PROYECTO FIN DE CARRERA

Título:	Diseño e Implementación de una Arquitectura de Agentes Basada en Web Hooks
Título (inglés):	Design and Implementation of an Agent Architecture Based on Web Hooks
Autor:	J. Fernando Sánchez Rada
Tutor:	Miguel Coronado Barrios
Departamento:	Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	Gregorio Fernández Fernández
Vocal:	Mercedes Garijo Ayestarán
Secretario:	Carlos Ángel Iglesias Fernández
Suplente:	Marifeli Sedano Ruiz

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

PROYECTO FIN DE CARRERA

DESIGN AND IMPLEMENTATION OF AN AGENT ARCHITECTURE BASED ON WEB HOOKS

Alumno: Juan Fernando Sánchez Rada

Tutor: Miguel Coronado Barrios

Ponente: Dr. Carlos Ángel Iglesias Fernández

Octubre 2012

Tempus fugit,
quo vadis?

Resumen

Esta memoria es el resultado de un proyecto cuyo objetivo es presentar una arquitectura para agentes inteligentes basada en eventos, en concordancia con las nuevas tendencias en la Web de Eventos.

En primer lugar se presenta la motivación del cambio de paradigma propuesto, así como las tecnologías utilizadas en la actualidad en cada campo (desarrollo y comunicación web y desarrollo de agentes). Este análisis lleva a la inevitable conclusión de que es necesario unir ambos campos. Por tanto, se compararán tecnologías que hacen una aproximación a dicha unificación, así como las limitaciones que presentan y los motivos por los que dichas soluciones no unifican completamente ambos campos.

Tras esta visión sobre el estado actual, se describe el análisis de requisitos llevado a cabo para casos de uso típicos de agentes y Web Hooks. Se pretende dar solución a un amplio rango de usos, por lo que se examinan diversos escenarios en los que se hace uso de diferentes servicios.

Con los requisitos anteriores en cuenta, se plantea una propuesta de arquitectura genérica, tanto a nivel externo o de comunicación entre agentes como a nivel interno de organización de información en las bases de conocimiento. Externamente, consta de una pieza central que orquesta el intercambio de información entre las diferentes entidades conectadas a ella. Dichas entidades pueden ser agentes o simples nodos, siendo el único requisito la comunicación mediante el esquema de mensajes definido. Internamente, se exponen unas pautas para el tratamiento de la información recibida. Esta arquitectura, llamada Maia, junto con las decisiones de diseño se describen en detalle en el correspondiente capítulo del documento.

La arquitectura Maia también ha sido llevada a la práctica en la implementación de un prototipo que utiliza Node.js, el entorno de JavaScript en servidor orientado a eventos, para la comunicación entre agentes. Mediante este prototipo se muestran los beneficios de la arquitectura en un caso real. Concretamente en el desarrollo de un agente personal para reserva de trenes que combina acceso a servicios, linked data y razonamiento de sentido común.

Para acabar, se presentan los retos de futuro así como las posibilidades de expansión y utilización.

Palabras clave: Agentes, webhooks, Live Web, Web de Eventos, JASON, hook.io, node.js, JavaScript, Java.

Abstract

This project aims to introduce an event-based architecture for intelligent agents, in accordance with the new tendencies in the Evented Web.

The reason for this change is that agent communication is no longer suitable for the inmense amount of data generated nowadays and its nature. At least, not for their use in evolving scenarios where data sources interact without previous configuration. This is exactly what the precursors of the Live Web envision, and it is beginning to show in the new generation of evented applications, which enable customized interactions and a high level of communication between different services.

The proposed architecture shown in this document, called Maia, is based on a central piece or event router, which controls the flow of information/events to and from the connected entities. These entities can be either event-aware agents or simply data sources and subscribers. Thus giving a higher flexibility than current technologies and easing the development of advanced systems by not requiring the complexity associated with agent systems in all of the nodes.

To demonstrate the feasibility and capabilities of the Maia architecture, a prototype has been implemented which is also explained in detail in this document. It is based on the event-driven I/O server side JavaScript environment Node.js for the event routing components, and adapted Jason BDI agent platform as an example of a subscribed multi-agent system. Using this prototype, the benefits of using Maia are illustrated by developing a personal agent capable of booking train tickets and that combines access to services, linked data and common sense reasoning.

Keywords: Agents, webhooks, Live Web, evented web, JASON, hook.io, node.js, JavaScript, Java.

Acknowledgement

My belief always was that acknowledgement should be a means to inform the reader about the context of the text and the writer, the situation that made it possible. It should not be the place to thank family and friends, as they should receive our gratitude every single day, and not just for a scientific text.

So allow me to thank the Group of Intelligent Systems (a.k.a. GSI) for providing the best working environment possible, and giving me the opportunity to participate in it. This thanks extends to every single member of the group that helped, either with technical support or just being there in the good and bad times. Being in this group for more than three years now has given sense to being enrolled in this university, as we turn our knowledge into real applications while learning a lot and having so much fun. I count is as one of the best things to ever happen to me, and I owe much to many of the members of the staff, professors and students.

Among the people in this department, there are some people I have to mention individually for several reasons. Firstly, César Monteserín, for this written form of my thesis is largely based on his previous research on latex templating and has allowed me to focus almost exclusively on adding content. Your work is amazing, and it keeps surprising me everyday. Once again, you sustain my belief that shared knowledge and open source is key to progress. Secondly, to Carlos Ángel Iglesias, for being a guide during these years not only in work related to the group, but other personal and professional issues. You teach us how to be real engineers, which may pass unthanked but is never in vain. And lastly, Miguel Coronado, who patiently helped me through the project that lead to my final thesis.

A big thanks should go to EESTEC as well, for helping to blur the boundaries between a professional career and a personal passion. I probably would not be the professional nor the person I am today without the experienced gained all around Europe. Most of my gratitude is to one particular committee, LC Madrid (known as EURIELEC in local level), for their members lit the spark that kept me passionate about being in this degree and improving both personally and professionally. From you I learnt most of the good stuff during these five years, and shared most of my time. And I am probably contradicting with you what I wrote about thanking friends, but we've achieved a unique mix of friends and colleagues that I will not easily give up to.

And last but not least, my full respect and gratitude to the Open Source community in all its flavours. It is your spirit that keeps the world turning and progressing instead of fighting over patents and legal disputes.

Agradecimientos

A mi familia, independientemente de lo escrito en el apartado anterior. Y es que según escribía esta memoria, me iba dándo cuenta de que no podía no dedicaros si quiera unas palabras. Principalmente porque si estoy aquí es por vosotros y vuestro apoyo incondicional en estos ya veintitrés años de vida. Porque me lo habéis dado todo sin esperar nada a cambio, pero sabiendo que con la educación que me habéis dado no podría sino darlo todo por vosotros. Porque esta memoria es la culminación de una andadura que comenzó hace mucho más de cinco años, y me habéis acompañado a cada paso del camino.

Pero sobre todo porque sé que no entenderéis una palabra de este proyecto de fin de carrera y aun así estaréis a mi lado en la lectura y os alegraréis incluso más que yo de que termine esta etapa. Lo cual no deja de ser una muestra más de lo que es el amor incondicional. Ese que hace que estemos al lado de los que queremos aunque no entendamos sus acciones o sus sueños, que lo demos todo simplemente por ver a la otra persona feliz, que da color y sentido a una rutina gris y vacía. Quizá sea poco profesional, pero no puedo dejar escapar esta oportunidad de dejar constancia de lo mucho que os debo.

A mis abuelos, por si tienen a bien leer este papel. Sé que es difícil que alguien se preocupe por mí como lo habéis hecho vosotros, y sólo espero haber estado a la altura.

A mi hermana Irene, porque ha sido siempre la cara de mi cruz. Porque sé que no siempre ha sido fácil convivir conmigo, y sin embargo aprendiste a no quejarte. Quizá no lo creas, pero por pocas personas me he preocupado como por ti.

Mamá, papá, puede que sea el primer ingeniero de la familia, pero vosotros habéis hecho algo mucho más grande de lo que la técnica consiguirá jamás: habéis dado la mejor educación posible a dos hijos que os quieren con locura, pese a no haber tenido siempre los medios adecuados. Sé que todo lo que soy os lo debo a vosotros, y nunca podré agradecéroslo lo suficiente. Y es que tener hijos es sencillo, pero para ser padres hace falta sacrificio, coraje, paciencia, tesón... Y vosotros lo tenéis todo.

Hacéis que me sienta orgulloso de vosotros cada día. Sólo espero que hoy podáis estar un poco más orgullosos de mí. Prometo seguir luchando, y hacerlo por vosotros.

Contents

Listings

List of Figures

Chapter 1

Introduction

"Ad initium, ad infinitum"

1.1 Rationale

This thesis is developed as part of the Web 4.0 project. The Web 4.0 Project proposes a new model of interaction with the user that is more complete and personalized, not only limited to showing information but also behaving like a magic mirror that gives concrete solutions for the user needs. It is an integration layer that represents a major step for the exploitation of the Semantic Web and its enormous possibilities. Eric Schmidt, CEO of Google Inc., once said in an interview: "The perfect search engine is the one that only gives one result, which is exactly what you're looking for". This is the basic principle in Web 4.0.

This new Web will be able to answer questions like "I want to be picked up by a taxi right now" through the combination of different techniques like:

- Natural Language Understanding (NLU)
- Context information processing: sentiment analysis, geolocation...
- An architecture of intelligent cloud agents that are capable of communicating among themselves and delegating the answer to the appropriate agent.
- A new model of interaction with the user, not only based on simple lists of results. Agents will even perform concrete actions by interacting with the mobile terminals

the user might be using. For instance, agents could make a call automatically to the nearest cab company, without direct user interaction, using geolocation.

Among the advantages and possibilities of this new model we could note the following:

- A more efficient exploitation of the Semantic Web, which is a highly demanded technology nowadays. Semantic search engines are tools that are being deployed in many systems. Search engines need new interaction paradigms beyond the keywords based searching. With the appearance of SIRI and Google now, natural language is becoming the preferred interaction, since it enables voice interaction from mobile phones. Here semantic representation of the Web is an essential requirement for the scalability of the solution. The semantic Web and, the open linked data initiatives are approaching this ultimate goal. This high demand will create new needs associated with the integration with the users. A new type of communication will be needed to represent information and make accessing it easier for everyone. The possible applications of these technologies are diverse and could start being deployed nowadays.

- Accessibility. Creating new models of user-machine communication will make it possible to bring Internet closer to people that can not understand the usage of a browser and the possibilities behind the Internet nowadays due to its complexity. The aim is to change the way people access information on the net by improving the paradigm of ubiquitous computer interaction formulated by Mark Weiser[1].

- Improvement of the user experience through customized agents. By including personalized agents that interact with the users and learn from them it is expected to improve their experience. The agent interacts with the user by offering custom content, always trying to lessen the difference between what the users want to find and what they finally find. It will also perform tasks for the user by accessing contextual information and communicating to other intelligent agents in the network to offer advanced information to the user that would otherwise need to be searched manually.

- Distributed computing of the information based on agents. The intelligent agents are elements that process information and communicate with each other. For the user the experience is similar to a centralized system. Each semantic agent will be specialized in one or several tasks available through an agent directory. As they are specialized, the requests will be sent to the agent that is more likely to give the right answer to that precise request. This means a very important saving of computational power.

1.2 Goals

In the long term, this project aims to provide an extensible and easy mechanism for communication between disperse agents by means of web hooks. This includes, but is not limited to, server modules that need to cooperate to achieve a common goal. In a bigger picture, this also includes client agents directly connected to the back-end.

Going further into details, the case scenario that will be used is the following: a personal manager in the cloud, accessible through an Android [1] application or an HTML5 [2] site, that will make use of cloud agents to provide the user with information about flights, accommodation and restaurants in response to a query in natural language. For this, we will need the following:

- A communication channel and protocol to connect Clients and Cloud Agent(s)
- A communication channel and protocol to connect Cloud Agents
- A generic and extensible schema for intents and actions
- A scalable platform to deploy Cloud Agents
- To develop all the complex logic needed in every specific Cloud Agent
- To define security constraints and authentication methods

Other general aims of this project are:

- To study and extend the current state of the art of Web Hooks and Web Intents
- To explore the capabilities of such technologies for inclusion in intelligent systems
- To explore and exploit the potential of Javascript in server applications (see Node.js in section 2.2.2)
- To demonstrate the versatility of the given models for communication, beyond programming languages or platforms
- To provide a robust and simple bidirectional connection between Java and Javascript programs

[1] `http://www.android.com`
[2] `http://dev.w3.org/html5/spec/`

1.3 Structure of the document

In order to make it easier for the reader to go through this document, here is a guideline of the contents of each chapter and the connection to each other:

- Chapter 1 gives an overview of the context and rationale of this thesis as well as the relation to the Web 4.0 project.
- Chapter 2 first shows the evolution of the technologies related to this thesis, including the Internet and Agent frameworks. This information helps contextualize this thesis and understand its importance and reason.
- Chapter 3 shows a series of case scenarios that helped shape the requirements of the architecture proposed in Chap. 4.
- Chapter 4 explains in details an Agent architecture, both internally and externally. In this chapter the relationship between components are described, and how the information flows in the system.
- Chapter 5 exemplifies the architecture explained in the previous chapter using Jason (2.3) and Hook.io (2.2.4), and shows a concrete case scenario for it which requires the definition of certain agents within Jason.
- Chapter 6 sums up the findings and conclusions found throughout the document and gives a hint about future development to continue the work done for this master thesis.
- Finally, the addenda provide useful related information, especially covering the installation and configuration of the tools used in this thesis.

Chapter 2

State of the Art

"Dicebat Bernardus Carnotensis nos esse quasi nanos, gigantium humeris insidentes."

— John of Salisbury

2.1 Historical Background

It is certainly impossible to understand the implications and essence of the concepts expressed in this document without fully understanding the nature of the Internet and its evolution, especially during recent years. The following sections try to give an answer to that and to introduce the basis for the rest of the document.

2.1.1 An inherited Internet

Since the birth of the World Wide Web in the form of glowing text in dark screens, we have seen it take over the world in many fields. From commerce to personal communications, including setting up and having meetings, we all have moved a vast part of our activities to the Internet and more precisely to the WWW. Yet the underlying model has remained unchanged for the time being: applications are thought and deployed in a client-server fashion, with the heaviest applications in powerful racks waiting for a request from a client to diligently answer with the correct data. And this model has worked, limitations and all, during the past twenty years. But that is about to change if we want the web to evolve and

see a new kind of applications spread.

The reason for the change of model is, ironically, the success of the Internet itself. We should never forget the beginnings of the Internet that made applications and protocols the way they are now: relatively static and limited data (from email to a company's public information) that was shown to the user on-request. Everyday, more and more users and companies move their activities to the Internet, and experts come up with new ways to exploit the capabilities of the web, so the flow of information is increasing quickly, not only in size but in diversity. The amount of information flowing has become unmanageable for any user, so we can no longer afford manually checking every source any more. To make it even worse, we not only want all the information, but we want it now. If we combine the inner client-server nature of the web with the need for immediacy, we end up with innumerable clients constantly polling a reduced number of servers to fetch new information. And all that information is filtered, so it is usually the case that those responses translate into "Nothing new". The same applies to server-to-server communications.

2.1.2 The Cloud

Over the past years, the popularity of the Internet has given birth to an overwhelming number of online services. Nowadays, we can find online editors for almost anything we can imagine: images, videos, audio, documents... The so-called *Cloud Services*, that make it possible to work seamlessly in any operating system, as long as it is loaded with a modern web browser. Moreover, some operating systems like ChromiumOS turn this browser into their cornerstone. With their the new "everything is in the browser" philosophy, there is no longer the need for local applications, delegating that task to either client applications written in JavaScript or server-side applications. This has also fostered the diversity of devices and terminals with limited processing power but highly connected.

On the other hand, the number of connected devices, those that could be considered clients in the classical server-client conception of the Internet, has reached the millions, being one of the leading factors in the transition to IPv6.

However, the services now being deployed are relatively isolated from one another. They are only interconnected and share information whenever they are provided by the same entity, limiting the potential combinations. In a way, this resembles the classical Unix model, in which every tool has a very specialized aim, only in the case of web applications we lack a connection mechanism like pipes in Unix.

For this reason applications known as mashups came to existence, which put together

different services to offer a more complex behaviour. The downside of such applications being that most of the times they were provided by third parties, needing an extra application server connected to the rest, acting as a proxy to those external services. This behaviour, though practical and interesting, has several disadvantages that have lead to evolving the mashup idea into more flexible and younger technologies.

2.1.3 Evolving the Web

To lessen the impact of some of the flaws mentioned before, and to adapt the web to the modern times and usage, a new paradigm is needed. As there are many different aspects or problems, there are several approaches that can be used. In particular, two currents will be covered in this document. The first one is a low-level mechanism to add synchronism and communication between traditional web servers, through *web hooks* 2.1.3.1. The second one takes a more ambitious goal and brings the new ideas about events and their importance to the web, turning it into *The Live Web* 2.1.3.2.

2.1.3.1 Web Hooks notion

Web hooks are a design pattern for connecting a server and a client. The server instead of offering an API of a server, offers an API for registering a URL which will be invoked when a service is requested. In this way, three kinds of web hooks can be obtained:

Push Those that only inform other parties about an update. For example, the GitHub example explained in section 2.2.1.

Pipe If the aim is to connect two or more services like one would do with pipes in a Unix system. An example would be a hook that connects your Flickr and Twitter accounts, posting a tweet with the tags of your picture and the link; and another hook from your Twitter account to your Facebook account, to post the same message there and upload the picture.

Plugin It is possible to program with web hooks in mind and provide an API that allows other developers to extend your application. As the concept of web hooks evolves, it is expected to find more applications that follow this pattern.

The web hooks allow service providers to expose some of their capabilities to third parties in a simple way. The underlying idea is to delegate tasks within our code, just as we would do with local hooks/calls in conventional programming languages. The difference is that,

while in local calls a different library will process the given data and return the results, with web hooks the data may be processed by a totally different entity, in a completely independent context. Even though this brings several design challenges, it also allows a flexibility never before known.

Conceptually, web services only have a request-based "input" mechanism: web APIs. By using the given APIs the user (that may even be other web service) requests the web application to perform some actions and, most often, it returns some data, e.g. ticket services, travel reservation, weather forecast services, etc.

To those regular services, APIs are enough. However, Internet is becoming a "real-time" service, and mobile application have pushed notification systems to the fore. If the users want to be informed whether the price of the train ticket changed or if there are new tickets available for the concert, they need to continually check the web service for changes (i.e real-time services based on APIs require polling)

However, polling is wasteful and unacceptable for applications that integrate a huge amount of services. Regular web services lack an event-based output mechanism to replace polling, and this is precisely the role of web hooks, since they propose a simple real-time event-driven alternative. Feeds represents another approach, but they do not suffice as the unique form of output for service integration. Strictly speaking they are based on polling, and so the user has to request the data. XMPP also provides a mechanism to avoid polling, and it is an established solution for data streams. However, it is hard to use, heavy weight and it is not suited for dealing with a large number of channels.

2.1.3.2 The Live Web

The term *Live Web* [2], also called *real-time web*, describes a new stage in web evolution that extends the web 2.0 interactive web. The Live Web is characterised by a new way of interacting with the web. Instead of simply browsing static web pages or even interacting with a web site or social network, the Live Web uses *dynamic streams of information to present contextual, relevant experiences to users* [2]. There are several sources of these dynamic streams of information a user can be interested to be notified by, such as social awareness streams, context awareness streams, activity awareness streams and transaction notifications. First, *Social awareness streams* [3] are an emerging lightweight social communication that helps people to maintain awareness of others. Most of the popular social

network sites such as Twitter [1], Facebook [2] or Foursquare [3] provide notifications containing updates about presence, collaborations or actions. *Context awareness streams* are receiving increasing attention in mobile computing since context awareness can enable adaptive applications, context based services as well as power management [4]. *Activity awareness* [5] provides users the ability to be notified about many potential events concerning actions carried out by collaborations in a work environment, such as pending tasks, tasks fulfilled, comments, or notifications in an approval process. Finally, *Transaction notifications* provide users an update of a transaction they have started (e.g. booking a flight or buying a book), so that users know the status of the transaction and of any impediment (rejected credit card, delay in shipping, out of stock, etc.).

The risk of overloading the users with notifications [6, 5] is evident and more and more problematic given the increasing online services users are subscribed to. Then, users are frequently interrupted by notifications and as a result, task performance degrades, and users experience a greater increase in anxiety and annoyance [7].

Thus, there is an emerging class of web applications which require asynchronous processing of incoming notifications, and differs from the requirements of traditional web applications based on the client-server model. As a result, *event-based programming* has emerged as a popular programming paradigm.

In his book, Phil Windley[2] demonstrates the possibilities of The Live Web 2.1.3.2 through a series of examples of web applications that make use of his Kinetic Rules Engine [8]. The reader is encouraged to take a look at the examples in the code repository and explore deeper by reading the book.

It is worth mentioning this, despite not having used any of these tools for this project, as some of the ideas in this document were inspired by either the words of Phil Windley or the applications available on the git repository.

2.1.4 Beyond human capabilities: Agents

Taking it a step further, we can erase human interaction and understand web hooks as a means of communicating and interweaving intelligent autonomous computer programs, or agents. Not only can we allow agents to exchange messages, but also to negotiate the execution of concrete tasks. BDI agents are meant to mimic user behaviour, but there are

[1] `http://www.twitter.com`
[2] `http://facebook.com`
[3] `http://foursquare.com`

have been few attempts to adapt them to match the needs of the evented web users use everyday. The architecture described in this document tries to shorten that gap and to enables agent to make use of the work being done for the web hooks and web applications.

As a result of adapting the traditional agent technology to support web hooks, it will also be demonstrated how it can also support Live Web requirements. On the one hand, agent technology can support Live Web requirements thanks to its distributed nature and its ability to deal with interactions in dynamic and open environments. On the other hand, assistant agents [9] can help users in reducing the information overload by filtering, intermediating and automating the processing of incoming notifications and executing transactions on behalf of their users.

Nevertheless, current agent platforms have not been designed for event processing and are instead focused on agent-based communications. Consequently, there is a need to bridge the gap between agent technology and event-driven Live Web.

The underlying motive of this work is to propose an agent architecture that can be aligned with the event programming architectural style that enables the development of agents for the Live Web.

2.2 Evented Web

The concepts explained in the previous section (2.1.3) are supported by a series of technologies that have either made them come true or have proven their concepts and potential right through a real world implementation.

2.2.1 Web Hooks

As explained in section 2.1.3.1, web hooks are a generalization of hooks in traditional programming in the form of user-defined HTTP callbacks. This section introduces the technical details of web hooks.

As an example, let's take the well-known social code versioning hosting service GitHub. In its website users have an option to set a URL to POST every time they push changes to their repository. This POST message has some relevant information about the event (repository update). This information is JSON-encoded like this:

```
{
  :before     => before,
  :after      => after,
  :ref        => ref,
  :commits    => [{
    :id        => commit.id,
    :message   => commit.message,
    :timestamp => commit.committed_date.xmlschema,
    :url       => commit_url,
    :added     => array_of_added_paths,
    :removed   => array_of_removed_paths,
    :modified  => array_of_modified_paths,
    :author    => {
      :name  => commit.author.name,
      :email => commit.author.email
    }
  }],
  :repository => {
    :name        => repository.name,
    :url         => repo_url,
    :pledgie     => repository.pledgie.id,
    :description => repository.description,
    :homepage    => repository.homepage,
    :watchers    => repository.watchers.size,
    :forks       => repository.forks.size,
    :private     => repository.private?,
    :owner => {
      :name  => repository.owner.login,
      :email => repository.owner.email
    }
  }
}
```

Listing 2.1: JSON template of GitHub postreceive hooks

The receiving end is thus integrated in the process by making it fully aware of the new status of the repository. This example is asymmetric, and the communication is only in one way. In a more complex scenario the emitting party (in this case, GitHub) could wait for the response to this POST and act accordingly. However, for such big services like GitHub it is inconceivable to incorporate this to their workflow as it is prone to attacks and vulnerabilities.

2.2.2 Node.js

Node.js [10] is an environment to execute server-side Javasccript based on Google Chrome's runtime (V8 engine). Due to javascript's event-driven nature, it is the perfect candidate to implement our web hooks architecture.

Moreover, node.js has a boiling community behind it despite being a very new technology[4]. In addition to the uncountable Javascript sources and code that can be found on the Internet and that are compatible with node.js or easily adapted, there are libraries for almost anything one can imagine. This demonstrates the vast possibilities of node.js and the ease of development using either Javascript or CoffeeScript.

Node.js is community driven, and most of the libraries available are open source. Through the Node Package Manager (npm)[5] there are hundreds of packages that can be installed, and their dependencies are automatically resolved.

2.2.3 Socket.io

Socket.io [11] is a protocol for message passing in long lived connections between clients and servers. It is also the client and server libraries that implement the protocol.

Socket.io aims to solve the heterogeneity of available connection methods among modern browsers by providing a transport-independent layer to establish the connection. One of the several transports available are websockets, and that leads to the common misconception that socket.io is a library for websocket connections in node.js. That is not the case, and a generic websocket client may not be compatible with a socket.io server, and vice versa.

Among the several client and server libraries that provide socket.io compatibility, we will be using the official libraries for server (node.js) and client.

An interesting feature of socket.io is that in addition to keeping a connection open

[4] The current stable version is v0.6.13

[5] `http://npmjs.org`

between client and server, it provides methods to: exchange messages in JSON format, acknowledge the reception of a message, emit events and broadcast messages. All of this with the event driven nature of Javascript, which means we can easily filter messages and define different methods for each of them.

2.2.4 Hook.io

Built on top of the socket.io library, hook.io [12] offers a framework to create and communicate sockets. Hook.io hooks are connected to a common channel, so every event is broadcasted to all the hooks using the same port. In fact, they still follow the Client-Server paradigm, and the first hook.io started (or the first one to start listening for connections) will forward all the events and messages sent to it from each hook to the rest. Additionally, events follow a hierarchical structure, which allows the creation of filters for a determined subset of events, using wildcards. And, what is more important, all these calls are made asynchronously, as explained in the previous section.

Except for these two basical differences from raw Socket.io connections, the Hook.io connections do not differ much from them. However, the ability to filter a range of events and sending events to all the other users [6] will be key to evolving into a Web Hook model. The details and the design considerations will be explained later.

The event names are of the form `hook::method::foo::bar`, being the first part automatically added when sending an event (emit(`method::foo::bar`) results in that name for the hook `hook`). To receive it (add it to the reception filters) in another hook, we can either use the full name `hook::method::foo::bar` or any of the wildcarded forms. For example, `hook::**` would get any event coming from `hook`, while `*::foo` would only receive the `foo` events coming from any hook but not `foo::bar` or similars.

2.3 Jason Agent Platform

Jason [13]is an open source platform for developing and executing BDI agents. It offers a framework to develop and deploy agents, using both Java and AgentSpeak. As shown in section 5.5 it is possible to customize the standard Jason deployment to suit the needs of the evented scenario. The following sections contain a deeper explanation of the inner workings of Jason, which will allow the reader to understand the adaptation process.

[6] In Socket.io only the server can broadcast

2.3.1 Architecture

The Jason architecture -either centralized, Jade or SACI- uses the Environment class to represent the simulated environment. That class, stores the information the agent perceives and provides the programmer some appropriate methods to access and modify that information. Usually, an auxiliary class called Model, is involved and it 'truly' represents the environment. It stores the value the sensors measure (in real time) and offers the methods to access the actuators. The Model class is not included into the Jason architecture, so the Environment class must provide the methods to capture the information kept in the Model.

Thus, we can say the Environment class, when appropriate, observes the value of the attributes in the Model class and changes the value of the percepts according to them, i.e. at the sampling instants it checks the value of the model and updates the percepts.

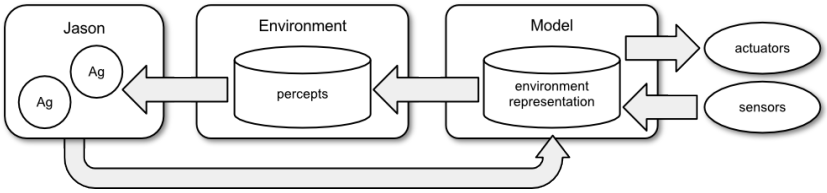

Figure 2.1: Environment and model functionality description.

As shown in Fig. 2.2, the first step in the reasoning cycle is perception, and acting is the last one. Acting may modify, depending on the action performed, the environment so it may modify the perception itself.

The perceiving process (perceive) that starts the Jason reasoning cycle, uses the Environment class to first check if the current perception data is correct. If not, it is updated with the information collected from the model, and provides the Jason agents a list of percepts to change the beliefs base. The detailed operation of this method is explained in greater detail below.

During the *act step*, Jason may execute an internal or external action. Internal actions cannot modify the environment, although they can modify the beliefs. However, external actions can definitely modify the environment, the data stored and so the perception. Specifically, the change made in the act process, is performed in the Model class, that will effectively be transferred to the Environment class -and so to the agent perception.

By keeping the model representation and the perception list separated, it is important to consider which is the best moment to read the model representation, i.e to update the information stored in the Environment. This matter is faced in the percept updating policies section 5.6.3.

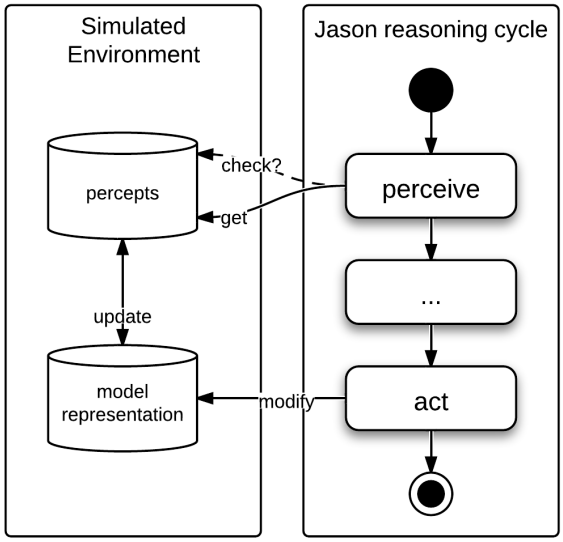

Figure 2.2: Jason reasoning cycle environment interaction

2.3.2 Detailed Jason perception model

In order to study and customize the Jason perception model, is is important to study, in detail, the operation of the Environment main processes. As far as we are concerned, it is composed by three different data collections, as shown in the UML model provided in section 2.3.

- percepts (List<Literal >) stores the general percepts, i.e. those common to all agents, so they will be transferred the belief base of every single agent defined in Jason.
- agPercepts (Map<String,List<Literal») stores the private percepts, i.e this list of percepts will be transferred to the belief base of the specific agent.
- uptodateAgs (Set<String>) keeps the names of the agents whose percepts in Jason system are up to date. So that when the systems request the percept list for any of the agents in the list, it will not be provided due to it is not necessary to change it. This is for performance purposes only.

These lists are accessed and modified through the addPercepts, addPercepts (agName), clearPercepts and clearPercepts (agName) methods of the Environment class. Calling any of the methods listed, as well as the effect listed in the table 2.1, make the involved agents to be removed from the list of up-to-date agents, to point out the Jason's belief base percepts for the agent need to be updated next time the perceive process is executed for the agent given.

Method	Acts on	Effect
addPercepts	Global percepts	Add percepts
addPercepts (agName)	Private percepts	Add private percepts
clearPercepts	Global percepts	Remove all the percepts
clearPercepts (agName)	Private percepts	Remove all the private percepts

Table 2.1: Perception related methods

The most important method, in the perceiving process, is getPercepts. It is called by the Jason Architecture to retrieve the perception list from the environment. Its flow chart is presented in Fig. 2.3.

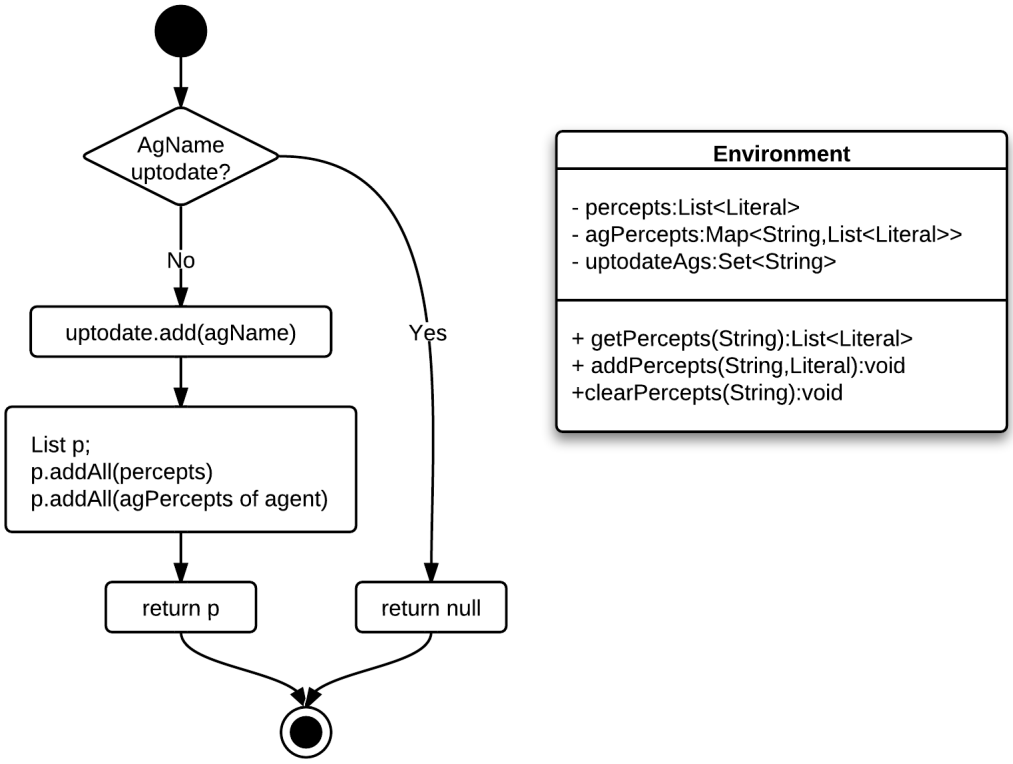

Figure 2.3: On the right the UML Environment class, on the left the Environment getPercepts flow chart.

As can be seen 2.3, getPercepts method checks if the current agent percepts in the belief base are up to date. In such case, no change is made. Otherwise, it combines the information of the common percepts and the private percepts to generate the list of percepts suitable for the agent.

2.3.3 Environment representation by the Model

Despite all this, Model class role in the perception process has not been clearly defined. In fact, the model class has never been named in the figure describing the getPercepts method 2.3, because it actually does not take part in the process. The information transferred to the agents is generated, as shown, from the data stored in the Environment class. However, Environment's information should reflect that contained in the Model class, since it keeps the real time information, measured by sensors, from the real environment.

Thus, the Model role -specially when the environment is particularly complex- is to establish a separation between the real time value represented by its attributes and the samples taken by the Environment class in the sampling instants.

Moreover, the Model class may provide -and it definitely will in this project- a public interface that can be accessed from external systems, platforms and services. Thereby external events can modify the Model without interfere in the Environment class.

Hence, we must point out clearly, the Model contains all the information fetched from the real environment, in real time, and it provides a public interface for external events. On the other hand, the Environment cannot be accessed from any external systems or even from the Model. The Environment class -when appropriate, according to the Jason reasoning cycle- consults the Model to update the percept list that will transfer to the Agent System, taking a sample of the model at a particular time.

2.3.4 Environment representation levels

At this point, according to the description given, we may see there are three different places where the information, collected from the environment, is represented in Jason architecture: the Environment class, the Model class and the beliefs base. The table 2.2 gathers, in summary, the information any of them represents and the way and the time they are modified.

2.3.4.1 Customization and limitations

As it will be shown in the next chapter 5, Jason allows for certain customization when it comes to changing the way beliefs are gathered and stored. It is also possible and usual to define Java methods to perform certain actions not available in AgentSpeak.

However, there are certain limitations associated to the use of AgentSpeak or the static

Environment representation level	Where is it stored?	How is it modified?	When is it modified?
Real time representation of the environment.	In the model class	By agent external actions o external events.	At any time.
Capture of the environment information taken at a particular time.	In the Environment class	By the addPercepts and clearPercepts methods. The Environment itself accesses the Model to capture the information.	Depending on the sampling policy (see section 5.6.3).
Available information for the agents	Belief base	It is modified by the Jason architecture.	In the perceive step of the Jason Reasoning cycle. They are only modified if the Environment class indicates they must be updated.

Table 2.2: Summary of different representation levels

typed nature of Java. Some of them are described in the process of adaptation in Chap. 5, especially in section 5.6.1.2.

2.3.5 SPADE

Simply put, SPADE (Smart Python multi-Agent Development Environment) [14] is an agent platform based on the XMPP/Jabber technology. This technology offers by itself many features and facilities that ease the construction of MAS, such as an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on XML, just like FIPA-ACL. Many other agent platforms exist, but SPADE is the first to base its roots on the XMPP technology.

The SPADE Agent Platform does not require (but strongly recommends) the operation of agents made with the SPADE Agent Library. The platform itself uses the library to empower its internals, but aside from that, you can develop your own agents in the programming language of your choice and use them with SPADE. The only requirement those agents must fulfil is to be able to communicate through the XMPP protocol. The FIPA-ACL messages will be embedded in XMPP messages.

2.3.5.1 Architecture

SPADE's framework consists in a server side that provides the XMPP message routing capabilities (i.e., XMPP server) as well as two main platform components for FIPA's Directory Facilitator (DF) and Agent Management System (AMS). Other services like a web interface, and proxies to support other MTS are also present, but are out of the scope of this project.

Once a SPADE platform is deployed, agents connect to it in the usual fashion for XMPP communications. From that moment on, agents communicate with each other through the server, using the DF or AMS components whenever they are necessary.

However, the SPADE agent library can be used to implement agents without the need of a full blown dedicated server. That is especially attractive considering the vast number of XMPP server available right now, some very popular and in use by millions of people. When used like this, the aforementioned components can either be connected to the server as normal agents, or be left out if their services are not necessary.

In the first prototypes of this system, different multi agent system platforms were evaluated. The most promising of them being SPADE, as it includes the XMPP protocol in its core and many of its communication features. SPADE also allows connecting to users by

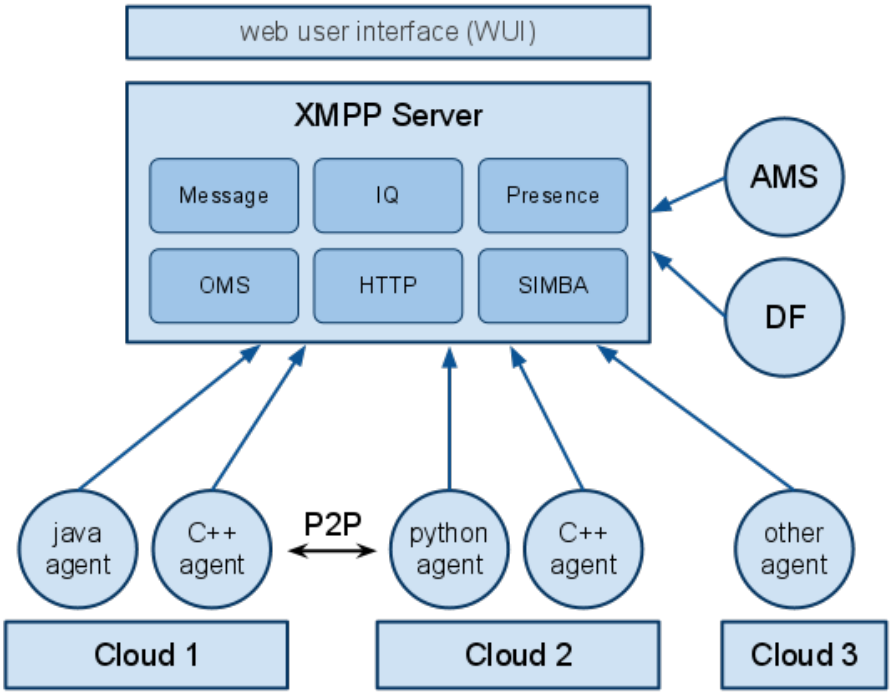

Figure 2.4: SPADE's architecture

using a common XMPP server. However this means losing many of the features provided by the server side components of the platform. In other words, we are constrained to the capabilities of the SPADE library for stand-alone agents. Even though it should be theoretically possible to use SPADE's built in XMPP server also for users, it proved to be hard.

Some attractive features in SPADE were also the ability to create communities (under development), a set of pre-set behaviours (one-shot, finite-state-machine...), its use of FIPA and being fully written in Python. But most importantly, it had the XMPP protocol's advantages: publish-subscribe mechanism to allow push updates, form-data to manage workflow between user, libraries for many programming languages and platforms, etc. It also has an event-based nature, but unlike our final proposal, it relies heavily on authentication against a server and point-to-point messages. Even though there are some solutions to add auto discovery and broadcasting/multicasting capabilities, they are not part of the basic XMPP messaging functionalities, and thus are not supported by all libraries.

Our final solution bypasses all those deficiencies or rough edges for our ad-hoc implementation and probably for a wider set of uses, while providing both a much more flexible treatment of events and a rules engine. Unfortunately, security, authentication and limiting the scope of certain components are aspects that need to be improved to equal the XMPP alternative as it will be discussed in more details in Chap. 6.

2.3.5.2 BDI Model

Michael Bratman's BDI agent model is based on 3 basic elements: Belief, Desire and Intention (you know, B-D-I).

Believes are entries in the agent's knowledge base. It represents what the agent "knows" about its environment.

Desires represent that which the agents wishes to achieve. The instantiation of a Desire is a Goal.

Intentions represent the deliberative state of the agent, that is, what the agent has decided to do in order to achieve its goals. The Intentions in execution are Plans, that is, a sequence of actions that the agent can execute in order to fulfil an Intention.

The BDI implemented by SPADE is slightly different. On the author's words: *We have tried to develop a distributed BDI system using Service Oriented Computation (SOC) together with dynamic compilation of services in SPADE, which we have called Goal Oriented Computation* [14].

This BDI model uses the same basic principles of the classic BDI model from Bratman, but it introduces a more complex model for the Intentions. In SPADE's BDI model, the following elements appear:

Belief: An entry of knowledge in the agent's Knowledge Base (KB). You can insert new knowledge, delete it and make queries to the KB about it.

Goals: SPADE's BDI model does not differentiate Desires and Goals, there are only Goals. When an agent expresses a Goal, it means that the agents wishes to accomplish the expression contained in such Goal. When a goal is selected for accomplishment, it becomes active.

Services: Given that SPADE's BDI model is grounded on SOC, it is necessary to include a Service element into the model. A Service is a method offered by the agent to the rest of the system agents. Services can be composed into a sequence forming the Plans. Services have in their description both a pre-condition (P) and a post-condition (Q). The pre-condition P represents a state of knowledge that must be present in order to execute the Service. The post-condition Q represents the state of knowledge that the agent will achieve once the Service has been invoked.

Plans: A sequence of Services that work their way to achieve the agent's Goals. Agents reach their Goals by executing Plans. Such Plans are composed of a sequence of Services (with their pre-conditions and post-conditions connected) and both a pre-condition P and a post- condition Q that define the whole Plan. Whenever a Goal G must be achieved, the agent will look for a Plan whose post-condition Q equals G and also whose pre-condition P is present as a pre-requisite for the Plan to start. But, wait for this, the Services composing a Plan's actions do not necessarily belong to the same agent. There is where the distributed nature of SPADE's BDI model comes into play.

2.3.5.3 Limitations

For the most part, SPADE is a great platform of choice to build multi-agent systems. It comes with an integrated server which once properly configured makes creating a complex system a breeze. However, due to a big number of issues and a fundamental difference in approach, SPADE is not a valid solution for the purposes of this project.

To be more precise, here's a list of reasons to discard SPADE for our intents:

Embedded XMPP server Despite being an advantage a priori, the embedded XMPP server turned out to be incompatible with regular users in an easy way. Even when the configuration was done to allow it, errors occurred. A possible solution is to use any regular XMPP server to connect agents with other users/nodes, but this way most of the additional features of SPADE were lost.

Subscription persistence Connected to the previous point, subscriptions are reset in every execution of the SPADE server, making it necessary to re-subscribe with the old username and password.

Knowledge Databases even though SPADE can connect to many different databases and be set to process different query languages, the support is not as advanced as in other tools like Jason.

Need for subscription Unlike the solutions like hook.io, using SPADE to communicate agents means subscribing them to a SPADE server and using the FIPA DF features to find other agents with certain capabilities. Although robust, this mechanism forces nodes to subscribe and expose their capabilities, and to search for other nodes specifically before connecting to them. Although this feature can be mimicked using multi-user chat rooms in XMPP, it is a fairly complex solution that bypasses many of the

rest of SPADE advantages, making them unnecessary.

Interoperability beyond agents Making SPADE agents exchange information with other XMPP clients required either to wrap FIPA ACL in XMPP Data-Forms [15], in an arduous process, or communicate using regular IQ stanzas in XMPP, again disregarding the agent oriented side of SPADE.

XMPP One of SPADE's strengths is at the same time a weakness for our scenario, in which services are meant to connect easily to one another. Adding an extra layer like XMPP, especially taking into account the low maturity of some of the advanced features in the current client libraries, is an ultimate reason not to use SPADE.

2.3.5.4 Customization

As explained before, SPADE is a complete framework more than just a library for agent creation. Since it is written using Python, all code is available for download and inspection. This, together with a permissive license, opens the door to any kind of modification. As a matter of example, during the testing phase of SPADE for this project, a new subscription method based on regular expressions was added to the agent library. This change was submitted to the developers, who added it to their following release. After this testing phase, other changes have been submitted by other developers and there's activity in the developers mailing list.

Apart from the expansion and license of the code, the behavioural component of every SPADE agent makes it easy to execute arbitrary python code following certain patterns of repetition. The code exposed to the programmer follows the evented paradigm, with control loops behind the curtains to control the acquisition of messages and execution/triggering of behaviours.

2.4 Personal agents

Bots and agents are designed to develop some particular tasks in a highly efficient way, hence the user is released from doing them. Intelligent agents that can aid a human in managing and performing complex tasks in an office desktop setting. Historically, the nature of those tasks is repetitive and the nature of the date they work with is well-structured, so the agents can easily manage it. However, nowadays much more ambitious intelligent agents are designed. The overall goal is to reduce the amount of effort required by the human to complete the tasks he intends. Agents cover many different topics and are applied to

a-Learning, e-Commerce, e-Health, information searching, travel planning, etc. Intelligent personal agents are focused on the user, on providing what he needs even before he requests it, and all about it is done by means of a proactive intelligent context analysis.

Before the rise of Social Networks, a personal agent was understood as a program that helps the user on organizing its files and emails, that checks its agenda and reminds him about upcoming meetings, etc. Social Networks paradigm shifts the concept of user profile to a new level. The behaviour of personal agents is based on user's profile, and not only the profile the agent may guess from user's interactions but the profile they may read from the user's social profile and his social networks activity. Connectivity is essential, personal agents connect to the user's agenda, his social network account, his email inbox, even his current presence and location and behave according to the information they retrieve.

The architecture of personal agents is changing too. Fast evolution of hardware capabilities in conjunction with fast wide area communications and availability of virtualization solutions is allowing new operational models for information technology. The advent of Cloud computing resulted in access to a wide range of services and personal agents can feed from those services to provide the user a better assistance.

Theories of collaborative problem solving clearly relate to the notion of proactive assistance: user–agent collaborative activity can be viewed as one aspect of task management. For the most part, these theories extend BDI models of agency to incorporate notions of joint beliefs and commitments. For example, Joint Intention theory [16] formalizes the communication acts between agents to establish and maintain joint belief and intention. SharedPlans theory specifies collaborative refinement of a partial plan by multiple agents, handling hierarchical action decomposition and partial knowledge of belief and intention.

There has been some recent work on agent technologies to aid people with cognitive disabilities in managing their daily activities [17, 18]. These systems monitor a person's actions to understand what he is doing, and interact when appropriate to provide reminders, situationally relevant information, and suggestions to aid in problem solving.

2.5 Personal Agents Taxonomy

When classifying personal agents into a fixed taxonomy it is needed to point out the criteria to be used. As ordering the agents based on their architecture is not the objective of this document, and cataloging them according to the resources they use is not big deal, the following taxonomy is made according to the task they develop. As shown in the use cases

description, those Multi Agent Systems tagged as Personal Agent Systems perform many different tasks in many different scenarios. Besides, systems do no consist of a single agent performing just one task, but of many different agents that communicate to achieve a big set of complex goals.

According to the task they perform, Personal Agents can be classified into five main categories described below. Some authors suggest an additional category named *Decision-making Agents* [19] that help user to take a decision. However, this category is too general since some other categories could be classified as decision-making.

2.5.1 Personal resources and information management agents

Personal Information Management (PIM) [20] refers to users' activities in acquiring, organising, retrieving, and processing information in their personal spaces. This category includes all those agents that handle personal resources. This is done in three ways:

- Taking care of the user's items that tend to be shared, such as books, CDs, DVDs or even toys, so that the user knows who's borrowed or given them back. This includes collection management.
- Organising Files, Web bookmarks and email [20]
- Keeping track of commercial transactions, providing the user with detailed information about his budget, debts and investments.

2.5.2 Purchasing and Trading agents

Buying agents's objetive is assisting the user in the shopping process by showing the user the best item to purchase, according to the search criteria he has achieved. To do so, they perform an exhaustive search in many shopping portals so they find the best price, because in many cases the cheapest match is the best option. Moreover, a user profile is developed to select the items to be presented as the best option, as recommendation systems do. Thus, in some cases being a trustworthy seller or providing a fast delivery may take precedence over price. However, purchasing agents do not make decisions, as the user prefers to select which item to buy. They simply present the user with a set of carefully selected items to choose from.

In this category, travel agents are the most common example, since the price of the trip is the prevailing factor when travelling, as opposed to other kinds of sale.

On the other side, we find the trading agents. This is a more specialised kind of agent, which does not need to deal such a segregated market as the former type. Nonetheless, they have to face the inner complexity of the stock market, and come up with the best investments.

Different approaches are being studied right now, being the latest those that involve social networks and by means of data mining try to guess the future changes in the market.

2.5.3 Task and Time management agents

This section covers two key areas: time management and task management. *Time management* [21] refers to the process of helping a user manage actual and potential temporal commitments. *Task management* [21] involves the planning, execution, and oversight of tasks. These tasks may be personal in that they originate with the user, or they derive from her responsibilities.

The aim of this type of agents is to get the most out of the user's time, relieving the user of routine tasks. Theoretically, these agents organise the tasks and duties of the user in order to make him more efficient. Task management agents fetch information about every task -its deadline, the priority, the workload...- and use optimization algorithms to generate possible schedules. Usually, this process has access to the personal network of the user group. If so, it can be analysed and the valuable data is used to improve the performance of the agent.

Despite being the most common, they have not been successfully implemented yet. This is due to the large number of factors taken into account when planning the schedule, the need for such planning to change based on contingencies that may arise, and the most important: people have their own standards for judging and scheduling [19]. For example, one may prefer to perform smaller and easier tasks at first and fail to perform important tasks because they tend to be bigger and more difficult.

A critical point in every user's working agenda are meetings, since they are very important and common events. Therefore, many time management personal agents focus on managing commitments and collaborative decision making [22]. They are designed to enhance user participation in a meeting through mechanisms that track the topics that are discussed, the participants' positions, and any resultant decisions, providing tools that improve decision making.

As shown with the examples given, personal task manager may also fall into this category, as they handle the own user's agenda or adapt the agendas of a group of people: working group, the members of a family...

Recently, *social task networks*[23] have been proposed as a social approach to task management, taking into account that tools such as to-do items usually show relationships among users' tasks, their social network and web resources.

2.5.4 Reminder agents

Those that check the user's calendar in order to remind him about important events. Some different implementations of this type of agents may be found according to the nature of the events they work with. Google Calendar is a great example of reminder agent since it tracks all the meetings, task deadlines and in general every event the user introduces and allows him to set custom alerts for each of them.

Other implementations make use of artificial intelligence techniques to determine what action should be performed, depending on the nature of the event. For instance, a meeting alert usually notifies the user some time before it starts, to guarantee he is attending, while birthdays are notified some days before so that the user has enough time to buy a present. Besides, depending on the locations of both user and event, the reminder will appear soon enough to allow the user to attend the meeting on time. This may cause the reminder to come out some days before the event if it involves long-distance traveling. For all this, intelligent personal agents require collaboration with many other personal agents in order to track the user's current location, to estimate the driving time...

Reminder agents are not classified as time managements agents since they are not responsible for structuring the work and meetings in the user's calendar but just for monitoring it and reporting about important events. In fact, Time Management Agents generally implement Reminder agents' tasks.

2.5.5 Recommender and filtering agents

Recommender agents are an important and widespread type of agents. Their work is to study personal preferences to make decisions on that basis in order to get user's satisfaction. The key issue in making recommendations is extracting the user's profile [24]. These agents appear on the Internet associated with other different agents, since almost all types of agents mentioned above use a recommender agent as well for some purpose. Some different recommender agents according to their application domain are e-commerce, e-mail filtering or web, movie, travel and news recommenders...

The difference, if any, between the recommender and filtering agents is that the latter

do their work without the user noticing. While recommender agents make suggestions to the user while he's surfing the Internet or buying in a e-commerce store, filtering agents just select the information to be displayed, discarding the presumably unimportant data. It is important to notice that filtering agents do not display additional information to the user, but they convert the information the user requests based on its profile. This is the case of e-mail spam filters, or the modern intelligent filters that sort e-mails according to their importance.

Within filtering agents, *intelligent filtering metasearch agents* [25] can be particularly interesting for the Web4.0 project. This kind of agents carries out the following tasks: (i) integration of data sources, which involves selecting the potentially relevant sources and integrating the result sets; (ii) rating of the obtained integrated result set; and (iii) optimisation of the result set. In order to rate the result set, different filtering strategies can be combined, such as content based rating based on the similarity between the user query and the documents, determining if the document fits with a domain relevant to the user, determining if the document matches individual user preferences based on last rated documents or collaborative filtering and spam filtering.

2.6 Personal agents Behavior

Proactive behavior is also seen as an essential characteristic of autonomous and semi-autonomous agents [26]. As said, the goal of personal agents is helping the user on completing a task. Agents may aid him directly by performing tasks on his behalf or in conjunction with him [27], and indirectly through actions such as providing context for her work, minimizing interruptions, and offering suggestions and reminders [28].

Consider a Personal Agent for Task Management, the following list includes some of the selected possible activities that an assistive agent might perform on behalf of its user to support task management in an office setting. We divide the list into four categories: Act directly, Act indirectly, Collect information, and Remind, notify, ask:

- Act directly
  - Perform the next step or steps of a shared task
  - Perform or prepare for future steps of a shared task now
  - Initiate the first step of a shared or agent task
  - Suggest (shared) tasks the agent can take over and perform

  - establish a learning goal (i.e., to learn new capabilities)

- Act indirectly
  - Suggest a user task be delegated to a teammate, or that the user offer to take on the task of a teammate
  - Suggest a meeting be rescheduled
  - Suggest a lower-priority task be postponed to free resources
  - Suggest a task be promoted or demoted in priority
  - Suggest (better) ways to achieve a (shared) task
  - Anticipate failures of (shared) tasks and look for ways to reduce the failure likelihood or the impact of failure

- Collect information
  - Gather, summarize information relevant to a user or shared task
  - Monitor the status of tasks delegated to a team-mate
  - Monitor and summarize resource levels and commitments
  - Analyze possible consequences/requirements of a (shared) task

- Remind, notify, ask
  - Remind of upcoming deadlines and events
  - Remind of the user's next step in a shared task
  - Ask for feedback or guidance from user
  - Ask for clarification or elaboration of a (shared) task
  - Monitor and filter incoming messages

Proactive behavior for an agent may be split into two types. The first type, which we call *task-focused proactivity*, involves providing assistance for a task that the user either is already performing or is committed to performing; assistance takes the form of adopting or enabling some associated subtasks. For instance, Task-focused proactivity behavior collects background information in support of a scheduled meeting.

The second type of proactive behavior, which we call *utility-focused proactivity*, involves assistance related to helping the user generally with her set of tasks, rather than contributing directly to a specific current task. An example of this type occurs when and assistant takes the initiative to recommend transferring a paper review task in response to the detection

of high workload levels. This action is triggered not by a motivation to assist with any individual task on the user's to-do list, but rather in response to a higher-level motivation (namely, workload balancing).

Consequently, when an agent with Proactive Behavior performs a task without having been instructed to do so the efficiency the user perceives is clearly superior. Apparently, the assistant completed its task instantly, even when it lasted quite some time. On the other hand, it is possible that the work done by the agent is not valuable, but its availability without having been requested gives a certain added value.

The point is to guess what user needs, depending on the context: you live your life, and the agent thinks/works in the meantime [29].

A proactive assistive agent must have an explicit model of user desires, in addition to current user goals and plans, as well as a theory that defines how those can be furthered by actions that the agent is capable of performing.

2.7 Personal agents shift to the Cloud

People are used to living in the Cloud [30], as it provides what the users need: access to their stuff in any place where they have access to the Internet. Web mailbox took the first step towards this ubiquity, its use spread to most of the users as its storage capacity got increased. Subsequently, the storage of files, photos, videos, and then the office-documents of all kinds shifted to the cloud. Online editor like GoogleDocs [31] also allow "editing in the cloud" and within a collaborative environment. Shops, games and services also became ubiquitous. Finally the processing capacity has also become ubiquitous, the cloud is the most powerful computer you have [30].

2.7.1 Services on the Cloud

The Cloud environment can be addressed in several forms IaaS (Infrastructure as a service), PaaS (platform as a service), AaaS (application as a service), etc. Whatever the term used, it refers to the services. The Cloud hosts thousands of services and the agents must be able to access and use them. As shown in Fig. 2.5 the available services available cover different appliance fields such as storage, search, weather forecast, billing, etc.

When the service provider is getting some profit from furnishing the service, he pays special attention on making the service available. The normal way is to offer a private API

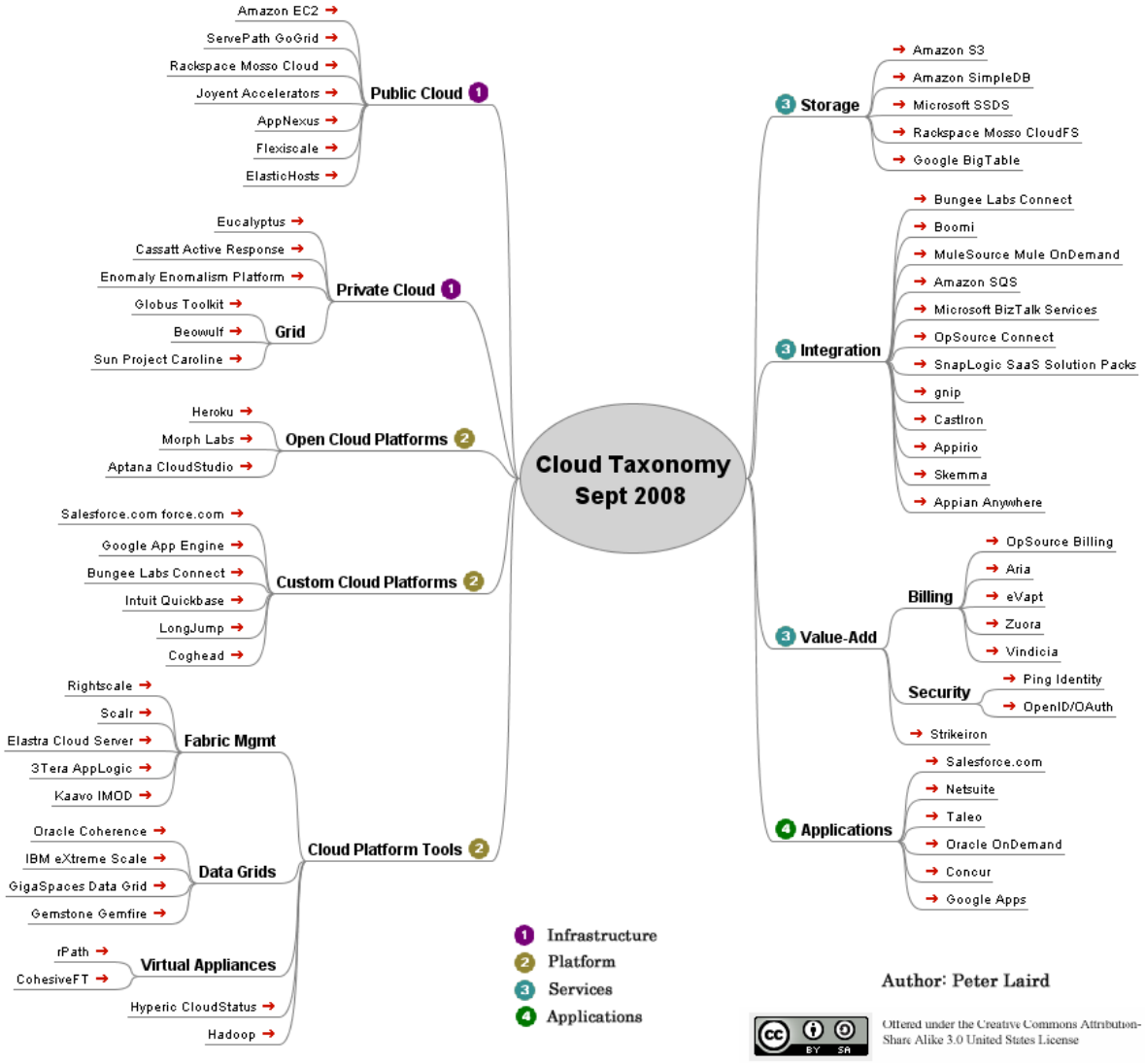

Figure 2.5: Example map of services in the cloud (Source 2008)

for accessing the service whose terms of use are clearly established. The API connectivity amplifies the power of services. The term private means that the API is defined by the service provider, that reserves the right to change it. This is a potential impediment to the development of systems that make use of all the services in the cloud, having to make an adapter for each API.

Within this landscape, OCCI (Open Cloud Computing Interface) [32] consists on a boundary API that acts as a service front end offering a clean, open specification and API for cloud services. Those services that provides a OCCI can be easily accessed by a common OCCI client (as shown in Fig. 2.6).

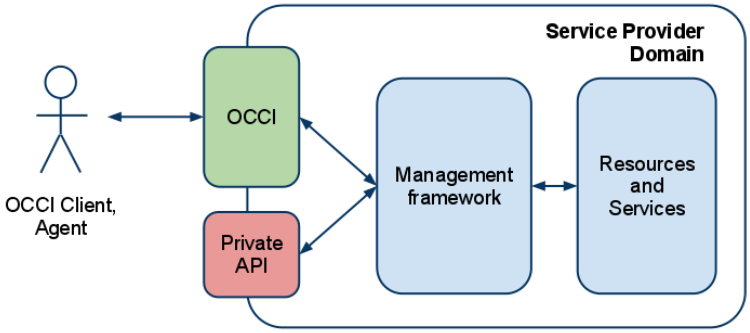

Figure 2.6: Open Cloud Computing Interface Architecture

2.7.2 Smartphones, Smart interfaces

Thanks to the paradigm of cloud, personal agents no longer need to be run on machines with processing capacity. Instead, they can focus on delivering better interface features, improved user experience. In addition to the library of available services, this is the best improvement the cloud offers to personal agent systems. Smart phones are the new computers. Their main task is to communicate with the user, to potentate his expressiveness and to understand correctly what he requests. Smartphones provide multiples capabilities compared to a conventional PC.

- Being wireless devices, they will be available for the user to employ them in the right moment he needs them. All of it without sacrificing the Internet connection.
- Their processing capacity is comparable to one-decade-old computers. However, they focus on the interface, so they are not supposed to evaluate complex algorithms. Hence the available power is more than enough.

- They are equipped with multiple sensors, displays and devices (multi-touch screen, GPS, accelerometer, multiple cameras, etc.) to provide an enhanced user experience.

The aim is to achieve a high degree of fluency in the interaction with the user. This involves a bunch of technologies that are not fully available nowadays. Speech recognition and natural language understanding are two examples. However, this only slightly slows the progress towards achieving personal assistant.

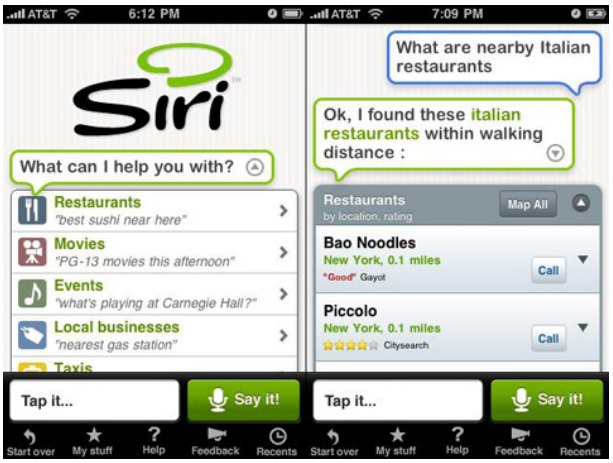

Figure 2.7: Smart interface of Siri

2.8 Case studies

This section introduces two well-known examples of personal assistants. CALO, *the Cognitive Assistant that Learns and Organizes*2.8.1, and the personal assistant Siri 2.8.2 are probably the most advanced personal agent at the present time. The way they approach to the solution is quite different. CALO focus on the proactive way, so it suggests the user information and actions even when him does not request CALO's assistance. Siri is centered on the Smart Interface, on providing the user what he exactly wants, taking care over the user experience.

2.8.1 CALO

CALO [33, 34, 35] was the most ambituous artificial intelligence project in US history. It lasted five years and brought together more than 300 researchers from 25 top university and commercial research institutions.

CALO provided six high-level functions [34]:

- Organising and priorising information from available sources (web, email, PIM, etc.).
- Preparing information artifacts. Assessment on creation of new documents based on previous documents.
- Mediating human communications. Provide asssitance in human meetings.
- Task management. Automatisation of routine tasks.
- Scheduling and reasoning in time. Learn preferences and schedule task completion.
- Resource allocation. As part of task management, learn to acquire new resources.

2.8.2 Siri

Siri [36, 37] is a virtual assistantsoftware developed by Tom Gruber and derived from the SRI's CALO project [33].

Siri is focused on helping users to complete tasks in their online lives, particularly in the mobile context. Siri is limited by design, it is scoped to a few vertical domains such as booking dinner reservations, buying movie tickets, getting local information, or finding things to do in the area (Fig. 2.8).

Figure 2.8: Assistance provided by Siri

The main purpose of Siri is shifting the interaction paradigm for the web from search to assistance [37]. The search engine interaction requires the user to express their intent as search keywords, and it returns a set of links to matching information sources. Instead, the assistance paradigm requires the user to express their intent in a conversation, as a request or goal statement, and the assistant requests more information if needed and guides the user through the process of exploring options and making a choice.

The main innovations of Siri are [37] a conversational interface which an interactive dialog, semantic auto-complete and service delegation.

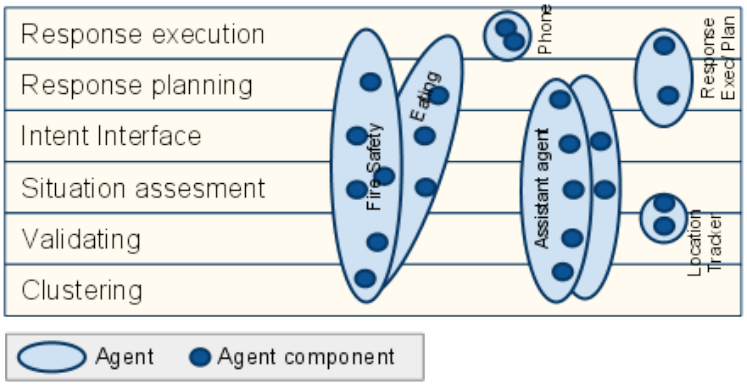

Figure 2.9: Agent component-base architecture

2.9 Architecture discussion

As features, capabilities and functionalities, the architecture of the personal agent systems is evolving over the past years. The concept of architecture that is most widespread is task-focused architecture, but this trend is moving towards a user-focused architecture.

2.9.1 Agent-component-based architecture

A few years ago, the popularity of multi-agent platforms such as JADE [38] unleashed a rapid increase in multi-agent systems. Architecture of these systems are focused on the component (Fig. 2.9). The mayor issue that arises from this architecture is how to handle multiple of a particular agent or component. There may be cases when we want to create a generic set of agents everyone can use, multiple agents needs the same agent-component and we do not want dozens of identical agent-components running at the same time, a specific vendor requests a specific capability or a specific vendor wants to sell a new improved agent or agent-component. This architecture had to handle the resolution and discovery of these capabilities [18].

There are even patents protecting this architecture [39].

2.9.2 Extended BDI architecture

The triumph of the architecture of CALO (Fig. 2.10) is it introduces the concept of proactive behavior, a step towards artificial intelligence. CALO agent focuses not only on solving the tasks marked for the user but to find out what other tasks that can help to solve, and acts accordingly, making suggestions, providing information, etc.

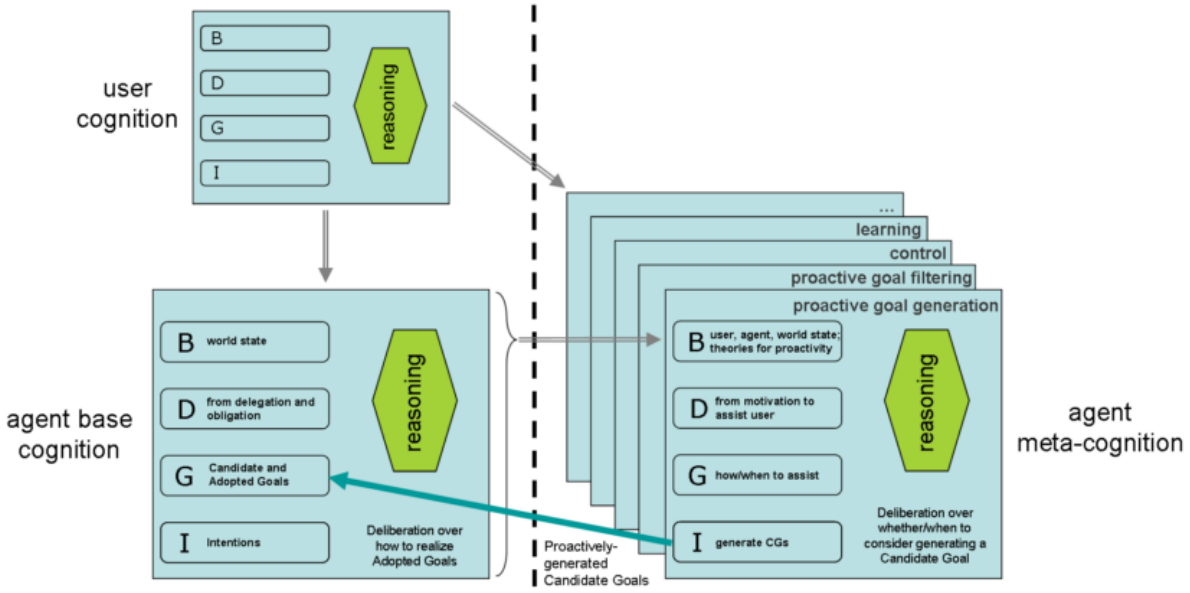

Figure 2.10: Extended BDI agent architecture for proactive assistance as defined in CALO

Fig. 2.10 depicts proactive goal generation in an extension of the delegative BDI agent architecture. As usual in a BDI formulation, the agent's base-level cognition reasons about how to realize Adopted Goals as intentions. Multiple forms of meta-cognition are depicted to the right. In addition to the usual BDI meta-cognition over aspects such as agent control—for example, over goal selection— it adopts proactive goal generation and filtering, an extension to the prior delegative BDI model. A personal assistive agent can be thought of as holding an overarching meta-desire of being a helpful assistant to its user. One desire might be to learn (although one could construe this as the agent bettering itself in order to become a better assistant). Indeed, in principle, the majority of an assistive agent's desires — or at least goals that might arise from them—can be considered as consequences of the overarching high-level meta-desire.

Candidate Goals (CGs) are created through two mechanisms. At the base level, they arise from the agent's motivations to achieve tasks delegated by the user. At the meta-level, CGs are generated proactively as depicted, as a result of deliberation over a theory of proactivity.

2.9.3 User-focused architecture

The architecture proposed in Siri [36] represents a change of concept. In functionalities design, Siri proposes a user-centered system and this extends to architecture design. While other agent systems focus on execution of tasks to achieve goals, the user-centric architecture focuses on the interaction with the user, understand their desires and then fed into the system and act in consecuence. The user experience becomes important, hence the system's intelligence is based on promoting the expression of the user. As mentioned before, terminals become Smart as focused on user interaction.

As Tom Gruber points out in [40] the assistant paradigm for human-computer interaction focus in task *completion*, intent understanding via conversation in context and learns and applies personal information.

The architecture centered on the used (Fig. 2.11) distinguishes how the intelligence lies in two elements: the smart terminal, which focuses on the user and transmits the information obtained / deduced intelligence in the cloud. The latter plan the resolution of the task, you understand, sorts and find a solution that perfectly fits the user's request. This is awarded of the services that are available in the Cloud so it used them when needed.

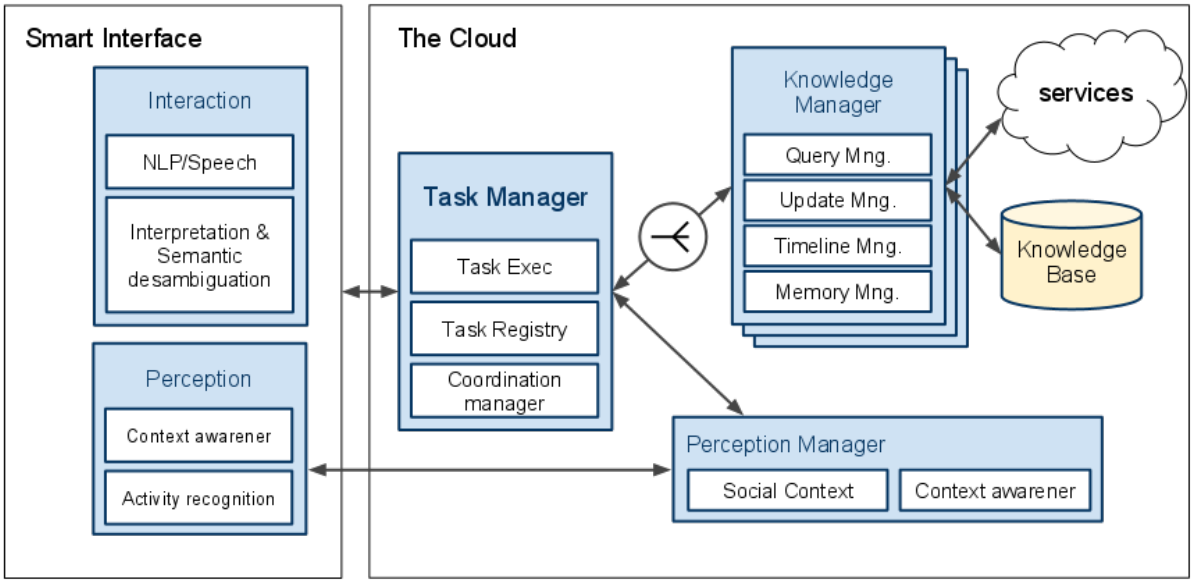

Figure 2.11: User-focused Agent Architecture

2.10 Recommender agents

In the personal agents taxonomy given in section 3, Recommender agents were declared as part of the classification. However, its importance transcends since many implementations of other kind of personal agents use a recommender agent internally. Therefore, it is important to go deeper in the study of recommender agents.

2.10.1 Social recommendation

Usually, the social recommendation methods collect ratings of items from many individuals, and use nearest-neighbor techniques to make recommendations to a user concerning new items. However, there are many factors which may influence a user in his preferences, thus ideally one would like to model as many of these factors as possible in a recommendation system.

There are two general approaches to this problems. The first is the called social-filtering methods [41, 42], in these the user of the systems provides ratings of some items and the systems make informed guesses about what others items the user may like. And the second is called content-based filtering [43] , in this case the system accepts information describing the nature of an item, and based on a sample of the user's preferences. In both cases, the objective is to learn a function that can take a description of a user and an item and predict the user's preferences concerning the item.

One of the most important issues today is the use of social data as context for making recommendations. As the social web is heavily used it could provide a better understanding of a user's interest and intentions. The proposed system gathers information about users from their social web identities and enriches it with ontological knowledge.

Before entering into the issue which we note are the various categories of social recommendation: collaborative is a kind of social recommender compare with traditional content-based approach, recommendations from friends can be online or offline, recommendation using social data as input by example social relationship, social network, social tagging... and recommender over social media. The latter utilizes friend of friend an usage data to make intelligent recommendations based on actions and interactions. Thanks a social media recommendation API we can build superior media applications, add social relevance to existing applications, platforms or sites and expand our understanding of our user's social context.

The context encompasses a set of interest from user profile, which are extracted from the

user's social web interactions and tagging activities. Dealing with context is accompanied by issues like, making the system fully understand the context of the task in hand without tedious efforts by the user finding and retrieving the desired data automatically, usually involving the integration of data from the different sources to draw useful conclusions without breaching security and privacy issues.

2.10.2 The importance of the user's profile

To make a good recommendation is necessary to consider the following points [44]: Define a generalized context model of user interest that serves as interpretation of user's intention and assist during recommendation or searching processes, this model is built initially by capturing user's social-web data, mostly tagging activities.

To generate and maintain the user profile, the system needs relevant information about the user's interests. When users interact with a computer, they provide a great deal of information about themselves. Successful interpretation of these data streams is necessary for computers to tailor themselves to each individual's behavior, habits and knowledge. As for the interaction of the user with these applications, the system can gather relevance feedback to learn his tastes, interests and preferences.

To develop a support system which can assist in providing brief and precise, then our system will identify and relate user's profiles across different social networks which has the capability of returning a set of URIs for a particular user. This will be ensured by gathering the public ally available information about user's tagging activities, important information about user's interest without requiring any help from them.

Furthermore as shared vocabularies and thesauri to model the user's interest domain. It will be achieved by linking tags used by a user, to meaningful concepts in the above mentioned ontologies. To conceptualize tags we tend to remove or minimize the vocabulary gap, the concept-tags can now more easily be mapped to more domain specific ontologies in order to support domain specific recommendations.

The system must have an architecture that intends to model user interest based on the user's social-web profiles, this is utilized in recommending cultural heritage resources that might be of interest for the user. In this case the frequency of use of certain tags indicates the interest of the user form the bases of the assumptions [45].

User interest profile modeling to enrich the set with the most related concepts in the domain. The resulting set of the concepts is added to the system as user's interest profile. The underling recommender system utilizes his interest model form making relevant

recommendations to the user.

Finally the recommender system will query this data to see how it improves the recommendation process and will also provide an interface to query the linked data over the web to suggest interesting things related to the cultural heritage that are present on the open linked web.

2.10.2.1 Common Sense Computing Initiative

The *Common Sense Computing Initiative* is a project intended to provide computers with the knowledge that we usually consider *common sense*. Such knowledge is usually known by everybody but hardly stated in knowledge databases, thus being of special interest the addition of common sense reasoning to computer programs to provide them a level of intelligence granted for human beings but rarely found on computers. [46]

Common sense is a large amount of knowledge shared by all humans which makes them able to understand each others. By granting computers the ability of common sense reasoning, we can make them understand human needs and behaviours, instead of relying solely on the user understanding software and adapting to its usage. Through this understanding, computers and software could foresee users' needs or actions even before they realize about them themselves.

This initiative comprises mainly two projects: OpenMind Common Sense - A system created to gather common sense knowledge from users; and ConceptNet - a database representing this knowledge. For our project, only the latter is relevant.

ConceptNet

ConceptNet [47] provides a formal representation for all the data gathered by the Open Mind Common Sense system. While assertions inserted into the later are stated in free-form natural language, computer applications would need a formal representation in order to be able to use reasoning methods on that common sense data.

In this direction, ConceptNet represents all the data in the form of a semantic network. Concepts such as objects or actions are nodes in the network, and the common sense information on them is represented as links between them. The system uses a limited set of possible relations, which has proven to be enough to represent the knowledge from Open Mind Common Sense and significantly eases the task of using the data in computer software. Fig. 2.12 shows a small example of some of the nodes and relations existing in ConceptNet

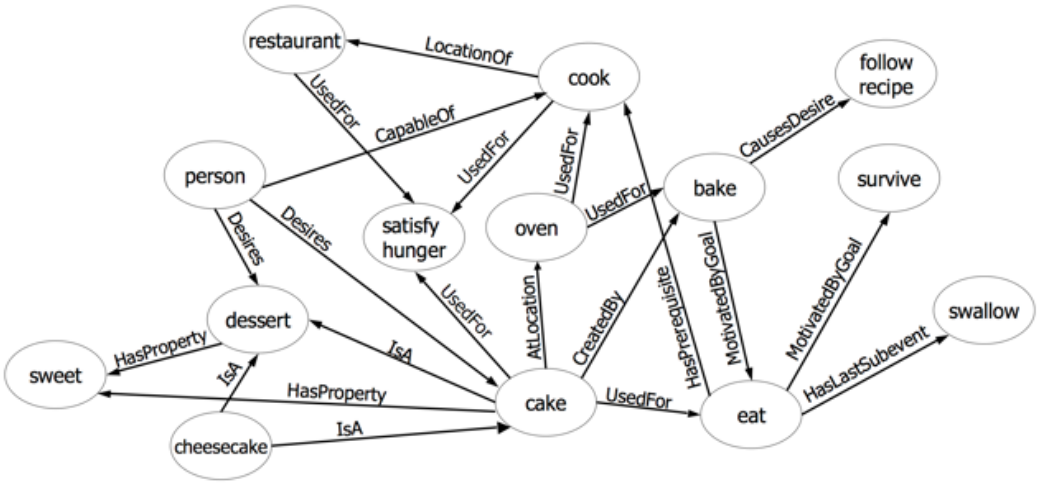

Figure 2.12: Some of the nodes and links in ConceptNet

ConceptNet offers several public APIs that anyone can use to add common sense reasoning to computer projects. These APIs have varied through the different versions of ConceptNet, usually offering not only a raw access to the semantic network but also some reasoning methods already implemented, such as finding the context of an action or making analogies. In the fourth version of ConceptNet, the reasoning tools was taken out and specialised into Divisi, an independent project also part of the Commonsense Computing project at the MIT Media Lab. Divisi, as any other reasoning method applied on ConceptNet, can infer additional relations that are not stated in the network, but that are also common sense knowledge.

ConceptNet 5

At the time of this writing, ConceptNet is already in its fifth version. Several changes have been made to the system, as well as several new sources of data have been included in the data acquisition procedures. Thus, the knowledge base has not only grown beyond the limits of the former ConceptNet 4 database, but has also been significantly enriched through other changes.

Originally, ConceptNet's semantic network was a graph in which nodes represented concepts and links (or *edges*) gave information on their relation. While this still applies to ConceptNet 5, the network has grown to become a *hypergraph*. In other words, edges can now act as nodes and have other edges pointing to them. We can have *edges about edges* which not only provides a much larger network, but a much richer one. Thus, relations will have additional information from which we can infer its accuracy, reliability or justification.

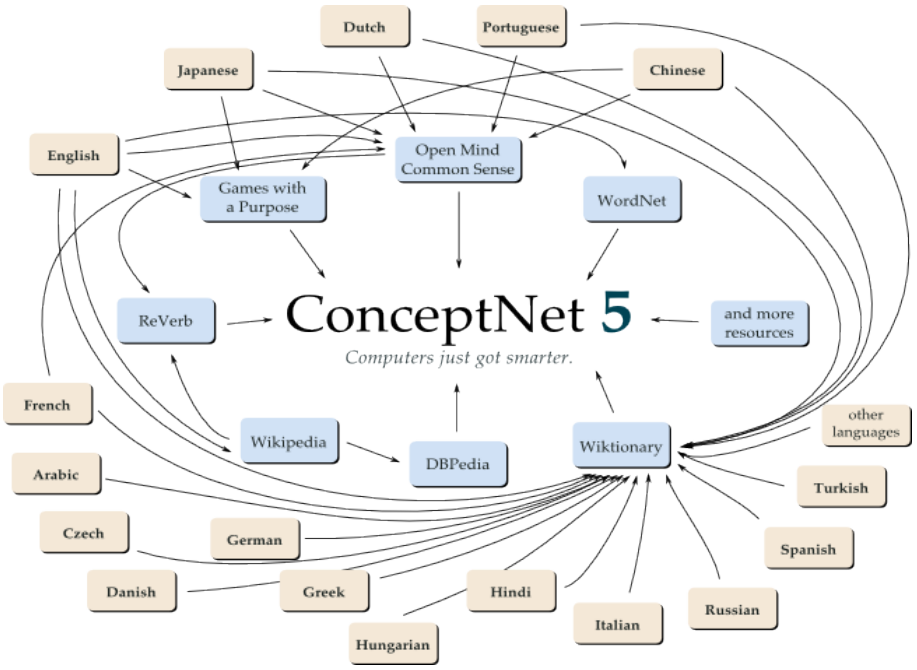

Figure 2.13: ConceptNet 5

Furthermore, relations are no longer limited to the specified set we had in previous versions. While it keeps being advisable to stick up to normalised sets when possible, additional relations can be created at convenience. In this direction, the fact of treating relations as nodes themselves comes in handy as a way of counterfeiting increased data sparsity or analogous relations which could indeed become a drawback on the quality of the network.

While previous versions of ConceptNet expressed nodes and relations exclusively using English language, ConceptNet 5 aims to represent also interlingual knowledge. For this purpose, a new type of relation "TranslationOf" has been added to the set of basic relations. This way, all the relations existing between English terms are automatically reachable from their translations through their "TranslationOf" edges, being very handy for non-English or multilingual applications. Furthermore, the translated nodes can be very useful themselves, in order to find the best translation of a word in a given context. As we mentioned before, any node can be an edge in ConceptNet 5. Having different language nodes could become a problem when being used to state relations. However, basic abstract notions such as "MadeOf" are still used in their English form to mean the same across all languages, and other language-specific edges can still be used in an interlingual context if the appropriate "TranslationOf" edges exist.

Furthermore, the acquisition procedures for the common sense statements have also

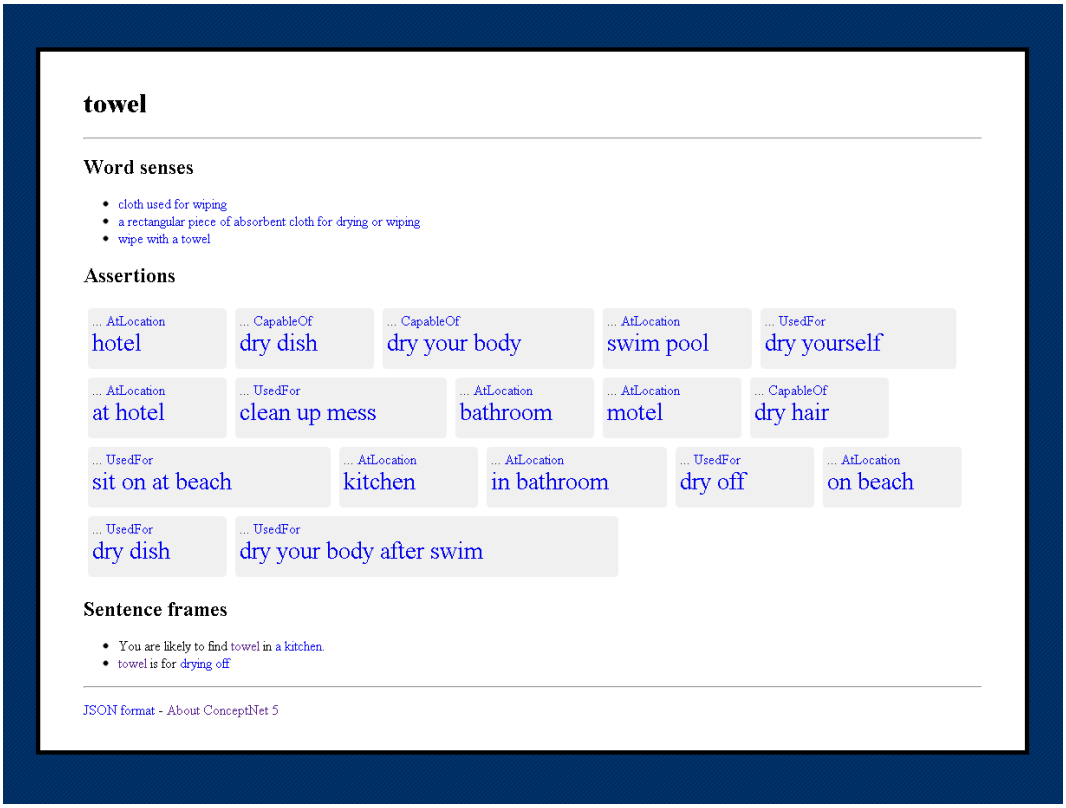

Figure 2.14: An quick example overview of some of the information about the concept "Towel" in ConceptNet 5

significantly changed since the first versions of ConceptNet. While initially relying uniquely on the Open Mind platform, and in users actively *teaching* things to the system, ConceptNet 5 also gathers knowledge from several RDF sources such as DBPedia [48] and even from general purpose websites using ReVerb [49] to extract relational knowledge from wikipedia and other sources.

Chapter 3

Requirements Analysis

"Laudem virtutis necessitati damus."

—Marcus Fabius Quintilianus (Quintilian)

3.1 Overview

It is important to perform a requirements analysis to make sure the final solution will be adequate for real life applications, and to broaden the variables taken into consideration, making it less likely to miss a key aspect in the design process.

3.2 Use Cases

Given the versatility of the system in pursue, there are a countless number of scenarios in which we can take advantage of the combination of social networks and online services. To illustrate this, we will cover some significant examples that could be interesting.

In the following sections four cases will be presented. The first three inside the scope of the Web 4.0 project (i.e., related to intelligent personal agents), and a fourth one that exemplifies the use of the new paradigm in a typical web service.

3.2.1 Birthday Present

Jane Doe is an average user of the application. One day, she is notified of the upcoming birthday of her mother. In addition to that, it offers her the option to look for possible presents. When Jane accepts the proposal, the client application queries the cloud agent for suggestions. Those suggestions will be based on the data available in Jane's social network. Based on the available data, the agent may suggest new books, trips, movies or a revenue.

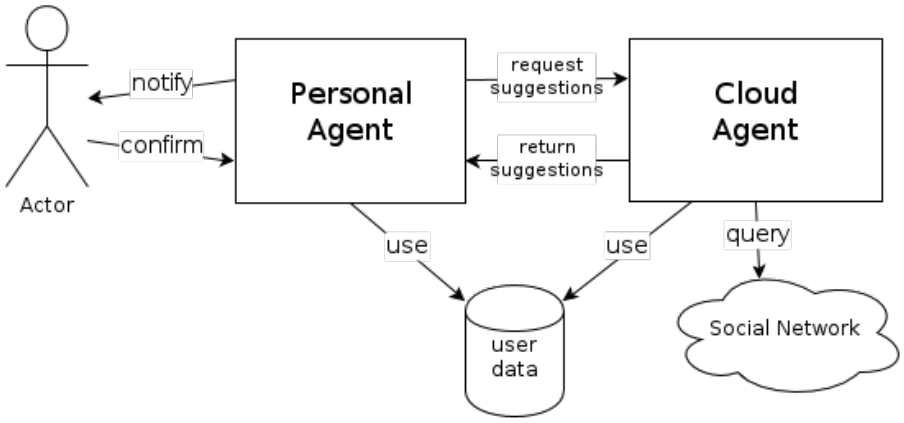

Figure 3.1: Birthday present use case

3.2.2 Summer Trip

John Doe, Jane's brother, decides to try the application too. In this case, he decides to use it to plan his annual summer trip. First, he queries the application for cheap flights to Bahamas in July. After querying the application, and receiving a list of the best offers found, John chooses a flight. Right after that, it is saved as a future plan, and the agent provides a list of TO-DOs for his trip: pack-up, get vaccinated, check passport, etc. As John completes these tasks, he can cross them, or he can even delete them if he does not find them necessary.

3.2.3 Movie Tickets

Mary Major decides to occupy her Saturday night watching a movie in the theatre. For that, she uses the application, asking for movies of her taste to watch in her area after 16:00. The application then gives her the list of cinemas and movies currently being shown in each of them, filtered by genre and cast according to her and her friends' likings in social networks and previous uses of the application. In addition to the list, she can ask for more information about any of the movies (plot, cast, IMDB rating, et cetera). After making

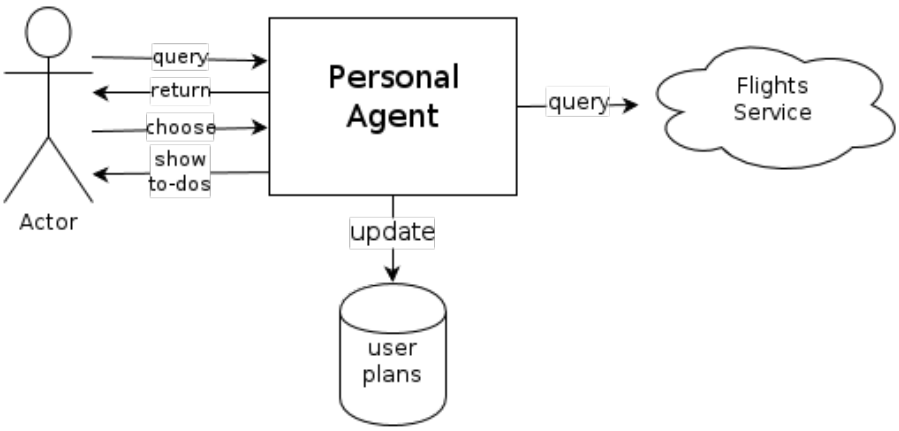

Figure 3.2: Summer trip use case

up her mind, Mary can then add that movie to her TO-DO list, and optionally share it on social networks or by email. Later, when she want to get to the cinema, she can go back to her TO-DO list and select the information of the cinema. The location will then show in an embedded map.

Finally, she can rate the movie in the application and share that rating online, deleting it from her list.

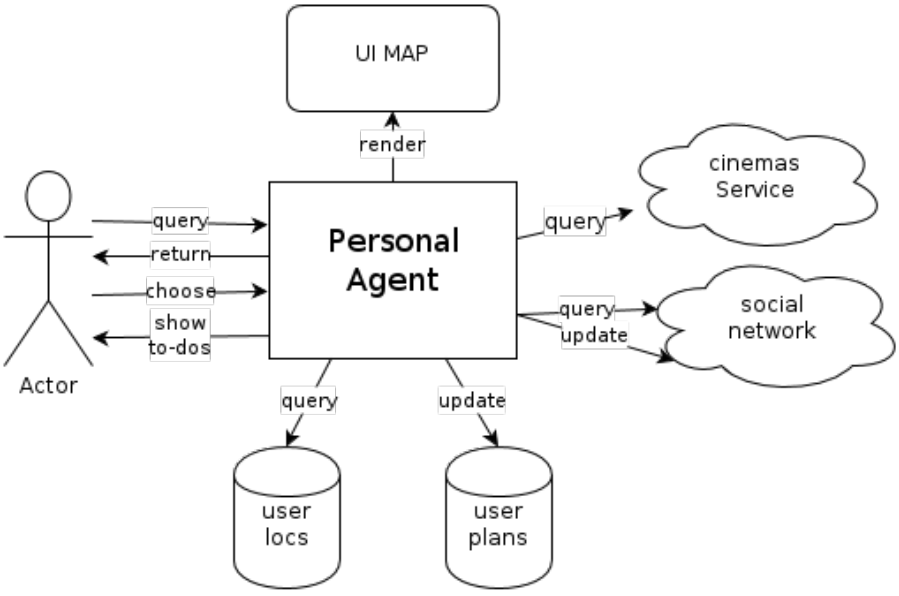

Figure 3.3: Movie tickets use case

3.2.4 Blogging site

In this case, a user is registered to a blogging platform. In this platform, there are many bloggers that post about different topics. Since there are many posters and the activity is very high, the platform offers the users the possibility to subscribe to certain topics. What makes this service different from the existing ones is that the topic selection is based on semantic analysis of both users' request and bloggers' entries. Once the users have subscribes to a certain topic, they receive a notification every time a new post has in common with their subscriptions.

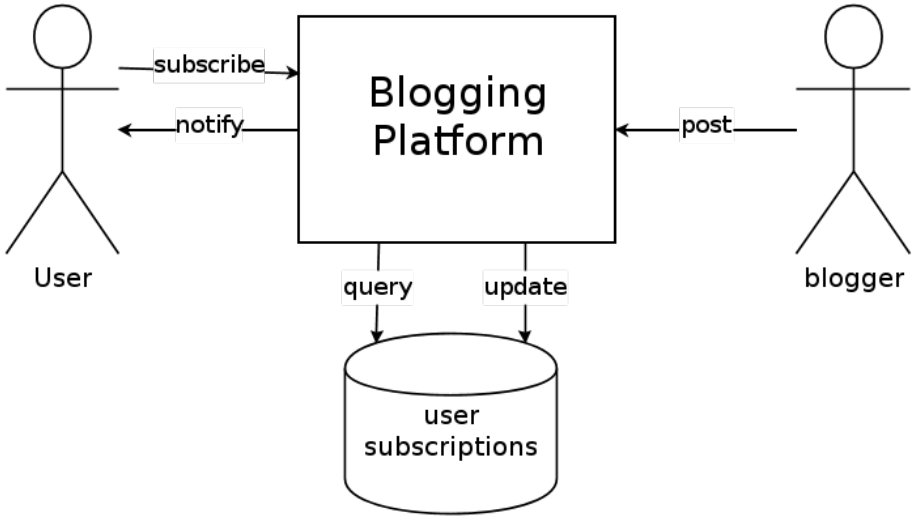

Figure 3.4: Blogging use case

3.2.5 Blogging site

With these scenarios, the endless possibilities of the agent application should have been shown. Not only for querying information, but also for generating it and filtering or modifying it, offering a customized experience and a bidirectional service.

3.3 Summary of requirements

After analysing the previous use cases, some clear requirements seem to stand out:

- The architecture must allow the connection of heterogeneous end-points
- Language agnostic, the final system will likely incorporate different languages and the corresponding design must allow for this

- Compliance with current web protocols, including HTTP
- Low latency, to achieve a real-time user experience and process a big amount of information live
- Pro-activity, not limiting the behaviour to the typical reactive nature of the Web
- Simplicity, to make the development of new components easy as fast, since the strength of the system relies on the quantity and quality of its available external connectors
- Openness, for the same reasons as the previous point
- Easy connection to third parties, both to reach other services and to allow connections from users
- Loose definition of actions and objects
- Flexible topology
- Adaptability, to evolve as web technologies and user needs change
- High connectivity between elements

Nevertheless, this list is not as extensive and formal as the usual requirements analysis results. This is mostly due to the ever-changing and evolving nature of the technologies involved, as well as the novelty of the field. Despite this, it can and should be taken as a solid guideline for the design and development of the desired architecture.

Chapter 4

Architecture

"Knowing how things work is the basis for appreciation, and is thus a source of civilized delight."

—William Safire

4.1 Event-based Agent Architecture

Based on the requirements identified in the previous section, a preliminary set of design criteria can be extracted for agent architectures. These criteria are especially interesting for those systems whose purpose is acting as personal assistants in the live web.

1. *Event centric.* In the Live Web, events must become first class citizens[50], and this change allows for a simplification of data sources. Not only can we communicate agents by means of events, but also acquire information seamlessly from plain modules that follow the same protocol. This may also apply to other agent communication protocols as long as the data source is adapted.

2. *Open and Homogeneous interface for processing events.* All external events are treated in the same fashion, which eases the process of developing new nodes to interact with the existing ones, especially those that only serve the role of data sources.

3. *Event subscription and filtering capability.* There are two problems in an event-centric system: starvation, as a result of an absence of incoming events; and flooding, which

makes impossible to tell useful information from noise. Subscription addresses the first problem in those situations in which broadcasting is not an option (probably to avoid flooding), and effective filtering tools and methods prevent the effects of an excess of information.

4. *Real-time processing capabilities.* Outdated information is of no use in real-time applications. Because of this, the process of analysing and handling information have to be almost instantaneous.

5. *Smooth transition.* In order to attract developers and offer a good experience to users, it is important to provide a good level of compatibility with existing technologies and services. It is also important to lessen the possible negative effects of the change of paradigm in the user.

There are two basic additions to a generic BDI multi agent system to follow the Maia principles. However the nature of the changes, which result in a complete change in the way of understanding agent communication, are deep and need to be understood thoroughly to make use of the advantages it offers.

The first one, following a top-down approach, affects the way agents communicate with each other. All communication is done asynchronously using events as defined in this context (see Section 4.2). What makes the Maia approach special is that fact that agents are adapted to the existing tendencies in online data sources rather than following ad-hoc solutions to gather information. Information passing is useless unless something is done with that information. That is the second requirement, how events are handled internally by the agents or agents platforms. As explained in the following section, events follow a fixed structure which makes them easy to treat and include within the knowledge databases.

4.2 Messaging and Communication

The cornerstone of communications within Maia are event messages. An event can be either informative or a request, in the sense that it may inform of an action performed or of an intention to trigger an action in a remote entity. This difference can be inferred by the context or by using a special namespace. The basic components of any event are the following:

Sender Unique identifier of the sending entity. It is recommended to prepend or append the identifier of the bus it is connected too if the scenario allows for shared messages

between them. This operation can be performed on the fly by the Event Exchanger, making it transparent to the sender.

ID Unique identifier of the event for the specified entity (sender).

Timestamp Time of the original emission. This makes time reasoning possible and prevent side effects of asynchronous communications.

Name Which describes the event, and is the only required field. Ideally, it will not only consist of a basic string, but of a complete namespace. This allows for a complex processing of the events and an advanced filtering for triggers. It is highly advisable to use an already existing naming convention, especially if there are tools available that use it. For an example of an implementation, see Hook.io's in Section 2.2.4.

Payload For any kind of non-trivial event, we will need more information about the entities involved in the event, or the parameters if it is a request. In FIPA-ACL this is the equivalent of the content, but this is a simpler approach as it mixes agents and plain modules to external web services that communicate in the same manner.

Callback Due to the asynchronous nature of event-driven communication, there is a need to specify a callback function to execute whenever there is data to return, or just to acknowledge the reception of the event. The actual method for acknowledgement is up to the designer. A working solution is to send an event with a special namespace (as explained in the next section) used only for this purpose, appending the original Sender identifier. The payload of that acknowledgement event will contain the returned data to be used in the callback function back in the origin.

4.3 Namespaces

The final goal for the name of the events is to make them descriptive and related to the nature of the action to be taken (either informative or inquisitive). Events can be understood as actions (verbs) performed or requested from one party to another. Since these actions may or may not be triggered due to the nature of our communication, we refer to them as intents. In Maia, events are defined as nodes of a taxonomy that defines and relates the. This brings up several advantages, the main one is that we define an ontology of intents, so every item in the ontology has a set of properties that define the action to be taken and the response given. Thus we can filter not only by class but by properties, and we can also reason using them.

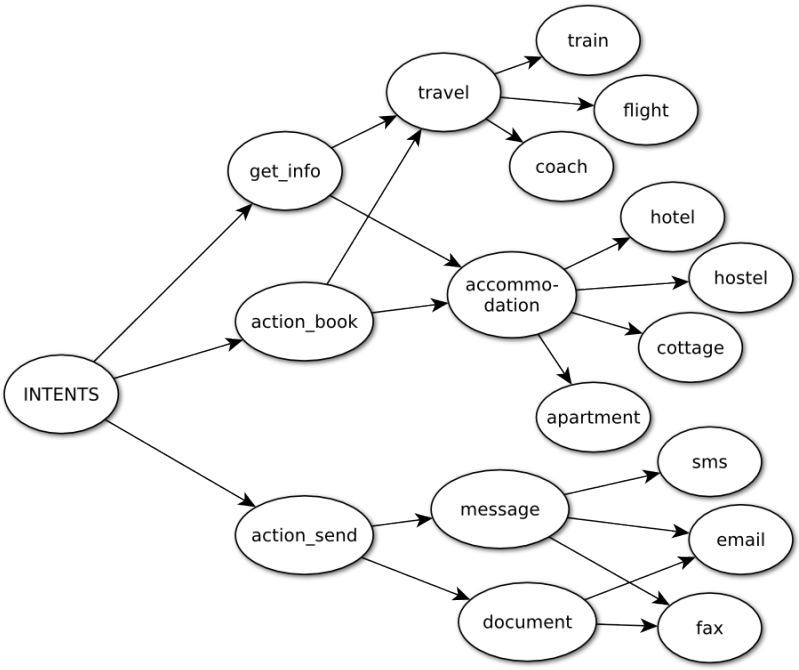

Figure 4.1: Excerpt of hook intents ontology

The figure 4.1 shows ans excerpt of the ontology defined.

Given the advantages of hooks and webhooks, using them is the best choice for implementing intents. Intents can be defined as events that are triggered, what make the hooks that are registered to those events activate and perform a certain action such as retrieving some information, or even triggering some other events.

Node.js and hook.io are perfectly suitable to develop massive event architectures, since their prepared for event filtering and great at scalability. And hook.io inherits namespaces from eventemitter2[1]. With a little modification, mostly by setting a schema for these names, it is possible to obtain both a strong definition and an already working technology to support it. Thus, intents are modelled as events in the format given:

```
somehook::someotherhook::get_info::travel::train::fares
```

Where, the first element is the hook that emitted the event. It can be used to trace the message to its origin and should always be added at the beginning once the message is sent. The second element is the potential receiver of the action, should there be any, or * if it can be any hook that can react to the intent in question. The third element in the namespace is the name of the intent, and traversing down the nodes of the taxonomy we

[1] `https://github.com/hij1nx/EventEmitter2`

append the subsequent elements. Fourth, and fifth elements represent the scope, and last elements are typically the properties. So, in the example given, the intent is attempting to retrieve information about fares of trains. Any other important information to complete the intent successfully is included in payload data.

In hookio, events, these intents are transmitted through a common bus. Hooks listen to the information in the bus, so they get triggered when particular events are transmitted, those for whom the hook is registered. Thus, registering a hook for a event is a lightweight process since the emitter does not have to send the event to all the registered hook, but to put is on the bus and it will get broadcasted. Hence, using hooks promote service auto-discovery.

Elements in the event name can be filtered, so those relevant to determine what hook should be interested in processing the intent must be included. For instance, a hook that can retrieve information about train and coach fares may listen to events that match the name `*::*::get_info::travel::*::fares` . So it does not filter by emitter or travel type. Any intent relative to travel, whatever type it is, e.g. `srchook::*::get_info::travel::coach::fares`, will be processed by the hook given.

Similarly, when the intent `srchook::*::get_info::travel::*::fares` is emitted the "srchook" wants all the hooks that can fetch information about any means of transport fares to process the intent.

4.4 Topology and hierarchy

As we learned from the evolution of the internet in section 2.1.3, the success of the Internet and all its supporting protocols was based mostly on the simplicity of their design. There are many examples of how complex technologies have failed to root on developers because they were either so intricate the learning curve was too steep, or they were so fine tailored and heterogeneous that applications grew difficult to extend, develop and upgrade to allow new uses. To mitigate this effect and allow for fast adoption and development, the Maia architecture is meant to be as simple as possible, whilst providing an elegant solution for most cases.

The key feature that shapes the proposed architecture and drives its design is, as already stated, events. All messages are passed in the form of events, broadcasted to all the members connected to the shared bus and forwarded by the event router to all subscribed nodes, as seen in Fig. 4.2. There may, or may not be, shared events between the several buses, depending on the subscriptions.

Entities can be BDI agents, plain modules or agent platforms, as long as they use the same protocol and events convention. This provides a high flexibility and separates the transport mechanism from the logic of the agents.

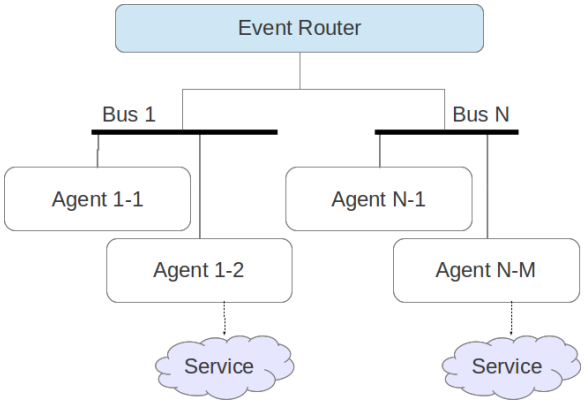

Figure 4.2: Internal distribution of nodes within Maia. Events are broadcasted to all members within the same bus.

As mentioned before, not all event are sent to all the entities in the platform, as each entity may have different interests. In Maia, there is at least a central piece that orchestrates the exchange of events, and we will refer to it as *event router* or simply *router*. This event router handles events and the way they are forwarded to each entity. To simplify this mechanism, there are two basic ways entities can be arranged: in the same bus/hub, or in two separate buses. The difference lies in the scope of the events shared, any entity can subscribe to any event sent to its bus, while events in other buses are inaccessibly by default. Depending on the specific configuration of the Event Router, they may or not be accessible at one given point in time.

When compared to other solutions like XMPP, used in SPADE, it is easy to spot many similarities. For example, the presence of a central node that every other node connects to and serves as a message router. These similarities are not coincidental and find their basis in the previous analysis of current technologies and their advantages. There are, however, a big number of changes that will be presented in the following sections.

4.5 Clustering

With Maia, nodes are loosely coupled, which allows for a higher flexibility than in other architectures. All the information is contained within the message, including emitter and sender, which are included as strings in the name of the event. This makes routing events as easy as parsing event names and applying certain rules. Moreover, nodes are agnostic to

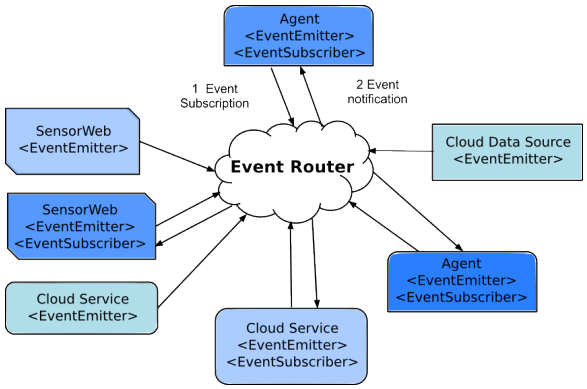

Figure 4.3: Flow of events. Entities can simply send event notifications to the Event Router, or subscribe to be forwarded a certain subset of events.

the central router as described in the previous section. Thus, the routing complexity can be as complicated as possible, or as simple as can get, and the only changes will be in the routers. Connected nodes are only influenced by this in the sense that they have access to a higher or smaller range of events from more or less sources.

Different layers of nodes can be connected to each other (by means of their routers), without needing to change the actual nodes/agents within any of them. Of course, security issues arise that require for more elaborate mechanisms, but the flexibility and simplicity of the base solution is undeniable.

What is more, in the total absence of a router, nodes can still exchange messages as long as they are connected to the same bus. This behaviour mimics that of the Internet Protocol, but providing it a looser nature.

4.5.1 Treatment of Beliefs

Apart from the modifications and adaptations needed to comply with the communication mechanism explained in the previous section, there must be a rearrangement of the inner working of the agent and the way it processes its information. The sole structure of events gives a hint of the proper way to store them, but we will go through the basic aspects as generally as possible.

Firstly, every event has a source, which should be treated as the source of information, as is the case in any common multi agent platform. Beliefs should be tagged this way when possible, or stored in a separate knowledge database.

Secondly, events are bound to a specific time, which should be relevant for almost any

possible application that makes an intensive use of events. Having references to the time events were generated permits removing old events, reasoning according to time constraints and correlating events in time. Since the systems we are aiming to design are likely to receive a high amount of information, it is advisable to set a strong policy on the persistence of incoming events.

Thirdly, it is important to note that every agent should subscribe only to the subset of events that is relevant to its functioning. If necessary, more agents will be created to handle different sources or events when they are correlated. This way, the amount of irrelevant information will be lower, and handling information using several smaller databases is more efficient than storing and processing all the information in the same point. Moreover, it follows the criteria of agent design, using different agents for different tasks.

Chapter 5

Case Study

"Difficile est tenere quae acceperis nisi exerceas"

— Pliny the Elder

5.1 General description

To demonstrate the capabilities of the architecture described in Chap. 4 a personal agent was developed. The aim of this agent is to assist the user in the organisation of leisure activities. This prototype was built using Jason, as well as several different services which work as data sources and which are further described in this document.

The idea of this prototype can be extended and used to organise every kind of event, from business trips to dinners, depending on the amount and characteristics of the data sources accessible. For this prototype, the scope is limited to travel of any kind. Therefore, our system will be able to assist the user in the task of organising travels, from the moment of choosing the date to the time to pack the luggage.

The system has two main working methods. The first and most interesting, in which the system can anticipate the travel opportunities or needs the user will have, and suggest suitable plans in advance. The second one is quite similar, but is the user who requests travel organisation actively.

Figure 5.1: Avatar used in the browser User Interface

5.1.1 Intelligent suggestions

The main functionality of the system is the ability to produce autonomous suggestions, based on several parameters to find the plans which have higher chances of being appealing to the user.

The procedure for obtaining suggestions can be seen in Fig. 5.2. The triggering event can be a calendar modification, a timed signal, or a user-activated event, depending on the system. For demonstration purposes, in this prototype this procedure will be manually triggered.

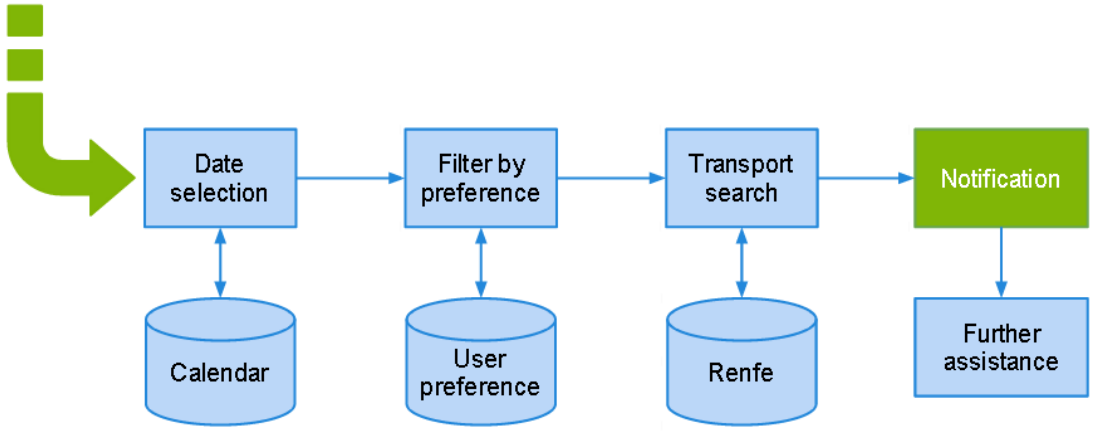

Figure 5.2: Workflow of intelligent suggestion generation

5.1.2 Description of the procedure

In order to build appropriate suggestions, the following variables are taken into account:

- Available dates
- Preferences

- Transport availability

In first place, it obtains a list of suitable days for the user to travel. This information is fetched directly from a calendar source. In order to make further discrimination between dates, a score has been added to each day, which computes from the free subsequent days. This way, if the user has a whole week off, Monday would get a score of 6, Tuesday a score of 5 and so on, thus prioritizing Monday as the day to organise plans, allowing making longer trips.

The second step is to get a list of places where the user is likely wanting to travel to. This is deeply described under section 5.2.2. The user preferences are stored as a list of places and scores, which is dynamically modified as the user makes plans.

The third step is to get the best offers for travels to the chosen destinations. This is obtained through web scraping, as described in section 5.2.3

With these parameters, suitable plans are built and presented to the user, which can select one of them and proceed with further assistance, as described in section 5.2.4

5.1.3 Requested planning

The functionality offered by this mode is quite the same as for "Intelligent recommendations". The difference is that suggestions here are not automatically generated as described in section 5.1.1 but using a query from the user as parameter.

The destination and dates are chosen by the user, entering a query in natural-language format, and the best offers for transport are presented by the system. Right now, the cheapest option is chosen, regardless of time, but other algorithms can be used.

Optionally, the user can specify a threshold or budget for the trip. Should the cheapest fare not comply with this requirement, the user is notified and is asked to raise this value to obtain results.

This way of organising events allows us to refine the preference database, as described in section 5.2.2

5.2 Functionalities

This section contains a deep description of the functionalities present in the prototype system and their implementation.

5.2.1 Synchronisation with Google Calendar

In order to obtain the most suitable dates to organise trips, the system can connect to the user's Google calendar and retrieve their free days.

This connection is made through a simple REST wrapper, but can be integrated easily whichever way is required by the platform in which we want to install the application. For instance, if we want to develop an interface for android devices, we could connect this to the device's calendar instead to a Google calendar directly. For prototype purposes, Google Calendar and a REST wrapper were chosen. However, this choice is irrelevant, as the availability information is normalised by whichever system is chosen.

Thus, the knowledge base will contain a list with the days in which the user is freely available to make plans. In order to make further discrimination between dates, a score has been added to each day, which computes from the free subsequent days. This way, as explained before, if the user has a whole week off, Monday would get a score of 6, Tuesday a score of 5 and so on, thus prioritizing Monday as the day to organise plans, allowing making longer trips.

Weekends are not computed in this direction, unless they are manually marked by the user in the calendar. The reason is that the inclusion of weekends on the knowledge base would result on many possibilities to take into account, and would generate much "noise" in the suggestions, as people usually do not want to travel *every* weekend. However, users can actively request a plan for the weekend if so they desire.

5.2.2 User preferences learning

One of the parameters used for the generation of intelligent suggestions (sect. 5.1.1) is the user preference of destinations. This information can be obtained from a vast variety of sources, such as social networks or user-entered information. Furthermore, what is really interesting is having the system progressively adjust these preferences, according to the plans being accepted by the user. Thus, apart of gathering information from the available sources, the system can *learn* from its own suggestions, and fine-tune these preferences for future suggestions.

Using whichever means to obtain the user preferences, the places are entered in the knowledge base of our agent. This information is built in the form of a "ranking", having a list which relates the places with a score, calculated by the number of times the place appears on the used source.

There are several ways to gather information, some of the analysed are the following:

- Friends' location through social networks
- "Liked" destinations on social networks
- Information on previously visited places through other sources
- Learning from the usage of the system

5.2.2.1 Acquisition from external sources

After the analysis of the possibility of gathering this information from social network and other location-based services, it was discovered that this task happened to be much more complex than expected, mostly due to the complexity of the services' APIs, or to the restrictions imposed by the services for privacy reasons.

As the objective of this project was not to exercise these aspects, it was decided to focus on the intelligent part of this feature and avoid putting a high amount of effort in gathering further data sources. Therefore, the acquisition from external data sources in our prototype is simulated through mock systems and not truly connected to Facebook or other data sources. However, this is irrelevant for the functionalities of our system, as it does not matter whether the information obtained is hard-coded or really obtained through whichever system. It is the following stages the ones which really matter, in terms of intelligent systems studies. This is a key factor of this programming paradigm: once modelled as external sensors, the nature of the data sources are no longer important as long as they populate the agents with the appropriate perceptions.

5.2.2.2 Learning from the usage of the system

Besides having a "pre-loaded" or directly obtainable source of preferences, it was decided to focus on the adjustment or fine-tuning of this information.

This is achieved by modifying our knowledge base every time the user accepts one suggested plan. When the user accepts travelling to a proposed destination, we can infer that the user has some preference of some kind for that place. Whether it is for business needs, family affairs or just preference, we can assume the user is likely travelling to this destination again for the same reasons.

Whenever the user travels to a place, this site is saved to the knowledge database, and given a "score". The score is increased every time a place is revisited by the user, inferring that the reasons to visit it are either strong or likely to happen again. Later, places with higher scores are more likely to be suggested again over others with low scores.

5.2.3 External services

In order to provide actual information about transport, available destinations, prices, etc. we need to connect our system with external services providing this information.

As this is not the objective of this project, the connection has been limited to a single service: the website of a Spanish train company - Renfe. This website is accessed through Scrappy - a web scraper which obtains train information such as pricing, times or destinations. The addition of extra sources is a significantly tedious task, as usually each of them have their peculiarities and a new approach has to be made for each of them.

5.2.4 Generation of tasklists

Planning and execution of all our activities is strongly driven by common sense. You know you can go on a long trip on holidays, that playing tennis requires a racket, or that you can only ski if there is snow. Furthermore, the whole process of organisation of any specific activity also relies in a vast amount of common sense statements. For instance, the fact of needing to pack your luggage before going to the beach for a week, *is* common sense, or going even further, the list of items that you are likely going to need when travelling to the beach falls into this category as well.

However, in spite of being common sense (which is allegedly owned by everyone) some of the tasks can be usually forgotten or not taken into consideration by most people. Is not uncommon to forget bringing a towel or sunscreen to the beach, while everybody would agree those things are clearly needed there... It is common sense!

The usage of common sense reasoning is of very high help in these fields, as most of the times, the knowledge about the prerequisites to perform an action or organise a successful event is only common sense knowledge. In other words, through the usage of common sense reasoning, the system can understand the needs of the user rather than just saving reminders set by the user or give information as response to a user-entered query. The system aims to pro-actively help in the organisation of these tasks instead of being a mere calendar and a source of information.

Description of the procedure

As mentioned before, common sense reasoning can help our system to understand the basic nature of the activity the user is planning, such as *organising a meeting* or *going to the beach*, and could hence infer common basic tasks such as *getting a projector* or *buying sunscreen.* We can not, however, aim to provide further aid regarding more complex aspects of the activity, such as those related to the topic of the meeting or the specific geographic location of the beach we are travelling to in our example, as that is not common sense knowledge.

One of the most relevant projects in this direction is being developed by the Software Agents Group of the MIT Media Lab: Anticipating User Tasks Using Commonsense Reasoning [51, 52]. For this project, a subset of the nodes and relations of ConceptNet was built into LifeNet [53], gathering only those nodes related to actions and their temporal related links. In order to get the appropriate suggestions for tasks related with the activity being organised, we can follow time-related relations such as "has prerequisite" or "has subevent". We can expect that all the previous tasks needed for the organisation are classified into one of said two categories, but of course not all the nodes with those relations are going to be tasks that we should suggest to the user. For instance, the node "Go to the beach" can have the subevent "sunbathe" while no task can be inferred from it.

The method followed in the aforementioned project to discern between possible tasks to suggest and any other related events, is to look for verbs whose meaning can indicate a necessary action, such as "buy", "bring" or "get" [51]. Using this criterion the system returns valid suggestions in most of the cases.

For our project, we plan to do a similar usage of ConceptNet to create a to-do list for user's activities. The latest features added to ConceptNet, as well as its increased complexity in form and content is still to be analysed, as these will tentatively open new possibilities in the reasoning methods on the common sense knowledge.

5.2.5 Natural Language Processing

To process the query from the user, the tool Unitex is used. More specifically, we use a set of custom dictionaries containing cities, currencies and companies, to tag certain keywords. And, more importantly, a Unitex graph is used to identify the relevant structures and convert the results into an xml format that the agent will understand. In this graph, we have the basic structures for requesting information about travels or asking for advice, and some conversions are also performed to translate some words like the name of the months into digits and tokens that the agent can process.

As with the rest of the modules, the interaction with Unitex is made through a wrapper. This wrapper was developed especially for this scenario, and includes code in python. In contrast with other parts already mentioned, it was decided not to simulate this functionality by replacing it with a mock-up because the results achieved with this method are much more versatile. Once the barrier of making the wrapper made to work is passed, modifying the behaviour via the Unitex graph is an easy task. Moreover, this feature is closer to the object of study than connecting to the API of a social network, and thus it was considered a priority.

5.3 Agent Network Design

One of the goals with this prototype is to develop a Network of Agents that cooperate to achieve the goals the systems identifies. In general terms, this will be understanding what the user asks, using natural language, giving assistance and trying to perform that task if the system's capabilities allow it to do so. These capabilities are defined by those of the web services the system is able to use. Using a web service, for the MAS system means making a request, either by using an API given by the service provider or scrapping a web page, and understanding and classifying the data received to be used by the system.

The figure 5.3 shows the different agents that compile the Agent Network designed to assist the user for the prototype system. That figure is drawn using the Prometheus notation for Agent Networks [54].

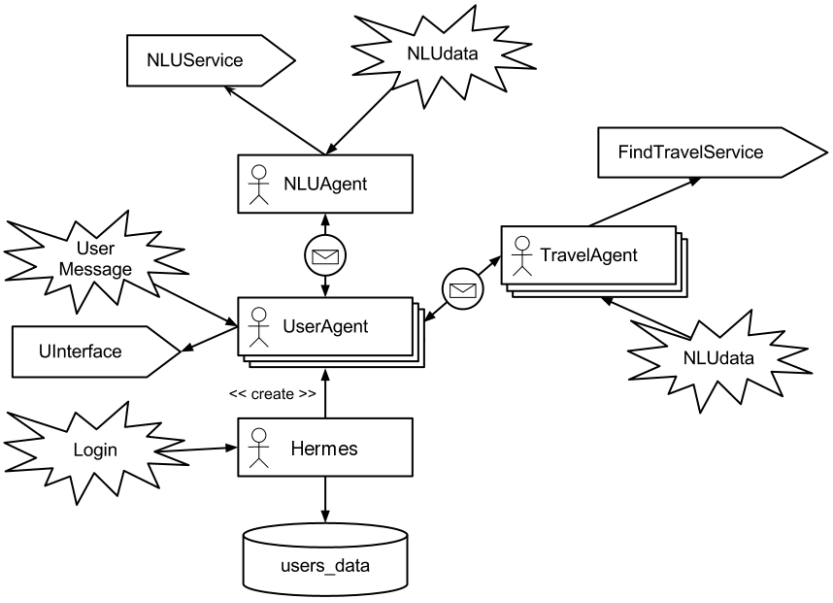

Figure 5.3: Agent Network Overview

As shown in Fig. 5.3, there are three different type of agents, plus the Hermes back-end. Each has different capabilities so they can cooperate to achieve their assignments. Similarly, they react with different events so the information the system gathers is distributed among the agents.

5.3.1 Agents

The user agents within Jason are responsible for interacting with the user. They capture the information the client sends to the system, i.e. the messages in natural language the users texts in the User Interface and the context information the client may send the system to inform of its capabilities, current state and even some positioning information. They also present the user the results of the execution of the tasks so the user can validate the decisions.

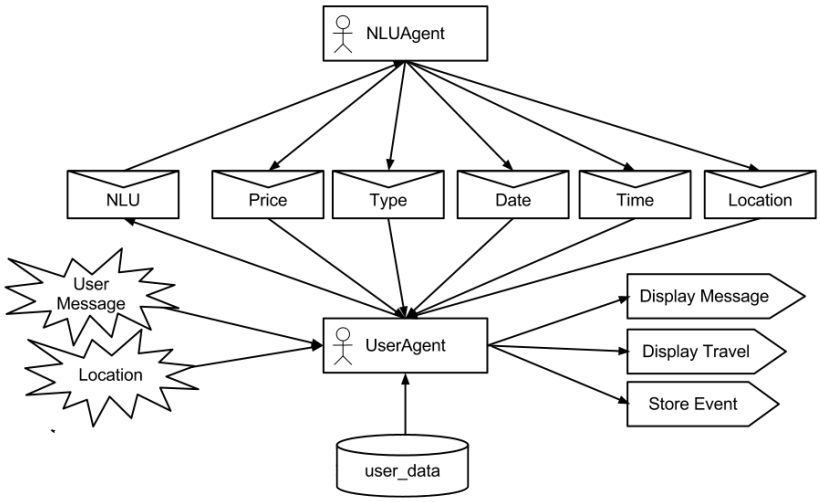

Figure 5.4: User Agent and neighbours details

In a normal multi-user scenario, each user would have a single user agent. For this prototype, there are only single-user capabilities and so a single user agent is used. Again, it is a simplification that does not affect the final results or the objective of this thesis, which is to show the capabilities of the proposed architecture.

5.3.2 NLU Agent

The NLU Agent is responsible for processing the natural language messages that the user types on the client interface. They are received as part of messages sent by the User Agents,

and the NLU Agent sent them back a set of messages with the literals it was able to extract from the message.

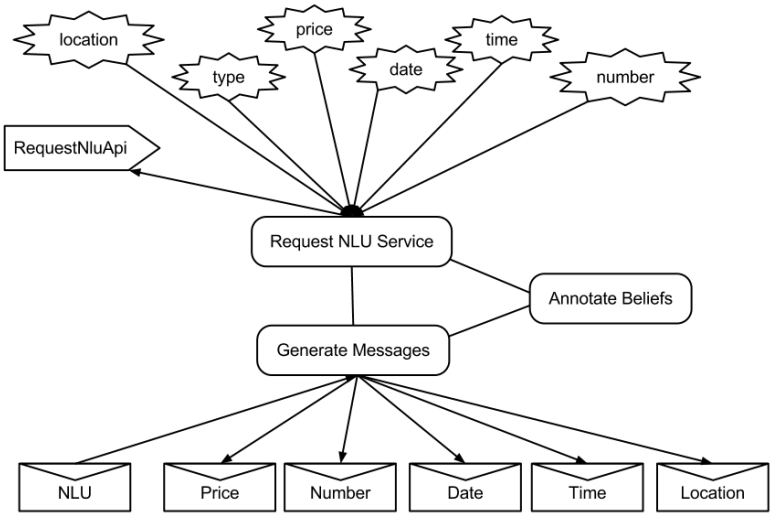

Figure 5.5: NLU Agent

The NLU Agent has tree main capabilities:

- It can access the API of the NLU System, so after each request it fetches a JSON object with a predefined structure and the data. The information of the JSON object is transformed into literals and added to the NLU Agent list of percepts.
- It processes the received information: filters those percepts that are empty, i.e. have no information, and annotates them with the particular domain they belong to (this is travel, business, revenue, etc.)
- Finally it prepares a set of messages and sends them to the proper User Agent that requested them.

5.3.3 Travel Agents

The Travel Agents are responsible for finding a journey -or a combination of them- that matches the criteria established by the user. Travel Agents use particular web services, such as travel search engines and train and flight companies web services, as source for the retrieving the travel information.

As a matter of scalability, each Travel Agent is entrusted with a different web service, so new data sources can be plugged or unplugged dynamically.

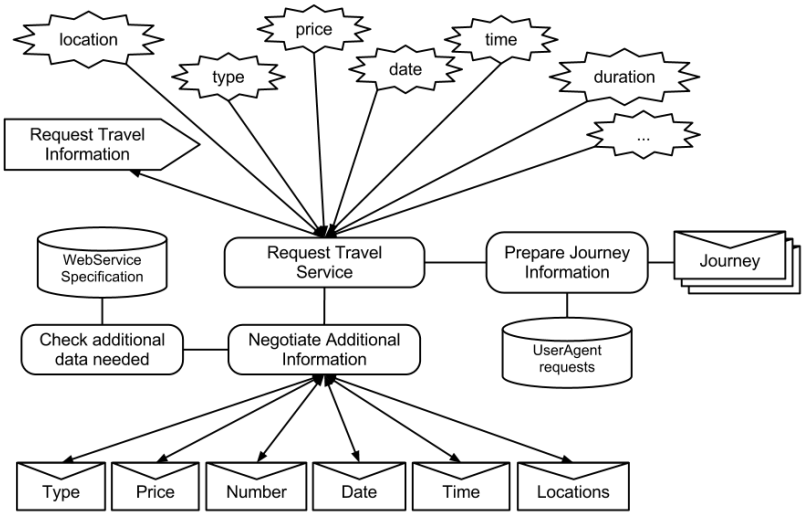

Figure 5.6: Travel Agent

The Travel Agents has four main capabilities:

- They access the travel web services, either by using its API or scrapping its web page.
- As Travel Agents have the specification of the web service the work with, they can check if any additional information is needed. This capability is particularly important as far as different web services need different input parameters, even although they may provide the same information.
- When the information provided is not enough, Travel Agent may negotiate with the User Agent the missing information.
- Once the journey -or journeys- information is obtained, the Travel Agent brings together all that information into a set of messages, each of them containing the information of a particular journey, that will be set to the User Agent.

5.4 Agent Implementation

The actual agent configuration needed for this prototype is fairly simple, yet it demonstrates the capabilities of the infrastructure. It consist of the following:

UserAgent responsible of initiating a conversation with the users and acting as a mediator between them and the rest of the users. It is this agent which keeps track of user preferences and past trips as well.

NLUAgent which connects to the NLU service and fetches all relevant information for the next steps. It is capable of understanding the intention of the message and calling the appropriate agent.

TravelAgent which performs the appropriate search according to the user criteria or informs the userAgent of any problem or lack of information.

These agents are developed using a particular implementation of the declarative language Agent Speak called Jason. This is supported by the agent platform of the same name, that allows the developer to choose a centralised infrastructure or a distributed one like JADE or SACI (although for the matter of agent development this is not relevant).

In the Jason platform, each agent is described by a source file -written in Agent Speak- that contains all the plans and goals descriptions of a BDI agent. Jason lets the programmer define additional external actions so he can deal with any library he needs. This is done in the Environment class associated to the MASs system. To be more precise, this does not need to be done in the Environment class, as long as the definition class were correctly instantiated in the environment.

5.4.1 NLU Agent

As explained, the NLU Agent has to transform the JSON object received into a set of beliefs. Lets consider the JSON object shown in List.5.1.

```
{"domains": {
    "travel": {
        "dates": {
            "return": 2012-03-31,
            "depart": 2012-03-29
        },
        "price": {
            "currency": EUR,
            "max": 250,
            "min": null
        },
        "queries": [ ],
        "locations": {
            "to": "barcelona",
```

```
            "from": "madrid"
        },
        "number": "1",
        "time": {
            "return": 10:00,
            "depart": 19:00
        },
        "type": [
            {  "vehicle": null,
                "providers": [ ],
                "type": "train",
                "business": null
            }
        ]
    }
},
"query_id": 123af87eb,
"queries": [ ]}
```

Listing 5.1: Excerpt of a JSON message received from the NLU Service.

Processing this JSON data will produce the set of beliefs listed in List. 5.2.

```
location(from, madrid)
location(to, barcelona)
date(daparture, 29, 03, 2012)
date(return, 31, 03, 2012)
time(departure, 10, 00)
time(return, 19, 00)
price(max, 250)
currency(eur)
type(train)
~scales
```

Listing 5.2: Except of a JSON message received from the NLU Service.

Before being sent to the User Agent, these beliefs are annotated with the query_id and the domain, so the User Agent may deal with different queries.

```
location(from, madrid) [query(123af87eb), domain(travel)]
location(to, barcelona) [query(123af87eb), domain(travel)]
date(daparture, 29, 03, 2012) [query(123af87eb), domain(
   travel)]
date(return, 31, 03, 2012) [query(123af87eb), domain(travel)]
time(departure, 10, 00) [query(123af87eb), domain(travel)]
time(return, 19, 00) [query(123af87eb), domain(travel)]
price(max, 250) [query(123af87eb), domain(travel)]
currency(eur) [query(123af87eb), domain(travel)]
type(train) [query(123af87eb), domain(travel)]
~scales [query(123af87eb), domain(travel)]
```

Listing 5.3: Excerpt User Agent source code for sending the Travel Agent all the information received from the NLU Agent.

5.4.2 User Agents

The code for the User Agent in the source code is simple. The excerpt shown in List. 5.4 is all the code needed to send the NLU Agent the user messages received. It checks whether the query is properly tagged, and if it is not, it generates a new random query id to uniquely identify a query.

```
+user_msg(Msg) : not query(_)
  <-  .random(Query);
      +query(Query);
      sendNLU(Query, Msg).
+user_msg(Msg) : query(Query)
    <-  sendNLU(Query, Msg).
```

Listing 5.4: Excerpt User Agent source code for sending the NLU Agent the users messages.

The excerpt shown in List. 5.5 is used for sending the Travel Agent the pieces of data about the user's request. As soon as it receives the information it is sent to the Travel Agent, who will save them until it has enough data.

```
+price(Terms, Price)[query(Query), domain(travel)] : true
    <- .send(travelAgent, tell, price(Terms, Price)[query(
       Query)]);
       .print("Percibido: price ",Terms, " ", Price ).

+date(Terms, Day, Month, Year)[query(Query), domain(travel)]
   : true
    <- .send(travelAgent, tell, date(Terms, Day, Month, Year)
       [query(Query)]);
       .print("Percibido: date ",Terms, " ", Day, " ", Month,
           " ", Year).

+time(Terms, Hours, Minutes)[query(Query), domain(travel)] :
   true
    <- .send(travelAgent, tell, time(Terms, Hours, Minutes)[
       query(Query)]);
       .print("Percibido: time ",Terms, " ", Hours, " ",
          Minutes).

+location(Terms, Place)[query(Query), domain(travel)] : true
    <- .send(travelAgent, tell, location(Terms, Place)[query(
       Query)]);
       .print("Percibido: location ",Terms, " ", Place).

+type(Terms)[query(Query), domain(travel)] : true
     <- .send(travelAgent, tell, type(Terms)[query(Query)]);
       .print("Percibido: type ",Terms).
```

Listing 5.5: Excerpt User Agent source code for sending the Travel Agent all the information received from the NLU Agent.

Additionally, when it notices it has received all the information available, it sends the

Travel Agent a message to ask it to achieve the find travels goal.

5.4.3 Travel Agents

Each travel agent contains a set of rules, defined according to the web service specification, that tells the agent when the data to fill in all the mandatory inputs of the form is available. In List. 5.6, a sample rule is shown.

```
canFindTravel(Query) :- location(from,_)[query(Query)] &
                        location(to,_)[query(Query)] &
date(departure,_,_,_)[query(Query)].
```

Listing 5.6: Web service mandatory fields checker rule

List. 5.7 shows simple set of plans to call the web service when needed data is available.

```
@findTravel1
+!findTravel(Query) : not canFindTravel(Query)
    <- .print("Not enough data").

@findTravel2
+!findTravel(Query) : canFindTravel(Query)
  <- ?location(to, To);
    ?location(from, From);
    ?date(departure, Day, Month, Year);
    findTravel(From, To, Day, Month, Year).

@findTravelFailure
-!findTravel(Query) : true  <- !findTravel(Query).

-!findTravel(Query) : errorMsg(Msg)
  <-    .print("Problema al encontrar viajes:", Msg);
    !findTravel(Query).
```

Listing 5.7: Simplified plans for finding a travel (and informing of failure)

5.5 Communication with Jason and Hook.io

This section clarifies some of the concepts about the architecture explained in chapter 4 and puts them in context. For this prototype we will go through the process of implementing the communication architecture using Jason as the agent platform and Hook.io and Socket.io to provide two separate communication buses. For this purpose, it will necessary to perform certain modifications to the vanilla Jason installation.

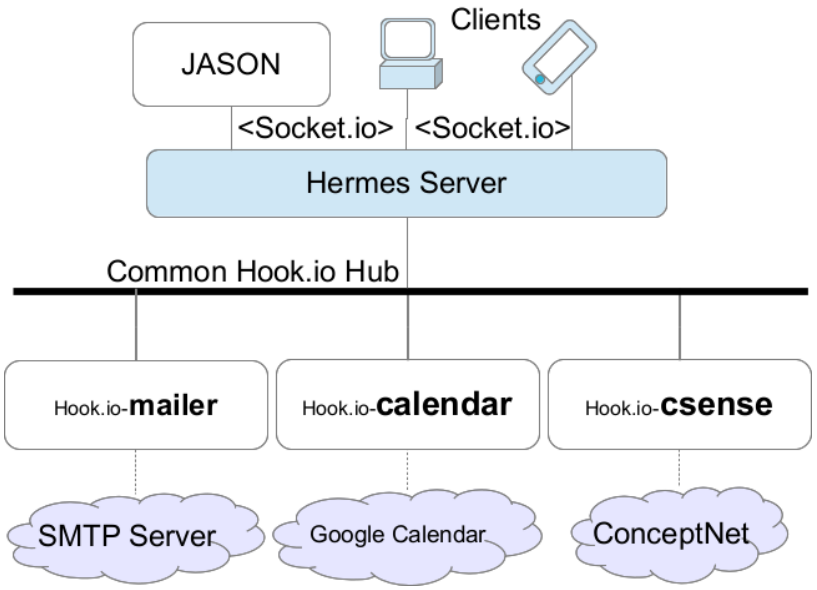

Figure 5.7: Different communication channels. In detail, hooks to external sources.

5.5.1 Data Sources

Every agent within Jason has its own knowledge database, which is populated by data from the different connectors.

To be able to actually modify the perceptions of the agents, a custom Jason Environment is needed, along with an ad-hoc model for this scenario. By modifying the basic Jason Environment one can not only to control the sources through which new information is added, but the life cycle of such information.

More precisely, the custom model follows the *data inbox* concept, the same as regular mailboxes. All information received by the agent is volatile, and will be discarded after it is fetched. Should the agent find the information interesting or necessary in the future, it will save it as beliefs in its permanent knowledge database.

Using these data inboxes it is incredibly easy to integrate Java code and agents in

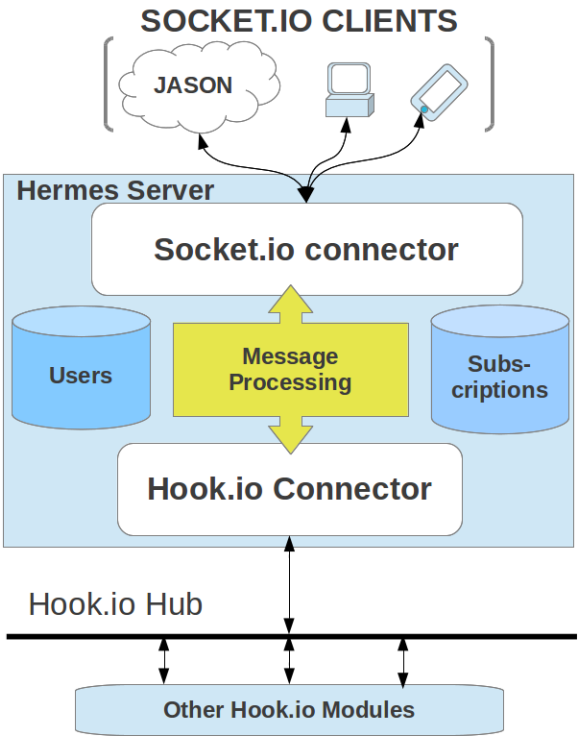

Figure 5.8: Interconnection of the different buses and protocols

AgentSpeak. One can call the appropriate function from within a Java method to send information to a certain agent, and create a wrapper function to allow our agents to call any Java method, which will probably return data by sending it to its data inbox.

There is a whole section describing the inner working of Jason in section 5.6.1.2.

5.5.2 Messaging and Communicating

While it is be useful to use this kind of model to add a list of web scrappers to fetch information from the web and an http server to communicate with the users (as illustrated in Fig. 5.9) , it is not as interesting as connecting it in a generic way that allows for easy incoming and outgoing connections. This, in addition to the benefits of event-based communications, makes making our Jason instance a node in a bigger platform like Hook.io a natural choice.

As seen in Fig. 5.8, the clients use Socket.io to connect to the back end, while the server components use a common hook.io channel. Thus a gateway or translator is needed. There are several reasons to follow this model instead of trying to unify it using a single hook.io bus, but the main important ones are:

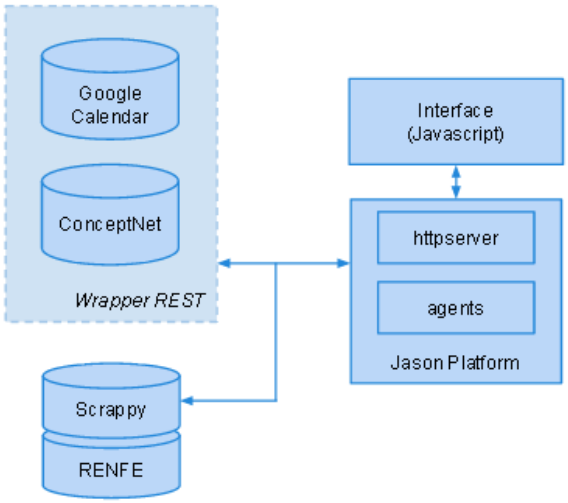

Figure 5.9: Architecture using http wrappers.

Security: Hook.io hooks are interconnected and adding the clients as well would mean that any client connected can see and tamper with the messages sent between modules of the backend. Separating both buses prevents users from reading all the events broadcasted in the hub.

Overhead: In an ideal scenario, where we would want to have a big number of users connected to the system, having all their messages and events forwarded to all the hook.io instances would mean a high overhead.

Asymmetry: While server modules will most probably be connected most of the time, clients need not be.

Robustness: By separating the back end from the front end we effectively decrease the chances of having a failure both in the client and the server side. Of course, a single point of failure is introduced for the client side (the gateway), but none of the other server functions would be affected by its failure.

There is another reason, purely technological, and it is that Hook.io is a new technology and there is not any suitable implementation in other languages.

For this last reason, Jason is also connected to the Hook.io hub using a Socket.io shim (embedded in the *Hermes Server*), instead of making Jason the central piece, until there is a mature implementation in Java. In this case, this disadvantage was turned into an advantage by allowing a pure Socket.io-to-Socket.io communication between the end users and the Jason agents. This prevents flooding the hub with user messages, as well as allowing

some kind of connection even in the event of a failure in the main hub. Moreover, it falls under the possible configurations of the proposed architecture, with more than a single hub.

5.6 Treatment of external events

The scenery proposed in this project does not fit the closed world assumption, i.e. it considers that external events may interact with the model, and so the agents may sense the changes they cause. It involves redesigning some modules of the Jason architecture, due to it has not been developed to explicitly support data importation.

This has two important effects: on the one hand, the need to change the architecture so it listens to external events; and on the other hand, the need to adapt it to let the agents correctly perceive their effects -at the particular point of the Jason reasoning cycle.

5.6.1 Web Service calls

Web service calls can be modelled as external actions in Jason. They may be performed using the functionalities SACI and JADE platforms offer, but they can also be made from the Model class.

Web service calls, as external actions in general, modify the information stored in the model with the data get in response to the service request. However, they cannot be modelled as simple sensors, because the data provided is not sustained over time, and therefore some mechanisms to ensure it is received by the agents must be developed. This will be discussed in the next sections.

5.6.1.1 Concurrent calls

Jason architecture has been developed to bear with execution of concurrent external actions, so it can deal with those whose execution time lasts more than what it is normal for a single action; either because it depends on some external resource or because its computational complexity is too high. Thus, Jason is prepared to concurrently run different external actions, so the reasoning cycle never stops.

This is the case of web service calls: once they are waiting for response they yield the processor to other processes until they get the requested data. Jason architecture, internally works associating the execution of actions -internal and external- to intentions. So one could say the execution of an intention consist on a sequential execution of the actions that belong

to the intention. Every time an external action is carried out, Jason architecture change the context so it keeps on the execution a of different intention. When the execution of the external action ends, the intention is marked as resumable, so the architecture can resume its execution. If necessary, Jason architecture may carry out as many steps -relative to other intentions- until the execution of the external action ends.

The figure 5.10 shows the UML sequence diagram that explains the execution steps taken by the Jason architecture when calling a web service.

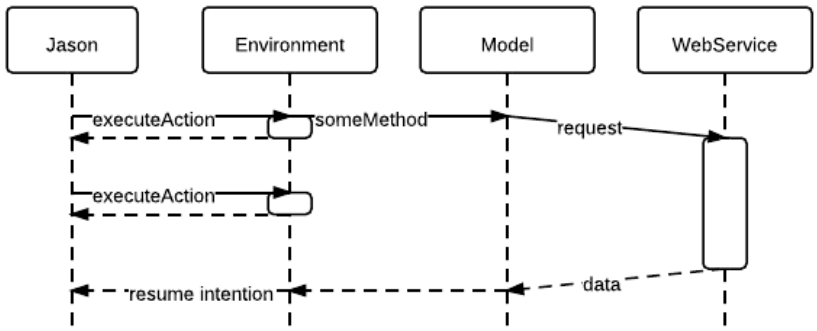

Figure 5.10: UML sequence diagram that involves calling a web service.

The proposed design means an important advantage, due to agents can complete their plans, by resuming intentions, even if they involve long idle intervals, without loosing the context.

5.6.1.2 Data representation in Jason system

The design of Jason architecture and language was not developed taking into account the needs of including external data into the platform. It is definitely prepared to cope with sensors and long execution time tasks -as mentioned before. However, to represent the information read by the sensors, the designer of the system must choose a format, and develop the code that is needed to translate that measurements into Jason believes with the given format. This forces the designer to know, in advance, all the possible states of the environment sensors, and this is not always acceptable. Sometimes it simply cannot know it, specially when working with external events and web services whose response does not fit a standardization.

Let's consider a web service that provides the weather forecast in Madrid. At some point, the forecast is sunny, and one hour later it is partly cloudy. We need to have a glossary that embraces all the possible terms to describe weather forecasting. What is more difficult is

to have a dictionary that contains all the possible locations, so some rules must be used to transfer that information into the belief base. The representation is as follows:

```
forecast(madrid, time(18,00), wheather(sunny)).
forecast(madrid, time(19,00), wheather(partly_cloudy)).
```

Listing 5.8: Data representation in Jason

Jason supports string handling, however strings cannot be treated as literals to use all the capabilities of Jason reasoning engine, i.e. in Jason string are just textual messages for the user. Thus, the following translations were made to put the information into the belief base as literals:

- In Jason, words that start with a capital letter are variables, so the input is put into lowercase.
- Jason does not accept white-spaces in literals, so they are replaced by '_'.
- A specific format is used to represent the time, splitting the time into hours and minutes values, since Jason cannot work with punctuation marks.

Although the translations made may be obvious, it is important to establish some rules so every programmer knows them to avoid misunderstanding. For instance, there no reason to user whitespace replacement by '_' instead of using camel case to represent white-spaces, it is just a matter of agreement.

The main difficulty is trying to reverse the process and obtain the original expression in its exact format. Due to some of the rules mentioned before does not have a reverse rule that uniquely returns the original expression from the adapted terms.

In those cases, where the User Interface is important, one may consider using a so called translation service -either an agent or an auxiliary module- that lets the system undo the transformation; not using rules but retrieving the correspondence from a temporal database that stores the original expressions translated.

5.6.1.3 Availability of the data received

Data received from web services cannot be processed as sensors data, it has some singularities that make us adapt the architecture to treat them properly. Although, the way it is put into

the Jason system is by representing it as percepts -with all the considerations of formatting agreed in section 5.6.1.2-, the data received received from web services are not effect of sensor detection or measurement. The date received is effect of the request made to the web service, thus the availability of that information does not follow the same rules as physical or state information measured by sensors.

The main difference is, as they do not represent a physical state, its availability by the sensors, which normally indicate that a perception is no longer available when they stop detecting it. Web service data is information received that, therefore, must be removed from agent perception as soon as that agent received them. Since, unlike what happens with sensors (where whether a perception does not lasts enough to be processed by the involved agents it means that the perception is not relevant) the information has been requested by the agent and the architecture must ensure it receives it.

The solution proposed is building, in the Model class, a service-data inbox that stores the incoming data. So that, the data is stored according to the agent that will receive them. Once the agent has accepted them, they will be removed from the data inbox -as someone remove a letter from the mailbox. As they are defined as part of the private perception, i.e. that perception that is particular of any single agent, it will only be removed from the inbox when the perceiving agent is the addressee of the data. Thus, it guarantees the information is correctly received, regardless the delivery may be delayed.

On the other hand, as it is explained before, the updating protocol defined in the Environment class must take into account the special nature of the serviceDataInbox, to guarantee the agents effectively receive the information it stores.

5.6.2 Modelling input events

By taking into account the considerations made about the availability of the data from web services and the solution proposed, we can consider modelling two different kind of external events. Those that modify the state or the representation of physical objects, in the Model, whose effect will be perceived by the agents until it is changed or removed. And those that can be modelled as notifications or external messages -to particular agents or to the group- that will be included in the serviceDataInbox as mentioned.

Once the approach is defined, we must bear in mind that those attributes of the Model, including the serviceDataInbox, that can be modified from the outside may be accesses simultaneously by some different sources, and the information may get corrupted. Hence, the model must provide synchronizes method to assure the integrity of the data.

5.6.3 Percept updating policies

We call percept updating policies -or protocols- to the different mechanisms by which the Environment class captures the representation of the environment from the Model class and updates its own percepts.

In a first approach, we may be tempted to update the environment whenever any data changes in the model. However, if some particular situations this proposition may cause problems. Lets consider a high-accuracy temperature sensor, the value it measures is continuously changing due to its high precision. That means the sensor is changing the value that represents the temperature in the model every short period of time. Let also consider, incoming notifications as mentioned in previous sections, the policy we should use is different in both cases.

There are different suitable updating policies depending on the nature of the data involved.

5.6.3.1 Update base on external action execution

Update base on external action execution, is the easiest and most used way in simulated environments on Internet Jason examples involves updating the Environment percepts after every execution of an external action. This policy assumes the system respond to the closed world assumption, so the Model can only be modified by external actions executed inside the Jason architecture. Applying this method when the assumption is not met, the architecture cannot guaranty the percepts transferred to the agents were correct.

5.6.3.2 Continuous updating

Continuous updating policy means reflecting every single change made in the Model in the Environment class as soon as it occurs. The objection of this policy lies on situations where the changes in the model are fulfilled very fast. In this cases it becomes very inefficient. Moreover, in most of the cases, this frequency is not required, e.g. in the high accuracy temperature sensor environment, the small changes should not be reflected in the model because they are not relevant, temperature change at a certain speed, and this defines the relevant variation frequency. Furthermore, the implementation of this updating protocol happens to let the Model change the Environment -or at least inform it- what is against the main design of Jason architecture.

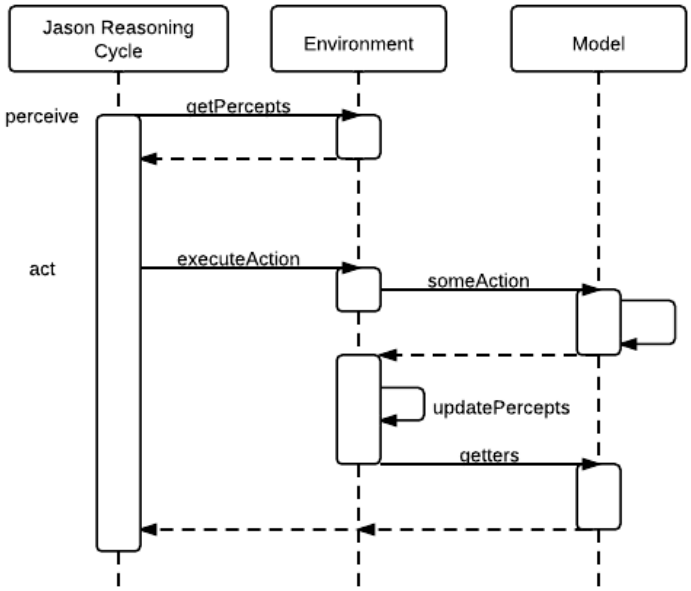

Figure 5.11: Update base on external action execution

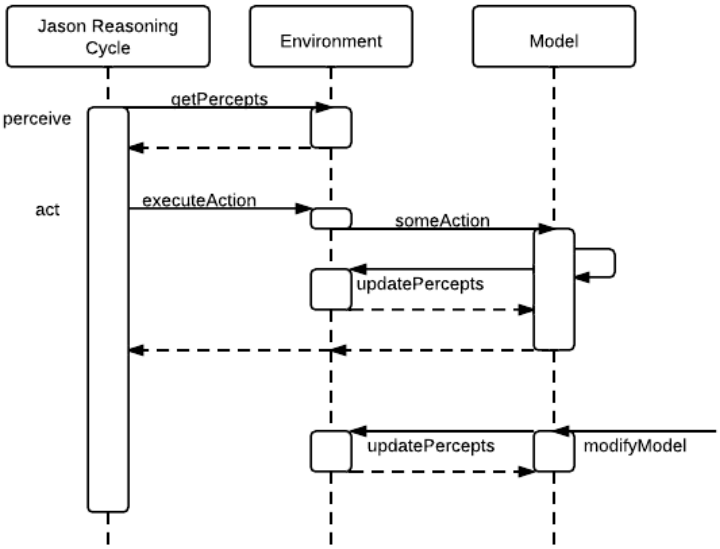

Figure 5.12: Periodic Sampling

5.6.3.3 Periodic sampling

Periodic sampling policy means updating the Environment perception capturing the Model data only when it is necessary. Either because something has changed in the model or because of the execution of an external action that requires updating the percepts. To do so, the model must provide some testing methods that indicate if the update must be done.

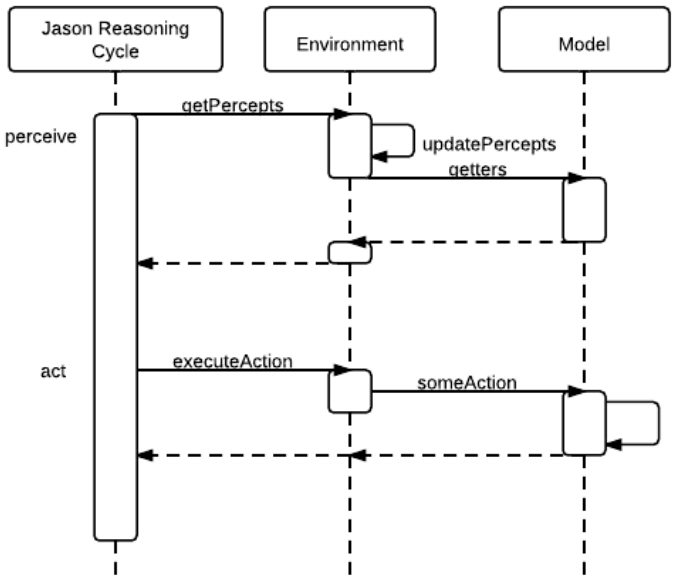

Figure 5.13: Continuous updating

Chapter 6

Conclusion and future work

"It is always wise to look ahead, but difficult to look further than you can see."

— Winston Churchill

6.1 Conclusions

By following a simple set of rules, in the new paradigm exposed in this master thesis, it is possible to achieve modern systems that combine the potential of intelligent agent systems and the interconnection and limitless applications of the modern web.

Moreover, we can make good use of the existing technologies, as it has been shown. The more important shift is in the way we understand agents and agent communication.

Following the event oriented programming paradigm and using the simple set of rules defined for Maia, working with multiple data sources was a breeze. Partly due to the agility provided by Node.js and using Javascript for both server and client sides.

There is a growing community of developers that are following these principles, even if it is only by means of using Javascript for their projects. This fact is reassuring, and means that the continuity of the new ideas discussed in this document is guaranteed.

It is impossible to know how the Web, or more generally the Internet, will evolve. However, it is clear that more and more services and applications are embracing the evented paradigm, at least in the way of simple web hooks. Because of this, it is quite easy to en-

vision a new Internet, as described in *The Live Web* book [2]: services interacting without previous consensus, unimaginable interactions between applications, new sources of information like DVRs or home appliances...

6.2 Achieved goals

This document started with a set of goals to be achieved with this master thesis. To wrap up, here is a summary of the final outcomes:

- A communication channel and protocol to connect Clients and Cloud Agent(s). Examples were given of both hook.io and socket.io working, which gives more than one channel and protocol. The protocol defined for Maia, consisting of the event and namespace definition, can also be used with any other technology.
- A communication channel and protocol to connect Cloud Agents. The same as previous point. One of the advantages of the proposal in this document is how simple the communication mechanism is. For details, read Chap. 4.
- A generic and extensible schema for intents and actions. Although it has not been the key outcome, an extensible ontology was created as described in Sec. 4.3 that describes intents and actions.
- A scalable platform to deploy Cloud Agents. See Chap. 5 for an example implementation using a modified Jason instance as the agent platform.
- To develop all the complex logic needed in every specific Cloud Agent. See Sec. 5.3.
- To define security constraints and authentication methods. Some guidelines are introduced in Sec. 6.3.1.

As for the more general aims of this project:

- To study and extend the current state of the art of Web Hooks and Web Intents
- To explore the capabilities of such technologies for inclusion in intelligent systems
- To explore and exploit the potential of Javascript in server applications (see Node.js in section 2.2.2)
- To demonstrate the versatility of the given models for communication, beyond programming languages or platforms

- To provide a robust and simple bidirectional connection between Java and Javascript programs.

6.3 Future Work

There are many lines that can be followed to continue this work. Some compromises were made to achieve a working prototype that could show the potential of the new design paradigm, so some aspects were abandoned for the sake of simplicity and speed.

Now that the concept has been proven to work and be very promising, it is time to continue growing both the architecture and the developed tools to give a more solid solution to a wider range of scenarios. In the following sections some fields of study or improvement are presented to the reader.

6.3.1 Security

One of the main aspects to improve from a pragmatic point of view is the security of the information being exchanged and the scope in which it is visible. The explained hierarchy based on hubs is just a practical first approach to limiting these problems, but it is important to apply more advanced security policies and authentication of entities. This is especially important if the information being shared is confidential or if not all the parties in contact are trusted, which is likely to happen in open endpoints on the internet.

It would be advisable to apply the same simplicity principles applied during the original design and implementation described in this document. There is a handful of simple solutions for authentication and limiting scopes for Web applications which can be applied or ported easily to Maia.

6.3.2 Chaining services using hooks

More important from the research side is to go one step forward in our definition of events and the propagation of plans using them. As previously said, everything being sent is an event, in the form of an intention or a change of information. This idea can be further expanded to allow the propagation of plans within an agent system using the Maia architecture.

There are some different approach for chaining services using hooks. The easiest consist on using to handlers for returning data to the emitter hook so it orchestrates the composition

based in the data received. The second design consist on using events to communicate between services and the agent platform.

This last approach, implies developing a communication protocol so the events emitted are used as messages passed between actors involved. In addition to the complexity add to the development process, using events to communicate among agents and services it is not a good practice, since the hook/events where designed for being used programmatically, not for being part of a communication protocol. Thus it is more convenient to use callbacks to return the resulting data to the emitter, so it can process it.

6.3.3 Better integrate Web Hooks

Right now, all the connectors to web services are ad-hoc solutions, and have been fine tailored for their use in this scenario. Developing a standard connector to translate regular web hooks into Maia event emitters and receivers. Having such a compatibility layer would make it possible to build rich applications that integrate both services that expose web hooks and services built using Maia nodes.

6.3.4 Define an ontology of objects

Standardizing namespaces and creating an ontology of actions is not sufficient in the long term, as more and more services begin to use events to communicate to each other. In the end, it is necessary to also provide a definition of the payload, or objects to be passed.

With the already existing ontology of intents, it should be trivial to add an extensible format of objects and make an ontology of it as well.

6.3.5 Interacting with the Kinetic Rule Engine

It would be interesting to use either the Javascript library to connect with a Kinetic Rule Engine (KRE) [2], or the one written for Node.js [55].

Even though these are only client libraries, and a Kinetic Rule Engine must be running also, the potential combination of Maia events and the processing capabilities of the KRE is really promising.

Another approach could be to port the KRE to Javascript, adding the many advantages of Node.js (see Sec. 2.2.2). But this task would be much more complex. However, given the rapid evolution and changes in Node.js it may be justified, should the technologies related

to the Kinetic Rule Language evolve and be adopted by a wider audience. The combination would be a really powerful tool, example of the blossoming technologies on the Internet that are leading to an evented web.

Appendix A

Installing Node.js and Hook.io

This tutorial goes through the process of writing a simple mailer hook capable of sending email alerts to your users. It covers the whole process, from installing node.js and hook.io 0.8.7 in Ubuntu 11.10 to running your test hook.

A.1 Install node.js

To install node, you can either compile the sources or use the binaries available in the ppa repository. We will cover the installation using the ppa repository as it's easier and will keep our installation updated effortlessly.

First of all, we add the ppa repository:

```
$ sudo apt-get install python-software-properties
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
```

Now, we install the node.js package, and npm (nodejs package manage) so we can install node modules easily later:

```
$ sudo apt-get install nodejs npm
```

Or, if you want to compile Node C++ modules:

```
$ sudo apt-get install nodejs-dev
```

You can now test your node.js installation. For this, open a new file called "`helloworld.js`" with the following code:

```
1 #!/bin/env node
2 var http = require('http');
3
4 http.createServer(function (req, res) {
5   res.writeHead(200, {'Content-Type': 'text/plain'});
6   res.end('Hello World\n');
7 }).listen(1337, "127.0.0.1");
8
9 console.log('Server running at http://127.0.0.1:1337/');
```

And run it with:

```
$ node helloworld.js
```

Now click the link in the terminal, or open your browser and go to:

`http://127.0.0.1:1337/`

Congratulations! You ran your first node.js application!

A.2 Install Socket.io

If you installed npm, installing hook.io is a piece of cake. But if you're using Ubuntu 11.04+, you will need to install some avahi dependencies or you'll get nasty errors in compile time:

```
$ apt-get install libavahi-compat-libdnssd-dev
```

Once the dependencies have been installed, we carry on with the real installation:

```
$ sudo npm install hook.io -g
```

The `-g` option causes npm to install hook.io globally, in the appropriate folder in `/usr/lib`. Without it, the module will be downloaded and installed in the current working directory. Later, you will need to install it locally, but first let's install it this way so you can use the hook.io tools from the command line.[1]

If everything went well, now you can test your hook.io by launching your first hook:

```
$ hookio
```

This instance will act as the server hook, as explained before. Now every instance of hookio launched will connect to the first one on port 5000. To make it more interactive, you can launch hookio with a read-eval-print loop:

```
$ hookio --repl
```

Once connected, you will be able to write to a node console. To send your first event, write:

```
$ hook.emit("foo","bar")
```

Which sends the `foo` event with `bar` as data/payload. If you go back to the first instance you executed, you should see the log of the `foo` event. However, if you have more instances open, you won't see any sign of the event in your log. Do not worry, I'll show you how to subscribe to events in the following chapter.

[1] For more information, visit:
`https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager`

Appendix B

How to use hook.io hooks

the power of the developed architecture comes from the ability to easily add data sources and endpoints. the simplest way is by using hooks, leaving all agents design aside. Listing B.1 shows the basic skeleton to get started with two simple hooks that communicate with each other. It does the following:

- Create a new hook (`hookA`) named `a`.
- With the method `on`, bind a callback function with the specified event filter. In this case, print event and data every time any hook (`*`) sends the `sup` event.
- Start the hook. This method tries to find any other hook instance using the default port and connects to it. Otherwise, it becomes accepting incoming connections on that port.
- Create another hook (`hookB`) named `b`.
- As soon as the new hook is ready, it will emit the event `sup`, using `dog` as data.
- Start `b`. Since `a` was started first, this will make `b` connect to it. This action will lead to a chain of events that will end with `a` printing `b`'s original event and data.

```
#!/env/nodejs
var hook = require('hook.io').hook;

var hookA = new Hook({
  name: "a"
});

hookA.on('*::sup', function(data){
  // outputs b::sup::dog
  console.log(this.event + ' ' + data);
});

// Hook.start defaults to localhost
// it can accept dnode constructor options ( for remote connections )
// these hooks can be started on different machines / networks / devices
hookA.start();

var hookB = new Hook({
  name: "b"
});

hookB.on('hook::ready', function(){
  hookB.emit('sup', 'dog');
});

hookB.start();
```

Listing B.1: basic hook.io hook template

B.1 Tips with Hook.io

Here is a list of useful things to take into consideration when developing a hook.io hook.

- Once emitted, event names are prepended their origin hook's name. For that reason, you have to use `on("*::event",...)` instead of just `on("event",...)`
- Your server hook should have no methods at all, use a vanilla hook
- Follow the node.js convention: the main js in `/bin/index.js`, the libraries in `/lib/`, and a package.json. This will make it easier to upload your hook for use with npm.

B.2 Troubleshooting and known bugs

Hook.io is a new technology, not even in its 1.0 version, so the number of bugs and feature requests is incresing. You can see the full list in their github page[1], but I'll give you a short list of the main problems I encountered:

- If you can't install hook.io using the npm version in the repositories, try installing the latest version as root:

```
$ curl http://npmjs.org/install.sh | sh
```

- Disconnecting/killing the server instance will make the clients enter a loop.
- It's not possible to remove a wildcard event once added with `hook.on`
- All the hooks have to use the same hook.io version or you'll get an error
- You can't emit using a filter of the form `foo::*` as it will raise an exception

B.3 Further reading

For more information about hook.io and how to set it up, please visit:

- `http://www.nodebeginner.org/`
- `http://ejeklint.github.com/2011/09/23/hook.io-for-dummies-part-1-overview/`

[1] `https://github.com/hookio/hook.io/issues`

Appendix C

Excerpts of code

This section contains some of the code of the Jason Agents that are part of the prototype explained in Chap. 5.

For more information and code to modify Jason (see 2.3) to connect to Socket.io (see 2.2.3) visit the SOJA project in GitHub:

`https://github.com/balkian/SOJA`

For information about the hook.io and socket.io event router Hermes, visit:

`https://github.com/balkian/Web4.0`

A set of use examples can be found in:

SparQl full demo `https://github.com/balkian/Hook.io-Sparql-Demo`

Spotlight hook `https://github.com/balkian/Hook.io-Spotlight`

Mailer hook `https://github.com/balkian/Hookio-Mailer`

All of which use the Web 4.0 Hook.io (see 2.2.4) hook class defined in this GitHub repository:

`https://github.com/balkian/hook.io-web40`

C.1 travelAgent

```
1  // Agent travelAgent skeleton
2  //
3  // 'findTravel' external action is provided. The exact format
4  // of the action is as follows:
5  //
6  //    findTravel(Queryid, From, To, Day, Month, Year)
7  //
8  // where the Queryid is a number that may be used
9  // to identify the the results of a particular call.
10 // this external action always successes. When the data
11 // is processed it includes the beliefs in the agent
12 // belief-base. The format
13 // of the beliefs is as follows:
14
15 //  journey(<From>, <To>, <Departure>, <Arrival>,
16 //                                    <Fare>)[query(Queryid)]
17 //   journey(madrid, barcelona, time(11,00), time(14,30),
18 //                        fare(FName, FPrice))[query(12345)]
19
20 /* Store results */
21 @storejourney1
22 +journey(From, To, Departure, Arrival, fare(FName, FPrice))[
      query(Query)]
23   :   true
24   <-  .print(journey(From, To, Departure, Arrival, fare(FName
       , FPrice)));
25    +journey(From, To, Departure, Arrival, fare(FName, FPrice
       )).
26
27 @findtrip
28 +!find_trip[source(nluAgent)]
29   : true
30   <-  .print("Finding trip");
31    ?location_to(To);
```

```
32      ?location_from(From);
33      ?day(D);
34      ?month(M);
35      .date(YY,MM,DD);
36      if ((MM+1) > M) {
37        findTravel(12345, From, To, D, M, YY+1);
38      }
39      if ((MM+1) < M) {
40        findTravel(12345, From, To, D, M, YY);
41      }
42      if ((MM+1) == M) {
43        if (DD > D) {
44          findTravel(12345, From, To, D, M, YY+1);
45        }
46        if (DD <= D) {
47          findTravel(12345, From, To, D, M, YY);
48        }
49      }
50      .wait(500);
51      .findall(viaje(FPrice, F, T, Departure, Arrival, Fare),
52           journey(F, T, Departure, Arrival, fare(Fare, FPrice)
                ),
53           Journeys);
54      .max(Journeys, Cheapest);
55      Cheapest = viaje(CPrice, CFrom, CTo, CDeparture, CArrival
           , Name);
56      !check_price(CPrice);
57      .send(userAgent, achieve,
58          show(CFrom, CTo, CDeparture, CArrival, fare(Name,
                CPrice)));
59      .print("Done!");
60      .abolish(_[source(_)]).
61
62  -!find_trip[source(nluAgent)]
63    : true
64    <-  .send(userAgent, achieve, fail);
```

```
65      .abolish(_[source(_)]).
66
67  /* +?location_to(To): not location_to(_)
68    <-  .print("No to");
69      -doable(yes);
70      ?missing_message(MM);
71      -missing_message(MM);
72       .concat(MM,"Hacia?",MM);
73       -+missing_message(MM).
74  +?location_from(From): not location_from(_)
75    <-  .print("No From");
76      ?missing_message(MM);
77       .concat("Desde?",MM,MM);
78       -+missing_message(MM);
79       -doable(yes).
80
81  +?day(Day): not day(_)
82    <-  .print("No day");
83       ?missing_message(MM);
84       .concat("Dia?",MM,MM);
85       -+missing_message(MM);
86      -doable(yes).
87
88  +?month(Month): not month(_)
89    <-  .print("No month");
90      ?missing_message(MM);
91       .concat("Mes?",MM,MM);
92       -+missing_message(MM);
93      -doable(yes).
94
95  +?doable(_): not doable & missing_message(MM)
96    <-  .print("Not doable");
97       .send(userAgent, achieve, send_user(MM));
98       .remove_plan(find_trip[_]). */
99
100 +!check_price(Price): not price(_)
```

```
101   <-  .print("No price limit").
102
103 +!check_price(Price): price(Max) & Max > Price
104   <-  .print("Price");
105     .print(Price);
106     .print("Limit");
107     .print(Max);
108     .print("The price is under the limit").
109 -!check_price(Price): price(Max)
110   <-  .print(Max);
111     .print(Price);
112         .send(userAgent, achieve, send_user("Maybe you should
113        consider spending a bit more."));
114     .remove_plan(find_trip[_]).
```

Listing C.1: travelAgent code

C.2 userAgent

```
1  // Agent userAgent skeleton
2  //
3  // Messages from the user are expressed as beliefs with the
4  // format:
5  //
6  //    user_msg("message string here")
7  //
8  // 'sendUser' external action is provided. It can be used to
9  // send a literal or a string to the user.
10 //
11 //    sendUser(<literal>|<string>)
12 //
13 // When the literal has the specific format
14 //
15 //  journey(From, To, Departure, Arrival,
```

```
16 //                                fare(FName, FPrice))
17 //
18 // the client prints the data in a special format, as
19 // it recognizes it as a journey.
20 //
21 /* Initial beliefs and rules */
22 //
23 //user_msg("Sugiere algo").
24 //user_location(madrid).
25 //user_msg("Quiero un tren de Madrid a Barcelona
26 //                   por 60 euros el 27 de Julio").
27 /* Initial goals */
28 //
29 /* Plans*/
30
31
32 +user_msg("Gracias") : lastjourney(From,City,Start,End,fare(
       Name,Price))
33   <- findTasks(City);
34     .wait(500);
35     .findall(D,task(F,D),
36          Tasks);
37     .concat("Vas a ",City,", no olvides: ",Tasks,Y);
38     !send_user(Y);
39     -user_msg("Gracias").
40
41
42
43
44 @msg_thanks
45 +user_msg("gracias") : lastjourney(From,To,Start,End,fare(
       Name,Price))
46   <-  !send_user("De nada").
47
48 +user_msg("gracias") : true
49   <-  !send_user("No hay anterior viaje").
```

```
50
51  @msg
52  +user_msg(Msg) : true
53    <-  .send(nluAgent, tell, query(Msg)).
54
55
56  +user_location(Location) : true
57    <-  .print("Location is ",Location);
58      .send(nluAgent,tell,user_location(Location)).
59
60  @senduser
61  +!send_user(Msg) : true
62    <- sendUser(Msg);
63      .abolish(_[source(_)]).
64
65  @show_journey
66  +!show(From, To, Start, End, fare(Name, Price))
67    : true
68    <-  .print("Last JOURNEY:",lastjourney(From,To,Start,End,
        fare(Name,Price)));
69      sendUser(journey(From, To, Start, End, fare(Name, Price))
          );
70      -+lastjourney(From,To,Start,End,fare(Name,Price));
71      !updateTrips(To).
72
73  +!updateTrips(To): trips_to(To,N)
74    <-  .print("YOU ALREADY WENT THERE");
75      Nn = N+1;
76      -+trips_to(To,Nn).
77
78
79  +!updateTrips(To): true
80    <-  .print("The user hasn't been there.");
81      +trips_to(To,0).
82
83  +task(For,Do): true
```

```
84    <-  .print(">",tasks(For,Do));
85      +task(For,Do).
86
87  @fail_nlu
88  +!error
89    : true
90    <-  sendUser("Parlez vous francais? Do you speak English?")
          ;
91      .abolish(user_msg[source(_)]).
92
93  @no_match
94  +!fail
95    : true
96    <-  sendUser("No hay resultados. Lo siento");
97      .abolish(user_msg[source(_)]).
```

Listing C.2: userAgent code

C.3 nluAgent

```
1   // Agent nluAgent skeleton
2   //
3   // A sample plan has been given. The format of the triggering
4   // event is optional, since it depends on the message send by
5   // the userAgent (defined by the programmer)
6   //
7   // In order to develop the multi-agent system before the
        graphs
8   // of the unitex module, a plan like 'demo_msg' may be used.
9   // That plan uses the 'sendNLU' external action is provided.
10  //
11  // It must be used to send data to the unitex module. The
12  // format of the action is as follows:
13  //
```

```
14 //  sendNLU("msg to nlu")
15 //
16 // This external action always succeeds. When the data is
17 // processed it includes the beliefs in the agent belief-base
18 //
19
20 /* Plans */
21
22 @demo_msg
23 +!demo_msg(Msg, Queryid)[source(Ag)] : true
24     <-  .send(Ag, tell, location_from(madrid));
25       .send(Ag, tell, location_to(barcelona));
26       .send(Ag, tell, departure_date(8,5,2012));
27       .send(Ag, tell, num_people(3)).
28
29 @demo_nlu
30 +!demo_nlu : true
31   <- sendNLU("Quiero un viaje para 3 personas de Madrid a
       Valencia").
32
33 +!ask_agent: true
34   <-  .print("Asking the agent");
35     .send(travelAgent, achieve, find_trip);
36         .abolish(_[source(_)]).
37
38 @query_nlu
39 +query(Msg): true
40   <-  .print("Told to query");
41     -query(Msg);
42     sendNLU(Msg);
43     .wait(500);
44     +finished.
45
46 +finished: not query_type(suggestion)
47   <-  !ask_agent.
48
```

```
49 +finished: user_location(Location)
50   <- .print("Sugerencia");
51     .print("Location:",user_location(Location));
52     findHolidays;
53     .wait(500);
54     .findall(holiyad(R,Y,M,D),
55          holiday(M,D,Y,R),
56          Holidays);
57     .min(Holidays, Best);
58     Best = holiyad(Rep,Year,Month,Day);
59     .print("Rep:",Rep);
60     .print("Year:",Year);
61     .print("Month:",Month);
62     .print("Day:",Day);
63     .print("Location:",Location);
64     -query_type(suggestion);
65     -+month(Month);
66     -+day(Day);
67     -+year(Year);
68     -+location_from(Location);
69     -+location_to(valencia  );
70     -+price(1000);
71     -+finished.
72
73
74
75 +query_type(suggestion): true
76   <- +query_type(suggestion).
77
78 +location_from(From): not query_type(suggestion)
79   <-  .send(travelAgent,tell, location_from(From)).
80
81 +location_to(To): not query_type(suggestion)
82   <-  .send(travelAgent,tell, location_to(To)).
83
84 +number_people(N): not query_type(suggestion)
```

```
85    <-  .send(travelAgent,tell, number_people(N)).
86
87  +price(P): not query_type(suggestion)
88    <-  .send(travelAgent,tell, price(P)).
89
90  +day(D): not query_type(suggestion)
91    <-  .send(travelAgent,tell, day(D)).
92
93  +month(M): not query_type(suggestion)
94    <-  .send(travelAgent,tell, month(M)).
95
96  +holiday(D,M,Y,R): true
97    <-  .print(">",holiday(D,M,Y,R));
98      +holiday(D,M,Y,R).
```

Listing C.3: nluAgent code

Bibliography

[1] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, July 1993.

[2] P.J. Windley. *The Live Web: Building Event-Based Connections in the Cloud.* Course Technology, 2011.

[3] Mor Naaman, Jeffrey Boase, and Chih-Hui Lai. Is it really about me?: message content in social awareness streams. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, CSCW '10, pages 189–192, New York, NY, USA, 2010. ACM.

[4] Hans W. Gellersen, Albrecht Schmidt, and Michael Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications*, 7:341–351, 2002. 10.1023/A:1016587515822.

[5] L. Ardissono, G. Bosio, and M. Segnan. An activity awareness visualization approach supporting context resumption in collaboration environments. In *International Workshop on Adaptive Support for Team Collaboration*, pages 15–25, 2011.

[6] T. Gross, W. Wirsam, and W. Graether. Awarenessmaps: visualizing awareness in shared workspaces. In *CHI'03 extended abstracts on Human factors in computing systems*, pages 784–785. ACM, 2003.

[7] B.P. Bailey, J.A. Konstan, and J.V. Carlis. The effects of interruptions on task performance, annoyance, and anxiety in the user interface. In *Proceedings of INTERACT*, volume 1, pages 593–601. IOS Press, 2001.

[8] Kinetic rules engine. `https://github.com/kre/Kinetic-Rules-Engine/`.

[9] M. Luck, P. McBurney, and C. Preist. *Agent technology: enabling next generation computing (a roadmap for agent based computing).* AgentLink/University of Southampton, 2003.

[10] Website of node.js. `http://nodejs.org/`.

[11] Website socket.io. `http://socket.io/`.

[12] Github repository of hook.io. `https://github.com/hook.io/`.

[13] Official website of the jason project. `http://jason.sourceforge.net/`.

[14] Miguel Escrivá Gregori, Javier Palanca Cámara, and Gustavo Aranda Bada. A jabber-based multi-agent system platform. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 1282–1284, New York, NY, USA, 2006. ACM.

[15] Xep-0004: Data forms. `http://xmpp.org/extensions/xep-0004.html`.

[16] Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42:213–261, March 1990.

[17] Karen Zita Haigh, Liana M. Kiff, Janet Myers, Valerie Guralnik, Kathleen Krichbaum, John Phelps, Tom Plocher, and David Toms. The independent lifestyle assistant tm (i.l.s.a.): Lessons learned, 2003.

[18] Karen Zita Haigh, John Phelps, and Christopher W. Geib. An open agent architecture for assisting elder independence. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, AAMAS '02, pages 578–586, New York, NY, USA, 2002. ACM.

[19] I. Ohmukai, H. Takeda, and M. Miki. A proposal of the person-centered approach for personal task management. *2003 Symposium on Applications and the Internet, 2003. Proceedings.*, pages 234–240, 2003.

[20] Julita Vassileva. A review of organizational structures of personal information management. *Management*, pages 1–19, 2008.

[21] Karen L. Myers, Pauline Berry, Jim Blythe, Ken Conley, Melinda T. Gervasio, Deborah L. McGuinness, David N. Morley, Avi Pfeffer, Martha E. Pollack, and Milind Tambe. An intelligent personal assistant for task and time management. *AI Magazine*, 28(2):47–61, 2007.

[22] M. Indiramma and K.R. Anandakumar. Collaborative decision making framework for multi-agent system. In *Computer and Communication Engineering, 2008. ICCCE 2008. International Conference on*, pages 1140 –1146, May 2008.

[23] Yolanda Gil, Paul Groth, and Varun Ratnakar. Social task networks: Personal and collaborative task formulation and management in social networking sites. In *Proceedings of the AAAI Fall Symposium on Proactive Assistant Agents*, 2010.

[24] Miquel Montaner. A Taxonomy of Recommender Agents on the Internet. *Artificial Intelligence Review*, pages 285–330, 2003.

[25] Andreas Lommatzsch, Martin Mehlitz, and Jérôme Kunegis. A multi-agent framework for personalized information filtering. In *Proceedings of German e-Science 2007 (GES'07)*. Max Planck Digital Library / German e-Science Conference, 2007.

[26] Donald A. Norman. How might people interact with agents. *Commun. ACM*, 37:68–71, July 1994.

[27] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *ARTIFICIAL INTELLIGENCE*, 86(2):269–357, 1996.

[28] Mary Czerwinski, Eric Horvitz, and Susan Wilhite. A diary study of task switching and interruptions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 175–182, New York, NY, USA, 2004. ACM.

[29] Tom Gruber. Intelligence at the interface: Semantic technology and the consumer internet experience, 2009.

[30] Tom Gruber. The perfect storm: Bringing itelligence to the interface, 2009.

[31] Google docs. `https://docs.google.com/`.

[32] Occi: Open cloud computing interface. `http://occi-wg.org/`.

[33] Sri calo web site. `https://pal.sri.com/Plone/framework`.

[34] Wikipedia entry for calo. `http://en.wikipedia.org/wiki/CALO`.

[35] Neil Yorke-Smith, Shahin Saadati, Karen L. Myers, and David N. Morley. Like an intuitive and courteous butler: a proactive personal agent for task management. In Carles Sierra, Cristiano Castelfranchi, Keith S. Decker, and Jaime Simão Sichman, editors, *AAMAS (1)*, pages 337–344. IFAAMAS, 2009.

[36] Siri web site. http://siri.com/.

[37] Nora Spivack. How siri works. interview with tom gruber, cto of siri, January 2010.

[38] Java agent development framework. `http://jade.tilab.com/`.

[39] Christopher A. Miller, Wende L. Dewing, Karen Z. Haigh, David C. Toms, Rand P. Whillock, Stephen V. Geib, Christopher W.and Metz, Rose Mae M. Richardson, Stephen D. Whitlow, John A. Allen, Lawrence A. King, John A. Phelps, Victor A. Riley, and Peggy Wu. *System and method for automated monitoring, recognizing, supporting, and responding to the behavior of an actor*. Number 20040030531. February 2004.

[40] Tom Gruber and Adam Cheyer. Siri: A virtual personal assistant, 2010.

[41] Will Hill, Larry Stead, Mark Rosenstein, and George Furnas. Recommending and evaluating choices in a virtual community of use. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 194–201, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[42] Upendra Shardanand and Pattie Maes. Social information filtering: Algorithms for automating "word of mouth". In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 210–217. ACM Press, 1995.

[43] Michael Pazzani, Jack Muramatsu, and Daniel Billsus. Syskill & webert: Identifying interesting web sites. In *In Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 54–61. AAAI Press, 1996.

[44] Salma Noor and Kirk Martinez. Using social data as context for making recommendations: an ontology based approach. In *Proceedings of the 1st Workshop on Context, Information and Ontologies*, CIAO '09, pages 7:1–7:8, New York, NY, USA, 2009. ACM.

[45] Xin Li, Lei Guo, and Yihong Eric Zhao. Tag-based social interest discovery. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 675–684, New York, NY, USA, 2008. ACM.

[46] H Lieberman, H Liu, P Singh, and B Barry. Beating Common Sense into Interactive Applications. *AI Magazine*, 25(4):63–76, 2004.

[47] H Liu and P Singh. ConceptNet — A Practical Commonsense Reasoning Tool-Kit. *BT Technology Journal*, 22(4):211–226, 2004.

[48] Christian Bizer and Freie Universität Berlin. DBpedia Querying Wikipedia like a Database. *World Wide Web Internet And Web Information Systems*, pages 1–13, 2007.

[49] Anthony Fader, Stephen Soderland, and Oren Etzioni. Identifying Relations for Open Information Extraction. *Network*, pages 1535–1545, 2011.

[50] S. Tarkoma. Distributed event dissemination for ubiquitous agents. In *10th ISPE International Conference on Concurrent Engineering (CE-2003)*, pages 105–110. Citeseer, 2003.

[51] Henry Lieberman, Alexander Faaborg, Jose Espinosa, and Chris Tsai. A Calendar and To-Do List with Common Sense. Technical report, MIT Media Laboratory, 2004.

[52] Jose Espinosa and Henry Lieberman. EventNet: Inferring Temporal Relations Between Commonsense Events. In Alexander Gelbukh, Álvaro De Albornoz, and Hugo Terashima-Marín, editors, *MICAI Fourth Mexican International Conference on Artificial Intelligence*, volume 3789 of *Lecture Notes in Computer Science*, pages 61–69. Springer, 2005.

[53] William Williams. LifeNet : A Propositional Model of Ordinary Human Activity. *Learning*, 2003.

[54] Lin Padgham and Michael Winikoff. Prometheus: A methodology for developing intelligent agents, 2002.

[55] Kinetic rules engine for node.js. `https://github.com/kynetx/kynetx-node`.