requested to the Runtime Registry. Unfortunately, Vertx does not support ECMAScript 2015, which forced the creation of the temporary authoriseSync() method to execute the authorization flow in a synchronous fashion.

The authorization flow of the Messaging Node Policy Engine is considerably more simple, as its only task is to enforce policies and generate an authorization decision.

5.3.4 Actions Service

The reTHINK framework is a very rich environment, where a number of services and identities are managed in a seamless way for the user. One of the features that was planned to add value to the framework was the possibility of redirecting communication attempts to different destinations, similarly to what we have in the telephony services. Another desired feature would be to send an automatic response, similarly to what currently can be configured in our email accounts. These feature should be an option given to the user and should not be hardcoded in the system. Consequently, it is a feature to be implemented in the Policy Engine through policies. The Actions Service is the module responsible for the execution of actions like the ones just introduced, or other actions that the enforcement of policies may require. As described in Section 5.3.1, the Actions Service is instantiated when the PEP is created and the execution of the actions associated with a policy is triggered during the policy evaluation, where the array of actions is swept and each of them is executed asynchronously.

The redirection of messages is possible to configure in the Policy Engine in two different ways: first, by providing the hyperty URL of the destination; second, by providing the user identity of the destination. To redirect a message to a hyperty URL, the user must have the hyperty URL of the new destination, which is not the same throughout the hyperty instance life-cycle. Since it is part of the future work of reTHINK to reuse the URLs allocated at hyperty instantiation time, this way of allowing redirection will make better use in the future. The redirection process itself is simple: the message destination is replaced by the new destination, and then it is posted in the Message Bus to be delivered to the redirection address. To redirect a message to a user identity, there is only one extra step over the previous option: a request is sent to the Runtime Registry to retrieve the hyperty URL assigned to the given identity. Then, similarly to the first procedure, the destination of the original message is replaced by the retrieved hyperty URL, and then posted in the Message Bus to be delivered to the redirection identity.

Regarding the sending of automatic messages, it is achieved by posting a response message in the Message Bus. This message is has as destination the source of the original message and includes in
the body the user defined message. Consider this example: Alice is an important entrepreneur who is on vacations. Even though she wants to have some rest, she does not want that the managers of the many businesses she owns to think she is ignoring them on purpose. To achieve this, she sets a policy that sends an automatic message with the content ‘I am currently on vacations, I’ll be back around the 1st of September!’ When the Policy Engine intercepts a communication attempt, it generates a new message with the format presented in Figure 5.6.

The execution of actions in the Actions Service is the last task to be executed before the authorization decision is enforced, which finishes the authorization flow of the Policy Engine. The specification of policies in the reTHINK framework that carry out the features presented in the previous sections is a task that requires some insight into the JSON format and the general structure of policies. In order to assist the user in this task, a graphical user interface was created. This component of PoliTHINK is presented next.

5.4 Graphical User Interface

One of the requirements of this thesis is that the specification of policies has low complexity. Even though JSON is very easy for humans to read and write, it is expected that the average user does not understand how to build a well-formatted JSON file, or to easily understand the meaning of each JSON key and possible values. In order to facilitate the specification of policies in the end device, we created an administration page that offers a simple graphical user interface. This interface can be used by the user to specify policies by the means of input fields, buttons and dropdowns, which is a language the average user is familiar with. Since the focus of this component of PoliTHINK is to facilitate the specification of policies, we focused on providing the means for the user to specify the most basic rules; nevertheless, we also provide a means to import a JSON file, so the most advanced users to take advantage of the policy specification language full potential.

Before proceeding to the description of the implementation details of the administration page, we will describe the placement of the administration page regarding the reTHINK application for the browser. The reTHINK application is a webpage where the user can choose a hyperty to be instantiated from
Figure 5.8: Policy creation modal

a list of the available hyperties. When the application is started, the Core Runtime is downloaded and deployed on an iframe to isolate its execution from the application code. Then, the user can choose a hyperty, for instance, a hyperty for group chat communications or a hyperty for video calls, which is executed in a separate sandbox. To give access to the Policy Engine API, the administration page is also embedded in the Core Runtime iframe. The homepage of the reTHINK application shows a settings button in the upper right corner, which when clicked is expanded to show the administration page for policy administration and identities administration. To achieve this, the settings button has a listener for clicks, and when clicked posts a message for the Runtime UA stub (responsible for managing the deployment of the Core Runtime in the browser) through the browser native method `postMessage()`.

This message has the 'showAdminPage' string in the `method` field, that when intercepted by the Runtime UA stub changes the height and width of the iframe to 100% of the page, showing the administration homepage. When the EXIT button of the administration homepage is clicked, another message is sent to the Runtime UA Stub, this time with the 'hideAdminPage' string as the value of the `method` field. When the Runtime UA stub intercepts this message, resizes the iframe back to the size of the settings button, and the administration page is hidden, which allows the user to see the reTHINK application homepage. The reTHINK administration page manages two separate administration pages, one for policies management and another for identities management. Next we will describe the features offered by the policies’ administration page, presented in Figure 5.7.

To manage the creation of policies through the administration page, two classes were implemented: the Policies GUI and the Policies Manager. The first is responsible for handling the user interactions, i.e., dynamically builds the HTML elements necessary for the presentation and collection of the information, and collects the user input. The Policies Manager receives and organizes that information, which is
then sent to the Policy Engine API and the policy instances. For a unified user experience, we used the Materialize framework for the styling of the administration page, which was already being used for the styling of the reTHINK application. Implementation details about the two created classes will be provided next, as the administration page is described.

5.4.1 Policies management

The policies’ administration page is divided in two parts: the first for policies management, and the second for groups’ management. When the user clicks in the first section header, the list of existing policies and corresponding rules is presented. Next to it, a dropdown enables the user to select the currently active policy, and below a button allows creating a new policy. When the administration page is used for the first time, the list of policies is empty. If the ‘NEW POLICY’ button is clicked, the user is prompted with the modal shown in Figure 5.8. This modal allows the user to input the policy key in the text field, and the policy combiningAlgorithm in the dropdown, which offers the following options: ‘Block Overrides’, ‘Allow Overrides’ and ‘First Applicable’. Since the meaning of these options may not be clear enough for the user, a HELP button is provided, which shows a brief description of the combining algorithms. Consider that Alice uses the administration page to create a policy for her vacations. The title she introduces in the modal is ‘Vacations’, and she does not set any combining algorithm - which defaults to the block overrides algorithm. After introducing the titles and clicking in the OK button, the Policies Manager uses the addPolicy (‘USER’, ‘Vacations’, [], ‘blockOverrides’) method of the Policy Engine to add a policy with the key and combining algorithms provided, but no rules or actions. Clicking the EXIT button will ignore the information provided and will show the policies’ administration homepage.

Alice now creates the first rule for her ‘Vacations’ policy, which blocks all communications on Sundays. To do so, she must click the ‘+’ icon under the policy to which assign the rule, which will show a new modal offering the system attributes the Policy Engine can verify as options of a dropdown. When clicked, four sections are shown: first, a brief description of the implication of the setting in the reTHINK application; second, an input field for the user to be able to input the parameter related to the preference for the chosen system attribute; third, a dropdown for the rule target; fourth, two radio buttons for the user to select if the authorization decision is to allow or block the communication attempts that fall within the configurations provided. The first two sections are dynamically created for each attribute by retrieving the information and fields to present from a hashtable. The dropdown that enables the user to choose the rule target is built by requesting to the Policies Manager, which by its turn requests the Policy Engine, that asks the Identity Module and the Runtime Registry, respectively. The creation of a rule regarding the weekday attribute is shown in Figure 5.9. In this figure, clicking the ‘Select a weekday’ dropdown displays the weekdays, from which Alice selects ‘Sunday’. Then, the Alice must select the target of the rule being created by clicking the ‘Apply this configuration to’ dropdown, which displays a list of Alice emails.
registered in the Runtime Registry, followed by a list of the hyperties names instantiated, and finally the option ‘All identities and hyperties’. If Alice clicks on one of her emails, the scope assigned to the rule is ‘identity’, and the target is the clicked email; if Alices clicks on a hyperty name, the scope assigned to the rule is ‘hyperty’, and the target is the clicked hyperty name. If Alice clicks on the ‘All identities and hyperties’ option, the scope assigned to the rule is ‘global’, and the target is also ‘global’, which is the option Alice wants. Regarding the radio buttons for the authorization decision, if Alice chooses the ‘Allow’ button, the decision field of the rule is true; if Alice chooses the ‘Block’ button, the decision field of the rule is false. Therefore, Alice chooses the ‘Block’ radio button and then clicks on the OK button. The Policies GUI collects the information provided by Alice regarding the new rule for the policy, namely the policy title, the system attribute, the chosen weekday, the target, and the authorization decision. Then, the Policies GUI requests the Policies Manager to do the translation to the policy specification language, which invokes the createRule (false, ‘weekday equals Sunday’, ‘global’, ‘global’ ‘0’) method of the Policy class to add this rule to the Policy Engine. After Alice clicks on the OK button, the modal is closed, revealing the policies’ administration page as presented in Figure 5.10, where the vacations policy has a rule representing that Alice wants to block all communications on Sundays.

As shown in Figure 5.10 the description of the rule is followed by a ‘X’ icon, which allows the user to
delete the rule that precedes it. If this button is clicked, the policy title and the rule priority associated with the rule being deleted are retrieved and a request to delete the rule with that priority value and that key is sent to the Policies Manager. In its turn, the Policies Manager invokes the `deleteRule (rule)` method of the policy with the given key, consequently removing that rule from the Policy Engine. A similar process applies to the deletion of a policy: its title is followed by a trash icon that allows the user to delete the policy. If this button is clicked, the policy title is retrieved and a request to delete the policy with that key is sent to the Policies Manager. In its turn, the Policies Manager invokes the `removePolicy ('USER', key)` method of the Policy Engine, consequently deleting the policy.

To have a policy being applied to future communications, the user can click the ‘Click to active a policy’ button, which will display two options: ‘Vacations’ (which is the policy title), and ‘Deactivate all policies’. If the user clicks on the ‘Vacations’ option, the Policies Manager changes the value of a Policy Engine variable that holds the key of the active policy; if the user clicks on ‘Deactivate all policies’, that variable will be set to undefined, and no user policy will be applied.

The administration page gives to the user the opportunity to change the rules’ priority by using the arrows that precede the rule title: clicking the arrow pointing up increases the rule priority; clicking the arrow pointing down decreases the rule priority. Changing the priority associated with rules is immediately reflected in the order by which the rules are presented in the list. This feature is achieved by assigning a different id to each of the arrows, that when clicked generate a request to the Policies Manager to change the rule priority.

The presented policy management section allows the specification of policies in a very simple way, and at the same time providing a means for the more advanced users to specify more complex policies through the import of policy files. Next, the description of the groups’ management section will be
5.4.2 Groups management

As introduced in the beginning of Section 5.4, the reTHINK administration page is divided in two parts: the first for policies management, which was described in the previous section, and the second for groups management, which will be described next. One of the features of the Policy Engine is to verify if the identity of the received message is part of a group of users to generate an authorization decision. The representation of that condition in a policy is simply stated as ‘source in <groupname>’, for instance ‘source in family’. The Policy Engine is tasked with converting the group name to the array of user emails that correspond to the family user group, and only then the condition applicability can be verified. To make this possible, the policies’ administration page has a section to create and delete groups of users, and to add and remove members from them, as shown in Figure 5.11. When the user clicks in the ‘Groups management’ section header, the list of existing groups and corresponding members is presented. Below it, a button allows creating a new group. If the user clicks in the ‘NEW GROUP’ button, a modal requesting the name of the group pops up. The user inserts the name, for instance, ‘Family’, and by clicking the ‘OK’ button the modal is closed, the inserted group name is obtained by the Policies GUI class, which requests the Policies Manager to create a group with that name. To do so, the Policies Manager invokes the `createGroup(groupName)` method of the Policy Engine, which by its turn creates a new entry in the hashtable that holds the groups of users, setting its value with an empty array.

After creating the group, the user can add members to it by clicking the ‘+’ button placed near the group just created. This will open a new modal with a text field for the email to be added to the group, for instance, ‘alice@gmail.com’. When the ‘OK’ button is clicked, the modal is closed, and the
inserted email and the group name are obtained by the Policies GUI class, which requests the Policies Manager to add the new email to the corresponding group. This is done by invoking the `addToGroup ('Family', 'alice@gmail.com')` method of the Policy Engine, which pushes the new member email to the corresponding entry of the groups hashtable. The groups’ administration section is then revealed as presented in Figure 5.11.

Similarly to the rules of a policy, each member of a group has a ‘X’ icon, which allows the user to remove members from the corresponding group. If this button is clicked in the provided example, the group name and the user email are obtained by the Policies GUI and a request to remove that member from the group is issued to the Policies Manager. To do this, the Policies Manager invokes the `removeFromGroup ('Family', 'alice@gmail.com')` method of the Policy Engine, which removes the that email from the array in the ‘Family’ entry of the groups hashtable. It is also possible to delete a group by clicking the trash icon next to the group name. This triggers the execution of the `deleteGroup('Family')` method of the Policy Engine, again issued by the Policies Manager on receiving the request to delete the group from the Policies GUI.

The presented groups’ management section allows creating groups of users to be used for a more expressive policy specification. It is a very simple and intuitive section that brings significant value for the expressiveness of the policy specification language.

5.5 Summary

In this chapter the implementation description of the three subcomponents of PoliTHINK was presented: the policy specification language, the Policy Engine and the graphical user interface. The policy specification language is organized in a hierarchy of classes that together mimic the JSON format while taking advantage of the definition of methods. The Policy Engine was shown to be built in a way that allows its reuse throughout the reTHINK components, fulfilling the portability requirement of this work. The graphical user interface was also presented, revealing itself a very useful tool to lower the specification complexity of policies in our language, another requirement of the developed system. In the next chapter we will present the evaluation methodology and results for the quantitative and qualitative measurement of the developed system.
This chapter presents the evaluation of the PoliTHINK system. It is organized as follows: Section 6.2 measures memory usage; Section 6.3 measures the loading time of policies from both non-persistent memory and persistent memory; finally, Section 6.4 measures the evaluation time of policies.

6.1 Methodology

Before presenting the obtained results of PoliTHINK evaluation, we will first present the methodology used in the execution of the tests. One of the goals of this thesis was the implementation of a solution that is efficient both in terms of memory usage and processing time. The time it takes to process a policy was separated in two stages: first, the loading time, second, the evaluation time. The tests to evaluate these variables will be performed for an increasing complexity of policies specified on our Policy Specification Language and also on XACML, the OASIS standard for policy specification. To test the loading time of XACML policies, we used an XML parser provided by JavaScript. To test the evaluation time of XACML policies, we built a specific program to read the XML DOM tree in order to extract the information necessary for the verification of the conditions represented in the policy. Each experience was performed a number of times that, by experimentation, provided stable results: 10000 times. To improve the reliability of the tests results, each experience was repeated 11100 times, distributed by 11 runs of 1010 repetitions each. From the 11 runs, the first one was ignored, and for each run, the first 10 repetitions were also ignored. This way, a total of 10000 repetitions were used, while it was possible to discard the warm up periods and equalize the obtained results, further improving their reliability.

The complexity of a policy varies, on the one hand, with the number of rules, and on the other hand, with the size of the condition in a rule. The evaluation of each metric was done for two groups of policies. The first group is characterized by policies composed of 1, 10, 100 and 1000 rules, each holding a simple condition to be examined. The second group is characterized by policies composed of one single rule, with an advanced condition that combines the result of 1, 10, 100 and 1000 simple conditions through logical operators. When the reTHINK application is started, the Policy Engine is not populated with the policies that users or service providers may have specified in previous sessions. To populate it, these policies are loaded from the persistent memory and stored in a local hash table, which is used to retrieve policies for message evaluation in that session. The processing time of the two ways of loading policies was evaluated, as well as the processing time of policy evaluation. These tests were carried out in the two environments where the Policy Engine is integrated in: the Core Runtime and the

---

1OASIS (Organization for the Advancement of Structured Information Standards) is a global organization that works on the development and adoption of standards for security, Internet of Things, and other areas
6.2 Memory Usage

To evaluate the memory consumption, we took into account the fact that the Policy Engine can be integrated into two different environments: the Core Runtime and the Vertx Messaging Node. In both environments, policies can be stored both persistently and not persistently. The persistent way of storing policies in the Core Runtime is done through the Persistence Manager, which uses the Local Storage. It is a storage type that provides a means to store data with no expiration date within the user's browser. The Vertx Messaging Node policies are stored in a file, which is loaded when the first message is sent to the Policy Engine for validation. To understand the impact of loading policies from the persistent memory when the system is booted, we must quantify how many bytes their representation requires. Figure 6.1 presents the number of bytes occupied by policies with 1, 10, 100 and 1000 simple conditions and by policies with 1 rule composed of 1 advanced condition that combines 1, 10, 100 and 1000 simple conditions. Figure 6.1 (a), presents the number of bytes used by the Persistence Manager to store each policy; Figure 6.1 (b) presents the measurements obtained for the policies stored in JSON/XML files. For better readability, Figure 6.1 (c) illustrates the measurements presented in Figure 6.1 (a) in a chart, and 6.1 (b) illustrates the measurements presented in Figure 6.1 (d). This comparison aims to clarify which is the impact of the environment constraints in the policies storage.
Taking into account the results obtained for PoliTHINK and for XACML in both platforms, we observe that our solution is always more efficient than XACML in terms of memory usage. On average, a policy represented in PoliTHINK is 7.6 times more concise than XACML to specify the same behavior. Also, we can conclude that the number of bytes used to represent policies is always higher when using the Persistence Manager, as depicted in Figure 6.1. This is justified by the encoding used in the different environments. On the one hand, the Local Storage used by the Persistence Manager stores the policy as a JavaScript string, which uses UTF-16 encoding, i.e., each character is represented by 16-bit blocks. On the other hand, the default encoding of the Vertx Messaging Node is UTF-8, i.e., each character is represented by 8-bit blocks. For the most verbose policies, namely XACML, the measurements made for the file are approximately half of what was measured for the Local Storage, which clearly reflects the impact of the encoding on each platform. Adding to these observations, it should also be noted that representing the same conditions on a simple policy and on an advanced policy implies a big difference when it comes to the number of bytes needed. While a simple policy with X conditions requires the specification of X rules with fixed fields repeated X times, an advanced policy only requires that these fields are represented once for the rule holding the advanced condition. For this reason, the discrepancy between the occupied memory for simple and advanced policies increases rapidly for 1, 10, 100 and 1000 conditions.
6.3 Policy Loading Time

One of the requirements of PoliTHINK is that it provides an efficient solution in terms of the necessary time for loading and evaluating policies. Similarly to the previous section, this section presents the measurements obtained for the loading time of simple and advanced policies with an increasing number of rules and conditions. Next, we will present the time measurements obtained when loading policies from persistent memory and from the local hash table, starting with the former.

6.3.1 Loading Time from Persistent Memory

The policies loading time from persistent memory is measured differently in both environments where the Policy Engine is integrated. For the Core Runtime, policies are stored using the Persistence Manager; for the Vertx Messaging Node, policies are stored in a text file. To measure the loading time in the Core Runtime, a timer is set using a library that generates the timestamp before the policy is loaded from the Persistence Manager, and a second timer is set right after that request is fulfilled. Loading a policy with a given key from the Persistence Manager consists of requesting the Local Storage to retrieve the value that corresponds to the policy key, which is then parsed to JSON format. Subtracting the two timers gives the time it takes for the policy to be loaded from the Persistence Manager. To measure the loading time in the Vertx Messaging Node, a timer is set before reading the file, which consists on instantiating a FileInputStream and a BufferedReader given the path to the JSON/XML file holding the policy, and then reading the file line by line, concatenating the contents in a string. The measurements obtained for the policies loading time are presented in Figure 6.2, where Figure 6.2 (a) shows the loading time of policies when using the Persistence Manager, and Figure 6.2 (b) presents the measurements obtained for the same policies but when they are loaded from JSON/XML files. Figure 6.2 (c) illustrates the measurements presented in Figure 6.2 (a), and 6.2 (b) illustrates the measurements presented in Figure 6.2 (d).

Taking into account the results obtained for PoliTHINK and for XACML in both platforms, we observe
that our solution is always more efficient than XACML in terms of loading time from persistent memory. The policy loading time is closely linked with the corresponding memory usage, i.e., as the complexity of the policies increases, they imply a higher number of bytes for their representation, and consequently more time to load them. This relationship is illustrated in Figure 6.3, where Figure 6.3 (a) illustrates the memory usage for policies in the Core Runtime and 6.3 (b) illustrates the policy loading time in the same environment. In this figure it is possible to clearly observe the trend of the policies loading time from persistent memory, which follows the trend of memory usage in the Persistence Manager.

When the policies are loaded from the persistent memory to the Policy Engine, they are stored in a local hash table. This hash table is then accessed for each message if there is an active user policy (in the Core Runtime), or if there is an applicable service. Next, we present the measurements obtained when loading the policies from the local hash table.

### 6.3.2 Loading Time from Local Hash Table

A hash table is a data structure that can map keys to values through a hash function in constant time. To avoid the overhead introduced by loading the policies from the persistent memory for each message that requires policy evaluation, the Policy Engine uses a hash table to store the policies that correspond to a given policy key. Therefore, when a message is intercepted by the Policy Engine, the policy to apply is retrieved from the hash table instead of from the persistent memory, which greatly improves the loading time. To examine the performance of loading the policies from the hash table, a test similar to the one performed in the previous section was carried out: a timer was set before accessing the hash
table, and a second timer was set after reading the policy that corresponds to a given key. Subtracting the two timers gives the time it takes for the policy to be loaded from the non-persistent memory. The measurements obtained for this test are presented in Figure 6.4, where (a) shows the loading time of policies in the Core Runtime, and (b) shows the loading time of policies in the Vertx Messaging Node. The charts presented in (c) and (d) correspond to the values provided in (a) and (b), respectively.

Taking the presented measurements into account, we observe that loading policies from the hash table is considerably faster than loading policies from the persistent memory. We also observe that the load time is independent of the stored amount of information and also independent of the platform. On average, loading a policy from the hash table takes approximately 1.75 microseconds. This value corresponds to approximately 10 times less what was obtained for the best case of the persistent memory loading evaluation - which corresponds to the PoliTHINK simple policy with 1 rule. To restrict the overhead introduced by the loading of policies to a minimum, policies are cached, i.e., each policy is loaded only once from the persistent memory when the session starts, and until the end of that session they are loaded from the hash table. After loading a policy, it is necessary to evaluate it in order to generate an authorization decision. Next, we present the time measurements obtained for the evaluation of policies.

6.4 Policy Evaluation Time

A policy is composed of a set of rules, which by their turn are composed of 1 or more conditions, simple or advanced. The evaluation of a policy consists of evaluating each rule on the array of rules; the evaluation of a rule consists of verifying the condition applicability and returning the specified authorization decision if the condition applies. Consequently, the evaluation time of a policy is expected to be increasingly higher when the number of conditions to evaluate also increases. To measure the impact of policy evaluation on both the Core Runtime and the Vertx Messaging Node, a timer was set before starting the evaluation of a policy, and a second timer is set right after the evaluation completes. Subtracting the timers gives the time it takes for a policy to be evaluated. The measurements obtained for the evaluation of the test policies are presented in Figure 6.5, where (a) shows the evaluation time of policies in the Core Runtime, and (b) shows the evaluation time of policies in the Vertx Messaging Node.

Taking into account the results for PoliTHINK and XACML in both platforms, we observe that our solution is always more efficient than XACML in terms of policy evaluation time. Such difference exists due to the means of extracting the information using each language. On the one hand, policies represented in the PoliTHINK Policy Specification Language are objects, and the policy properties are accessed directly. On the other hand, extracting the information from a policy represented in XACML requires traversing the node tree, which is computationally more expensive. Adding to this observation, and as expected, we observe that increasing the number of conditions to be evaluated results on a higher evaluation time. Also, we observe that the policy evaluation time in Vertx Messaging Node is, on average, 1.8 times higher that the policy evaluation in the Core Runtime. Since the Vertx Messaging Node uses Nashorn \(^2\) to embed JavaScript code in Java applications, this outcome would be expected: to execute the JavaScript code of the Policy Engine, Nashorn compiles it into Java bytecodes and runs

\(^2\)Nashorn is a JavaScript engine developed by Oracle.
them on the JVM (Java Virtual Machine) itself. Consequently, the overhead introduced by this process was reflected in the policy evaluation time.

### 6.5 Summary

In this chapter we presented the evaluation of the implemented solution. For the two types of policies, i.e., that are composed of simple or advanced conditions, we studied the behavior of the solution by observing the time it takes to load and evaluate policies while increasing the policies’ complexity. Our implementation revealed itself a more efficient solution for the reTHINK framework when compared to the XACML policy specification language, both in terms of memory usage and processing.
Conclusions and Future Work

7.1 Conclusions

This dissertation presented PoliTHINK, a subsystem developed for the specification and enforcement of policies in the reTHINK framework. First, we provided an overview on the main components of the framework, and also provided a detailed description of the two components of the reTHINK framework where the Policy Engine was integrated: the Core Runtime and the Vertx Messaging Node. Providing the description of the background of this work is important for the correct understanding of the role of the Policy Engine in the framework. Then, a survey on the related work regarding policies was carried out. We detailed the existing theoretical models that base the access control mechanisms, i.e., DAC, MAC, RBAC and ABAC, and provided an overview on security policies on both targets of the reTHINK project, the mobile and the browser environments. We then dived in the state of the art or policy specification languages, where four implementations were described: CSP, WS-Policy, XACML and Ponder. Thereafter, four mechanisms for the enforcement of policies were described: JSand, ConScript, WebJail and XACML.

Taking into consideration the conclusions drawn from the analysis of the described implementations, we presented our solution, PoliTHINK. PoliTHINK is a subsystem that is divided into three subcomponents: a policy specification language, a policy enforcement mechanism (the Policy Engine), and a user interface for a more convenient way of specifying policies for the average user. Our policy specification language provides a flexible means of representing user preferences and service provider constraints. The specification of policies can be done in two different ways: by importing a file with the policy specification in JSON format, or by using the administration page for a more user-friendly option. After their specification, these policies are added to the Policy Engine to be enforced. Policy enforcement is carried out by having a hook on the communication channel, thereby guaranteeing that all messages circulating in the framework are validated against the existing policies.

To understand the impact of policy specification and enforcement on both platforms where the Policy Engine is integrated, quantitative tests were performed. These tests consisted on measuring two important variables, the memory usage and the processing time of policies, which was separated in two: the loading time and the evaluation time. These measurements were made for both platforms where the Policy Engine is integrated, the Core Runtime and the Vertx Messaging Node, and the performance of our solution was compared with the standard for policy specification, XACML. Each test examined the behavior of the two implementations for an increasing complexity of policies, and our solution revealed itself as the most efficient one in terms of memory usage, loading time and evaluation time.

The developed work is currently integrated in the reTHINK framework and is aligned with the requirements defined in this dissertation.
7.2 Future Work

The reTHINK framework is composed of a number of components that can take advantage of policy enforcement. Among these, the most urgent is perhaps the two implementations of Messaging Nodes that were not covered in this work: the Matrix Messaging Node and the Node.js Messaging Node. As future work, the developed Policy Engine should be extended to support these two implementations.

Regarding the existing implementation of the Policy Engine in the Vertx Messaging Node, we should seek a more sophisticated implementation of the policies storage, for instance, by using a database to store and retrieve policies. Moreover, we should solve the incompatibility of the `authorise()` method on Vertx, since that environment does not support the means to execute it due to the Nashorn incompatibility with EcmaScript 2015 used in the Policy Engine implementation. Another possible improvement to add value to the solution is to support the acquisition of the user calendar and also the user location, which are possible to integrate in the Policy Engine by interacting with the Google Calendar API and the HTML Geolocation API, respectively. Regarding the administration page, even if it aims at easing the policy specification for the average user, it would be useful to have the possibility to specify policies with advanced conditions through the graphical interface. This would allow the specification of increasingly complex policies through the HTML elements of the page instead of requiring the user to create a JSON file. With these improvements, PoliTHINK would be more complete and would provide extra value to the reTHINK framework.


