

Roberto Yus Peirote

Semantic Management of Location-Based Services in Wireless Environments

Departamento
Informática e Ingeniería de Sistemas

Director/es
Mena Nieto, Eduardo

<http://zaguan.unizar.es/collection/Tesis>

Tesis Doctoral

SEMANTIC MANAGEMENT OF LOCATION-BASED SERVICES IN WIRELESS ENVIRONMENTS

Autor

Roberto Yus Peirote

Director/es

Mena Nieto, Eduardo

UNIVERSIDAD DE ZARAGOZA
Informática e Ingeniería de Sistemas

2016



Semantic Management of Location-Based Services in Wireless Environments

Roberto Yus Peirote

Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza



Universidad Zaragoza

January 2016

Semantic Management of Location-Based Services in Wireless Environments

Roberto Yus Peirote

Supervisor

Eduardo Mena	University of Zaragoza, Spain
--------------	-------------------------------

Dissertation Committee

Heiner Stuckenschmidt	University of Mannheim, Germany
Jesús Bermúdez	University of the Basque Country, Spain
Sergio Ilarri	University of Zaragoza, Spain

Alfredo Goñi	University of the Basque Country, Spain
Thierry Delot	University of Valenciennes, France

International Reviewers

Nalini Venkatasubramanian	University of California, Irvine, USA
Francesco Guerra	University of Modena and Reggio Emilia, Italy

Submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Systems Engineering with “International Doctor” Mention in the Computer Science and Systems Engineering Department, Escuela de Ingeniería y Arquitectura, Universidad de Zaragoza. This work has been supported by the Spanish Government under the FPI fellowship program, ref. BES-2011-043934.

January 26th, 2016.

“Do. Or do not. There is no try.”
Jedi Master Yoda

Acknowledgements

First, I would like to thank my adviser, Eduardo Mena, for his support all these years. Eduardo taught me how to do research and what I know right now I owe it to him. Also, he had the patience needed to deal with a student who sometimes thought he knew it all ☺. Also, I owe being here to Sergio Ilarri who asked me if I would be interested in joining the group and doing research and spared the world another software engineer. That question turned everything upside down and I am very grateful to Sergio for it. I am also very grateful to Tim Finin and Anupam Joshi from University of Maryland, Baltimore County, and Sharad Mehrotra from University of California, Irvine, for accepting me in their research groups. They gave me a nice research experience and I look forward to continuing the collaboration we started in the future.

There are many people I want to thank from Unizar. Starting with the (temporary or not) members of the SID group that collaborated with me in different parts of this thesis and made this work possible (apart from Eduardo and Sergio commented before): Carlos Bobed, Fernando Bobillo, Enrique Solano, Guillermo Esteban, Juan Mengual, and Jorge Bernad. I want to include in this list Arantza Illarramendi and David Antón from the University of the Basque Country who are not directly part of the SID group but almost! All of them have always been accessible and ready to help. I also owe thanks to the rest of the group for the meetings and discussions we shared: Ángel, Jorge, Óscar, María del Carmen, Ramón, and Raquel.

The many nice people that stopped by the lab 1.03a (my second home at Zaragoza!) during these five years made it a great place to work. Naming them all would be difficult and I am sure that I would forget someone, so thank you all (specially to Ricardo!). The members of the *Se Come* lunch group: Andres, Cristian, Jorge, José, Miguel Ángel, Raquel, Sergio, Simona accepted this PhD student in their lunch group and it has been a pleasure to share laughs and talks at lunch with you. We also shared many good talks around an *espresso*

machine, that has to be close to graduation too because we started almost at the same time, and which also deserves some acknowledgment. To José Ramón Asensio who has been my *comrade-in-arms* and friend in the department (you can finally stop asking “when are you finishing your thesis?” ☺).

I want to thank all the nice people I met at UMBC (Varish, Prajit, Sunil, Vlad, Abhay, Jennifer, Lisa, and Claire) and at UCI (Drubh, Joyce, and Xikui). I went there thinking that with luck I will find a collaborator or two and I found many which, most importantly, I can call friends. The common factor in these visits (we met at UMBC and UCI!) deserves a special mention, my dear *rafiki* Primal Pappachan. Apart from being a great collaborator (our research discussions are frequently vehement and very fruitful!) he has been a great support in the tough final stages of the PhD and proved to be the best friend. I hope I can return you the favor in your PhD which I foresee will be full of successes.

My friends from the undergrad times, Ángel, Alberto, Carlos Pérez, Carlos Vara, Diego, Gorka, Guillermo, and Victor, shared with me many laughs, biers, and on-line matches that helped to decrease the stress levels. Finally, I want to thank my parents, Carmen and Antonio, and my sister, María. I am sure that they did not understand why someone would spend most of his time in the lab regardless of festivities or daylight hours and working for peanuts (who can blame them? nobody outside of this world would understand it). However, they have always been there for me and their support made everything easier.

Zaragoza, January 2016
Roberto Yus Peirote

Contents

1	Introduction	1
1.1	Context of the Thesis	2
1.1.1	Mobile Computing	2
1.1.2	Knowledge Management	3
1.2	Motivation	4
1.3	Overview of the System	5
1.3.1	Knowledge Management	7
1.3.2	Request Management	8
1.4	Structure of the Thesis	11
2	Technological Context	13
2.1	Knowledge Management	13
2.1.1	Ontologies	13
2.1.2	Description Logics	16
2.1.3	Representation Languages	23
2.1.4	Query Languages	25
2.1.5	Tools for Developing Semantic Applications	27
2.2	Mobile Computing	28
2.2.1	Location-Based Services	28
2.2.2	Context-Aware Computing	29
2.2.3	Agent Technology	30
2.2.4	Software Agents	30
2.3	Knowledge Management on Mobile Devices	32
3	Overview of the System	37
3.1	Motivating Scenarios	37
3.1.1	Looking for Transportation	37
3.1.2	Helping Firefighting	39
3.1.3	Live Broadcasting of Sport Events	39

3.1.4	Emergency Management	41
3.1.5	Common Challenges	43
3.2	Architecture	45
3.2.1	Interaction with the User	46
3.2.2	Knowledge Management	47
3.2.3	Request Management	47
3.2.4	Overview of a SHERLOCK-Enabled Device	48
3.3	Summary of the Chapter	48
4	Knowledge Management	51
4.1	Modeling SHERLOCK Devices and Services	51
4.1.1	Contextual knowledge	53
4.1.2	Service and Scenario Knowledge	58
4.2	Providing Access: Knowledge Endpoint Agent	62
4.2.1	SHERLOCK's Query Language	63
4.2.2	Context-Aware Privacy Policies	70
4.3	Summary of the Chapter	72
5	Knowledge Update	75
5.1	Updating Context: Context Updater Agent	75
5.1.1	Context Extraction	76
5.2	Updating Ontology: Ontology Updater Agent	81
5.2.1	Knowledge Maintenance	82
5.2.2	Knowledge Exchange	83
5.2.3	Knowledge Integration	84
5.3	Related Work	96
5.3.1	Works on Discovery of Subsumption Relationships	96
5.3.2	Works on Context Enrichment	97
5.4	Summary of the Chapter	98
6	Request Management	99
6.1	Request Generation	99
6.1.1	Service Selection	101
6.1.2	Parameter Provision	104
6.1.3	Service Handling	105
6.2	Request Processing	108
6.2.1	Processing of SHERLOCK Queries	109
6.2.2	Invocation of External Services	112
6.2.3	Execution of Service Plans	113
6.3	Related Work	115

6.3.1	Location-Based Services	115
6.3.2	Service-Oriented Architectures	116
6.4	Summary of the Chapter	116
7	Processing of SHERLOCK Queries	117
7.1	User Request Processor Agent: Coordinating the Network of Agents	119
7.1.1	Creation of a Network of Helping Agents	120
7.1.2	Maintenance of the Network of Helping Agents	122
7.1.3	Results Correlation	123
7.1.4	Movement Evaluation	123
7.2	Tracker Agent: Monitoring a Location of Interest	126
7.2.1	Movement Evaluation	127
7.2.2	Creation of the Network of Updater Agents	127
7.2.3	Updater Agents Network Maintenance	133
7.3	Updater Agent: Obtaining Results	133
7.3.1	Query Execution and Results Correlation	134
7.3.2	Query Extension	134
7.3.3	Movement Evaluation	137
7.4	Related Work	137
7.5	Summary of the Chapter	138
8	Multimedia Information Management	141
8.1	Motivation	142
8.2	Processing of Camera Views	144
8.2.1	Kind of View Obtained	145
8.2.2	Percentage Viewed of an Object	146
8.2.3	Percentage of a Part of an Object	149
8.2.4	Estimating Future Views	150
8.3	Measuring the Similarity of Camera Views	155
8.3.1	Percentage Difference	156
8.3.2	Viewpoint Difference	156
8.3.3	Location Difference	157
8.3.4	Summing Up: Differences in Each Object	159
8.4	Preserving User Privacy in Photos	159
8.4.1	Creating Privacy-Aware Pictures	160
8.5	Related Work	162
8.5.1	Works on Selection of Camera Views	163
8.5.2	Works on Privacy on Pictures	165

8.6	Summary of the Chapter	166
9	Dealing with the Motivating Scenarios	169
9.1	First Scenario: SHERLOCK for Looking for Transportation . .	169
9.1.1	Knowledge for the Scenario	170
9.1.2	Steps Followed	170
9.2	Second Scenario: SHERLOCK for Helping Firefighting	174
9.2.1	Knowledge for the Scenario	175
9.2.2	Steps Followed	175
9.3	Third Scenario: SHERLOCK for Handling Sport Events Broad- casting	180
9.3.1	Knowledge for the Scenario	181
9.3.2	Steps Followed	181
9.4	Fourth Scenario: SHERLOCK for Emergency Management . .	186
9.4.1	Knowledge for the Scenario	186
9.4.2	Steps Followed	187
9.5	Other Extra Scenarios	190
9.5.1	SHERLOCK for Obtaining Tourist Information	191
9.5.2	SHERLOCK for Meeting Fellow Researchers	193
9.5.3	SHERLOCK for Helping Health-Care Workers	195
9.6	Summary of the Chapter	200
10	Conclusions	203
10.1	SHERLOCK: Main Contributions	203
10.2	Other Contributions	204
10.3	Research Results	206
10.4	Future Work	210
	Relevant Publications Related to the Thesis	213
A	Semantic Technologies on Mobile Devices	217
A.1	Porting Semantic Technologies to Android	218
A.2	Evaluating the Use of Semantic Web Technologies on Mobile Devices	222
A.2.1	Experimental Setup	222
A.2.2	Comparing the Reasoners for the OWL 2 DL Profile . .	228
A.2.3	Comparing the Reasoners for the OWL 2 EL Profile . .	233
A.2.4	Other Experiments	237
A.2.5	Analyzing the Impact of Memory	237
A.2.6	Analyzing the Impact of the Virtual Machine	239

A.2.7	Discussion	241
A.3	Related Work	245
A.3.1	Reusing and Evaluating DL Reasoners on Mobile Devices	245
A.3.2	Evaluating DL Reasoners on Desktop Computers	246
B	Prototypes for the Semantic Management of LBS	255
B.1	SHERLOCK Prototype	255
B.1.1	PC Prototype	256
B.1.2	Android Prototype	259
B.1.3	Testing the Android Prototype	260
B.2	DUCK: Exchange and Integration of Knowledge in Wireless Environments	269
B.2.1	Architecture of DUCK	269
B.2.2	Prototype of DUCK	270
B.2.3	Evaluating the Extraction of Subsumption Relations . .	273
B.3	Triveni: Shared, Semantic Context Models for Mobile Devices .	277
B.3.1	Architecture of Triveni	277
B.3.2	Prototype of Triveni	279
B.3.3	Evaluating the Extraction of a Shared Context	279
B.4	MultiCAMBA: Multi-CAMera Broadcasting Assistant	286
B.4.1	Architecture of MultiCAMBA	286
B.4.2	Prototype of MultiCAMBA	288
B.4.3	Evaluating the Processing of Multimedia Information .	293
	Bibliography	315

List of Figures

1.1	Interaction of different SHERLOCK nodes.	6
2.1	Example of ontology.	14
2.2	Two sample ontologies describing knowledge about universities.	15
2.3	Semantic relationships that exist among the concepts of two ontologies.	17
2.4	Definition of OWL 2 class expression.	25
2.5	Number of semantic mobile apps per year.	34
2.6	Distribution of semantic mobile apps per platform.	35
2.7	Wordcloud with the semantic technologies used by the different apps.	36
3.1	Different transportation options in our first motivating scenario.	38
3.2	A team of firefighters suppressing a fire in our second motivating scenario.	40
3.3	The rowing boat race in our third motivating scenario.	41
3.4	The traffic accident in our fourth motivating scenario.	42
3.5	High-level architecture of a SHERLOCK node.	46
3.6	Agents to interact with a user and manage knowledge in SHERLOCK devices.	49
4.1	Different knowledge managed by a SHERLOCK device.	52
4.2	An excerpt of the context ontology.	55
4.3	An object-of-interest modeled in our system.	56
4.4	Modeling a camera: pan (horizontal plane) (a), and tilt (vertical plane) (b).	57
4.5	Basic ontology for service modeling in SHERLOCK's services.	58
4.6	Example of the definition of a SHERLOCK querying service to obtain transports.	59

4.7	Two examples of the definition of external services: 1) to take pictures and 2) to obtain transports from a web service.	60
4.8	Example of the definition of a complex service with a workflow to find cameras in an area and request them to take a picture.	62
4.9	Simplified Grammar of SHERLOCK's query language.	73
4.10	Sample views, recreated using Google Earth, of three cameras (<i>CAM1</i> (a), <i>CAM2</i> (b), <i>CAM3</i> (c), and <i>CAM3</i> after 4 seconds panning to the right (d)) covering a rowing race.	74
5.1	Context Updater agent (CU) tasks.	76
5.2	Ontology Updater agent (OU) tasks.	82
5.3	Steps involved in the management of knowledge.	83
5.4	Main steps in our approach to extract subsumption relationships from two source ontologies.	86
5.5	Subsumption degree (Y-axis) between two concepts, C_s and C_S , depending on the number of roles of C_S (denoted by the different curves) and their shared roles (X-axis).	89
5.6	Some of the subsumption degrees obtained for our running example.	94
6.1	User Request Manager agent (URM) tasks.	100
6.2	Steps to obtain relevant services for a user.	102
6.3	User Request Processor tasks.	109
6.4	Flow diagram of the execution of a SHERLOCK query.	110
6.5	Sequence diagram of the execution of a service plan.	114
7.1	Mobile agents to process a user query and obtain results.	118
7.2	User Request Processor tasks.	119
7.3	Query to obtain the features of devices around a coordinate.	126
7.4	Example of the monitoring of a location of interest.	128
7.5	Creation of Updater agents by a Tracker agent to cover a location of interest.	131
7.6	Semantic relationships exploited when extending a service.	137
8.1	Real camera footage (a) and interesting and other objects in the scene (b).	143
8.2	A real camera shot of a rowing boat (a) and the recreation of the view in our system with the top (red) and rear (blue) of the boat highlighted (b).	147

8.3	Computing the percentage of a target object in a shot: scene in <i>Google Earth</i> (a), selecting the target (b), painting the FOV (c), and covering the target completely (d).	149
8.4	Initial state of a target object in (x_0, y_0) and a camera in (x_c, y_c)	151
8.5	t_{f+} (t_{f-}) time to focus the target object rotating the camera to the left (right) side.	153
8.6	Estimation of trajectories and time needed to focus a rowing boat, “Kaiku”, from another boat, “Orio”.	154
8.7	Horizontal angles involved in the computation of the similarity between two images (a) and (b).	158
8.8	Images involved in the process of obtaining a privacy-aware picture: (a) picture of a user; (b) face identifier of the user generated by the system; (c) picture taken by a SHERLOCK user; and (d) privacy-aware picture generated by SHERLOCK.	161
8.9	Example of a context-aware privacy policy to create privacy-aware pictures.	162
8.10	Handshake diagram for the creation of privacy-aware pictures.	162
9.1	Excerpt of the ontology for the “Looking for Transportation” scenario.	171
9.2	Screenshots of the SHERLOCK prototype executing the first scenario.	172
9.3	Excerpt of the ontology for the “Helping Firefighting” scenario.	176
9.4	GUI of the PC prototype for the fire monitoring scenario.	177
9.5	Deployment of the agent network in the fire monitoring scenario.	178
9.6	Results shown to the user in the fire monitoring scenario with the location of the mobile agents deployed.	179
9.7	Service to find cameras that could obtain a certain shot.	182
9.8	Excerpt of the ontology for the “Handling Sport Events Broadcasting” scenario.	183
9.9	Definition of the shot that the technical director wants to broadcast.	184
9.10	Camera view recreated in a 3D engine by the system.	185
9.11	Some of the camera views returned to the TD.	185
9.12	Camera feed broadcasted.	186
9.13	Excerpt of the ontology for the “Emergency Management” scenario.	188
9.14	Emergency management scenario.	189
9.15	Excerpt of the ontology for the “Obtaining Tourist Information” scenario.	192

9.16	SHERLOCK obtaining: (a) interesting touristic points and (b) information about a researcher.	193
9.17	Excerpt of the ontology for the “Meeting Fellow Researchers” scenario.	194
9.18	Excerpt of SHERLOCK’s ontology for the CHW scenario: diagnosis module.	196
9.19	Excerpt of SHERLOCK’s ontology for the CHW scenario: patient context module.	197
9.20	Excerpt of SHERLOCK’s ontology for the CHW scenario: stats module.	197
A.1	Results (finished tasks/average time) for the complete OWL 2 DL ontology set.	229
A.2	Average computing time for each ontology group in the minimum set of OWL 2 DL ontologies processed by all the devices and reasoners.	232
A.3	Results (finished tasks/average time) for the complete OWL 2 EL ontology set.	234
A.4	Average computing time for each ontology group in the minimum set of OWL 2 EL ontologies processed by all the devices and reasoners.	236
A.5	Comparison of the number of finished classifications of OWL 2 DL ontologies.	238
A.6	Comparison of the number of finished classifications of OWL 2 EL ontologies.	239
A.7	Comparison of the Pellet reasoner on Android with limited and “unlimited” memory.	240
A.8	Comparison of the Pellet reasoner on the Dalvik and ART virtual machines.	241
B.1	Different modules of the SHERLOCK prototype.	256
B.2	Screenshots of the SHERLOCK prototype for PC.	257
B.3	Screenshots of the simulator for the PC prototype.	258
B.4	SHERLOCK app: Home interface.	261
B.5	SHERLOCK app: Settings.	262
B.6	SHERLOCK app: Map interface.	263
B.7	SHERLOCK app: Selecting a service and its parameters.	264
B.8	SHERLOCK app: Displaying results.	265
B.9	SHERLOCK app: Interacting with objects.	266
B.10	SHERLOCK app: Interacting with other users.	267

B.11 SHERLOCK app: Handling multiple requests.	268
B.12 Sequence diagram of the exchange and integration process in the DUCK prototype.	270
B.13 DUCK app: Screenshots of the prototype.	272
B.14 High-level architecture of Triveni.	278
B.15 Triveni app: Screenshots of the prototype.	280
B.16 Motivating use case for Triveni: Users being part of a study group.	281
B.17 Precision and recall of the context obtained for a user in different scenarios before and after using the three approaches of Triveni (conservative, optimistic, and semi-optimistic).	284
B.18 Computation time and memory usage on a Triveni node with increasing number of devices.	287
B.19 Main steps followed by the MultiCAMBA prototype.	288
B.20 Technical architecture diagram of the MultiCAMBA prototype.	289
B.21 MultiCAMBA app: Graphical User Interface (GUI) for the Technical Director.	290
B.22 MultiCAMBA app: Example of low-level input form.	291
B.23 MultiCAMBA app: Snapshot of the Query-by-Example 3D interface.	292
B.24 Quality of the result set (i.e., cameras fulfilling the user require- ments): number of cameras in the answer set (blue line) and number of wrongly chosen cameras (red dashed line).	296
B.25 Error in the estimated time needed for the cameras to view the target object: the error is localized at two specific time intervals (the start and the end of the race).	297
B.26 Error in the estimated percentage of the target that a camera will view in the first eight seconds.	299
B.27 Testing the system against real camera footage (the information generated by our system is on the top): in two consecutive seconds.	300
B.28 Query images used in the tests.	302
B.29 Set of 45 images used in the tests.	303
B.30 Number of images selected by the users as similar to each query image.	304
B.31 Comparing the ranking of images obtained by our system (s) and the users' rankings (u_i) for each query (normalized Kendall tau distance $K(u_i, s)$).	306
B.32 Tester Disagreement and System Disagreement.	308
B.33 Ranking obtained by the system for <i>Query 4</i> : before (a) and after (b) modifying the weights of the similarity formula.	309

B.34	Ranking of images obtained for a user query: before (a) and after (b) modifying the weights of the similarity formula. . . .	310
------	---	-----

List of Tables

2.1	Constructors and their meanings for \mathcal{ALC} DL.	18
2.2	Expressivity and complexity of reasoning in some important DLs.	19
2.3	Semantic Web reasoners and some of their characteristics. <i>FRG</i> : Fragment; <i>APX</i> : Approximated.	22
2.4	Main RDF-S constructs.	24
2.5	SPARQL-DL supported query patterns.	27
2.6	Semantic mobile applications presented in the literature.	33
4.1	Information stored about the context of the device.	53
4.2	Information stored about the context of the user.	54
4.3	Semantics of the introduced DL operators: Domain and Range.	66
5.1	Contextual information shared about location.	78
7.1	Information of the different communication technologies consid- ered by a Tracker to cover an interesting area.	132
9.1	Table used by a Tracker agent to dynamically maintain its network of Updaters.	179
A.1	Android support for some semantic APIs and DL reasoners.	249
A.2	Comparison of classification time (in seconds) for two Android devices. <i>OOM</i> : Out Of Memory; <i>UDT</i> : Unsupported Data Type.	250
A.3	Comparison of classification time (in seconds) for PC and An- droid. <i>UDT</i> : Unsupported Data Type.	250
A.4	Errors for uncompleted tasks in the DL ontology set.	251
A.5	Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 DL ontology set.	251
A.6	Errors for uncompleted tasks in the EL ontology set.	252

A.7	Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 EL ontology set.	253
B.1	Precision and recall of our prototype for different ontologies. . .	274
B.2	Results for the tests performed: context creation with consistent information (Test1), context enrichment with consistent information (Test2), context correction with inconsistent information (Test3).	285

Chapter 1

Introduction

The context of the work presented in this thesis is *mobile computing* and *knowledge management*. We focus on the usage of semantic technologies for representation, sharing, and integration of knowledge about services (specially Location-Based Services) as well as agents to discover devices which can provide interesting information for users. Therefore, our main contribution is *a general and flexible agent-based architecture based on the use of semantic technologies to provide services to mobile users*.

Location-Based Services (LBS) provide added value by customizing the information offered to mobile users based on their locations. Current LBS are usually designed for specific scenarios and goals with predefined schemas for the modeling of the elements involved in their scenarios. Apart from specific LBS, some approaches of architectures to provide users with LBS have been presented before. However, these approaches assume either a centralized architecture or its distribution using a fixed infrastructure. In addition, these approaches usually act as mere repositories of services and do not deal with their execution to obtain the information that the user needs.

Our proposal is the system *SHERLOCK* (*System for Heterogeneous mobile Requests by Leveraging Ontological and Contextual Knowledge*) that offers a general and flexible architecture to provide users with LBS which might be interesting regarding their context. SHERLOCK is based on semantic and agent technologies: 1) ontologies are used to model the information about users, devices, services, and the world around a device whereas a semantic reasoner is used to manage these ontologies and infer non-explicit knowledge; and 2) an agent-oriented architecture enables SHERLOCK devices to autonomously exchange knowledge keeping their local ontologies updated, and to process user information requests, finding what the user needs wherever it is. The

use of these two technologies helps SHERLOCK to be flexible in terms of both the services it offers to the user (which are *learned* from the interaction between devices), and the mechanisms to find the information that the user wants (which adapt to the underlying communication infrastructure).

In this chapter, we first describe the context of our work and our motivation. Then, we provide an overview of our work describing our approach for the management of knowledge and user information requests. Finally, we present the structure of the thesis.

1.1 Context of the Thesis

The work presented in this thesis belongs in the fields of mobile computing and knowledge management. More specifically, it focuses on the semantic management of Location-Based Services in mobile scenarios.

1.1.1 Mobile Computing

The main implication of mobile computing is the possibility of computers being transported around by users. Indeed, in the last few years, we have witnessed a massive spread of mobile computing which has been shaping our daily lives. This has been undoubtedly helped by the pervasive connectivity that the current wireless networks provide us with and the affordable prices of current mobile devices (such as smartphones and tablets). Mobile devices have been fast replacing other stationary devices as *de-facto* medium for on-line browsing, social networking, and other applications. This has attracted a huge community of developers that are continually releasing new mobile applications (or *apps*) which usually utilize the location of the user and therefore, can be viewed as LBS. In fact, the most popular mobile application (*app*) stores crossed the one million apps mark in 2013. For example, the Google Play market¹, the app store of Google, contains more than 1,800,000 available applications in December 2015².

Mobile devices have some limitations compared to traditional forms of computing regarding, for example, processing power, battery, and memory. Therefore, most of current systems and apps rely on the “cloud” for performing their tasks [FLR13]. However, in this vision of mobile computing, mobile devices are regarded as mere *terminals*, devices used to enter data into and display data from. Also, this model presents problems, apart from the obvious

¹<http://play.google.com>

²<http://goo.gl/7oeQD>, all the URLs in this thesis have been last accessed in 2016-02-07.

privacy issues that arise with information being transmitted to the cloud, such as the reliability on the connectivity to the cloud. Although, these days Internet connectivity is mostly assured in major cities and other locations, there are still areas where the connectivity is not assured and situations where the user might not be interested in using it (for example, due to monetary costs or energy consumption). We believe that the increasing capabilities of current mobile devices, from the point of view of computing power and communication with other devices around, can be leveraged in these scenarios.

1.1.2 Knowledge Management

Systems and applications need to manage some knowledge about their context to accomplish their goals. A traditional way of representing this information is the use of ontologies, defined by Tom Gruber as “an explicit specification of a conceptualization” [Gru95]. Ontologies allow modeling and capturing the semantics of different knowledge domains, providing a means to share definitions, and reach an implicit agreement on the meaning of the published information. In addition, ontology modeling languages based on Description Logics (DL) [BCMNP03], such as OWL, make it possible the use of semantic DL reasoners to infer non-explicit knowledge from the explicit facts and the model defined. These features enable the development of smart applications, such as smart LBS [IIMS11].

By using semantic technologies, applications on mobile devices can benefit from the advantages of the Semantic Web. For example, apps can use information from the Linked Data cloud [BHBL09], as well as publish and subscribe to various data sources without worrying about app or device specific schemas, and even reason over information to derive non-explicit facts.

The use of semantic technologies on mobile devices has been subject of interest from the early stages of the Semantic Web [WRSOS05]. However, currently the use of semantic technologies on mobile applications is not widespread in comparison with the overwhelming amount of existing apps. In [YP15] we presented a systematic review of semantic mobile applications which covers the breadth of semantic mobile apps and the depth of semantic data management. To this end, we analyzed more than 400 papers and found that at least 36 semantic mobile apps have been presented in the literature over the last 10 years.

Our results show that most of the “semantic mobile apps” presented act as clients which rely on external servers for the handling of semantic data. This means that although they consume data which comes from Linked Data

points and ontologies, in many cases this data is preprocessed on a server which returns the data in a semistructured format (e.g., JSON) or just as strings. There are only a few recent apps exploiting the capabilities of current mobile devices to handle semantic data locally (from which majority were developed as part of this thesis). Therefore, as with the case of current LBS, mobile devices do not usually manage knowledge locally. Although, obviously more limited than their fixed counterparts, the capabilities of current mobile devices make them suitable for the semantic management of data locally on the device. However, research should focus on dealing with the problems related to this new scenario (e.g., devices with limited capabilities which generate large amounts of highly-dynamic data) to popularize the use of semantic technologies for the local management of knowledge in apps on these existing and future devices.

1.2 Motivation

In the last years the interest in mobile computing has grown due to the ever-increasing use of mobile devices and their pervasiveness. The low cost of these devices, along with the high number of sensors and communication mechanisms they are equipped with, make it possible to develop useful information systems. Using special kinds of sensors, location mechanisms enable the development of *Location-Based Services (LBS)* [SV04]. These services provide added value by considering the locations of the mobile users to offer customized information. For example, LBS for taxi searching [SCC10], helping firefighting [JCHWTL04], detecting nearby friends [AEMPW07], or multimedia retrieval in sport events [IMIYLM12] have been presented, among many others.

However, current LBS are usually designed for specific scenarios and goals. Moreover, the knowledge they manage is not explicitly represented but embedded in their code; that is the reason why they only work for one specific goal. Moreover, developing services ad hoc for specific purposes leads to the fact that there exist thousands of them (even with the same purpose), and therefore it is difficult to choose the most suitable one.

For example, imagine an attendee of a conference that has just arrived at the airport of a foreign city, and needs a way to reach the conference hotel; this information could be obtained by visiting a tourist office, searching a local transportation website, or even downloading a mobile app. After checking in, she could be interested in finding other nearby conference attendees to talk to them or even to go sightseeing (again, he should browse the Web to find information about interesting places to visit). Thus, it is the user herself who is in charge of knowing/finding the interesting and updated information sources

and gathering and correlating all this information; even worse, she will have to know/find all these updated information about each city she would visit.

Therefore, how to provide the LBS that the user needs at the moment regardless of their specific goals is still an open problem. Some ad hoc solutions have been proposed to provide users with LBS (e.g., [GL06; IMI06]) but there is a lack of a general architecture able to provide different LBS and obtain the information that the user needs wherever it is. Existing approaches to the problem present several key disadvantages that motivate our work in this area. To build such a general system by simply merging preexisting LBS is not straightforward: it is a challenge to provide a common framework that allows 1) managing knowledge obtained from data sent by heterogeneous devices (textual data, multimedia data, sensor data, etc.); and 2) considering situations where the system must adapt itself to contexts where the knowledge changes dynamically and in which devices can use different underlying wireless technologies (fixed, wireless, ad hoc, etc.).

In this thesis, we attempt to overcome the aforementioned points by designing a general and flexible architecture to provide interesting LBS to mobile users.

1.3 Overview of the System

In this thesis, we present *SHERLOCK*, a general and flexible system to provide LBS based on the use of semantic techniques for knowledge management and mobile agents for finding the information the user needs wherever it is. The system is based on the collaboration of devices to satisfy information needs of their owners. In *SHERLOCK*'s distributed architecture every device acts as an independent node which communicates with others to exchange information that might be interesting for their users (see Figure 1.1). *SHERLOCK*-enabled devices use their communication mechanisms (e.g., WiFi and 3G) to create Peer-to-Peer (P2P) networks. The benefits of the communication between devices are twofold, it allows devices to: 1) Exchange information about services and their surroundings and thus, every device learns from the different interactions; and 2) Answer requests posed by others by using the information they store locally.

As its namesake, the well-known Arthur Conan Doyle's character, *SHERLOCK* uses deductive reasoning to infer information to answer user requests. In our opinion, the use of semantic techniques can enable the development of intelligent LBS [IIMS11]. Thus, each node uses ontology reasoning and alignment methods, locally on the device, to represent and manage, in a distributed

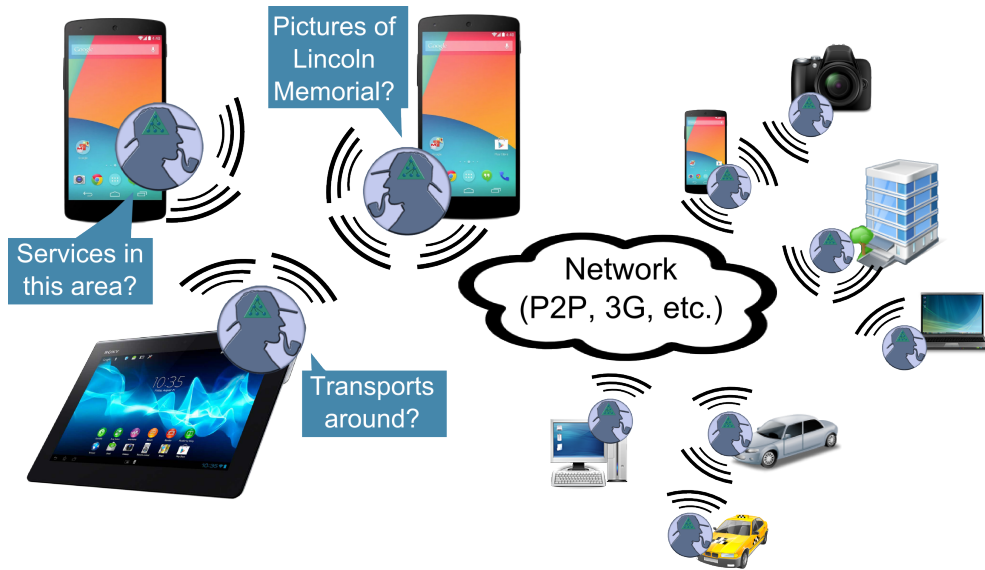


Figure 1.1: Interaction of different SHERLOCK nodes.

way, the knowledge about services and the world around the user. This way, the system guides the user in the process of selecting the LBS that best fits her needs; the participating devices can cooperate and exchange data and knowledge among them to relieve the user from knowing and managing such knowledge directly. This knowledge can be defined by providers of services or even extracted from ontologies in the Web using Semantic search engines (elementary, SHERLOCK can use the services of WATSON [dM11] to find ontologies in the Web). Furthermore, thanks to the use of mobile agents [LO99], it is possible to monitor devices (e.g., inside a certain geographic area), which could provide the information the user needs, wherever they are. Also, the agents help to distribute the load of the system (both the CPU power and the communication costs) wherever it is needed in the wireless environment. This way, the required processing tasks can be carried on the most appropriate device in the scenario. In summary, the main benefits offered by our system, from the user's point of view, are:

1. It offers to the user all the available LBS which might be interesting for her at each moment given her current context. After choosing one of them, it helps the user to express her information needs by pro-actively querying the local knowledge at the user device. Therefore, it relieves

the user from managing specific knowledge about LBS.

2. It reconciles the different views of the world and the vocabulary used to describe objects and requests. This is achieved by supporting a decentralized and dynamic discovery of new kinds of services, providers, and information about the surroundings of the user. This way, the system manages up-to-date knowledge about the LBS provided to the user.
3. It manages heterogeneous (fixed or mobile) devices that can be part of the system, each of them having different capabilities. Moreover, it adapts itself in run-time to different underlying networks, such as fixed infrastructures (e.g., such as 4/3G and wired networks) and Mobile Ad-hoc Networks (MANETs) [CCL03].
4. It finds the information that the user needs wherever it is. A mobile agent network is deployed to obtain the information from other SHERLOCK-enabled devices. Also, different sources are considered, such as the local knowledge in the device and third-party providers.
5. It continuously carries the processing to the most appropriate nodes in order to balance the processing load and communication tasks, by using mobile agents. This is important to alleviate the limited CPU power, storage, and communication capabilities of mobile devices.

The system presented is flexible enough to deal with different types of services and scenarios. In addition, new services can be added to the system by providing SHERLOCK with an ontology which models them without architectural changes. In the following sections we describe in more detail how the system manages the knowledge about services and scenarios as well as information requests by users.

1.3.1 Knowledge Management

SHERLOCK-enabled devices manage knowledge modeled as OWL ontologies with the help of a Description Logics (DL) semantic reasoner running on the device. The different categories of knowledge managed by the system include: user context, device context, services, and scenarios. The knowledge management task is performed by different agents whose main goal is to keep the local knowledge on the device updated, as that will mean offering more interesting information to their users, and provide access to this information. In the following we briefly summarize the different agents in charge of this task and their goals:

- The *Knowledge Endpoint* agent provides access to the local knowledge on the device to other (local or external) agents. The access is done through queries in SHERLOCK's query language which is based on SPARQL (with extensions to handle geospatial data and DL ontologies).
- The *Ontology Updater* agent updates the local ontology on the device by exchanging it with other devices. To integrate the knowledge received into the local ontology, it makes use of a technique based on the extraction of subsumption relationships between concepts defined in the ontologies.
- The *Context Updater* agent keeps the user context information updated and collects information about the context of other devices. Through the integration of the information received, it creates a shared context model used to improve the context of the user managed.

A special type of knowledge managed by SHERLOCK is related to multimedia information. Users might be interested in LBS to obtain, for example, pictures of a certain location or object (e.g., pictures of the Lincoln Monument in Washington D.C.). So, SHERLOCK manages information provided by cameras attached to SHERLOCK-enabled devices by extracting high-level features of their views regarding objects in their Field-of-View and specific details such as the viewpoint of these objects or the amount of them being covered. To this end, our system relies on a 3D model of the scene generated based on information about the interesting objects and cameras in the scenario (location, direction, and approximate 3D extent). Also, the system is able to compute an objective value to compare the picture that the user wants to obtain with the shots provided by the different available cameras. Finally, when handling pictures taken by SHERLOCK-enabled devices, the system tries to preserve the privacy preferences of users in the pictures in which, for example, they would not like to appear depending on their context.

1.3.2 Request Management

The knowledge that SHERLOCK-enabled devices *learn* from their interactions with others is used to offer interesting services to their users. First, SHERLOCK captures the user information needs by guiding her to select an appropriate LBS and then it generates a formal user request with the service selected and the user preferences. Then, it processes the generated request against different sources to obtain the actual information the user is interested in.

Request Generation

A first step in order to fulfill the user information needs is to capture those needs and formalize them into a formal request in order to avoid ambiguities. To provide SHERLOCK with enough expressivity and flexibility, we have designed a SPARQL-like query language that is used by the system making it possible to express semantic location-based queries (e.g., “Retrieve taxis around me”) and non-location based queries (e.g., “What is the age of Barack Obama?”) as part of LBS or other services, respectively. This language requires knowledge about SPARQL and SHERLOCK’s ontology, so it might be too complicated to be used by non-advanced users. Therefore, our approach helps users to define their interests, guiding them, and capturing their requests. To do so, the system relies on the *User Request Manager* agent (URM) to guide users when defining their information needs. The URM performs three main tasks summarized in the following:

1. The URM is in charge of obtaining the services (based on a location or not) that are relevant for a user in a particular situation. The result of this task is a list of services the user can see and select.
2. Given the service selected by the user, the URM retrieves the information needed to invoke it (its formal parameters) from the ontology, and, if needed, handles the interaction with the user required to obtain the actual values for the parameters.
3. Given the service, and a set of parameters with the selected values, the URM generates the appropriate service request/invoke.

This way, the user does not have to be aware neither of the details of the query language nor the schema and available services.

Request Processing

The next step is to process the user request to obtain the information that the user needs. The user request might imply the processing of a location or non-location based query, the call to an external service provided by a third-party, or the execution of different actions defined as a plan. The User Request Processor agent (URP) deals with these three different types of user requests. The high-level protocol performed by the URP agent to process a request in the form of a SHERLOCK query can be summarized in the following steps:

1. *Execute the query against the local ontology on the user device* that could contain the information requested from previous interactions.
2. *Evaluate the need of querying external sources*: The results obtained in the previous step are analyzed to evaluate if they are good enough for the user in terms of their timestamp and the number of results. External sources include third-party repositories and other SHERLOCK devices.
3. If the query has to be posed to other SHERLOCK devices, the URP will try to query devices in the geographic area relevant for the query (if any), to maximize the chances of obtaining information interesting for the user, as follows:
 - (a) *Split the query for each non-overlapping geographic constraint in it*: Tracker agents will be created to monitor the geographic area associated with each geographic constraint. Each Tracker agent will autonomously move toward the centroid of the area. For this purpose, the Tracker will discover devices around the area (if possible) or around the device it is currently residing in and will move to them. Once a Tracker discovers devices that might be able to partially (or even totally) *cover* the area (i.e., their communication mechanisms enable them to communicate with devices inside such an area), it creates *Updater* agents on them.
 - (b) *Send agents to devices in the relevant area*: Each tracker agent creates Updater agents that will move to the best device to cover the relevant area considering their capabilities. These agents also continuously evaluate if the device they are residing in is the best under the given circumstances and their goals. They keep themselves in the device that maximizes the communication with others and pose the user query to these SHERLOCK devices around.
 - (c) *Each device executes the query against its local ontology* and returns as answer the information fulfilling the query constraints (including geospatial and DL constraints, the latter evaluated by the DL reasoner on each device). If the device does not contain the requested information, an Updater makes use of different mechanisms to maximize the chances of obtaining results. For example, it can ask the SHERLOCK on the other device to execute the query (and therefore, create its own network of agents if needed) and even extend the query by deducing devices that would be able to produce the information requested (e.g., it can deduce that cameras could

obtain the pictures that the user requested and execute a request to find such cameras).

- (d) *Correlate the results obtained* by each Tracker agent from the different Updater agents to detect redundant information. This step can minimize the information that will be sent back to the URP agent through the network of agents.

Finally, the URP correlates the results obtained from each tracker and presents them to the user. In the case of continuous queries (i.e., queries which have to be reevaluated continuously), the URP maintains the network of agents adapting it to the current situation (e.g., the current network condition).

1.4 Structure of the Thesis

This thesis has been structured into ten chapters including this one where we have summarized our motivation and contributions.

In Chapter 2, we review the technological context of this work. In particular, we describe aspects related to mobile computing and knowledge management.

In Chapter 3 we first present four motivating scenarios where a system like the one presented in this thesis can be helpful. Each scenario include unique challenges that have been taken into account in the proposed architecture and which make the system general enough to help in many other scenarios. Also, we describe the high-level architecture of the system, highlighting the different parts and modules that will be explained in detail in the rest of the thesis.

In Chapter 4 we describe the management of knowledge done by the system. We include the management of information related to the context of the user and the services and scenarios which the system considers. We explain how the system models this knowledge and the mechanism developed to access it through a language we designed based on SPARQL.

In Chapter 5 we explain how the system keeps its local knowledge updated. We describe the process of sharing information with other devices in order to help each SHERLOCK-enabled device to learn from the interaction with others.

In Chapter 6 we present the process of managing user information requests. First, we explain how the system helps users to define their information needs in order to translate their needs into a formal request using SHERLOCK's query language. Then, we explain how these requests are processed.

In Chapter 7 we present the processing of SHERLOCK queries by using different sources. We focus on the deployment of a network of mobile agents

which are in charge of finding SHERLOCK devices which might contain the requested information. The agents execute the queries against such SHERLOCK devices.

In Chapter 8 we describe the management of multimedia information in SHERLOCK. In particular we focus on the efficient management of camera views using information about the context of cameras and objects in the scenario. We explain the algorithms developed to extract high-level features of the views of such cameras and compare the result to the camera view the user needs.

In Chapter 9 we explain how the developed system deals with the motivating scenarios used through the thesis. We detail the steps involved and highlight how the system would deal with other similar scenarios.

Finally, in Chapter 10, we present the conclusions and main contributions of this thesis, as well as some future work and lines that have been opened.

We also include two appendices where we present: 1) Our thorough analysis of the performance of Semantic Web technologies, such as semantic reasoners, on mobile devices, which is one of the foundations of the SHERLOCK architecture (Appendix A); and 2) The different prototypes developed, based in our approach for the semantic management of Location-Based Services in wireless environments, and tests (Appendix B).

Chapter 2

Technological Context

In this chapter, we describe concepts and technologies related to our work in order to help understanding the rest of the thesis. First, we focus on *knowledge management* that is one of the pillars of SHERLOCK. We describe important concepts such as ontologies and languages used to model them, semantic reasoners to infer non-explicitly defined knowledge in these ontologies, and query languages to retrieve information from such knowledge. Second, we focus on *mobile computing*, which is the other pillar of SHERLOCK, describing concepts such as Location-Based Services and mobile agents. Finally, we focus on *knowledge management on mobile devices* and more precisely we overview how mobile applications in the literature use semantic data.

2.1 Knowledge Management

In this section we present the main semantic technologies used in the thesis. First, we use ontologies to model the knowledge managed by our system and ontology alignment techniques to integrate the knowledge shared by SHERLOCK devices. We selected Description Logic (DL) [BCMNP03] as the formalism to represent these ontologies. Also, we use semantic reasoners based on DL to infer non-explicit knowledge from the explicit facts defined. Finally, we use a query language based on SPARQL, and its GeoSPARQL and SPARQL-DL extensions to formalize user requests in the system.

2.1.1 Ontologies

The term ontology was defined by Tom Gruber as “an explicit specification of a conceptualization” [Gru93]. Therefore, ontologies allow to model and capture

the semantics of different knowledge domains, providing a means to share definitions, and reach an implicit agreement on the meaning of the published information. There exist different formalisms to define ontologies and the basic elements in them include:

- *Instances*, which are objects of the world.
- *Classes*, which are sets of instances usually organized in hierarchies.
- *Attributes*, which represent relationships between classes of the domain, or between classes and a *datatype* value.

Figure 2.1 shows a very simple example of ontology. This ontology models the class *PhDStudent* as a subclass or *Student*, having two properties *dissertationTitle* (which takes a string value) and *hasSupervisor* (which connects *PhDStudent* and *Professor* classes). According to the ontology, *RobertoYus* (an instance of *PhDStudent*) has supervisor *EduardoMena* (an instance of *Professor*) and his dissertation title is “Semantic Management of LBS in Wireless Environments”.

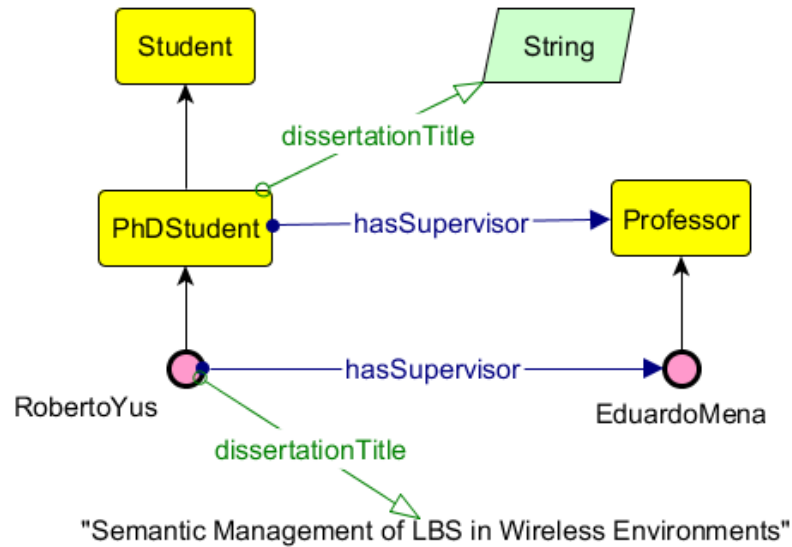


Figure 2.1: Example of ontology.

The Semantic Web [BLHL+01] was proposed to make the information on the Web readable by machines. One of the main tasks to achieve that goal is to

annotate the content, and ontologies have been traditionally used for it. Since then, different efforts have been made to structure the content of the Web and nowadays, thanks in part to the Linked Data [BHBL09] movement, hundreds of resources/ontologies are available¹. As an example, Figure 2.2 shows two ontologies, O_1 and O_2 , extracted from the website of two universities.

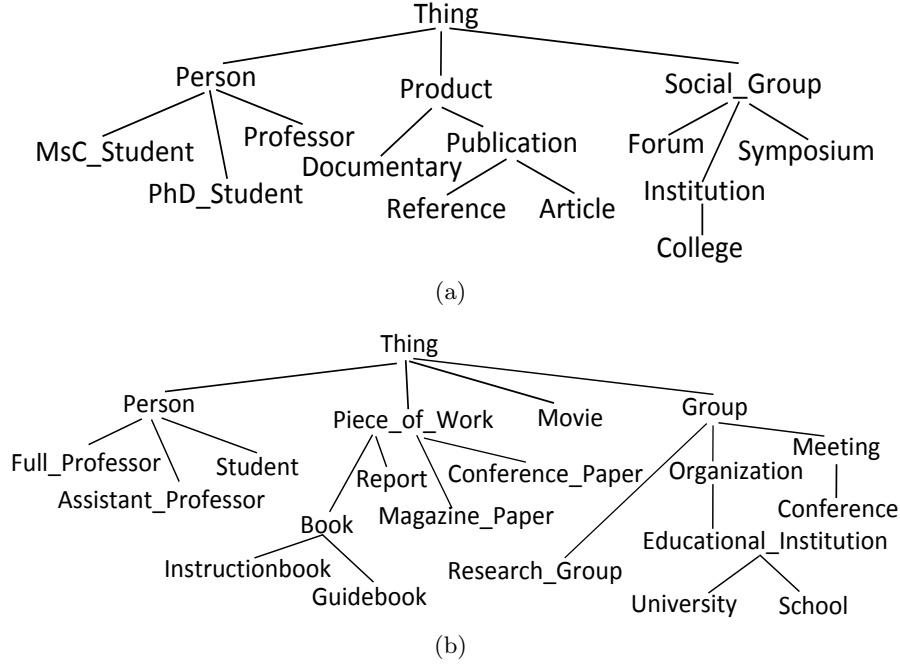


Figure 2.2: Two sample ontologies describing knowledge about universities.

One of the main goals of ontologies is to provide a common model for annotating content and thus help systems to interoperate. However, interoperability problems still remain as usually terms defined in different ontologies are related according to their definitions (sometimes with a varying level of detail) but not directly linked. As an example consider the sample ontologies, O_1 and O_2 . The knowledge that they model is related, as both represent information about universities, but their definitions are slightly different. Thus, *ontology alignment* [ES+07] systems were designed to try to find semantic relationships such as, for example, *synonymy*, *hypernymy*, and *hyponymy*, which could relate elements from different ontologies.

¹<http://www.linkeddata.org>

Ontology Alignment

There are several definitions of ontology alignment, however the most accepted one is given by Sowa [Sow99]:

“[...] the process of finding common elements between two different ontologies A and B to produce as result a new ontology C that favors the interoperability between computer systems based on the domains of both original ontologies”.

There has been a considerable amount of work in the ontology alignment area [SE13]. Most of the efforts have been made in the alignment of ontologies through the extraction of synonymy relationships (i.e., extracting that the concept A from ontology O_1 and the concept B from ontology O_2 are equivalent).

Synonymy is a very strict relationship that implies, in fact, that the two entities have the *same meaning*. On the contrary, in the real world it is much more common to find terms that are quite similar but not exactly the same (e.g., one of the terms could be more general than the other, it could *subsume* the other term). There are only a few works focused on discovering subsumption relationships and they are based on: the use of external sources of information where the relationships could be defined [BBCCGMMV00; SdM08] (but sometimes the relationships are not defined anywhere); instances in the ontologies [KLXWL05] (but not all the ontologies contain enough instances for that); and in classification and training methods [SVV08] (which depend on the training data).

To integrate these ontologies the relationships between their terms have to be discovered. Considering our running example, an ontology alignment system would have to discover the existing relationships between their concepts (see Figure 2.3). In this case, there exist two synonyms (e.g. $O_1\#College \equiv O_2\#University$ and $O_1\#Person \equiv O_2\#Person$) and many *subsumption* relationships such as $O_2\#FullProfessor \sqsubseteq O_1\#Professor$ and $O_1\#PhD-Student \sqsubseteq O_2\#Student$.

2.1.2 Description Logics

Description Logics languages (DLs) are “formal languages for representing knowledge and reasoning about it” [BCMNP03]. In DL-based ontologies, the basic ontological representation primitives (also called ontology elements) are *individuals* (which are the instances explained before), *concepts* (classes), and *properties* (attributes). In addition, there exist also the following primitives:

- *Datatypes*, which represent concrete data values such as numbers (e.g.,

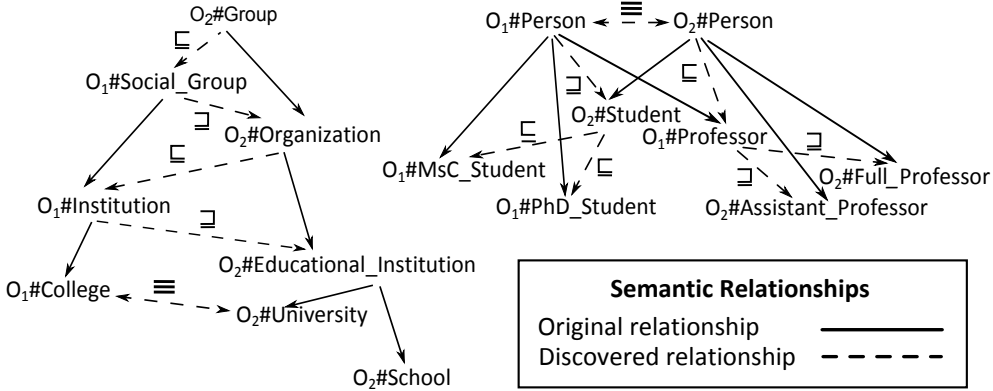


Figure 2.3: Semantic relationships that exist among the concepts of two ontologies.

real, rational, integer, nonnegative, etc.), strings, booleans, dates, times, or XML literals, among many other possibilities.

- *Axioms*, which are formal conditions to be verified by the elements. An ontology can be seen as a finite set of axioms, usually divided in three parts: an assertional box (*ABox*), a terminological box (*TBox*), and a role box (*RBox*), with axioms about individuals, concepts, and roles, respectively. For example, an *ABox* can assert that *University of Zaragoza* is a member of the concept *University*, and a *TBox* can assert that the concept *College* is equivalent to *University*, usually denoted $College \equiv University$, or that the concept *PhD Student* is a subclass of *Student*, usually denoted $PhDStudent \sqsubseteq Student$.

DLs are a well-known formalism providing a good trade-off between expressivity of the representation and efficiency of the reasoning. Each DL is denoted by using a string of capital letters which identify its expressivity. For instance, the standard language for ontology representation OWL 2 is equivalent to the DL $\mathcal{SROIQ}(\mathbf{D})$. The expressivity of a DL translates in what kind of constructors can be used to form new concepts. For example, if letter \mathcal{C} is in the expressivity of a DL, it means that it can use the constructor \neg to express the contrary of a concept ($Woman \sqsubseteq \neg Man$, a woman cannot be a man). We summarize informally in Table 2.1 the most common constructors that lead to a DL with expressivity \mathcal{ACL} , while other logics and their allowed constructors are presented in Table 2.2. For more formal details, see [BCMNP03].

\top	any element
\perp	no element, empty set
A	atomic concept
$\neg C$	elements that are not in C
$C \sqcap D$	elements in C and D
$C \sqcup D$	elements in C or D
$\forall R.C$	elements a such that if a is related with b by the property R , then b is in C
$\exists R.C$	elements a such that are related by property R with an element b in C

Table 2.1: Constructors and their meanings for \mathcal{ALC} DL.

DL Reasoners

A semantic reasoner is a software able to infer logical consequences from a set of facts. According to Sirin et al. [SPCGKK07], a practical OWL reasoner should provide the following set of DL inference services:

- *Consistency checking*, which checks whether an ontology contains any contradictions or not.
- *Concept satisfiability*, which checks if a class can have instances.
- *Classification*, which computes the complete class hierarchy based on the subsumption relation between the ontology classes.
- *Realization*, which finds the most specific concepts a given individual is an instance of.

The use of semantic reasoners would enable the development of more intelligent applications capable of discovering new knowledge, inferred from the available information. Many reasoners have been proposed in the literature and we review in the following some of them.

Logic	Expressivity	Complexity class
\mathcal{AL}	$\top, \perp, \sqcap, \forall, \neg A, \exists R. \top$	P _{TIME}
$\mathcal{ALC}(= \mathcal{ALUE})$	$\top, \perp, \sqcap, \sqcup, \forall, \neg, \exists$	EXP _{TIME}
$\mathcal{SHIF}(\mathbf{D})$ (OWL Lite)	$(\mathcal{S} =) \mathcal{ALC}$ + transitive roles, role hierarchies (\mathcal{H}), inverse roles (\mathcal{I}), functional roles (\mathcal{F}), concrete domains (\mathbf{D})	EXP _{TIME}
$\mathcal{SHOIN}(\mathbf{D})$ (OWL DL)	\mathcal{SHI} , nominals (\mathcal{O}), non-qualified numerical restrictions (\mathcal{N}), concrete domains (\mathbf{D})	NEXP _{TIME}
$\mathcal{SROIQ}(\mathbf{D})$ (OWL 2)	$\mathcal{SHOIQ}(\mathbf{D})$, complex role inclusion (\mathcal{R}), self-restriction, and additional role axioms	N ² EXP _{TIME}
$\mathcal{EL}^{++}(\mathbf{D})$ (OWL 2 EL)	$\top, \perp, \sqcap, \exists$, role hierarchies, nominals, concrete domains (use of constructors with syntactical restrictions)	P _{TIME}
DL-Lite (OWL 2 QL)	$\top, \perp, \sqcap, \exists, \neg$ (use of constructors with syntactical restrictions)	LOGSPACE
DLP (OWL 2 RL)	$\top, \perp, \sqcap, \sqcup, \forall, \neg, \exists$, cardinality restriction (0..1) (use of constructors with syntactical restrictions)	P _{TIME}

Table 2.2: Expressivity and complexity of reasoning in some important DLs.

CB^2 [Kaz09] (Consequence-Based) reasoner supports a fragment of OWL 2 (Horn- \mathcal{SHIF}). As its name suggests, the reasoner algorithm does not build models but infers new consequent axioms. CB is implemented in OCaml and, as far as we know, the only supported reasoning task is classification. It can be used from command line, as a Protégé plug-in, and through the OWL API.

ELK^3 [KKS14], implemented in Java, is a Consequence-Based reasoner for a subset of OWL 2 EL. It supports different reasoning tasks, which include classification, consistency checking, subsumption, and realization. The classification procedure is different from other algorithms for OWL 2 EL. For instance, it includes several optimizations such as concurrency of the inference rules. ELK can be used through several interfaces, including OWL API.

²<http://www.cs.ox.ac.uk/isg/tools/CB>

³<http://elk.semanticweb.org>

*HermiT*⁴ [GHMSW14] implements a hypertableau reasoning algorithm with several optimization techniques. It supports OWL 2 and DL safe rules. Historically, it was the first DL reasoner that was able to classify some large ontologies (such as GALEN-original) thanks to a novel and efficient classification algorithm. Inference services include concept satisfiability, consistency, classification, subsumption, realization, and conjunctive query answering. *HermiT* is implemented in Java, and is accessible through several interfaces, including the OWL API and a Protégé plug-in.

*jcel*⁵ [Men12] is a Java implementation of a tractable classification algorithm for a subset of OWL 2 EL. *jcel* is based on *CEL*⁶ [BLS06] (Classifier for \mathcal{EL}) reasoner, a Common LISP implementation of a rule-based completion classification algorithm. Both reasoners are open source and accessible through the OWL API; *jcel* can also be used using a Protégé plug-in.

*JFact*⁷ is a Java port of the reasoner *FaCT++*, although it does not include all of its parts and provides an improved datatype support. *FaCT++* reasoner⁸ [TH06] is a successor of *Fact* reasoner (*FAst Classification of Terminologies*) [Hor98]) using a different architecture and a more efficient implementation (*FaCT* was written in Common Lisp, and *FaCT++* in C++). Both *FaCT++* and *JFact* completely support OWL 2 and implement a tableau algorithm [BCMNP03] with several optimization techniques. Supported reasoning tasks include concept satisfiability, consistency, classification, and subsumption. From a historical point of view, *FaCT++* was the first reasoner fully supporting OWL 2. Both reasoners can be used through the OWL API and are available under a GNU license.

*MORe*⁹ [ARCGHJR13] is an OWL 2 metareasoner that exploits module extraction techniques to divide complex reasoning tasks into simpler ones that can be solved using different reasoners. The modules of the ontology in the OWL 2 EL profile are solved by the ELK reasoner, and the more expressive ones are handled using *HermiT* and *JFact*. *MORe* currently supports classification and concept satisfiability. It is implemented in Java, open source, and accessible through the OWL API and using a Protégé plug-in.

*Pellet*¹⁰ [SPCGKK07] supports full OWL 2 and DL safe rules. It implements a tableau algorithm with several optimization techniques. It was the

⁴<http://www.hermit-reasoner.com>

⁵<http://jcel.sourceforge.net>

⁶<http://lat.inf.tu-dresden.de/systems/cel>

⁷<http://jfact.sourceforge.net>

⁸<http://owl.man.ac.uk/factplusplus>

⁹<http://code.google.com/p/more-reasoner>

¹⁰<http://clarkparsia.com/pellet>

first reasoner fully supporting OWL 1 DL. Inference services include concept satisfiability, consistency, classification, subsumption, realization, and conjunctive query answering. Pellet is implemented in Java and has multiple interfaces to access it, including OWL API.

*TrOWL*¹¹ [TPR10] is implemented in Java and supports OWL 2, offering sound and complete reasoning for OWL 2 EL and OWL 2 QL, and approximate reasoning for OWL 2 DL. Inference services include classification and conjunctive query answering. TrOWL includes an OWL 2 EL reasoner (*REL*) to compute the classification and an OWL 2 QL reasoner (*Quill*) to answer conjunctive queries. Reasoning with OWL 2 DL ontologies is achieved by means of a syntactic approximation into OWL 2 EL or a semantic approximation into OWL 2 QL, depending on the reasoning task. TrOWL can be used through several interfaces, including OWL API.

*TReasoner*¹² [GI13] supports a subset of OWL 2, namely the Description Logic $\mathcal{SHOIQ}(\mathbf{D})$. *TReasoner* solves classification, concept satisfiability, and consistency using a tableau algorithm with several optimization techniques. It is implemented in C++ and supports the OWL API.

Table 2.3 shows a summary of every reasoner introduced before. There exist many others that we have not consider in this thesis. We will enumerate now, in alphabetical order, only those of them that will be mentioned at some point of this thesis document:

*BaseVISor*¹³ [MBK06], *Chainsaw*¹⁴ [TP12], *ConDOR*¹⁵ [SKH11], *DB*¹⁶ [DK09], *ELepHant*¹⁷ [Ser13], *fuzzyDL*¹⁸ [BS16], *KAON2*¹⁹ [MS05], *Konclude*²⁰ [SLG14], *OWLIM*²¹ [BKOTV11] (a family of repositories including *SwiftOWLIM* reasoner), *Racer*²² [HHMW12], *SHER* [DFKSS09], *SOR* [LMZBWPY07], *SnoRocket*²³ [LB10], *WSClassifier*²⁴ [SSD13], and *WSReasoner*²⁵ [SSD12].

¹¹<http://trowl.org>

¹²<http://code.google.com/p/treasoner>

¹³<http://vistology.com/basevisor/basevisor.html>

¹⁴<http://sourceforge.net/projects/chainsaw>

¹⁵<http://code.google.com/p/condor-reasoner>

¹⁶<https://code.google.com/p/db-reasoner>

¹⁷<https://github.com/sertkaya/elephant-reasoner>

¹⁸<http://webdiis.unizar.es/~fbobillo/fuzzyDL>

¹⁹<http://kaon2.semanticweb.org>

²⁰<http://www.derivo.de/en/produkte/konclude>

²¹<http://www.ontotext.com/owlim>

²²<http://www.ifis.uni-luebeck.de/index.php?id=385>

²³<http://github.com/aeherc/snorocket>

²⁴<http://code.google.com/p/wsclassifier>

²⁵<http://isew.cs.unb.ca/wsreasoner>

Reasoner	Profile	Language	License	OWL API
CB	OWL 2 DL (FRG)	OCaml	LGPL	Yes
ELK	OWL 2 EL (FRG)	Java	Apache 2.0	Yes
HermiT	OWL 2 DL	Java	LGPL	Yes
jcel	OWL 2 EL (FRG)	Java	LGPL/Apache 2.0	Yes
JFact	OWL 2 DL	Java	LGPL	Yes
MORe	OWL 2 DL	Java	GPL	Yes
Pellet	OWL 2 DL	Java	Dual	Yes
TReasoner	OWL 2 DL (FRG)	Java	GPL	Yes
TrOWL	OWL 2 DL (APX)	Java	Dual	Yes

Table 2.3: Semantic Web reasoners and some of their characteristics. *FRG*: Fragment; *APX*: Approximated.

DL Reasoners Designed for Mobile Devices

Apart from the DL reasoners presented in the previous section, several DL reasoners were specifically designed to run on mobile devices. We dedicate this section to overview them in a chronological order. *Pocket KRHyper*²⁶ [SK05] was the first reasoning engine specifically designed for mobile devices. It can be seen as a version of the reasoner KRHyper [Wer03] for devices with limited resources, thus disabling some of its original capabilities (such as default negation and term indexing). It is implemented in J2ME (Java Micro Edition) and implements a hypertableau algorithm for the DL *SHI*. However, the reasoner suffers from scalability issues, as the authors state in [Kle06].

Later on, Müller et al. [MHLN06] reported the implementation of tableau algorithm for mobile devices, introducing some optimizations to reduce the memory usage such as assigning natural numbers to concept expressions to reduce comparisons to integer operations. Their system is implemented in J2ME and supports the DL *ALCN* with unfoldable TBoxes, but it does not have a known name and is not publicly available.

mTableau [SKG09; SK08] is a modified version of Pellet 1.5 to work on mobile devices. The main idea is to introduce some novel optimization techniques, namely selective application of consistency rules, skipping disjunctions, and ranking of individuals and disjunctions leading to potential clashes. This

²⁶<http://mobilereasoner.sourceforge.net>

reasoner is not publicly available.

Delta [MHK12] is designed to be used on mobile devices, but no implementation details are given. The reasoner uses RDF to store the ABox and OWL RL to represent the TBox axioms. The main reasoning task is conjunctive query answering, which is solved by translating TBox axioms into rules to expand the RDF triple store. The reasoner uses incremental reasoning techniques to avoid recomputing all the inferences every time there is an update of the ABox facts. A preliminary evaluation is performed, obtaining sub-second query response times. This reasoner is not publicly available.

*Mini-ME*²⁷ [RSSGL12] (Mini Matchmaking Engine) is a mobile reasoner implemented from scratch. The supported DL is the DL \mathcal{ALN} , whereas the supported reasoning tasks are consistency, classification, concept satisfiability, subsumption, and other non-standard inference services (abduction, contraction, and covering). It is implemented in Java and can be run on Android devices as well as on desktop computers. Mini-ME can be accessed through the OWL API, as a OWLink server, or using a Protégé plug-in. The authors have empirically compared the performance of Mini-ME in a mobile device and in a desktop computer. It turned out that reasoning times are roughly one order of magnitude higher in the Android device [RSSGL12]. However, it should be stressed that these results only hold for the not very expressive logic \mathcal{ALN} , having polynomial computational complexity. The authors also performed some experiments proving that the Android version of Mini-ME outperforms an older version developed in J2ME [RSS11].

2.1.3 Representation Languages

Ontologies represent the vocabulary of some domain from a common perspective using a formal language. In the following we explain the most important languages in the Semantic Web: RDF and OWL.

RDF and RDF-S

Resource Description Framework (RDF) [MMM+04] is a family of World Wide Web Consortium (W3C) specifications designed to represent information about resources on the Web. The representation followed by RDF is a triple $\langle s p o \rangle$ where s is the subject, which is the resource, p is the predicate, which is an aspect or trait of the resource, and o is the object. To denote the resources in these triples, RDF statements use the uniform resource identifier (URI), or blank

²⁷<http://sisinflab.poliba.it/swottools/minime>

nodes. Probably the most important piece of vocabulary introduced by RDF is *rdf:type* which can be used to state that a resource is an instance of a class. For instance we can use the triple $\langle \textit{RobertoYus} \textit{ rdf : type } \textit{PhDStudent} \rangle$ to define that the resource *RobertoYus* is an instance of the class *PhDStudent*.

RDF-S (RDF Schema) [BG04] is a “general-purpose language for representing simple RDF vocabularies on the Web”. Therefore, RDF-S allows the description of relations between RDF resources (see Table 2.4 for the main RDF-S constructs). For instance, RDF-S allows to define that *PhDStudent* is a subclass of *Student* and the different properties associated with it.

Classes	Properties	Utility properties
rdfs:Resource	rdfs:domain	rdfs:seeAlso
rdfs:Class	rdfs:range	rdfs:isDefinedBy
rdfs:Literal	rdfs:type	
rdfs:Datatype	rdfs:subClassOf	
rdfs:XMLLiteral	rdfs:subPropertyOf	
rdfs:Property	rdfs:label	
	rdfs:comment	

Table 2.4: Main RDF-S constructs.

Web Ontology Language (OWL)

Web Ontology Language (OWL) [HKPPSR09] is a family of knowledge representation languages for describing ontologies. OWL is characterized by formal semantics and indeed, the current version of the language, OWL 2, is equivalent to the DL *SR_QIQ(D)*. OWL extends RDF-S and allows the representation of ontological knowledge missing in RDF-S such as range restrictions that apply to some classes only, classes that are disjoint (whose individuals cannot belong to them at the same time), or union of classes. Therefore, OWL allows the definition of more expressive ontologies than RDF/RDF-S. For instance, OWL 2 supports the definition of complex classes using the class expressions in Figure 2.4²⁸.

In OWL 2 there are three sublanguages or profiles which can be more simply and/or efficiently implemented:

²⁸The complete OWL 2 syntax can be found in <https://www.w3.org/TR/owl2-syntax>


```

ClassExpression :=
  Class |
  ObjectIntersectionOf | ObjectUnionOf | ObjectComplementOf | ObjectOneOf |
  ObjectSomeValuesFrom | ObjectAllValuesFrom | ObjectHasValue | ObjectHasSelf |
  ObjectMinCardinality | ObjectMaxCardinality | ObjectExactCardinality |
  DataSomeValuesFrom | DataAllValuesFrom | DataHasValue |
  DataMinCardinality | DataMaxCardinality | DataExactCardinality

```

Figure 2.4: Definition of OWL 2 class expression.

1. *OWL 2 EL*, which is a fragment that has polynomial time reasoning complexity.
2. *OWL 2 QL*, which is designed to enable easier access and query to data stored in databases.
3. *OWL 2 RL*, which is a rule subset of OWL 2.

These profiles are independent of each other and the choice of a profile to model an ontology depends on its structure and reasoning tasks to consider²⁹.

2.1.4 Query Languages

Query languages are computer languages used to query databases and information systems. For example, SQL is the standard language used to query relational databases. In the following we introduce SPARQL, the standard query language of the Semantic Web, as well as two extensions of this language, GeoSPARQL and SPARQL-DL, used in our system to query geospatial data and OWL ontologies, respectively.

SPARQL

SPARQL [PS+08] (a recursive acronym for SPARQL Protocol and RDF Query Language) is an RDF query language, that is, a semantic query language for databases, able to retrieve and manipulate data stored in Resource Description Framework (RDF) format. It was made a standard by the RDF Data Access Working Group (DAWG) of the World Wide Web Consortium, and is recognized as one of the key technologies of the Semantic Web.

There are four different query variations in SPARQL for different purposes:

²⁹More information about the features of each profile can be found in <https://www.w3.org/TR/owl2-profiles>

- *SELECT*, which is used to extract raw values.
- *CONSTRUCT*, which extract results in RDF.
- *ASK*, which obtains a true/false results for the query.
- *DESCRIBE*, which extract an RDF graph containing information about the given resource.

In the rest of this document we will use mainly *SELECT* queries to ask directly for facts and data.

GeoSPARQL

GeoSPARQL [BK11] is a standard for representation and querying of geospatial linked data for the Semantic Web from the Open Geospatial Consortium (OGC). In particular, GeoSPARQL provides for:

- A small topological ontology in RDF-S/OWL for representation using:
 - Geography Markup Language (GML) and well-known text (WKT) literals.
 - Simple Features, RCC8, and DE-9IM (a.k.a. Egenhofer) topological relationship vocabularies and ontologies for qualitative reasoning.
- A SPARQL query interface using:
 - A set of topological SPARQL extension functions for quantitative reasoning.
 - A set of Rule Interchange Format (RIF) Core inference rules for query transformation and interpretation.

SPARQL-DL

According to [SP07] “it is harder to provide a semantics for [SPARQL] under OWL-DL semantics because RDF representation mixes the syntax of the language with its assertions. The triple patterns in a query do not necessarily map to well-formed OWL-DL constructs.” Therefore, Sirin et al. defined SPARQL-DL [SP07]: “a substantial subset of SPARQL that can be covered by the standard reasoning services OWL-DL reasoners provide.” SPARQL-DL uses the SPARQL syntax and is fully aligned with the OWL 2 standard. Table 2.5 shows the supported query patterns for the SPARQL-DL language.

Classes	Properties	Individuals
Class(a)	Property(a)	Individual(a)
EquivalentClass(a, b)	PropertyValue(a, b, c)	Type(a, b)
SubClassOf(a, b)	EquivalentProperty(a, b)	DirectType(a, b)
StrictSubClassOf(a, b)	SubPropertyOf(a, b)	SameAs(a, b)
DirectSubClassOf(a, b)	StrictSubPropertyOf(a, b)	DifferentFrom(a, b)
DisjointWith(a, b)	DirectSubPropertyOf(a, b)	
ComplementOf(a, b)	ObjectProperty(a)	
	DataProperty(a)	
	Functional(a)	
	InverseFunctional(a)	
	Transitive(a)	
	Symmetric(a)	
	Reflexive(a)	
	Irreflexive(a)	
	InverseFunctional(a)	
	InverseOf(a, b)	

Table 2.5: SPARQL-DL supported query patterns.

2.1.5 Tools for Developing Semantic Applications

Finally, we want to mention two important Semantic Web APIs we used in our prototypes to handle ontologies:

*Jena*³⁰ [McB02] is an ontology API to manage OWL ontologies and RDF data in Java applications. Jena is appropriate to manage OWL 1 Full ontologies, but support for OWL 2 is not available yet. However, it is much more used for the serialization of RDF triples and the manipulation of RDF graphs. Jena can interact with semantic reasoners to discover implicit knowledge. The latest versions of Jena are split into two packages, namely *jena-fuseki* (with the Jena SPARQL server), and *apache-jena* (with APIs, SPARQL engine, RDF database, and other tools).

*OWL API*³¹ [HB11] is an ontology API to manage OWL 2 ontologies in Java

³⁰<http://jena.apache.org>

³¹<http://owlapi.sourceforge.net>

applications and provides a common interface to interact with DL reasoners. It can be considered as de facto standard, as the most recent versions of most of the semantics tools and reasoners use the OWL API to load and process OWL 2 ontologies. The OWL API is able to process each of the OWL 2 syntaxes defined in the W3C specification (functional, RDF/XML, OWL/XML, Manchester, and Turtle) and to identify the OWL 2 profiles (OWL 2 DL, OWL 2 EL, OWL 2 QL, and OWL 2 RL). The OWL API is less appropriate for the management of OWL 2 Full or RDF ontologies.

2.2 Mobile Computing

Mobile computing implies the possibility of computers being transported around by users. Indeed, in the last few years, we have witnessed a massive spread of mobile computing which is shaping our daily lives. This has been undoubtedly helped by the pervasive connectivity that the current wireless networks provide us with and the affordable prices of current mobile devices (such as smartphones and tablets). In this section we present the main technologies related to mobile computing used in the thesis. First, as the thesis presents a system to provide Location-Based Services, we present some information about them and their building block, location-dependent queries. Then, we explain the context information, which mobile devices can infer from their equipped sensors, and used in our system to select services which might be interesting for a user. Finally, we introduce the concept of software agent focusing on mobile agents, which have been previously used for distributed and mobile computing applications. In our system, these agents are used to perform different tasks such as, for example, finding the information that the user needs wherever it is.

2.2.1 Location-Based Services

In the last years the interest in mobile computing has grown due to the ever-increasing use of mobile devices and their pervasiveness. The low cost of these devices, along with the high number of sensors and communication mechanisms they are equipped with, make it possible to develop useful information systems. Using special kinds of sensors, location mechanisms enable the development of *Location-Based Services (LBS)* [SV04]. These services provide value added by considering the locations of the mobile users to offer customized information. For example, LBS for taxi searching [SCC10], helping firefighting [JCHWTL04], detecting nearby friends [AEMPW07], or multimedia retrieval in sport events [IMIYLM12] have been presented, among many others.

Location-Dependent Queries

Location-dependent queries are a special type of query for which, opposed to traditional queries, their answer depend on locations of objects. Thus, the location of an object determine whether the object is part of the answer or not. For example, the query “find taxis within 2 miles” depends on the location of the user and a taxi will be retrieved if its location is less than 2 miles apart from the location of the user. Therefore, location-dependent queries are a fundamental building block of LBS which use them to obtain the information that the user needs.

There exist multiple types of location-dependent queries considered in the literature (see [IMI10] for a complete classification) among which we would like to highlight:

- *Range queries* [TWHC04], which retrieve objects in a region, which can be fixed or even move, within a range (e.g., “find museums in Zaragoza”). Range queries are called *within-distance queries* [TS03] when the range is a circle (e.g., “find taxis within 2 miles”).
- *Nearest neighbor queries (NN queries)* [TPS02], which retrieve the object which is closest to a location or object (e.g., “find the closest gas station”). They are called *kNN queries* if more than one object must be retrieved (e.g., “find the closest five restaurants to the Lincoln Monument”).

There are other possible classifications of location-dependent queries according to temporal or semantic factors, among others. From them, we want to highlight the concept of *instantaneous* and *continuous* queries for which the answer is computed only once or it is reevaluated, respectively. For example, queries which depend on the location of the user are usually treated as continuous as the user might move and the result should be updated accordingly.

2.2.2 Context-Aware Computing

Current mobile devices are equipped with sensors that enable them to go beyond *location awareness*, the capability of determining their location. Therefore, they can determine, for example, that the user is standing up, talking, in the middle of a research meeting with her peers, in the meeting room of the building, and on a Friday at 2pm. This information is part of the *context* of the user and so, *context awareness* [SAW94] is a property of current mobile devices.

Context awareness enables devices to react based on the environment and has been leveraged in many ubiquitous systems. For instance some classic

examples of context-aware applications (extracted from [CK00]) are: 1) a call forwarding system that detects the location of the user and her activity to forward her calls; and 2) a shopping assistant system that displays information of items whenever the user enters a shop and recommends what to buy according to the user preferences.

The concept *context* has many interpretations in the literature and so, there is no unified definition of it. Most of the definitions agree that *context has something to do with the interactions between the users and the computing systems* [CK00]. Arguably one of the most accepted definitions for context was suggested by Dey and Abowd [ADBDSS99]:

“[...] any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and application, including the user and applications themselves.”

Dey and Abowd also decompose context into two categories: *primary context pieces* (i.e., identity, location, activity, and time) and *secondary context pieces* (context aspects that are attributes of the primary context, e.g., a user’s phone number can be obtained by using the user’s identity).

Also, there are many approaches on how to infer the context of a user using context providers and synthesizers. According to Ranganathan et al. [RAMC04], a *context provider* is a sensor and/or other data sources of context information. In the same way, a *context synthesizer* is a mechanism that gets the information obtained by context providers and deduces a higher-level notion of context (e.g., the location or activity of a user).

2.2.3 Agent Technology

Agent technology has been used in the literature to develop distributed and mobile computing applications [Fer99]. In the following we give a brief explanation of software agents including mobile agents that are used in the architecture proposed in this thesis.

2.2.4 Software Agents

There is no consensus in the definition of what an *agent* is [Nwa96]. However, we can consider that a software agent is *a program that acts on behalf of a user*. Among the many different properties that might be associated with agents we highlight the following (extracted from [Ila06]):

- *Autonomy* [FG97]: ability to act without direct human intervention.

- *Sociability* [SS02]: ability to think about itself or about others (altruistic and egoistic agents).
- *Reactivity* [GDFLP02]: ability to respond to changes in the environment.
- *Proactivity* [RM00] or goal-directed behavior.
- *Temporal continuity* [FG97]: persistency over long periods.
- *Learning/adaptivity* [FG99]: ability to learn from the environment and from other agents to improve performance.
- *Reasoning* [RNCME96]: decision-making mechanism.
- *Mobility* [CHK97]: ability to migrate to other components.
- *Cooperation* [Les99]: interaction with other agents to achieve a goal.
- *Negotiation* [Smi77] (e.g., to allocate subtasks to different agents).

Autonomy has been usually considered inherent to all the agents but not all of them have to provide the rest of functionalities [RT98].

Mobile Agents

Agents that have the mobility property mentioned before are called *mobile agents* [CHK97]. Mobile agents are programs that execute in context denominated *places* and can autonomously travel from place to place resuming their execution there. Thanks to their mobility, mobile agents offer interesting benefits [LO99; Ila06]:

1. *They encapsulate protocols.* As they can move to remote computers to achieve their goals, they avoid the need for installing specialized server processes on every machine to provide access to all the required services.
2. *They reduce the network load.* Thus, a mobile agent can travel where the data are and access to them locally, filtering out the data that do not need to be sent over the network.
3. *They overcome network latency.* Thus, they can move to another computer in order to optimize the response time.

4. *They are asynchronous and autonomous.* A mobile agent does not need to keep contact with its source computer while performing its task. This is particularly important with mobile devices, which usually communicate via an expensive and unreliable wireless connection.
5. *They adapt dynamically to their environment.* They are able to sense the environment and can be programmed in order to react autonomously to adapt themselves to changes. For example, they could travel to another computer when the current computer is overloaded.
6. *They contribute to a seamless system integration.* Usually, hardware and software components are highly heterogeneous in a network. Mobile agents help to overcome heterogeneity issues because they are generally computer-independent and transport-layer-independent.
7. *They are robust and fault-tolerant.* Their ability to react dynamically to unfavorable situations and events makes it easier to build robust and fault-tolerant distributed systems. If a host is being shut down, all agents executing on that machine can be warned and given time to move and continue their tasks on another host in the network.

Mobile agents have been used in many different areas because of these benefits, such as distributed information retrieval, parallel processing, monitoring and notifying applications, and personal assistance, among others.

2.3 Knowledge Management on Mobile Devices

As we explained before, semantic technologies have been traditionally used for knowledge representation and management. The use of semantic technologies on mobile devices has been subject of interest from the early stages of the Semantic Web [WRSOS05]. However, currently the use of semantic technologies on mobile applications is not extended in comparison with the overwhelming amount of existing apps. In the following we present a summary of the results of a systematic review of semantic mobile applications which we published in [YP15].

Semantic Mobile Applications

We consider that a semantic mobile application is a application designed for mobile devices which uses semantic technologies for the management of

the information it considers. We analyzed more than 400 papers and found that at least 36 semantic mobile apps have been presented in the literature over the last 10 years (see Table 2.6). We want to highlight that 3 out of this 36 semantic mobile apps were developed as part of the work presented in this thesis (SHERLOCK, FaceBlock, and Rafiki in the previous table). In the following we provide some information from the study to show the distribution of these semantic mobile apps according to the year when they were presented, their domain, the platform where they were deployed, and the semantic technologies used.

[RSFIBS14]	Alive Cemeteries [MK14]
mSWB [MGK14]	Donate-N-Request [SSLPMC13]
WeReport [SSLPMC13]	Krishi-Mantra [KDNCB13]
Rafiki [PYJF14]	Who's Who [CDH11]
Cinemappy [ONMRS12]	SHERLOCK [YMII14]
PediaCloud [TJV13]	TouristGuide [DL14]
HDTourist [HMFC14]	CURIOUS Mobile [NBWMPW14]
RealFoodTrade [CVNMMNORU14]	ParkJam [KD12]
csxPOI [BSS10]	Urbanopoly [CCCCDVF12]
FaceBlock [YPDMJF14]	RDFContentProvider [DE10]
LinkedQR [ELLL12]	RDF On the Go [PPRH10]
[dNM11]	Linked Sensor Middleware [LPQPH11]
GetThere [CEBMPN13]	[TFAEA11]
[SXJMTL11]	mSpace [WRSOS05]
Person Matcher [WCT10]	LOD4AR [VDV14]
OntoWiki Mobile [EHTA11]	[RSILS12]
DBpedia Mobile [BB09]	[AAA13]
Mobile Wine Agent [PM09]	[AWH10]

Table 2.6: Semantic mobile applications presented in the literature.

Figure 2.5 shows the number of papers presenting a semantic mobile app per year (the figure do not include one app published in 2015³²). Notice that

³²This study has been finished in May 2015 so more semantic mobile apps might be presented in 2015.

there is a gap between 2005 and 2009, we believe that this might be related to two milestones: the release of the iPhone in June 29, 2007, and the release of the first commercial version of Android in September 23, 2008. With the more powerful and affordable devices, high speed Internet, and better tools available, the number of mobile semantic web apps doubled in 2010 and 2014 whereas it remained stable in between.

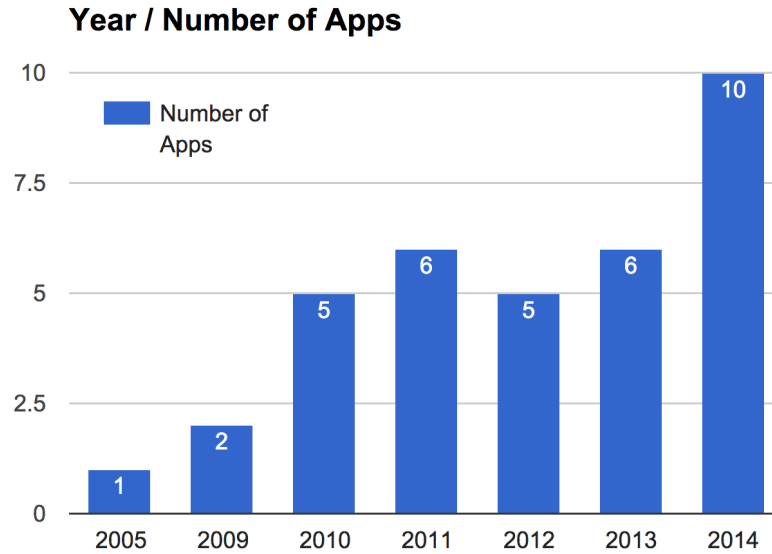


Figure 2.5: Number of semantic mobile apps per year.

The majority of the apps reviewed, 27 apps out of 36, can be classified as Location-Based Services (LBS). This was expected as mobile devices are equipped with sensors which are able to obtain the location of the user in real-time. Among these LBS apps, the most common functionality is providing information about Points Of Interest (POI), 14 apps.

In general all the apps are deployed on smartphones, except for [WRSOS05; WCT10] which were deployed on Personal Digital Assistants (PDAs), as they were developed when PDAs were the most popular mobile devices. Figure 2.6 shows the distribution across different operating systems with Android being the most common choice for semantic mobile apps (27 out of 36). 3 of the apps were developed for iOS whereas 2 are Windows Mobile apps. Also, there are 4 apps that have been developed as web applications and thus are cross-platform. The dominance of Android could be attributed to two factors: It has the most number of users worldwide, and it is based on Java as most of

the popular semantic tools.

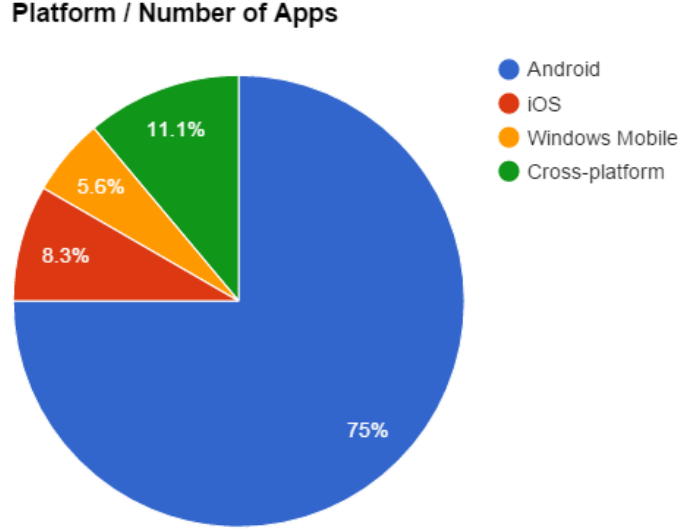


Figure 2.6: Distribution of semantic mobile apps per platform.

Figure 2.7 shows a wordcloud generated with the different semantic technologies that the apps reported using. For management of semantic data on the device, the most common libraries used are: Androjena (in 4 apps), OWL API (in 3 apps), and Sesame API (in 2 apps). Regarding Linked Data endpoints, apps use mainly DBpedia (in 7 apps) and OpenStreetMap/LinkedGeoData (in 6 apps). With regards to semantic reasoning, the following reasoners have been reported: Mini-Me, JFact, and Hermit.

Most of the apps, 23 out of 36, use a client-server approach in which the mobile app itself acted as an interface to present the results returned by the server. These type of client apps were also reported to be majority in [EKA14] (where they were called “thin client apps”). However, 9 of these 23 apps processed Semantic Web languages on the device. 13 apps do not follow the client-server approach and manage semantic data on the device (which can obtain from other devices or directly from Linked Data sources). Also, just 6 apps use a semantic reasoner/matcher on the device to infer facts.

Our results show that most of the “semantic mobile apps” presented act as clients which rely on external servers for the handling of semantic data. This means that although they consume data which comes from Linked Data points and ontologies, in many cases this data is preprocessed on a server which returns the data in a semistructured format (JSON) or just as strings.

Chapter 3

Overview of the System

There exist many scenarios where users need information related to their location which can be obtained from different sources, including other users and their devices. For instance, tourists looking for transportation after arriving in a foreign country, the coordinator of a firefighter team that needs to obtain information from the team, a technical director in charge of the broadcasting of a rowing race who needs pictures and videos of the event, or people involved in a traffic accident who need help. In this section, we present these four examples of such scenarios and explain the challenges that a system to support them would face. Then, we introduce *SHERLOCK* (System for Heterogeneous mobile Requests by Leveraging Ontological and Contextual Knowledge), the system which we designed to support the previous scenarios and many others. We show the high-level architecture of the system and introduce its different modules, which will be explained in the following chapters.

3.1 Motivating Scenarios

In this section, we present four motivating use cases (as examples of many others) that show the heterogeneity and complexity, as well as the interest, of having a flexible and global system as a common framework to provide mobile users with different LBS.

3.1.1 Looking for Transportation

Imagine a person that has just arrived at the airport of a city in a foreign country and wants to get to a certain hotel but does not know the best way to go there. Indeed, in that city there probably exist several transportation

services that could satisfy her demands (Figure 3.1 shows some of the possible transportation options), but in addition to their typical characteristics (e.g., cost), they may also have other specific features (e.g., shareable, door to door, etc.). So, the user needs to ask first tourist offices or websites, or search for a mobile app about transportation in that city; she could be easily overwhelmed with information and many options, and it could be difficult for her to determine which ones are relevant according to her preferences.



Figure 3.1: Different transportation options in our first motivating scenario¹.

So, it would be very interesting for this person to just indicate the name of the destination hotel and her preferences (for example, she could prefer to pay more to reach the hotel as soon as possible) and obtain on her smartphone the real-time location of the best possible transportation means around her. To enable this, the system would have to deal with challenges such as obtaining information about the transportation means (considering geographic information about the city) and keeping it updated, showing the results to the user, etc. The interest of a system like this is beyond doubt: currently, although many transportation services and hotels publish their information on the Web and there exist useful services such as Google Maps, a user traveling to a certain

¹Photos source: https://commons.wikimedia.org/wiki/Main_Page

city will probably have to deal with all the previous applications at the same time to try to arrange her trip.

3.1.2 Helping Firefighting

A wildfire has broken out in a wide area of forest (see Figure 3.2); the designated coordinator person is in charge of managing all the firefighters and emergency vehicles in order to suppress the wildfire. The main task of this team coordinator is to solve the problem as quickly as possible but, at the same time, keeping the team members safe. Due to the lack of a network infrastructure (the fire could have damaged it or there could be no network coverage in that area), firefighter team members usually use walkie-talkies to describe their location and the wildfire evolution. However, it is difficult to provide an accurate oral description of the situation while fighting a wildfire (due to smoke, geographic features of the terrain, and the stressing situation). So, monitoring firefighting units in a dangerous area (and instructing them to reach a safe one), during the suppression of the wildfire, turns out to be a very challenging task for a team coordinator.

Therefore, it could be interesting for a firefighting coordinator to see, on a map displayed on her tablet, the location of all the firefighting units and the evolution of the wildfire in real-time; she would also have to keep a continuous communication with all the team members to get their last smoke and heat sensor readings. Thus, she could be able to notice changes of the wildfire that could put the life of firefighters in danger. The main challenge for a system that deals with this scenario is to monitor moving team members deployed in an environment where it is not possible to rely on a fixed network infrastructure, while detecting automatically firefighting units that could be in danger.

3.1.3 Live Broadcasting of Sport Events

In many scenarios, it is important to select among many cameras the one whose view is the most interesting. For example, in the live broadcasting of sport events, a Technical Director (TD) has to make quick decisions to select the camera whose video stream will be broadcasted. Nowadays, broadcasting organizations are increasing the number of cameras covering sport events (e.g., Sky TV uses 24 cameras in Premier League matches). Furthermore, the audience of the sport event could provide TDs with interesting shots too. The higher the number of cameras available, the richer the content that can be

³Photo source: <https://www.flickr.com/photos/usfwsq/8597688091>



Figure 3.2: A team of firefighters suppressing a fire in our second motivating scenario³.

obtained, but the more complicated is for the TD to select the best one. And they must take this kind of decisions very frequently. For example, consider the TD in charge of the live broadcasting of a rowing race in San Sebastian (Spain). From the TD perspective, the scenario includes multiple cameras available for the broadcasting (in the rowing boats, in a helicopter, in the harbor, and in a nearby island). We should not forget either that many people among the crowded audience of the event are equipped with mobile devices (e.g., smartphones and tablets), in boats around the race and in the promenade (Figure 3.3 shows a picture of the scenario with the locations described before).

In this context, it would be very helpful to have a system where the TD could define her interest on a certain view (e.g., *a view of the front of two rowing boats*) and obtain the list of cameras (including broadcaster and audience cameras) that could provide it, currently or in a matter of seconds. Thus the system would monitor cameras for the TD, automatically looking for predefined shots or even detecting certain predefined situations (overtakings or

³Photo source: <https://www.flickr.com/photos/sansebastian2016/3903530582>



Figure 3.3: The rowing boat race in our third motivating scenario⁴.

other incidents), whose work could be focused on selecting the best camera to broadcast from the different preselected lists of cameras selected by the system. The main challenges in this scenario are: enabling the TD to express the kind of shot she is interested in, processing the views of the multiple cameras in the scenario in real-time to find those that can provide the requested shot, and ranking the results to help the TD to select the most similar shots to her request easily.

3.1.4 Emergency Management

Many well-known real emergency situations have shown that without a good coordination and information flow, the task of emergency teams is very difficult and could even become very dangerous. The bigger the scale of the emergency, the more valuable any information about the real situation; and multimedia information can become priceless in almost 100% of emergencies, especially when such data are obtained in real-time and propagated to emergency teams. In the last decade, the use of mobile information technologies in emergency management has attracted a lot of interest because of the potential value it can provide in this kind of scenarios (e.g., [Cro12; WYZ11]). For example, the use of P2P communications can help when the network infrastructure is damaged or it is just not available; thus, users in the area could provide the very valuable information that emergency teams need to optimize their work.

Let us imagine that an accident has occurred in the highway (see Figure 3.4

⁴Photo source: <http://trainingfirstaid.ca>



Figure 3.4: The traffic accident in our fourth motivating scenario⁵.

for a possible picture of the scenario). First, the people involved in the accident, or even their vehicles if equipped with sensors to detect it, need to alert emergency teams. The emergency team that received the alert needs to know the location of emergency vehicles and staff (e.g., policemen, firemen, etc.) in the surroundings that could help. In this scenario, to obtain real-time multimedia information (images/videos) of the accident area would be very useful for the emergency team to determine the severity of the accident: number and type of vehicles, condition of injured people and how many, urgency due to a fire or presence of dangerous substances, etc. Thus, they could send the most appropriate resources to the accident area. Also, such multimedia information could help emergency units on their way to the accident in order to know what they will face once there. Moreover, we could assume that users in this scenario (for example, drivers in the vicinity of the accident) would be willing to help because lives could be in danger. Notice that in addition to all the challenges commented in the previous scenarios, in this case there is a need to define complex services which require the interaction of devices. Also, the distribution of information to different devices is a challenge of this scenario (e.g., pictures of the accident to the emergency vehicles which could be on their way towards the accident area).

3.1.5 Common Challenges

In the previous use cases, some common needs appear which a system to support them would face. We have grouped these challenges into two categories, challenges related to: 1) the knowledge that such a system must consider; and 2) mobile computing.

Knowledge Management

The following common challenges related to the management of knowledge of the specific scenario arise in the above scenarios:

1. The system must be an expert in the different kinds of elements in the scenario, their features and capabilities, and the geographic information about the scenario. So, it is the system, and not the user, who is in charge of knowing all the details about all the LBS available at each location.
2. The system must handle information about the current context of the user (e.g., her location and activity) and the context of her device (e.g., its current capabilities or the readings of its sensors).
3. A flexible user interface, able to help the user to define the kind of multimedia information (text, images, videos, etc.) she wants, is needed. Indeed, a textual description or a form might not be enough to, for example, define the picture or video to obtain.
4. Integrate results from different sources detecting which results could be more interesting for the user. For instance, which results are more updated or fulfill the user requirements, or are closer to the information that the user requested.

Therefore, the system must know or learn, for example, transportation options in the foreign city of the first motivating scenario or how to help a TD in the broadcasting of the rowing race of the third motivating scenario. The third item is specially challenging for the TV broadcasting in our third motivating scenario. Of course, the coordinator in charge of the traffic accident of the fourth scenario could also be interested in specific images (e.g., showing the front of the cars involved in the accident). However, the level of detail in the definition of such interesting shots will always be higher in the case of a TD, who might have a very specific shot in her mind.

Mobile Computing

The following common challenges related to the managing of the user requests arise in the above scenarios:

1. Finding devices which could provide the information that the user wants and establishing communication with them. This might imply analyzing in real-time the information retrieved and the features of the devices which could capture it. For example, this helps to discard those users/devices that could not be able to provide the requested data and to rank the results obtained according to the matching with the kind of answer wanted.
2. Tracking the location of objects of interest (the transports in the first scenario and emergency teams around the accident area in the fourth) as well as cameras (owned by the broadcaster in the third scenario and from other users in the last two scenarios). Notice that some of the cameras might be attached to moving objects (e.g., boats or people) and be controlled just manually or remotely.
3. Discovering users equipped with cameras and asking them to share recent pictures/videos already stored on their devices. Also, it could be needed to ask users to capture certain pictures or videos. In the third scenario users might need an incentive to do so and in the last scenario they might do it altruistically or forced by authorities (as lives could be in danger). Following privacy and trust policies is mandatory in this case.
4. Maintaining a continuous communication among the user of the system and the rest of the devices involved in the distributed scenario in order to provide the final user with an updated answer continuously.
5. The system must deal with the distributed nature of the environment (which is particularly challenging when it is not possible to rely on fixed infrastructures and ad hoc networks have to be considered) and deal with continuous request processing, scalability and fault tolerance, deployment of computations to specific geographic areas, etc.

In the case of the ambulances in the fourth scenario, the last challenge is especially difficult as the system has to, not only obtain information but also, send it to the emergency units moving to the accident area. This would mean tracking those moving units at any time and establishing a communication to send them multimedia information while on the move.

Therefore, in the rest of this thesis, we propose *SHERLOCK*, a system that is able to address the challenges described above by applying semantic and distributed processing techniques. The system is able to support the previous four motivating scenario as well as any similar scenario where a user is interested in obtaining information about moving objects and performing actions in highly-dynamic distributed scenarios.

3.2 Architecture

SHERLOCK is based on the collaboration of devices to resolve information needs of their users. In SHERLOCK's distributed architecture every device acts as an independent node which communicates with others to exchange information that might be interesting for its user. SHERLOCK-enabled devices use their communication mechanisms (e.g., WiFi and 3G) to create Peer-to-Peer (P2P) networks. The benefits of the communication between SHERLOCK-enabled devices are twofold, it allows devices to: 1) exchange information about services and their surroundings and thus, every devices learns from the different interactions; and 2) answer requests posed by others by using the information they store locally.

To offer the previous functionalities each SHERLOCK node is composed of modules in charge of the following tasks (see Figure 3.5 for the high-level module architecture): Interaction with the user, knowledge management, and user request management. These tasks are handled by different static and mobile agents. In addition to the different agents created to address SHERLOCK goals, our architecture reuses some of the agents presented in [IMI06], to process location-dependent queries, and in [MRM04], to adapt user interfaces, extending them with semantic capabilities and with the ability to take into account the decentralized scenarios considered in SHERLOCK.

For the management of the agents involved in the architecture, SHERLOCK uses a mobile agent platform [TIM07], which provides an abstraction level for the development of distributed agent-based cooperative systems⁶. In SHERLOCK mobile agents communicate to each other directly and the mobile agent platform is in charge of managing the low-level details of this communication. For instance, when there is no direct communication (due to the lack of a fixed network infrastructure, which makes an ad hoc network the only possible option), the mobile agent platform may use an underlying multi-hop ad hoc

⁶In our prototype, we used a version of the mobile agent platform SPRINGS [ITM06], which offers RPC-based synchronous communications to support the cooperation among agents on the same device or on different devices, ported to Android.

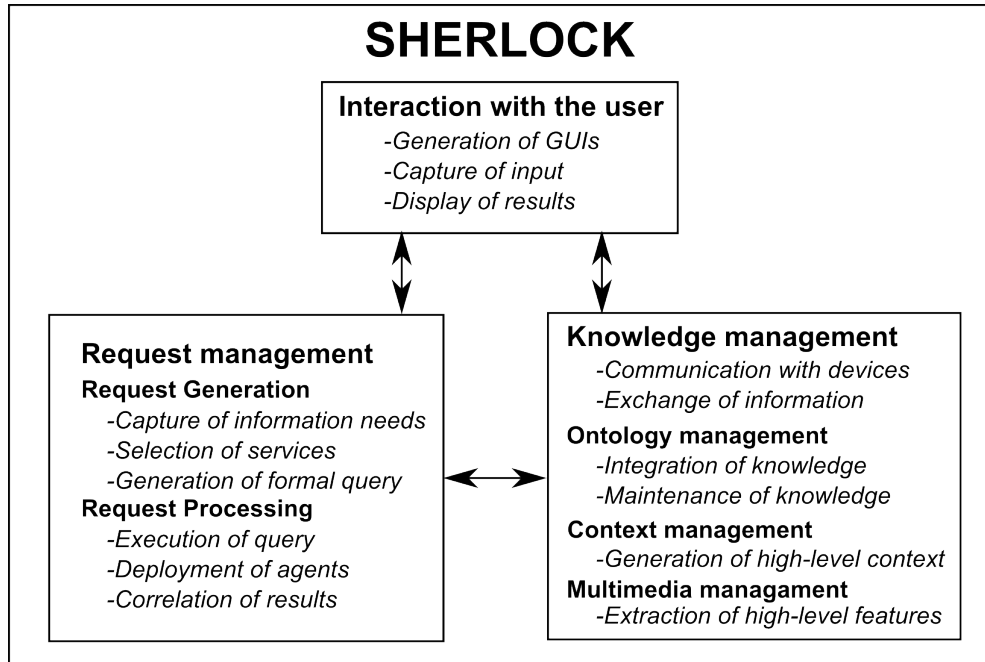


Figure 3.5: High-level architecture of a SHERLOCK node.

routing protocol [BTAABT11] to allow its agents to communicate with each other; for more low-level details about the communication protocol see the referenced paper.

In the following sections, we introduce the main functionalities of the system with the static and mobile agents involved in each task.

3.2.1 Interaction with the User

The module to interact with the user takes care of tasks such as the generation of Graphical User Interfaces (GUIs) to obtain information from her and to show results to her. However, for this module we reuse the work in adaptive interfaces presented in [MRM04] with some minor extensions. The following agents are involved in the interaction with the user:

- *ADUS*, which generates graphical user interfaces (GUIs), adapted to the user profile and device capabilities, by rendering a GUI description provided by incoming agents that want to interact with the user.

- *Alfred*, which specializes on interacting with the user and stores as much information as possible about these interactions.
- *SHERLOCK Endpoint (SE)*, which provides access to SHERLOCK's user request processing capabilities to external applications, advanced users, and other SHERLOCK devices.

3.2.2 Knowledge Management

The knowledge management module of the system handles the knowledge including information about the user and the device as well as services and scenarios. This knowledge is used to: 1) offer the user services and mechanisms to express her information needs, and 2) answer the user information requests. To provide users with interesting information, each SHERLOCK device keeps this knowledge continuously updated through the interaction with other devices. We explain this module in Chapter 4. The following agents are in charge of the knowledge management task:

- *Knowledge Endpoint (KE)*, which is in charge of providing access to the knowledge stored on the device to other agents (see Section 4.2).
- *Context Updater (CU)*, which specializes on knowledge about the user and the context of her device (see Section 5.1).
- *Ontology Updater (OU)*, which shares and integrates new knowledge, obtained from other objects, into the local ontology on the user device (see Section 5.2).
- *Multimedia Manager (MM)*, which analyzes multimedia information (i.e., camera views) to obtain further information from it (see Chapter 8).

3.2.3 Request Management

The request management module of the system handles the user information requests. First, it helps the user in the definition of her request by capturing her information needs. From the information captured, this module deduces the most appropriate service for her and generates a formal query expressing it. Then, it processes the request by executing the formal query against different sources including the local knowledge in the device and third-party external knowledge bases. Finally, it deploys a network of mobile agents to obtain the information requested directly from other devices, if needed. We explain

this module in Chapter 6. The following agents are in charge of the request management task:

- *User Request Manager (URM)*, which helps the user to generate a request that defines her information needs using ontology-guided mechanisms (see Section 6.1).
- *User Request Processor (URP)*, which continuously processes the user request, with the help of Tracker agents, and returns the results to the URM (see Section 6.2).
- *Tracker*, which continuously monitors its assigned relevant area, with the help of Updater agents (see Section 7.2).
- *Updater*, which accesses the data from the target objects inside the relevant area and communicates the information obtained to its Tracker (see Section 7.3).
- *Remote Request Execution (RRE)*, that executes non location-based queries against the local ontologies of devices around the user (see Section 7.1.1).

3.2.4 Overview of a SHERLOCK-Enabled Device

A SHERLOCK-enabled device hosts a series of static agents taking care of the interaction with the user and management of the local repository of knowledge (see Figure 3.6). The latter agents communicate with their corresponding agents on other devices to exchange knowledge autonomously. The static and mobile agents involved in the processing of a user request are created the moment the user interacts with SHERLOCK and needs to express her information needs. In the following chapters, we will detail the components and agents introduced before. First, we will explain the local knowledge on a SHERLOCK device and the static agents in charge of managing this knowledge in Chapter 4. Then, we will detail the static and mobile agents in charge of the management of a user request in Chapter 6, and the processing of location-based queries in Chapter 7. Finally, we will explain the management of multimedia information in Chapter 8.

3.3 Summary of the Chapter

In this chapter, we presented four motivating scenarios where different users are interested in obtaining information from other devices and/or users around:

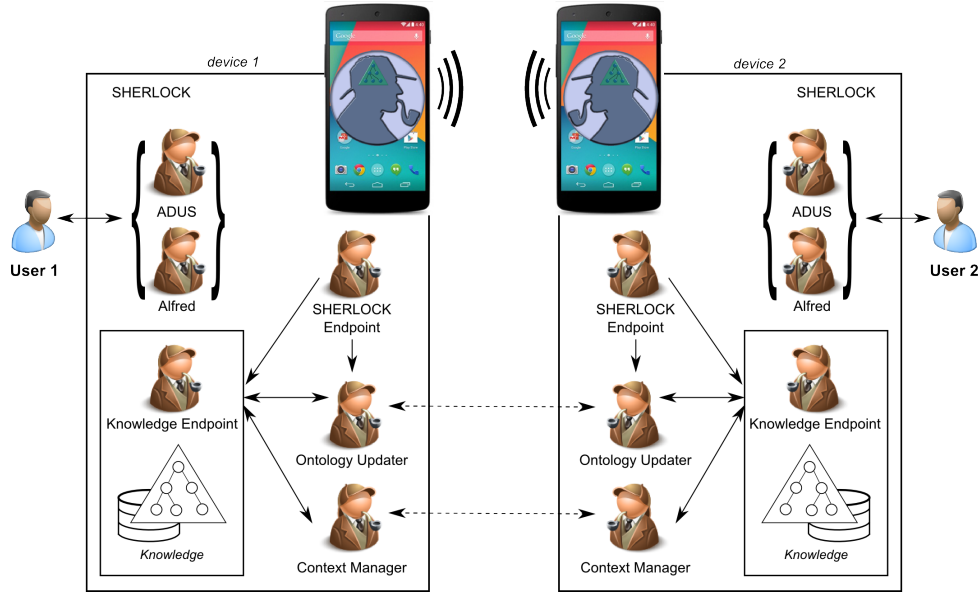


Figure 3.6: Agents to interact with a user and manage knowledge in SHERLOCK devices.

a tourist that needs to find transports to reach her hotel after arriving in a foreign country, a coordinator of a firefighting team suppressing a wild fire who needs information about the team and the fire outbreaks, a technical director in charge of the live broadcasting of a rowing race who needs pictures and videos of the race taken by cameras under her control and even audience members, and the authorities handling a traffic accident who need to coordinate the first response elements and obtain information from the vehicles involved in the accident. With the help of these four scenarios, which work as examples of many others, we highlighted the different challenges a system to handle them will face. Among them we introduced challenges related to: 1) the management of knowledge that such a system must consider (such as the problem of keeping the system updated by incorporating information relevant for new scenarios on the fly), and 2) the processing of user requests (such as finding devices which can provide the system with the information that the user wants and monitoring areas). Finally, we presented the SHERLOCK system which we designed to manage the four motivating scenarios (and any other similar to them) by taking into account the challenges described before. We introduced the multi-agent architecture of SHERLOCK giving an overview of each of its static and mobile agents.

Chapter 4

Knowledge Management

The knowledge that SHERLOCK manages enables it to provide all of its functionalities. By using ontologies to model this knowledge, SHERLOCK is able to integrate new information (and therefore new functionalities) easily and without architectural changes. We divide the knowledge managed by the system in two parts. On the one hand, there is information related to the context of the user and her device. On the other hand, there is information related to the services that SHERLOCK provides. In addition, the system enables accessing to this knowledge through a query language which we designed based on SPARQL. An agent is in charge of processing such queries that can be posed by users or other SHERLOCK agents from other devices. In this chapter, we present the modeling of this information and the mechanisms developed to access it.

4.1 Modeling SHERLOCK Devices and Services

SHERLOCK uses ontologies [Gru95] to model information about the user, her device, the different services she can use, and the scenario around her. These ontologies are represented using OWL¹, de facto standard language to implement expressive ontologies in the Web that makes it possible the definition of complex knowledge. Moreover, as OWL has formal semantics based on Description Logics [BCMNPS03] (DL), it is possible to perform several reasoning tasks to deduce implicit knowledge (i.e., logical consequences of the knowledge in an ontology) using semantic reasoners (i.e., DL reasoners). Indeed, SHERLOCK-enabled devices use a reasoner on the device to manage

¹OWL Web Ontology Language, <http://www.w3.org/TR/owl-primer>

their local ontologies. Having a local reasoner on the device prevents problems associated with relying on external reasoning services such as, for example, the connectivity with the services might not be possible in some situations and privacy issues related to some of the information used in the reasoning which could be sensitive (e.g., the user context). In Appendix A we include our thorough experimental evaluation on the use of semantic reasoners on mobile devices which show the feasibility of this approach. Also, we used these experiments with more than 300 ontologies and 10 popular reasoners, to select HermiT [GHMSW14] as the specific reasoner used in our prototype.

In addition to the OWL models, part of the factual data which are more prone to change dynamically is stored separately, for scalability's sake, using a database manager. For instance, information such as the current capabilities of the device (e.g., current battery available) or the GPS location of the user are highly dynamic whereas information about services is less prone to changes. Therefore, the former is stored in a database whereas the latter is modeled in an ontology.

The knowledge handled by SHERLOCK can be classified into four different categories depending on the subject being modeled (see Figure 4.1): *user context*, *device context*, *services*, and *scenarios*. However, notice that these categories share common knowledge (for instance, services, scenarios, and user context are interlinked as services might be interesting for certain contexts and return elements from the scenario). We group the previous four categories into two modules in SHERLOCK's knowledge base: 1) Contextual knowledge and 2) knowledge about service and scenarios.

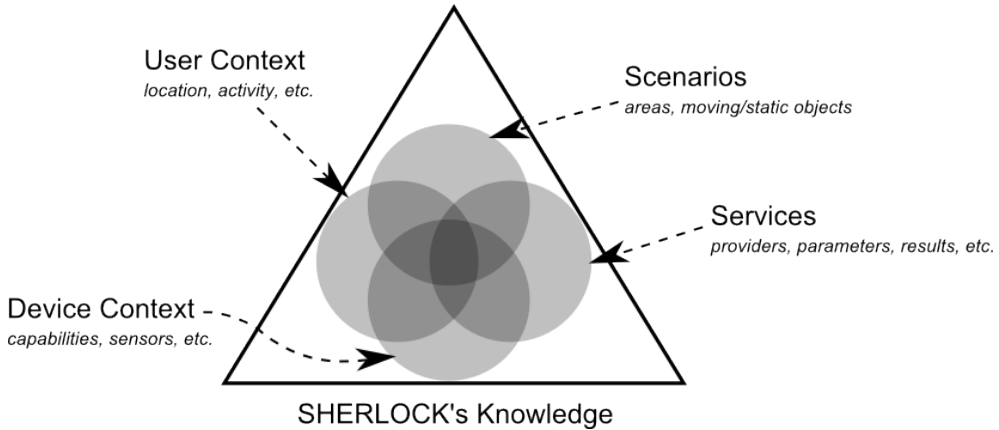


Figure 4.1: Different knowledge managed by a SHERLOCK device.

4.1.1 Contextual knowledge

SHERLOCK manages information about the context of users and their devices to infer different aspects of their status, and use such inferences to: 1) perform a context-aware service provision, and 2) keep their privacy preferences when sharing information with other SHERLOCK users.

The device context (see Table 4.1) includes the features of the device along with a snapshot of its current capabilities. This information is used by SHERLOCK’s mobile agents when processing a user request in order to distribute the workload appropriately.

Device model	The specific device which will be used to obtain information about the features such as processor, battery, memory, wireless interfaces, sensors, etc.
Available battery	An estimation of the remaining battery time. It might be infinite if the device is not running on batteries.
Available processor	Percentage of processor that is available at the moment.
Available memory	Amount of memory that is reserved to the system.
Available storage	Amount of persistent storage that is reserved to the system.
Available bandwidth	The bandwidth available for each wireless interface.
Coverage area	The coverage area that the device can see. It can be static (pre-known) or dynamic (updated in every data refreshment).
Sensor readings	Raw data extracted from the different sensors on the device.

Table 4.1: Information stored about the context of the device.

The user context (see Table 4.2) includes information about the profile of the user and her current context according to the broadly adopted definition by Abowd et al. [ADBDSS99] where context is split into “primary context pieces” (i.e., identity, time, location, and activity) as well as “secondary context pieces” (i.e., pieces of context related to the primary context pieces, e.g., a user’s phone number can be obtained by using the user’s identity). This information is used with a twofold goal: 1) to help the user selecting appropriate services, and 2) to evaluate her privacy preferences when sharing information with other SHERLOCK users.

The hierarchical structure of the information stored about the context of a user makes SHERLOCK able to use different granularities of context pieces depending on the situation. For example, the location of a user in our system can be viewed from her coordinates to the building, the city level, or the region level where the user is in. Ontologies have been widely used before to define and extract context [BBHINRR10]. Therefore, we model context by using an

Object class/es	The object is an instance of a class which shares with other devices (e.g., person, taxi, bus, firefighter, etc.).
Location	The physical position of the object which comprises the GPS coordinates as well as hierarchy of places.
Mobility	Whether the object is a moving object or not. From an ontological point of view, a mobile object with a maximum velocity of 0 is not the same as an object that cannot move at all.
Maximum speed	When dealing with moving objects, the maximum velocity can be used to estimate positions and make the system more robust against communication failures for example.
Direction	The direction of movement of a moving object which can be used, for example, to estimate future positions.
Extent	The area (2D) or volume (3D) that the object occupies physically.
Activity	The activity that the user is performing.
Secondary context	Pieces of information related to the activity and location of the user.

Table 4.2: Information stored about the context of the user.

ontology (see Figure 4.2²) although the dynamic data is stored in a database for efficiency. This way, in the case of location information the GPS coordinates of the user's location, which are highly-dynamic, are stored in a database. Then, this information is combined with the knowledge in the ontology about buildings or cities to infer a higher-level notion of the user location such as the building the user is in.

To explain the information modeled of users and other objects considered by SHERLOCK we will use as examples two elements of our third motivating scenario in Section 3.1.3: a rowing boat (see Figure 4.3), as an example of an object of interest, and a camera (see Figure 4.4), which is an important element of the system as it can capture multimedia information.

Modeling Objects

The location of an object is probably the most important context piece that the system has to manage as it is essential for LBS. For example, users can request information related to the location of an object such as whether the object is inside an area or not. The notion of location in our system includes both the

²The notation employed in this document for images of ontology excerpts follows the Graffoo specification [FGPSV14] (<http://www.essepuntato.it/graffoo>): yellow boxes for classes, green boxes for datatypes, blue arrows for object properties, green arrows for data properties, and black arrows for predicates.

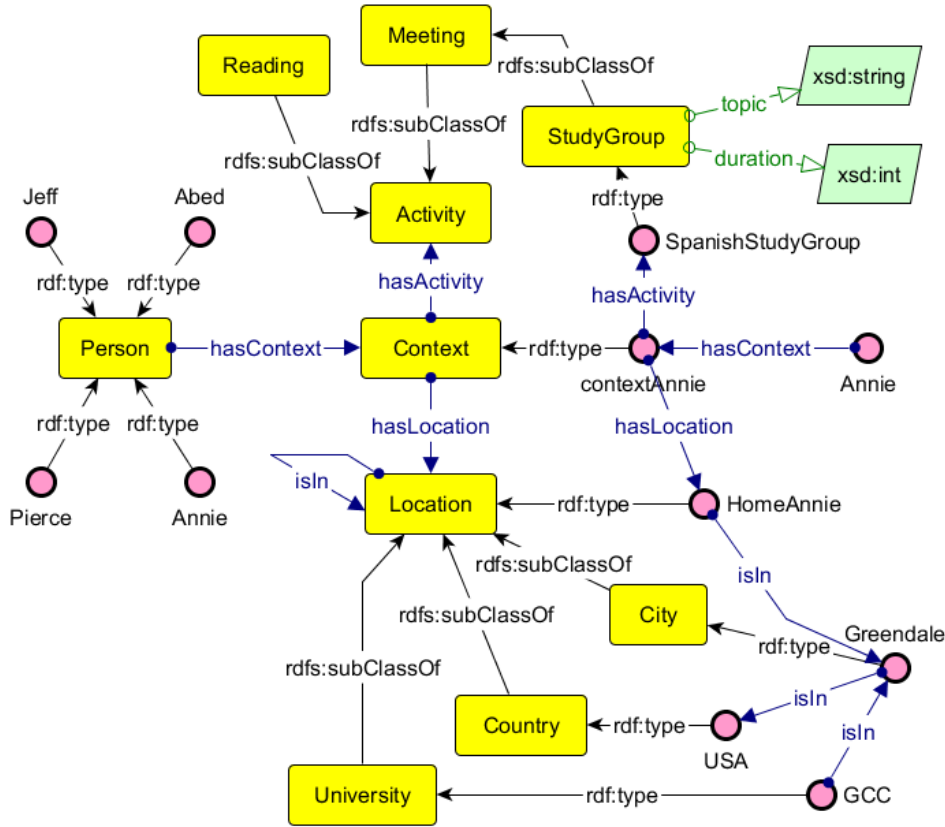


Figure 4.2: An excerpt of the context ontology.

position of an object (i.e., its coordinates) as well as the place where the object is (i.e., its geographic area). This information can be provided, for example, by a GPS and have to be continuously updated to obtain accurate results, as it is highly-dynamic data. The imprecision of the localization mechanism could lead to imprecise answers (e.g., in [SWW06] the authors report an accuracy of around 1 meter for some GPS receivers), but our approach is independent of the specific location mechanism used. So, it is possible to combine, if needed, several positioning mechanisms to increase the accuracy, even for indoor events (by using overhead cameras, sensors, Wi-Fi signal strength maps, etc.).

An special case supported by SHERLOCK is the capability to support requests related to cameras such as whether a camera could take a picture of a certain object or not. The process to achieve this, which we will explain in

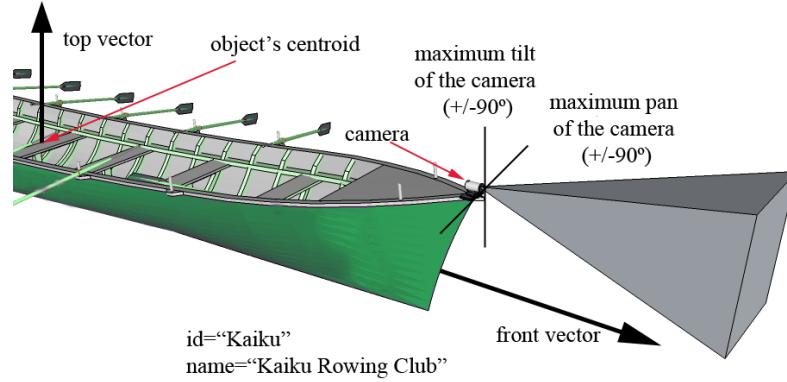


Figure 4.3: An object-of-interest modeled in our system.

Chapter 8, requires of other information of objects such as the direction of movement and the approximate volume (extent) of space that these objects occupy. Our system does not need a precise 3D mesh of these object to accomplish its main goal (as we show in our tests in Section B.4.3, where we used an approximate extent for the different types of objects-of-interest). Of course, the more precise the extent of an object-of-interest provided to the system, the more accurate the information it will obtain regarding the percentage of the object viewed by a camera. Thus, users could generate a simple 3D model for these objects or even search for similar already-generated meshes in 3D model databases (e.g., by using keywords or even real images, as studied in [ADV07; GWZTDZ11]). As the extent of objects-of-interest could change dynamically it could be interesting to request this information from other devices. However, in real life only small parts of these extents change (e.g., the rows of a rowing boat, the limbs of a person, etc.). Thus, the general accuracy of our approach to compute what are cameras viewing will not be affected significantly if non-deformable extents for objects-of-interest are used (in our tests we used fixed 3D models).

Besides, the front and top vectors, which are used to represent where the front and top parts of the extent are, must be defined for this extent (e.g., in the rowing boat of Figure 4.3 the front and top vectors are $[1, 0, 0]$ and $[0, 0, 1]$ respectively). This way, the system is able to support requests retrieving cameras that can view specific parts of the object, for example, cameras recording the front of the rowing boat. The views supported by these vectors are: top/bottom, front/rear, left/right, or any combination of two or three elements chosen from the three previous pairs. In our system, 90-degree

angles are considered between the front and top vectors, the sides and the front, and the sides and the top. Thus, no more than three viewpoints are going to be usually selected at the same time in a query for the same object.

Modeling Cameras

Concerning cameras, which are special types of sensors attached to devices and which play a key role in SHERLOCK as they are providers of multimedia content (see Chapter 8), we consider that they can rotate (both vertically and horizontally) and change their location (if they are attached to moving objects). We model a camera c as shown in Figure 4.4. In the figure, we identify several elements: β_h and β_v are the horizontal and vertical angle of view (that define the *Field of View* –*FOV*– of the camera), respectively; α , α_{max} , α_{min} , and α_{speed} are the current pan (i.e., the current horizontal rotation of the camera), the maximum pan possible, the minimum pan possible, and the pan speed (degrees/second) of such a camera, respectively; finally, θ , θ_{max} , θ_{min} , and θ_{speed} are the current tilt (i.e., the current vertical rotation of the camera), the maximum tilt possible, the minimum tilt possible, and the tilt speed (degrees/second), respectively³. Angles that represent a pan to the right (α) or a tilt upward (θ) from the corresponding vector are considered positive and those that represent a pan to the left (α) or a tilt downward (θ) are considered negative. Besides, each camera has the rest of features of a regular object (such as a unique identifier, location, etc.).

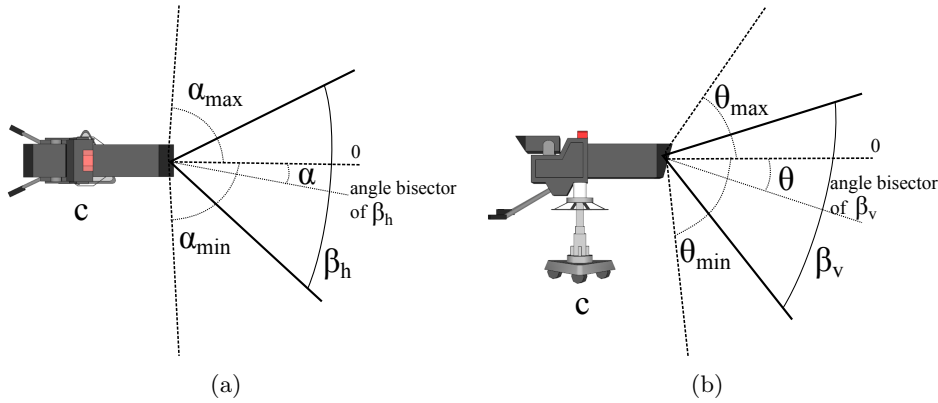


Figure 4.4: Modeling a camera: pan (horizontal plane) (a), and tilt (vertical plane) (b).

³In this work we do not deal with the possibility of zooming.

4.1.2 Service and Scenario Knowledge

To be aware of the available functionalities, SHERLOCK handles a structure of services which are defined extending a pre-shared ontology (see Figure 4.5). This module comprises knowledge about both the services themselves (e.g., which parameters it receives, how it is invoked, the type of the result -if any-), and the different entities needed to completely define their semantics (e.g., if we define a service to look for means of transport, in the ontology, the entity *Transport* should appear), which we refer to as “scenario”.

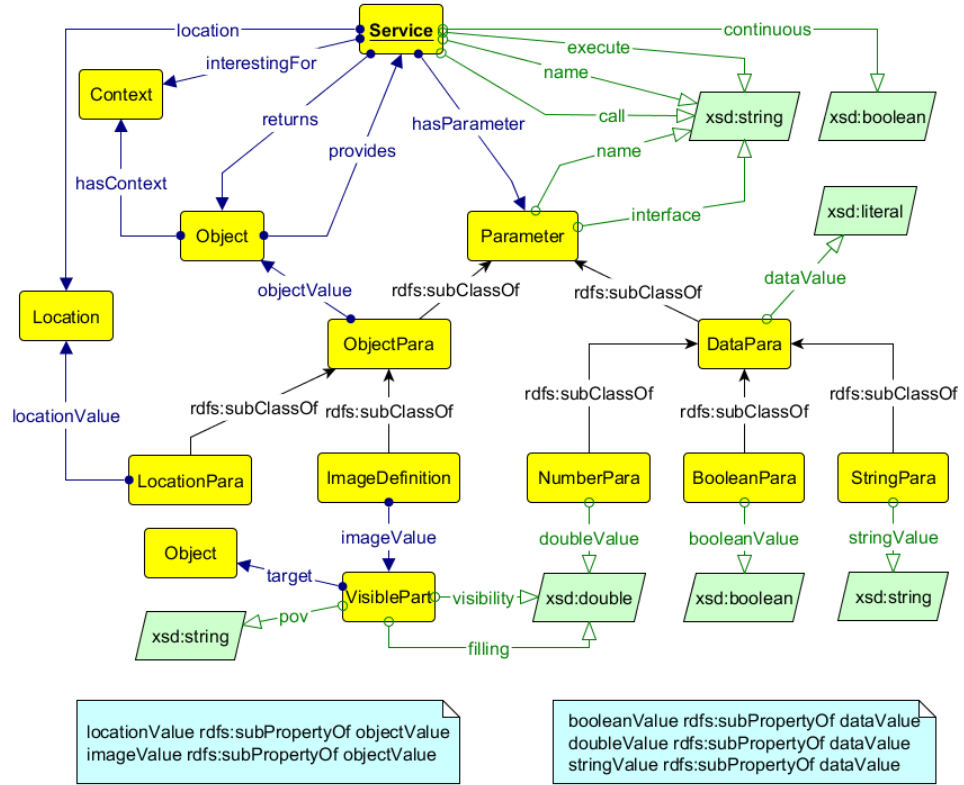


Figure 4.5: Basic ontology for service modeling in SHERLOCK’s services.

In our model, services are instances of the class *Service* in the ontology. Specializing this class *Service* in the ontology, the service definers (users and/or companies which want to extend SHERLOCK) can arrange services into *families of services* which share functionality (e.g., a concept *Find Buses* would be a family of services). In fact, our approach requires that service

definers select an existing family for their newly added services (or define a new one, extending the vocabulary)⁴.

Apart from their ontological classification, services defined in SHERLOCK can further be classified attending to the way they are processed into three types of services:

1. *SHERLOCK querying services*: Services which provide functionality to find objects and are similar to traditional *location-dependent queries*. However, SHERLOCK also handles queries which are not related to any location. The system itself provides contributors with a functionality to find objects in its local ontology and even deploys a network of agents to obtain them from other devices. To define these services, contributors can model the information that the service will obtain as a result (e.g., a service to find pictures will return pictures), and its parameters (if any). As an example, Figure 4.6 shows the definition of a service of this type to obtain transports.

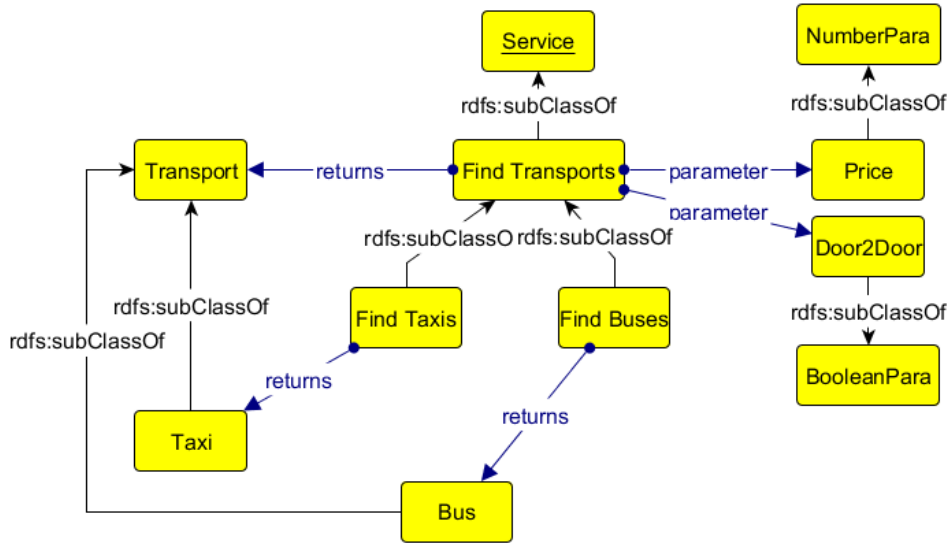


Figure 4.6: Example of the definition of a SHERLOCK querying service to obtain transports.

⁴SHERLOCK does not aim at matching directly services as in classical Semantic Web Services approaches. Our approach relies on integration of the shared schemas to discover new instances with similar functionality.

2. *External services*: Services provided by a particular provider object. This kind of services includes both third party external services and services offered by other SHERLOCK-enabled devices (which might involve notifying or interacting with another user). The definition of this kind of services is extended with information about: 1) their provider (via *provider* property), 2) the access mechanism to be used (via *call* property), and 3) for those which require user's interaction, an specification of the interface to be used⁵.

For example, Figure 4.7 shows a service to take pictures that is provided by SHERLOCK objects equipped with cameras and the service of the Metropolitan Transportation Authority of New York which returns the location of buses through a web service. Another interesting external service that can be defined is, for example, a service to query an external knowledge base, such as DBpedia, to obtain the location of POIs in an area (the *call* property would include the SPARQL query and the URL of the DBpedia endpoint).

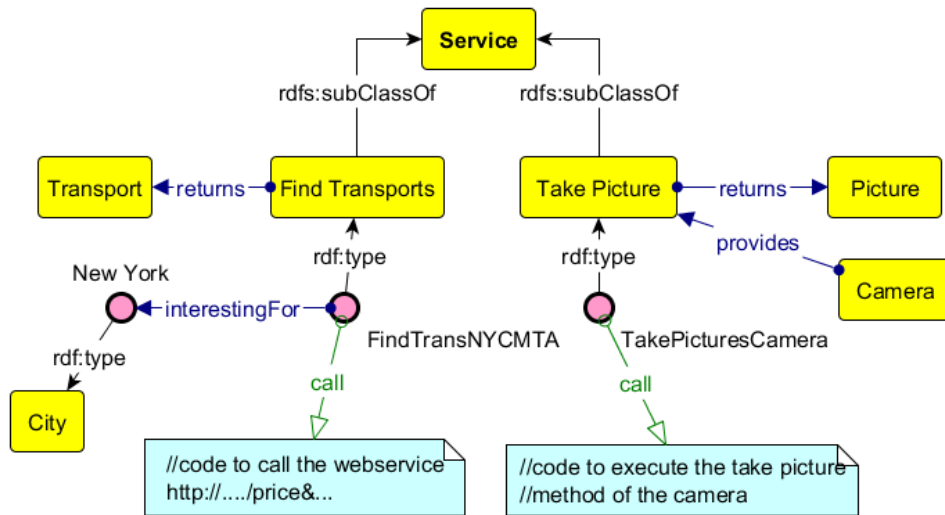


Figure 4.7: Two examples of the definition of external services: 1) to take pictures and 2) to obtain transports from a web service.

3. *Complex services*: Services defined via a composition of services of the

⁵As we will see, this specification will be used by the ADUS agent to create an user interface and interact with the user in another SHERLOCK-enabled device.

two previous types (i.e., these services use them as atomic actions). This composition is defined by a workflow specified in BPMN [Whi04], which is included in the service definition in XPDL format⁶ (taking into account that the “Activities” considered in BPMN are indeed “Services” in our architecture) using the *execute* property. Currently, we restrict our approach to support a subset of BPMN enough to model services executing atomic services in parallel, sequence, and combinations of the previous⁷.

For example, Figure 4.8 shows a service for the live broadcasting of a rowing race which needs to find cameras and rowing boats (two SHERLOCK query services) and then ask the cameras to take pictures of the boats (an external service provided by SHERLOCK objects with cameras). Notice that the specific services have to be defined as explained previously, and the composed service includes a reference to them in the XPDL code attached.

Any service of these three kinds will be translated into *user requests* when a user selects them and processed later, as we will explain in Chapter 6. In particular, a SHERLOCK querying service will be translated into a query in SHERLOCK’s language, an external service into an *external call*, and a complex service into a *workflow* composed of SHERLOCK queries and/or calls. Moreover, the way that atomic requests will be finally processed is also defined by the *continuous* property of their associated service, which defines whether they have to be continuously evaluated. For example, a service to ask a camera to take a picture might have to be evaluated once only, whereas a service to obtain taxis near the user is expected to be continuously reevaluated to obtain updated results until the user is not interested in it anymore.

Finally, to know which services are relevant to a particular user’s situation, these services are linked to the user context through a special property (*interestingFor*). This property has to be populated by the user or company who is modeling the service, deciding whether that particular service will be interesting for a certain context or not. In this case, the context for which the service is interesting has to be defined using its attached activity and/or location. For example, the service to find monuments in New York could be defined as interesting for contexts whose activity is “tourism” and whose location is “New York”.

⁶<http://www.xpdl.org>

⁷Note that our approach only supports the usage of simple atomic services within the workflows. We do not aim at supporting any kind of service composition (e.g., workflows within workflows)

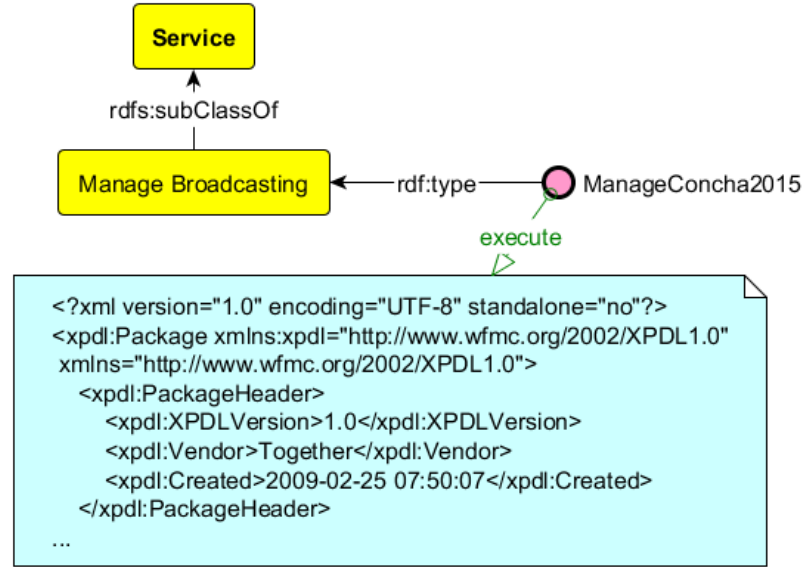


Figure 4.8: Example of the definition of a complex service with a workflow to find cameras in an area and request them to take a picture.

In the following, we detail present the agent in charge of providing access to the knowledge. The agents in charge of updating the knowledge and the local ontology will be presented in Chapter 5.

4.2 Providing Access: Knowledge Endpoint Agent

The Knowledge Endpoint agent (KE from now on) processes queries against the local ontology on the device posed by other agents. The handled information includes highly-volatile data (e.g., the specific location of the user and the surrounding objects, sensor readings of the device, or even instances of current providers of each service), as well as more static information (e.g., the model of the device and its features, or definitions of services). To handle the volatile part of the knowledge, the KE agent adopts the strategy presented in [BBIM14], where static and volatile knowledge is stored in ontologies and databases, respectively, and is detected and marked at modeling time, allowing continuous DL query processing with enough expressiveness.

Local agents on the device and external agents from other users pose queries, using SHERLOCK's query language, to the KE agent. The KE agent is able

to interpret this language and obtains results taking into account the privacy preferences of the user regarding the information to be shared with others. In the following, we explain the grammar of SHERLOCK's query language and the management of the privacy preferences of the user.

4.2.1 SHERLOCK's Query Language

The design of our system has been guided by criteria of maximizing both expressivity and flexibility of the system. Thus, in order not to be constrained to a predefined set of scenarios, with hardcoded queries and user needs, we made SHERLOCK capable of processing its own query language.

We designed this query language taking as basis our previous experience in the field of location-based queries. In particular, we took as baseline the expressivity of the SQL-like query language proposed in [IBM11], which allows to write location-based queries using different location granularities. As our approach manages knowledge modeled using ontologies, we based SHERLOCK's language on SPARQL [PS+08], a standard query language able to handle RDF data. To integrate both aspects (locations and DL semantics) in the same query language, we adopted (and adapted) the GeoSPARQL [BK11] and SPARQL-DL [SP07] extensions of SPARQL:

- GeoSPARQL [BK11] is a standard for representation and querying of geospatial linked data from the Open Geospatial Consortium (OGC).
- SPARQL-DL [SP07] is a subset of SPARQL extended with predicates that are fully aligned with OWL 2, and which covers the typical functions associated with OWL.

Also, we included some functions to handle information provided by cameras as SHERLOCK queries can be used to obtain multimedia information too. The adoption of this query language in our system has the following benefits:

- It provides part of the expressivity of SQL and complements it taking into account semantic and geographic technologies.
- As mentioned before, it decouples the system from a specific scenario, increasing its flexibility.
- As it is based on SPARQL, the system is able to query popular Linked Open Data SPARQL endpoints⁸ and integrate the results (in RDF format)

⁸There is a huge amount of information in these repositories, e.g., DBpedia [BLKABCH09] contains a great part of Wikipedia's information semantically annotated.

easily.

- As we have introduced in Section 3.2, it makes it possible to use each device in our system as a distributed knowledge endpoint via the exported query service. Moreover, such a service is also available to external applications which could use SHERLOCK to exploit its capabilities.

SHERLOCK's Language Grammar

The simplified grammar of SHERLOCK's language is shown in Figure 4.9⁹. We can distinguish two main parts in a SHERLOCK query:

- The *projections* clause, which declares the free variables that are used to match the result of the query. Note that as we do not have any attached schema (as, for example, in SQL), the meanings of these defined variables are not yet specified.
- The list of *where* clauses, which defines both the location constraints and conditions that the required objects have to met (*LICons* and *ObjectCons*, respectively), and the bindings of the previously declared variables to properties of such objects (*ProjectionsCons*).

Constraints in Language. The location constraints defined within the *LICons* fragment of a *where* clause impose conditions on the locations of the returned objects, defining the *relevant area* of the query. We explicitly separate the definition of location constraints from object ones to clearly distinguish from spatial patterns that are to be interpreted continuously (e.g., retrieve objects that *are* within Zaragoza) from spatial patterns that refer to static properties of the objects (e.g., retrieve people that were *born* within Zaragoza). Moreover, note that location constraints are not mandatory, allowing for both location and non-location based queries.

The object constraints within the *ObjectCons* fragment of a *where* clause define semantically the objects that are to be returned. The patterns in this fragment can appear modified by an OPTIONAL clause which makes them not mandatory, and/or grouped with the help of a CASE operator. This latter operator allows for grouping object definitions by expressing the shared properties and separating the particular patterns into different CASES. Formally, let be *OD* the set of patterns within a *ObjectCons* fragment, *S_{OD}*

⁹The complete grammar can be consulted at <http://sid.cps.unizar.es/SHERLOCK>

the subset of patterns in OD which are not within a CASE clause, and S_{CASE} the set of sets of patterns within CASE clauses in OD , then:

$$x \text{ satisfies}(OD) \Leftrightarrow x \text{ satisfies}(S_{OD}) \wedge \exists p \in S_{CASE}, x \text{ satisfies}(p)$$

This is, all the mandatory patterns (i.e., those that are not modified by an OPTIONAL operator) are so except for those which are inside a CASE function, which are added to the body of the definition following a logical OR semantics. For example, with the CASE clause it is possible to define objects of interest which are *available vehicles*, and specifically *Taxis* of a particular operator, or *Buses* in general in the same query:

```
OD{
  Type(?thing, sherlock:Vehicle),
  PropertyValue(?thing, available, <true>),
  CASE(Type(?thing, sherlock:Taxi),
    PropertyValue (?thing, operator, <TaxiCab Co.>),
    CASE(Type(?thing, sherlock:Bus))
  }
```

Finally, the projection constraints bind the free variables that are to be returned in the variable with the properties of the objects defined. These constraints are evaluated against the set of objects that satisfies both location and object constraints previously defined.

Predicates in the Patterns. Regarding the predicates that can be used to form the patterns, we can distinguish two main groups:

- Geographical predicates (*GeoFilter* in the grammar).
- DL-related predicates (*DLFunction* in the grammar).

Geographical predicates supported by the language are taken from the extension of SPARQL to handle geospatial information, GeoSPARQL [BK11]. We have adopted three different functions which we needed to express inside constraints. In particular, we reuse `geof:intersects` and `geof:within` tests, which test intersection and inclusion relationships between spatial elements, and `geof:buffer` operation, which performs the dilation of a spatial element by a given distance. We focused on predicates that allowed us to define inside constraints as other types of location-dependent constraints (e.g., nearest) can be expressed by using inside constraints (for more details, see [IMI06]).

DL-related predicates are mainly taken from SPARQL-DL [SP07]. We have adopted all the SPARQL-DL predicates (as shown in Table 2.5), keeping the

same semantics as in their original definitions¹⁰. Besides, we have included two DL functions to obtain the domain and range of given property. These functions in the DL extension are not part of SPARQL-DL, but are useful and used by SHERLOCK agents. They have as parameters a property and a class, allowing at most one free variable, and their exact semantics depend on the position such free variable (see Table 4.3).

Operator	Interpretation
Domain(C, R)	$true \Leftrightarrow O \models \exists R. \top \sqsubseteq C$
Domain(?C, R)	$\{x \in concepts(O) Domain(x, R)\}$
Domain(C, ?R)	$\{y \in roles(O) Domain(C, y)\}$
Range(C, R)	$true \Leftrightarrow O \models \top \sqsubseteq \forall R. C$
Range(?C, R)	$\{x \in concepts(O) Range(x, R)\}$
Range(C, ?R)	$\{y \in roles(O) Range(C, y)\}$

Table 4.3: Semantics of the introduced DL operators: Domain and Range.

Camera View predicates. SHERLOCK supports queries related to the views provided by cameras (as we will explain in Chapter 8). In order to enable access to all the features of our proposal to compute camera views, we define the following functions (*CameraFuntions* in the grammar):

- *checkKindOfView(target, cam, view)* returns a *true* value if the camera *cam* is obtaining the view *view* of the target object *target*. Another variant of this function is *checkKindOfView(target, cam, view, t)*, that performs the same operation but taking into account the estimated view that the camera will obtain after *t* seconds.
- *percentageShot(target, view, cam)* returns the percentage of the shot of the camera *cam* occupied by the target object *target*; *percentageShot(target, view, cam, t)* performs the same operation for the estimated view that the camera will obtain after *t* seconds. Notice that if the user selects a specific viewpoint for the *view* parameter, which is indeed optional, these functions will obtain the amount of the shot occupied by the viewpoint of the target object.
- *percentageObject(target, view, cam)* returns the percentage of the target object *target* that the camera *cam* is viewing; *percentageObject(target, view, cam, t)*, performs the same operation for the estimated view that

¹⁰The interested reader is referred to [SP07] for their detailed definitions.

the camera will obtain after t seconds. If the user selects a specific viewpoint for the *view* parameter, which is optional, these functions will obtain the percentage of the viewpoint covered.

- *preferenceDegree(target, cam, α)* returns a numeric value that allows the system to rank cameras that fulfill the user requirements according to how well their views fit his/her preferences. The user sets α , which represents the importance of the percentage of the shot occupied by the target with respect to the percentage of the object viewed (which will have a weight of $1 - \alpha$). In our prototype we advocate computing the preference degree as follows, in order to represent that the higher the percentage the better, but any other function could be used:

$$\text{percentage_of_shot_occupied} * \alpha + \text{percentage_of_target_viewed} * (1 - \alpha)$$

- *rotationToView(target, cam)* returns a “Rotation” instance composed of the pan and tilt angles that the camera *cam* has to turn to view the target object *target*. This function makes use of *panToView(target, cam)* and *tiltToView(target, cam)*, that obtain the pan and tilt needed to view the target, respectively.
- *timeToView(cam, <pan, tilt>)* returns the time needed by a camera to turn horizontally *pan* degrees and vertically *tilt* degrees. This function makes use of *timeToPan(cam, pan)* and *timeToTilt(cam, tilt)*, that obtain the time needed to pan and tilt a certain angle, respectively.
- *distance(target, cam)* returns the distance between a camera *cam* and a target object *target*.

Examples of SHERLOCK queries

For an example of the use of the language, we show in the following a SHERLOCK query to obtain entities which are *Statues*, or *Monuments* which have free admission, in Zaragoza and in the museums inside Madrid:

```
PREFIX sherlock: <http://sid.cps.unizar.es/ontology/sherlock/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX geof: <http://www.opengis.net/def/geosparql/function/>

SELECT ?name, ?lat, ?lon, ?desc
WHERE{
  LI{ // First location of interest: Zaragoza
    FILTER(geof:sfWithin(?thing, sherlock:Zaragoza)
```

```

    },
    OD{ // Definition of the object of interest
        // All the objects have to be opened
        PropertyValue(?thing, sherlock:parameter, ?paramOpen),
        PropertyValue(?thing, sherlock:name, 'open'),
        PropertyValue(?thing, sherlock:value, true),
        // Statues
        CASE(Type(?thing, sherlock:Statue)),
        // Or monuments with free admission
        CASE(Type(?thing, sherlock:Monument),
            PropertyValue(?thing, sherlock:parameter, ?param),
            PropertyValue(?param, sherlock:name, 'price'),
            PropertyValue(?param, sherlock:value, 'free')),
    },
    PropertyValue(?thing, sherlock:name, ?name),
    PropertyValue(?thing, sherlock:latitude, ?lat),
    PropertyValue(?thing, sherlock:longitude, ?lon),
    // Get description if available
    OPTIONAL(PropertyValue(?thing, sherlock:description, ?desc))
} OR WHERE{
    LI{ // Second location of interest: museums in Madrid
        Type(?place, sherlock:Museum),
        FILTER(geof:sfIntersects(?place, sherlock:Madrid)),
        FILTER(geof:sfWithin(?thing, ?place))
    },
    OD{
        PropertyValue(?thing, sherlock:parameter, ?paramOpen),
        PropertyValue(?thing, sherlock:name, 'open'),
        PropertyValue(?thing, sherlock:value, true),
        CASE(Type(?thing, sherlock:Statue)),
        CASE(Type(?thing, sherlock:Monument),
            PropertyValue(?thing, sherlock:parameter, ?param),
            PropertyValue(?param, sherlock:name, 'price'),
            PropertyValue(?param, sherlock:value, 'free'))
    },
    PropertyValue(?thing, sherlock:name, ?name),
    PropertyValue(?thing, sherlock:latitude, ?lat),
    PropertyValue(?thing, sherlock:longitude, ?lon),
    OPTIONAL(PropertyValue(?thing, sherlock:description, ?desc))
}

```

Notice that the query contains two interesting areas (Zaragoza and museums inside Madrid) and therefore contains two `WHERE` clauses. Also, notice that the definition of the second area of interest involves the use of two GeoSPARQL `FILTER` functions to select places which are museums inside Madrid. Finally, as the objects of interest that the system has to find are the same for the two areas, their definition in the `OD` clause is the same in both `WHERE` clauses. Also, in the `OD` definition, we show the use of several SPARQL-DL functions to define the class of the object and the value for different properties it has to fulfill.

To show the use of our custom functions about cameras in SHERLOCK's query language, we first consider an example where the Technical Director (TD) of our third motivating scenario asks the system about *cameras that view right now at least 30% of the "Kaiku" boat that fills at least 10% of the shot*. The request can be translated to the following query (we assume that for the TD the percentage viewed is more important than the percentage of the shot occupied and considers $\alpha = 0.4$):

```
SELECT ?id, ?score
WHERE{
  LI{
    FILTER(geof:sfWithin(?thing, sherlock:SanSebastianBay)
  },
  OD{
    Type(?thing, sherlock:Camera)),
    ?rotation=rotationToView(sherlock:Kaiku, ?thing),
    PropertyValue(?rotation, sherlock:pan, ?pan),
    PropertyValue(?rotation, sherlock:tilt, ?tilt),
    FILTER(?pan = 0),
    FILTER(?tilt = 0),
    FILTER(percentageObject(sherlock:Kaiku, 'any', ?thing) > 0.3),
    FILTER(percentageShot(sherlock:Kaiku, 'any', ?thing) > 0.1),
    ?score=preferenceDegree(sherlock:Kaiku, ?thing, 0.4),
  },
  PropertyValue(?thing, sherlock:id, ?id)
ORDER BY DESC(?score)
```

As the TD wants to obtain the cameras viewing the target object right now, the query includes a condition ($pan=0$ and $tilt=0$) that ensures that the cameras in the answer set fulfill this constraint. Besides, the function *preferenceDegree* is used to take into account the TD preferences in the ranking.

Now, considering that the TD asks about *cameras that can view the front, top and side of "Kaiku" in less than 20 seconds, sorted by the percentage of the target viewed and the time needed to view it* (the largest the percentage and the shorter the time, the more appropriate a camera is), the system generates the following query:

```
SELECT ?id, ?pct, ?time, ?pan, ?tilt
WHERE{
  LI{
    FILTER(geof:sfWithin(?thing, sherlock:SanSebastianBay)
  },
  OD{
    Type(?thing, sherlock:Camera)),
    ?rotation=rotationToView(sherlock:Kaiku, ?thing),
    PropertyValue(?rotation, sherlock:pan, ?pan),
    PropertyValue(?rotation, sherlock:tilt, ?tilt),
    ?time=timeToView(?thing, ?pan, ?tilt),
    FILTER(?time > 20),
```

```

    FILTER(checkKindOfView(sherlock:Kaiku, ?thing, 'front', ?time) = true),
    FILTER(checkKindOfView(sherlock:Kaiku, ?thing, 'top', ?time) = true),
    FILTER(checkKindOfView(sherlock:Kaiku, ?thing, 'side', ?time) = true),
    ?pct=percentageObject(sherlock:Kaiku, 'any', ?thing, ?time),
  },
  PropertyValue(?thing, sherlock:id, ?id)
ORDER BY DESC(?score) pct

```

Now let's show the information obtained by these functions applied to the camera views shown in Figure 4.10. First, the following high-level features are extracted by the system from a camera view:

- The specific objects viewed (e.g., in Figure 4.10(a), *CAM1* views the rowing boats “Kaiku” –in green– and “Urdaibai” –in red–) and some information about them:
 - The distance to the object (e.g., in Figure 4.10(a), the distance between *CAM1* and the boat “Urdaibai” is 17 meters).
 - The percentage of the object covered (e.g., in Figure 4.10(b), *CAM2* views 22% of the boat “Kaiku”).
 - The percentage of the shot occupied by the object (e.g., in Figure 4.10(b), “Kaiku” fills 26% of *CAM2* view).
 - The kind of view obtained of the object (e.g., in Figure 4.10(b), *CAM2* views the front and left side of “Kaiku”).
 - The percentage of the viewpoint of the object covered (e.g., in Figure 4.10(b), *CAM2* views 47% of the front and 29% of the left side of “Kaiku”).
- The percentage of the shot occupied by objects-of-interest (e.g., in Figure 4.10(a), both rowing boats fill 6% of the view provided by *CAM1*).

Therefore, *CAM1* and *CAM2* will be returned by the KE agent for the first sample query whereas *CAM1*, *CAM2*, and *CAM3* will be returned for the second (as *CAM1* and *CAM2* are already obtaining the required view, and *CAM3* will be able to do it in less than 20 seconds).

4.2.2 Context-Aware Privacy Policies

The KE agent executes queries posed by other agents against the local knowledge on the device. These agents might act on behalf of other users (e.g., Updater agents – which we will explain in Section 7.3 – execute queries of a

user against other devices). In this scenario, users might have some preferences regarding which data can be shared with whom. Semantic Web technologies have been used in the literature to represent and enforce these user privacy preferences [KFJ03], also called privacy policies. In [KFJ03] the authors propose a semantic policy language to represent the user privacy policies, and a policy engine that interprets and reasons over such language. The KE agent uses this approach to preserve the privacy of its user when processing queries from others. The policy language is based on the Semantic Web Rule Language (SWRL) [HPSBTGD04] whereas the policy engine in our approach is the KE which uses the semantic reasoner to evaluate the rules when information is requested. In addition, incorporating context to these policies allows a higher degree of granularity and control in the application of the privacy preferences of the user. For instance, the following context pieces are considered in the privacy policies handled by the KE agent:

- *Location*: In our scenario we treat location context semantically as in “at the bar”, “at the University campus” or “inside home”. For example, if a student has specified that she does not want to share her location when she is in University buildings, it is assumed that she does not want to share her location at the University library.
- *Activity*: It can be used to add another dimension to the meaning of location. For example, a classroom used for private meetings versus public lectures. An example of policy which would be activity dependent is a user do not want certain information from her local knowledge base to be shared when in a work meeting.
- *Identity*: The identity of certain users or groups can be used to model simple access control. For example, to specify what information on the device can be shared with strangers.
- *Time*: While time is ingrained into other aspects of context, it allows for an all-embracing notion of privacy without worrying about the location and activity of users.

Most user situations demand a combination of various types of context pieces described above. A possible example of such a scenario would be “do not allow my social network colleagues group (identity context) to obtain my (identity context) location when I’m in a party (activity context) held on a weekend (time context) at the beach house (location context)”. As proposed in [KFJ03], these rules should be encoded in SWRL and based on the

context ontology (see Figure 4.2)¹¹. For example, the previous policy would be expressed in SWRL as follows:

```

Person(?p) ∧ Colleague(?p) ∧ Context(?c) ∧
hasTime(?c,?t) ∧ hasDay(?t,?day) ∧ WeekendDay(?day) ∧
hasLocation(?c,?loc) ∧ BeachHouse(?loc) ∧
hasActivity(?c,?act) ∧ Party(?act) ∧ Location(?loc)
→ DenyAccessTo(?loc,?p)

```

Thus, whenever an agent representing another user poses a query requesting the location of the user, the KE agent evaluates the previous rule (with the help of the reasoner) to determine if the information can be shared.

4.3 Summary of the Chapter

In this chapter, we have presented the management of knowledge by SHERLOCK. First, we explained the modeling of the different categories of knowledge managed by the system: user context, device context, services, and scenarios. Then, we introduced the Knowledge Endpoint agent in charge of providing access to the local knowledge on the device to other (local or external) agents. We explained the SPARQL-like query language that we defined and which can be interpreted by the KE agent. This language, which decouples the system from specific scenarios, is based on GeoSPARQL and SPARQL-DL to handle geospatial and DL semantics, respectively. Also, we introduced the handling of user privacy policies when external agents, belonging to other devices/users, request information to the KE agent.

¹¹It is out of the scope of this work to deal with the definition of rules by users.

General query structure

Query	→	<i>SELECT</i> Projections Where
Projections	→	Var (',' Var)*
Where	→	<i>WHERE</i> '{' Conds '}' Where <i>OR WHERE</i> '{' Conds '}'
Conds	→	LICons? ObjectCons ProjectionsCons?

Location of Interest Constraints

LICons	→	<i>LI</i> '{' Patterns '}'
--------	---	----------------------------

Object Constraints

ObjectCons	→	<i>OD</i> '{' ObjectDefs '}'
ObjectDefs	→	TypeDef (',' TypeDef)*
TypeDef	→	OptionalDef ObjectDef
OptionalDef	→	<i>OPTIONAL</i> '(' ObjectDef ')'
ObjectDef	→	Patterns <i>CASE</i> '(' Patterns ')'

Projection Constraints

ProjectionsCons	→	ProjCons (',' ProjCons)*
ProjCons	→	<i>Property Value</i> (VarOrIRI , VarOrIRI , VarOrIRI)

Patterns Definition

Patterns	→	Pattern (',' Pattern)*
Pattern	→	DFunction GeoFilter CameraFunction
/* DL-related functions */		
DFunction	→	SPARQLDL DLExtension
SPARQLDL	→	.../* all the SPARQL-DL predicates */
DLExtension	→	<i>Domain</i> (VarOrIRI ','VarOrIRI) <i>Range</i> (VarOrIRI ','VarOrIRI)
/* GeoSPARQL functions */		
GeoFilter	→	<i>FILTER</i> '(' GeoFunction ')'
GeoFunction	→	<i>geof:sfIntersects</i> (Geometry , Geometry) <i>geof:sfWithin</i> (Geometry , Geometry)
Geometry	→	VarOrIRI <i>geof:buffer</i> (Geometry , Real , Units)
/* Camera View Analysis functions */		
CameraFunction	→	.../* checkKindOfView, percentageShot, ... */

Basic grammar productions

VarOrIRI	→	Var IRI
IRI	→	...
Var	→	...
...	→	...

Figure 4.9: Simplified Grammar of SHERLOCK's query language.

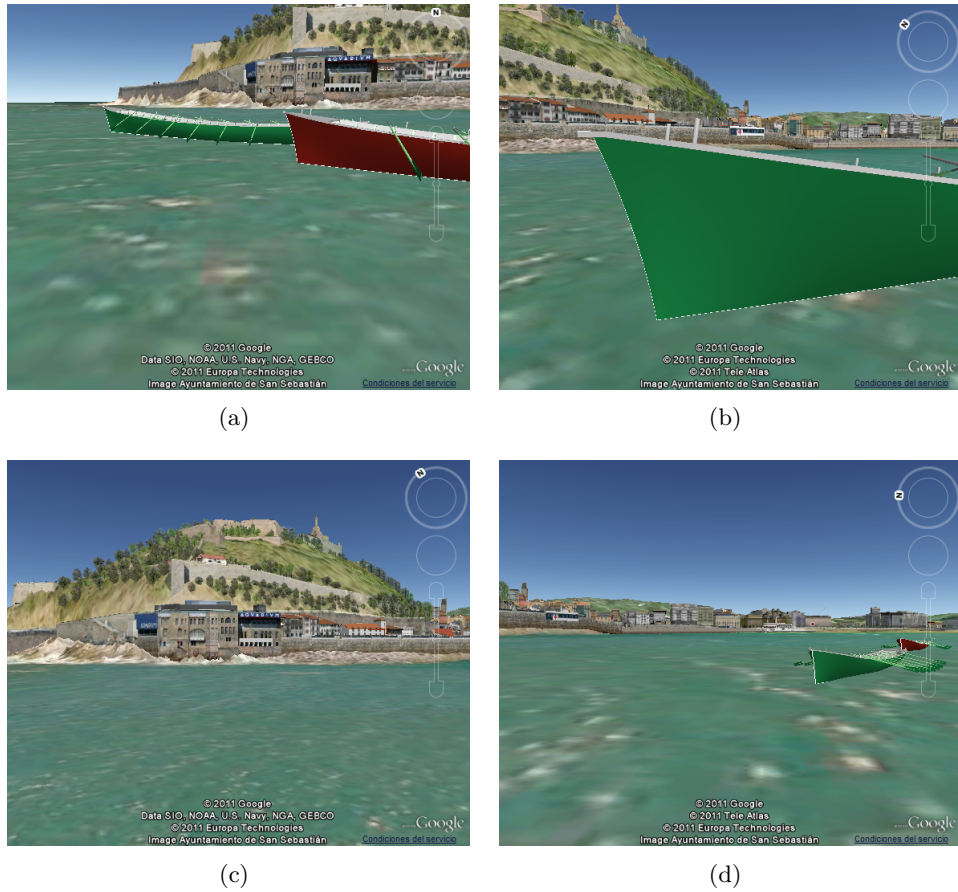


Figure 4.10: Sample views, recreated using Google Earth, of three cameras (*CAM1* (a), *CAM2* (b), *CAM3* (c), and *CAM3* after 4 seconds panning to the right (d)) covering a rowing race.

Chapter 5

Knowledge Update

To be able to provide its users with interesting information, each SHERLOCK device has to maintain its local knowledge updated. First, it needs to keep updated the information about the context of the device and user, as this information is used to determine whether a certain service might be interesting for her current situation. Also, as explained in the previous chapter, this information is leveraged to check her privacy preferences regarding the exchange of information with other devices. Second, it needs to keep updated the local ontology which contains information about services and the surroundings. This information is essential as, for example, a device might not have information about services in the current location. In this chapter, we present the *Context Updater* and *Ontology Updater* agents in charge of these tasks. These agents leverage the communication with other SHERLOCK devices to exchange information and learn from their interactions. In the case of the Context Updater agent it collect facts about the context of other users and their devices. The agent uses this information to improve the understanding of the context of its user (e.g., by including facts related to her current activity). In the case of the Ontology Updater agent, it exchanges ontologies modeling services and integrates them into its local knowledge by discovering subsumption relationships between their concepts.

5.1 Updating Context: Context Updater Agent

As explained before, the goal of the context information managed about the user and other users around is twofold. On the one hand, SHERLOCK uses the context of a user to offer services that might be interesting for her. On the other hand, SHERLOCK uses the context of other users/devices to answer

requests of the user. Also, the context information is used to specialize the privacy preferences of the user regarding her information being accessed by other devices.

The Context Updater agent (CU) handles the volatile context information belonging to the device, its user, and other users/devices around. For that, the CU agent performs the following tasks (see Figure 5.1):

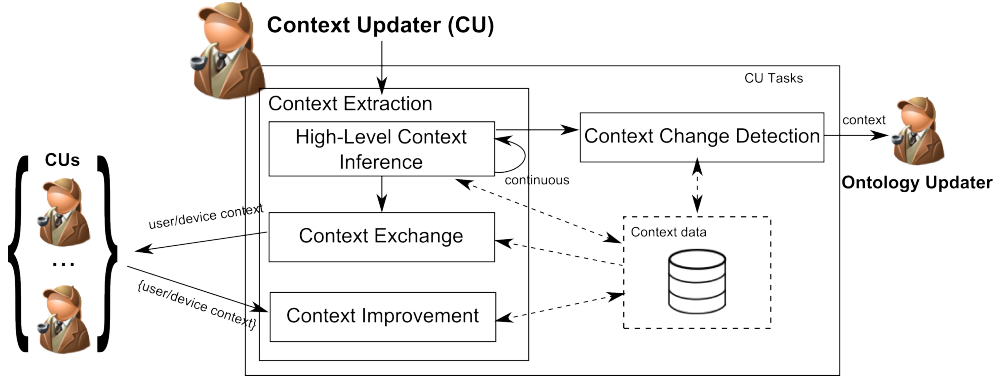


Figure 5.1: Context Updater agent (CU) tasks.

- *Context Extraction.* This task involves inferring a high-level notion of context, exchange this information with other devices, and integrate the information received to improve the inferences made.
- *Context Change Detection.* Whenever the CU agent infers a new context for the user, it is in charge of detecting significant changes (in our prototype, when the user moves to a different city and/or each month). These changes of context are used to reevaluate the information (e.g., services) that might be interesting for the user by the Ontology Updater agent as we will explain in Section 5.2.

In the following, we explain our approach for context extraction.

5.1.1 Context Extraction

The CU agent uses *context synthesizers* [RAMC04] to process the low-level sensory data produced by the device to extract a higher level notion of the context of the user. These context synthesizers are external modules that take as input the raw data (e.g., produced by sensors) and return a semantic notion

of context using terms from the context ontology (see Figure 4.2). Also, context synthesizers typically compute and return a value that expresses the confidence on the information inferred being correct. For instance, the context synthesizer presented in [PWLZB07] is able to use information from the microphone on a device and user calendar and deduce that the user is in a meeting with a certain confidence. As the device might have different sensors enabled at a given time and they can be inaccurate, the CU agent uses an approach to enrich (i.e., correct and improve) the information returned by context synthesizers. This approach is based on leveraging the information about devices/users around that the CU agents exchange continuously. The context enrichment task is based on the following steps:

1. Obtain the context information from the available context synthesizers. As mentioned before, these context synthesizers return a high-level notion of context (i.e., location and activity) along with a confidence value.
2. Communicate with devices discovered in the vicinity and request their context information. In our case devices exchange the information described in Table 4.1 and Table 4.2. The KE agent returns the previous information if the privacy preferences of the user allow it.
3. Integrate the context information collected to generate a shared context model. The shared context model integrates facts about location and activity, as well as any secondary context piece related to them.
4. Verify the information integrated in order to detect and resolve inconsistencies. A device could exchange erroneous information inferred by its context synthesizers (e.g., it could exchange a wrong location if the last GPS coordinate of the device was generated an hour ago).

In the following we focus on our approach to deal with the last two steps.

Context Integration

When mobile devices using different context providers and synthesizers exchange context information, some of which is possibly imprecise, sometimes there can be diverse information. For example, consider a scenario where multiple users (Annie, Abed, Jeff, and Pierce) are in a study room in the library of a university with their SHERLOCK-enabled devices (smartphones, tablets, and a laptop). The information that their respective devices extract about the context of their users depend on the current sensors enabled and the accuracy of the context

synthesizing phase. Imagine that the CU agents on their devices exchange the context information of their users that they inferred. Table 5.1 shows the information that the context synthesizers in each device inferred, the confidence they computed for each inference, and the normalized confidence. For example, let's focus on Annie's tablet which receives information about the location such as *Glendale Community College (GCC)* and *Study_room_F* while her own device thinks the location is *Annie's_Home*. In this situation and for each piece of context, the CU agent has to determine from all the possible values which ones are most likely. For this task, it uses a simple weighted majority voting scheme which has been proven to be successful in tasks such as pattern recognition [LS97].

Identity	Location	Source	Confidence	Confidence
Annie	Annie's Home	Calendar	0.75	0.23
Abed	GCC	GPS and GeoNames	0.8	0.25
Jeff	Study room F	Foursquare and GPS	0.7	0.22
Pierce	GCC	IP address	0.9	0.29

Table 5.1: Contextual information shared about location.

The CU agent uses the semantic reasoner and ontology to deduce if a given fact is supporting a different one. For example, Jeff's smartphone states that its location is *Study_room_F* and so, Jeff's device is implicitly supporting that its location is *GCC* as the study room is in GCC's library. Therefore, in the example of Table 5.1 three devices support that the location is GCC (Pierce's, Abed's, and Jeff's). The same situation can arise with activities, both Abed's and Annie's devices share that the activity performed is a *Study_Group* and so, they support the *Meeting* activity shared by Jeff's device.

To do this, the CU agent first obtains all the equivalent instances for a given fact by using the *owl:sameAs* property. For example, for the instance *GCC* the instance *Greendale_Community_College* will be obtained. Then, the CU agent obtains the list of supported facts by the selected one and its equivalents. This process varies depending on the type of fact:

- *Location facts*: The system uses the *isIn* property of the context ontology, that models geographic areas contained in others [BBMI13], to obtain the list of supported facts. As this property is transitive, it is possible to obtain the complete list of directly and indirectly (i.e., inferred by the

reasoner) supported facts. For example, as the following information has been modeled in the ontology $\langle \textit{Study_room_F}, \textit{isIn}, \textit{GCC_Library} \rangle$ and $\langle \textit{GCC_Library}, \textit{isIn}, \textit{GCC} \rangle$, a device sharing a fact stating that the location is *Study_room_F* is implicitly supporting that the location is also *GCC_Library* and *GCC*.

- *Activity facts*: The system obtains the class(es) of each individual by using the *rdf:type* property. Then, for each class, the system uses the *rdfs:subClassOf* property to obtain directly and indirectly supported facts. For example, the fact for activity shared by Annie's device is a *Spanish_Study_Group* which in turn is a subclass of *Study_Group* and *Meeting*. Therefore, Annie's device is implicitly supporting that the activity is also *Study_Group* and *Meeting*.

For each different context piece, cp_x (e.g., cp_{loc} for location), a CU agent has to compute a global confidence on each of the different facts shared, f_i , (e.g., *GCC*, *Annie's_Home*, and *Study_room_F*) taking into account that some of them can be supported by more than one device (e.g., *GCC* is supported by Pierce, Abed, and Jeff as mentioned before). First, the CU agent computes the normalized value by defining a normalized local confidence value, nc_i , as follows:

$$nc_i = \frac{\max(lc_i, 0)}{\sum_j \max(lc_j, 0)} \quad (5.1)$$

where the normalized local confidence value is positive (see column ||Confidence|| in Table 5.1). Then, let T be the set of normalized confidence values related to a piece of context cp_i , and let S be the set of normalized confidence values that support a context value f_i (e.g., location facts from Abed's and Pierce's devices support location as *GCC*). The CU agent sums up the values in S and normalize it over T , to compute the global confidence gc_i , as follows:

$$gc_i = \frac{\sum_i nc_i}{\sum_j nc_j} \quad \forall c_i \in S, nc_j \in T \quad (5.2)$$

After the context integration process, the CU agent will finally obtain a list of candidate context pieces with their computed confidence, $GC(cp_x)$. In our previous example, the final possible locations computed for the users along with their confidences are:

$$GC(cp_{loc}) = \{\textit{Annie's_Home}(0.23), \textit{GCC}(0.77), \textit{Study_room_F}(0.22)\}$$

Notice, that there is an inconsistency in this shared primary context information as there are two conflicting locations present, namely *Study_room_F/GCC* and *Annie's_Home*. In the following we explain how the CU agent detects and resolve these inconsistencies.

Context Inconsistency Resolution

Among the tasks that a semantic DL reasoner performs, consistency checking is defined as the operation “which ensures that an ontology does not contain any contradictory facts” [SPCGKK07]. To detect semantic inconsistencies, constraints should be modeled in the ontology. In our context ontology these constraints are mainly defined in two ways: 1) by using the *owl:disjointWith* class construct, to model that two classes have no members in common, and 2) by using cardinality constraints on properties (such as *owl:functionalProperty* to model that a property can have only one –unique– value for each instance) or classes (such as *owl:maxCardinality* to model that a class has at most N semantically distinct values). For example, in the context ontology that we defined for our prototype (see Figure 4.2) we stated that a user can only have one location (by defining the *hasLocation* property as functional), and that the activity class *Standing* is disjoint with the class *Running*.

The CU agent uses the context facts along with their confidence values for inconsistency detection and resolution. For each piece of context, cp_x , and the list of possible values, $GC(cp_x)$, the system performs the following steps:

1. Remove from the local ontology every fact related to cp_x . For example, the system removes the axiom $\langle \textit{Annie}, \textit{hasLocation}, \textit{Annie's_Home} \rangle$.
2. Reorder $GC(cp_x)$ according to the confidence computed for each element in descending order. For example, the list of possible locations will be reordered to $GC(cp_{loc}) = \{GCC(0.77), \textit{Annie's_Home}(0.23), \textit{Study_room_F}(0.22)\}$.
3. For each element of $GC(cp_x)$ create an axiom, include it in the local ontology, and use the reasoner to reclassify the ontology to check whether the ontology is still consistent. In the case of the reasoner inferring that the ontology is inconsistent, remove the last axiom materialized because its confidence will be lower than previous one(s).

The restriction on location context piece can be defined on the property *hasLocation*. Before creating a new axiom for the location of the user (e.g.,

$\langle \text{Annie}, \text{hasLocation}, \text{Study_room_F} \rangle$), the CU agent gets any existing location facts (e.g., $\langle \text{Annie}, \text{hasLocation}, \text{GCC} \rangle$) and checks whether the new instance of location (Study_room_F) is contained in the existing one (GCC) by using the *isIn* property explained before. If so, the previous axiom is replaced by the new one as it will preserve the existing semantics and make it more specific (e.g., the axiom $\langle \text{Annie}, \text{hasLocation}, \text{Study_room_F} \rangle$ implicitly states $\langle \text{Annie}, \text{hasLocation}, \text{GCC} \rangle$). Otherwise the axiom will not be created (e.g., the CU agent will not create the axiom $\langle \text{Annie}, \text{hasLocation}, \text{Home} \rangle$).

For expressing constraints on the activity context piece we use the OWL function *owl:disjointWith*. For the first element of $GC(cp_{act})$, the CU agent creates an axiom to state that the user is involved in an activity (e.g., $\langle \text{Annie}, \text{hasActivity}, \text{Activity1} \rangle$). Then, it creates an axiom that defines the class of the activity (e.g., $\langle \text{Activity1}, \text{rdf:type}, \text{Meeting} \rangle$). For the next elements of $GC(cp_{act})$, the CM agent simply creates axioms to make the class of the activity more specific (e.g., $\langle \text{Activity1}, \text{rdf:type}, \text{Study_Group} \rangle$) and then uses the reasoner to check whether the ontology is consistent or not. If not, the last axiom is removed.

In some scenarios it is possible that only a few SHERLOCK-enabled devices share interesting and precise information and so the confidence computed for them will be low (e.g., in our previous example only Jeff shares that the location is Study_room_F and then the confidence computed is the lowest). However, it is also possible that this low confidence is caused by wrong information being shared. A variety of approaches could be followed to tackle this problem, from conservative solutions (such as only use the context with the highest confidence) to optimistic approaches (such as use all context that is not inconsistent). The CU agent uses a semi-optimistic approach: use all context that is not inconsistent and whose confidence is greater than a threshold value. In Section B.3.3 we show the effect of each one of these approaches in our experiments and the threshold that we computed for the results obtained. Notice that this threshold could also be dynamically computed depending on the type of context (e.g., location, activity, etc.) by using machine learning techniques and even involving the user-in-the-loop as done by Kwapisz et al. [KWM11].

5.2 Updating Ontology: Ontology Updater Agent

The Ontology Updater agent (OU) is in charge of keeping the knowledge on the local ontology on the device updated. As stated before, OU agents from different devices *learn* from their interactions as they exchange part of their

local ontologies and data, integrating them in their local knowledge. In this scenario, appropriate knowledge management is crucial in order to keep the approach scalable (otherwise, a device would end up handling huge amounts of information, which might even not related to the current context of the user). For that, each OU agent performs the following tasks (see Figure 5.2):

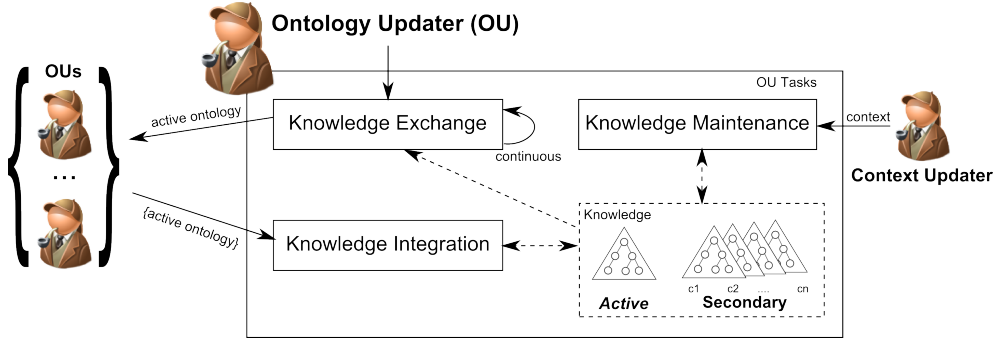


Figure 5.2: Ontology Updater agent (OU) tasks.

- **Knowledge Maintenance:** The OU agent divides the knowledge into an active and secondary knowledge to enable an efficient management of it.
- **Knowledge Exchange:** This task involves exchanging the active knowledge with OU agents in other devices.
- **Knowledge Integration:** The knowledge received is integrated into the local knowledge on the device by applying ontology alignment techniques.

In the following, we explain these three tasks.

5.2.1 Knowledge Maintenance

Instead of integrating all the knowledge in an ever-growing ontology (which might lead to scalability problems when reasoning or querying it), the OU keeps *active* just a module of the ontology which applies to the current user's context, while storing information that might be interesting in other contexts in *secondary* modules. Thus, the size of the ontology which will be used during the capturing of a user information need and its processing is minimized (local reasoning on current mobile devices has been shown feasible for small and

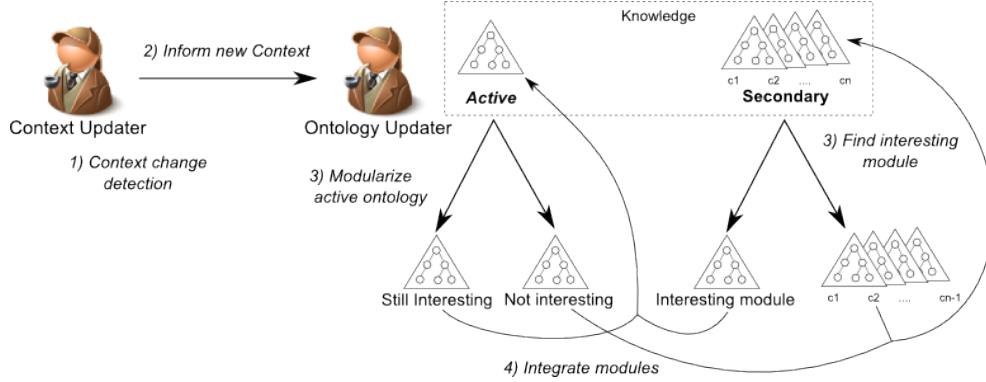


Figure 5.3: Steps involved in the management of knowledge.

medium ontologies in our experiments in Appendix A) whereas no knowledge is forgotten.

As Figure 5.3 shows, whenever the CU agent informs the OU agent about a significant change on the user context, the OU agent starts the process of selecting the knowledge that might be of interest to the user. For that, it first checks the current active ontology to obtain which part is still of interest (e.g., definitions of services which do not depend on the specific location of the user and so might be always interesting for her), and which not. To extract such knowledge, the OU agent uses ontology modularization techniques [SPS09] exploiting the information about the current context obtained from the CU agent (e.g., the new city where the user is). In parallel, the OU agent checks the secondary ontology, where different modules labeled using the context (in our case, the city) are, to find more interesting information. Afterward, the interesting knowledge from the active and secondary ontologies are integrated and becomes the new active module, whereas the rest is also integrated and stored in the secondary ontology.

5.2.2 Knowledge Exchange

Whenever two SHERLOCK devices meet, their OU agents exchange knowledge so both devices learn from the interaction, increasing the information that a particular SHERLOCK device has about its environment. To restrict the information exchanged, as in wireless environments the connection between devices might be short, OU agents only exchange their *active ontologies* (i.e., the knowledge relevant to the user's current context) and associated data. This process is performed continuously, tracking the list of devices recently

contacted to avoid exchanging the same information with the same devices all over.

Note, as the CU agent does, OU agents also check the privacy preferences of their users before exchanging knowledge in order to avoid disclosures. Finally, OU agents rely on a digital signature schema to enforce trust on the exchanged pieces of knowledge: each OU agent checks the validity of the signature/certificate of the user/company which defined each particular piece of knowledge before integrating it (note that the knowledge definer might be different from the knowledge exchanger).

5.2.3 Knowledge Integration

As we have seen in Chapter 4, SHERLOCK devices have a pre-shared ontology which is extended by service definers in order to describe ontologically their services and the terms needed to do so. While this predefined vocabulary is useful to provide a base common knowledge model, the vocabulary extensions made by different vendors are likely not to be completely aligned, even when dealing with similar domains. For example, two different contributors might define a service to find transports and a service to find taxis without explicitly linking them, even though that the relation might be obvious. Therefore, when receiving knowledge from other devices and before integrating them, the OU agent has to align the exchanged schemas [ES+07].

In SHERLOCK, we advocate to combine different approaches in order to extract synonym as well as subsumption relationships between terms (which is strongly helped by the preshared vocabulary). There are many works in the literature for the extraction of synonymy relationships [SE13] (e.g., to extract that the concept *A* from ontology *O1* and the concept *B* from ontology *O2* are equivalent) but, in particular, we use the technique explained in [GA13] to extract synonymy between concepts and roles from the two ontologies. This technique compares each pair of terms by extracting their ontological context and combining different elementary ontology matching techniques (e.g., lexical distances and vector space modeling). As synonymy is a very strict relationship that implies, in fact, that the two entities have the same meaning, we also incorporate an alignment algorithm that we developed based on [GA13] for extracting subsumption relationships between concepts (i.e., extracting that the concept *A* from ontology *O1* is more general than the term *B* from ontology *O2*). In the following we detail our proposed algorithm.

Discovering Subsumption Relationships

To integrate knowledge from other device into its local ontology, the OU agent discovers subsumption relationships between the concepts in the two ontologies¹. For this, the OU agent uses the *ontological context* [GLdSMM07] of the concepts. Given a concept C of an ontology O , we consider its ontological context as $\langle l, R, hypo, hype \rangle$. Where the following features are considered:

- Label, which is the name of the concept.
- Roles, which, in our approach, is every role r such that O entails r has domain C .
- Hyponyms, which are the concept names subsumed by C .
- Hypernyms, which are the concept names subsuming C .

Therefore, the goal of the OU agent is to discover the possible subsumption relationships among the concepts of the ontologies using their ontological contexts by considering $C_s \sqsubseteq C_S, \forall C_s \in O_1, C_S \in O_2$. For this, our approach computes a *subsumption degree* d that indicates the confidence of the OU agent on the existence of such a relationship as:

$$d = f(sub(l_s, l_S), sub(R_s, R_S), sub(R_s, hypo_S))$$

where the subsumption degree of the labels of the concepts ($sub(l_s, l_S)$), their roles ($sub(R_s, R_S)$), and potential co-hyponyms of C_s ($sub(R_s, hypo_S)$) are combined, and $R_O = \{r \in O_r | C_O \in domain(r)\}$ where O_r is the set of role names of the ontology O . Notice that some of the roles in R_s and R_S are inherited from the hypernyms of their concepts (i.e., if $C_s \sqsubseteq C' \wedge C' \in domain(r) \rightarrow C_s \in domain(r)$). Also, some of the roles in these sets are not explicitly asserted but inferred using a DL reasoner. Finally, a role r is “shared” by the concepts C_s and C_S if $r \in R_s$ and $r \in R_S$.

The set of all super & subconcepts of two concepts (i.e., their *Semantic Cotopy* [MS02]) has been compared before to discover synonymy. In our case, we do not directly compare these two sets as our focus is to extract subsumptions relationships (e.g., the concept *GradStudent* from an ontology can be a subconcept of *Student* from other ontology independently on how similar are the subconcepts of both *GradStudent* and *Student*). Nevertheless, we consider the roles that the concepts inherit from their superconcepts.

¹In DL subsumption exists also among roles, however we only focus on subsumption among concepts which is more common.

We propose the following steps to discover subsumption relationships along with their subsumption degree (see Figure 5.4):

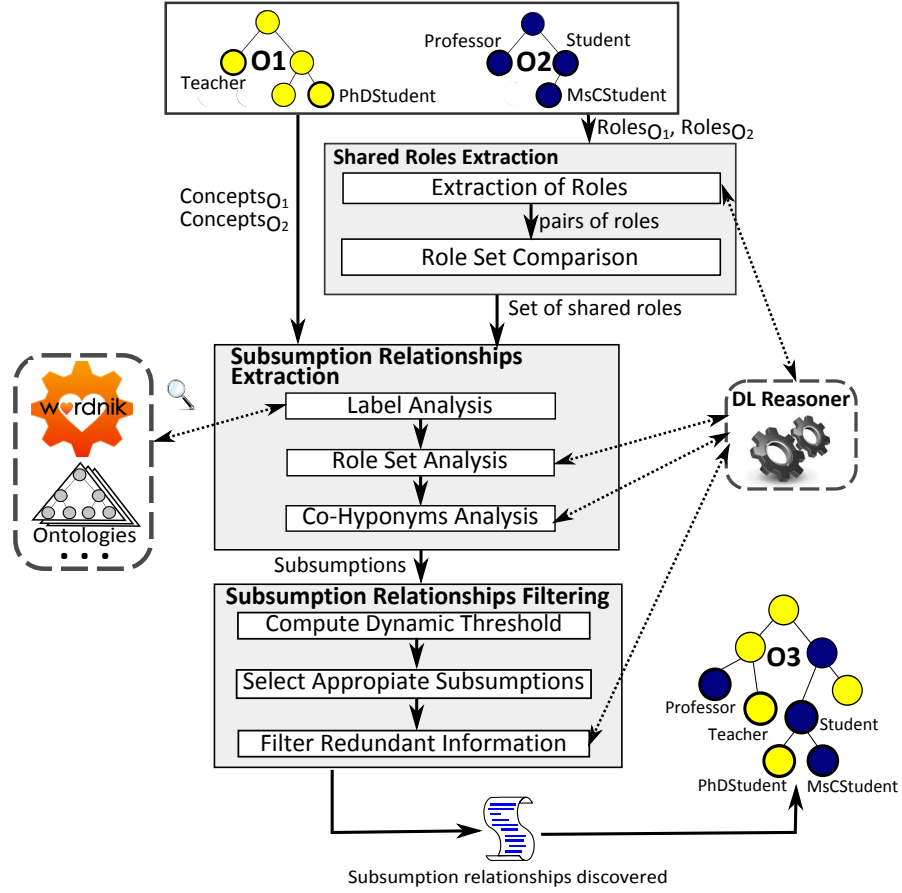


Figure 5.4: Main steps in our approach to extract subsumption relationships from two source ontologies.

1. *Shared roles extraction:* First, the algorithm extracts $R_{O_1} = \{r \in O_{1r} | C_{O_1} \in domain(r)\}$ and $R_{O_2} = \{r \in O_{2r} | C_{O_2} \in domain(r)\}$. For that, it uses the DL reasoner to check whether the concept C_{O_1} is a subconcept of the domain of r . Then, the set of roles of the two concepts are compared to extract the list of shared roles. Roles from different ontologies can be slightly different even when representing the same entity (e.g., the roles $O_1 \# StudiesAt$ and $O_2 \# StudentFrom$). Therefore, we should consider every possible combination of the roles of the concepts

being analyzed as shared. To decrease the number of comparisons the most probable pairs can be obtained by computing a similarity degree. In our case, as explained before, we use the technique described in [GA13] to compute a similarity degree between roles.

2. *Subsumption relationships extraction*: The subsumption degree among the concepts extracted from each ontology is computed by using their labels (and external sources of information), the set of similar roles from the previous step, and potential co-hyponyms (see Section 5.2.3).
3. *Subsumption relationships filtering*: The less probable relationships extracted are filtered out using a dynamic threshold to return a mapping file with the most probable subsumption relationships (see Section 5.2.3).

In the following we detail the previous steps.

Subsumption Relationships Extraction

The subsumption extraction method provides a measure, *subsumption degree*, that indicates the confidence on the concept C_s from an ontology, being subsumed by the concept C_S from a different ontology, by considering $C_s \sqsubseteq C_S$, $\forall C_s \in O_1, C_S \in O_2$. This method analyzes the ontological context of every concept (i.e., label, roles, and hierarchical relationships with other concepts) and so, it covers the major dimensions used in ontology matching according to [ES+07]:

$$\begin{aligned} \text{subsumptionDegree}(C_s, C_S) = & w_l * dLabel(C_s, C_S) + \\ & w_r * dRoles(C_s, C_S) + \\ & w_{ch} * dCohyp(C_s, C_S) \end{aligned} \quad (5.3)$$

where $dLabel(C_s, C_S)$, $dRoles(C_s, C_S)$, $dCohyp(C_s, C_S)$ obtain the subsumption degree between the labels (Section 5.2.3), the roles (Section 5.2.3), and the co-hyponyms (Section 5.2.3) of the concepts, respectively. Also, w_l , w_r , and w_{ch} are some weights (relative importance) assigned to each factor with $w_l + w_r + w_{ch} = 1$. Intuitively, the relative importance of w_r has to be greater than the others as it considers the semantic features associated with each concept, which differentiate them. Although, the information about co-hyponyms also contain semantic information about the concept, it cannot be used to distinguish between the given concept and their co-hyponyms.

Label Analysis

The names of the concepts can be used to obtain information about their relationships from third-party lexical databases [BBCCGMMV00; SdM08]. However, sometimes concept names could not be found in these lexical databases (e.g., in our running example the name *PhDStudent* does not appear in Wordnik, the resource used in our experimental evaluation) so, we also compare the two strings. Specifically, as a concept subsumed by another one should specialize it by definition, these subsumed terms sometimes contain the name of their subsumers (e.g., *PhDStudent* and *Student*, *FullProfessor* and *Professor*). In this case, we compute the similarity string metric between the names of the concepts [SSK05]. Otherwise, our approach does not compare the two strings at all, as this could lead to false positives (e.g., *Universe* and *University* look similar but there is not direct relationship between them). Finally, the information obtained from the external sources and the string comparison are weighted and added (intuitively, more importance should be given to the former as, if available, provides more information than the latter).

Role Set Analysis

The roles of the concepts can be used to find “hints” related to the features that every concept, C_s , subsumed by another concept, C_S , presents:

Statement 1 *C_s must have all the roles of C_S since a concept inherits all roles of its subsumer².*

Statement 2 *C_s should have more roles than C_S (i.e., it should be more specialized).*

In the ideal case, both statements should be followed; however, in a real scenario different situations may happen:

Statement 3 *As they belong to different ontologies, it is possible that C_s does not have all the roles of C_S although it is true that $C_s \sqsubseteq C_S$.*

Statement 4 *Some ontology roles are not characteristic enough to discover the semantics of a concept (i.e., the domain of such roles contains all the concepts in the ontology). So, we could even find concepts which do not have any characteristic role (all its roles are inherited from top concepts in the ontology).*

²A concept “has” a role if the concept is in the domain of such a role, according to the definition of the ontological context of a concept explained before.

Given two concepts and their set of roles (shared or not) we can define a formula that takes the previous statements into account to obtain their subsumption degree. The graphical representation of the desired subsumption degree with respect to the number of roles that the two concepts share, considering the number of roles of the subsumer concept, could be similar to the graph in Figure 5.5.

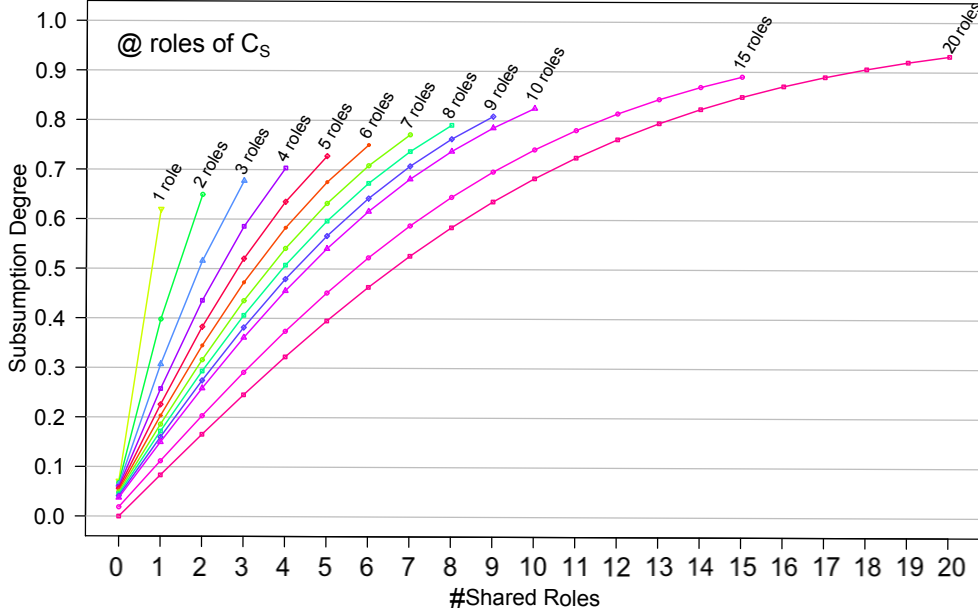


Figure 5.5: Subsumption degree (Y-axis) between two concepts, C_s and C_S , depending on the number of roles of C_S (denoted by the different curves) and their shared roles (X-axis).

The important aspect of the graph in Figure 5.5 is not the specific values shown (which simply correspond to our prototype) but that it models the following generic rules that capture the existence of subsumption relationships:

Rule 1 *The higher the percentage of roles of C_S that C_s has, the greater the subsumption degree.*

According to Statement 1, a subsumed concept should inherit 100% of the roles of its subsumers, although sometimes this percentage is less (Statement 3). Consider C_s^1 and C_s^2 which share 40% and 80% of C_S roles, respectively. The subsumption degree computed for $C_s^2 \sqsubseteq C_S$ should be greater than the degree

computed for $C_s^1 \sqsubseteq C_S$. In Figure 5.5, notice that the function is increasing for any number of roles.

Rule 2 *The higher the number of shared roles, the greater the subsumption degree.*

For example, if C_s^1 shares one role with C_S^1 (suppose that is 50% of C_S^1 's roles in this case) and C_s^2 shares six roles with C_S^2 (which is also 50%) the probability of a subsumption relationship between C_s^2 and C_S^2 should be greater than between C_s^1 and C_S^1 . I.e., according to the Duck Test: "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."; however, if it only swims like a duck, the probability of being a duck is lower. For example, in Figure 5.5, sharing 2/3 (two out of three) roles scores 0.52 while sharing 6/9 roles scores 0.64.

In the following we present three rules that detail Rule 2 in case of C_s sharing all, none, and n of C_S roles:

Rule 2.1 *If C_s shares all the roles of C_S (the ideal case according to Statement 1), the higher the number of roles C_S has, the higher the subsumption degree between them.*

For example, if C_s^1 shares one role with C_S^1 (that is 100% as C_S^1 only has one role) and C_s^2 shares six roles with C_S^2 (which is also 100% as C_S^2 has six roles), then the probability of a subsumption relationship between C_s^2 and C_S^2 should be greater than between C_s^1 and C_S^1 . Again, the Duck Test applies. In addition, the difference between these maximum values cannot be linear according to the number of C_S roles as, for any number of roles, the subsumption degree function should score between 0 and 1 always. Hence, the subsumption degree should grow slower for a high number of shared roles as beyond a certain amount of shared roles the subsumption degree should be close to 1. In Figure 5.5, notice that beyond 7/7 the subsumption degree scores above 0.8 (we configured our prototype to return high values when sharing at least 7 roles).

Rule 2.2 *If C_s shares no role with C_S (and this could happen according to Statement 3), the higher the number of roles C_S has, the lower the subsumption degree between them.*

For example, if C_s^1 shares no role with C_S^1 (C_S^1 has one role in this case) and C_s^2 shares no role with C_S^2 (C_S^2 has six roles in this case) the probability of a subsumption relationship between C_s^2 and C_S^2 should be lower than between

C_s^1 and C_S^1 . According to what we could call the *Opposite Duck Test*: if it does not look like a duck, does not swim like a duck, and does not quack like a duck, then it probably is not a duck. However, if it only does not swim like a duck, the probability of being a duck is higher. Also, subsumption degrees for all these minimum values have to be lower than situations where C_s shares one or more roles with C_S , which is always better than not sharing any role at all. In Figure 5.5, for example, 0/20 scores 0 and 0/1 scores 0.07; also, all these minimal values are lower than any other value when the number of shared roles is one or higher (which scores 0.08 and above).

Rule 2.3 *If C_s shares n of the m roles of C_S ($0 < n < m$), the higher the number of shared roles, the greater the subsumption degree.*

Intuitively, we could think that the number of non-shared roles, $m - n$, could “neutralize” the number of shared roles. However, according to the spirit of Rule 2, we believe that the greater number of shared roles, the more hints of C_s being subsumed by C_S . We could call it the *Weak Duck Test*: if it looks like a duck and quacks like a duck, then it is probably a kind of duck, although we are not sure that it swims like a duck. In other words, we advocate that shared roles score more than what non-shared roles penalize the subsumption degree. So, in Figure 5.5, 1/3 scores 0.3, 4/8 scores 0.5, and 13/20 scores 0.8; i.e., the subsumption degree increases with increasing number of shared roles, even when the number of non-shared roles (2, 4, and 7, respectively) increases as well.

The number of characteristic roles of C_s that are not inherited from C_S (Statement 2) do not affect the graph: To determine whether C_s is subsumed by C_S or not only shared roles are taken into account. For example, if the concept *Person* has four roles and the concept *PhDStudent* has these four and five extra roles, the latter would be a subconcept of *Person* regardless of its extra roles (the extra roles could, at most, indicate that *PhDStudent* might be subsumed by another concept). Also, not all the roles should have the same importance at the time of computing the subsumption degree. In general, the more concepts share the role the less important it is for extracting a subsumption relationship (Statement 4).

To model these rules, and so the trend described in Figure 5.5, we used a logistic function that obtains the subsumption degree between C_s and C_S :

$$dRoles(C_s, C_S) = 2 * \left(\frac{1}{1 + e^{-a*f(C_s, C_S)}} - 0.5 \right) \quad (5.4)$$

where the function $f(C_s, C_S)$ computes the subsumption degree between concepts C_s and C_S as:

$$f(C_s, C_S) = w_{sh} * \frac{sh(C_s, C_S)}{|C_S|} + w_{diff} * \frac{diff(C_s, C_S) - min}{max - min} \quad (5.5)$$

$$diff(C_s, C_S) = \frac{1}{\alpha} * sh(C_s, C_S) - |C_S| \quad (5.6)$$

where C_s and C_S represent their sets of roles and the two terms in Formula 5.5 are: 1) the percentage of C_S 's roles that C_s has (Rule 1), being $sh(C_s, C_S)$ the number of shared roles and $|C_S|$ the number C_S 's roles; 2) the number of C_S 's roles that C_s has (Rule 2) with respect to the rest of the ontology, being $diff(C_s, C_S)$ the difference between the number of shared and non-shared roles, α is a constant that models the importance of the number of shared and non-shared roles, and max and min are the maximum and minimum number of concepts that share the same role in the source ontology and are used to normalize the term. Also, w_{sh} and w_{diff} are used to adjust the importance of the shared roles and non-shared roles, respectively (w_{sh} should be greater according to Rule 2.3).

In addition, a modifier is applied to each role when computing $sh(C_s, C_S)$ that reduces the subsumption degree of concepts sharing very common roles in the ontology:

$$uniqueness(r) = 1 - k * \left(\frac{\#domains}{\#maxDomains} \right) \quad (5.7)$$

being r a role of the concept C , $\#domains$ is the number of concepts in the ontology that have the role, except C and the concepts subsumed by C , and $\#maxDomains$ is the maximum number of concepts that may have r as part of their definition. For example, imagine that an ontology has only three roles r_1 , r_2 , and r_3 which have 3, 10, and 5 concepts as their domains, respectively, then $\#maxDomains = MAX(3, 10, 5)$.

Co-hyponyms Analysis

In general, two co-hyponyms concepts, C_s and $C_{s'}$ such that $(C_s \sqsubseteq C_S) \wedge (C_{s'} \sqsubseteq C_S)$, will share the roles of C_S (remember Statement 1) but they

could share other roles too, especially in the absence of intermediate concepts in the ontology. For example, consider a “missing concept”³ C_m such that $(C_{s'} \sqsubseteq C_m) \wedge (C_s \sqsubseteq C_m) \wedge (C_m \sqsubseteq C_S)$. In this scenario C_s and $C_{s'}$ will share some roles that C_S does not have, the roles inherited from C_m . For example, consider $O_2\#Full_Professor$ and $O_2\#Assistant_Professor$ in Figure 2.2(b) where the concept *Professor* is missing; they share roles inherited from $O_2\#Person$ but also share roles that are common for professors. Thus, the subsumption degree of a concept *AssociateProfessor* with respect to the concept $O_2\#Person$ should be increased as it shares roles with the latter and with its potential co-hyponyms.

To compute the similarity of the concept C_s and the subsumed concepts of C_S , we compare the sets of roles of each subsumed concept, C_i , with the set of roles of C_s individually. This measure is calculated by computing the average between two terms: 1) the amount of roles shared by C_s and the subsumed concept C_i , with respect to the number of roles of C_s , and 2) the amount of roles shared with respect to the number of roles of C_i . Once the system has the similarity degree for each subsumed concept then it calculates the average similarity of all co-hyponyms.

$$dCohyp(C_s, C_S) = \frac{\sum_{C_i \in C_S^{chids}} \left(\frac{sh(C_s, C_i)}{|C_s|} + \frac{sh(C_s, C_i)}{|C_i|} \right)}{2 |C_S^{chids}|} \quad (5.8)$$

being C_S^{chids} the set of subsumees of C_S (i.e., the potential co-hyponyms of C_s) and $|C_S^{chids}|$ the number of elements in the set, C_i is the set of roles of the concept C_i and $|C_i|$ and $|C_s|$ are the number of roles of C_i and C_s , respectively.

This measure can be interesting not only to help extracting subsumption relationships between concepts of the ontologies, but also to detect missing concepts. As commented before, a large number of shared roles between two co-hyponyms could be a hint that indicates the existence of a missing concept. For example, in the explained scenario as $O_1\#MsCStudent$ and $O_2\#PhDStudent$ share some roles that are not inherited from $O_2\#Student$ we could detect a potential missing concept (e.g., *GraduateStudent*) not defined in these ontologies.

³We consider that a concept that exists in real life but has not been defined in the ontology is a missing concept.

Subsumption Relationships Filtering

Our approach computes the subsumption degree among all the pairs of concepts from the two ontologies (see Figure 5.6 for some of the results obtained for our running example). However, there are three major groups of relationships discovered according to their subsumption degree: 1) very probable as the degree is high, 2) clearly unrelated concepts as the degree is very low, and 3) questionable relationships with a neither high nor low degree. Our approach automatically discards those values that are not probable enough by using three filters to:

Relationship	dLabel	dHypRoles	dSimCohyp	Subsumption degree
$O_2\#Magazine_Article \sqsubseteq O_1\#Article$	0.91	0.823	0.0	0.797
...				
$O_2\#Student \sqsubseteq O_1\#Person$	0.75	0.750	0.472	0.624
$O_1\#Professor \sqsubseteq O_2\#Person$	0.75	0.741	0.5	0.618
...				
$O_1\#Professor \sqsubseteq O_2\#Student$	0.0	0.635	0.0	0.508
$O_1\#Person \sqsubseteq O_2\#Student$	0.0	0.635	0.0	0.508
...				
<i>Automatic threshold for these ontologies = 0.503</i>				
...				
$O_2\#Person \sqsubseteq O_1\#MsC_Student$	0.0	0.609	0.0	0.487
$O_2\#Assistant_Professor \sqsubseteq O_1\#MsC_Student$	0.0	0.609	0.0	0.487
...				
$O_2\#Person \sqsubseteq O_1\#PhD_Student$	0.0	0.581	0.0	0.465
$O_2\#Full_Professor \sqsubseteq O_1\#PhD_Student$	0.0	0.581	0.0	0.465
...				
$O_1\#Social_Group \sqsubseteq O_2\#Magazine_Article$	0.0	0.087	0.0	0.069

Figure 5.6: Some of the subsumption degrees obtained for our running example.

- *Discard subsumptions under a (dynamic) threshold.* The trend of the subsumption degrees computed depends on the ontology and thus, the threshold to filter out less probable relationships should be dynamic. For ontologies where most of the roles are shared by many concepts the overall confidence value can be low although, some of the extracted subsumptions can be correct. Instead of using a classifier that would require training on the domain, we propose using a clustering algorithm that automatically divides relationships into the three groups explained before. In the example of Figure 5.6, the threshold obtained by the *k-means* clustering algorithm in our prototype for the high-confidence cluster is 0.503 and so, 462 out of 560 possible subsumption relationships are filtered out (we detail the prototype and experiments performed in

Section B.2).

- *Select between hypernymy and hyponymy.* Our approach obtains the subsumption degree for all the possible combinations of concepts and so, values for both $C_s \sqsubseteq C_S$ and $C_S \sqsubseteq C_s$ are computed. In addition, if *Thing* is the domain of some of the roles, all the concepts inherit them and thus, there will be always a nonzero subsumption degree among them. Selecting both relationships, even when the degree of one of them is low, will mean creating a synonymy relationship between the concepts⁴ and that is out of the scope of the approach as it is not trivial and would require a further analysis. However, we do select both relationships when the degree is exactly the same. Therefore, in other situations this filter discards the relationship with the lower degree (e.g., in Figure 5.6 the filter discards $O_1\#Person \sqsubseteq O_2\#Student$).
- *Discard redundant relationships.* Some of the relationships extracted could involve the same concept C_s , for example $C_s \sqsubseteq C_S^1$ and $C_s \sqsubseteq C_S^2$, and they could be potentially redundant if $C_S^1 \sqsubseteq C_S^2$ or $C_S^2 \sqsubseteq C_S^1$. This filter selects the subsumption degree with the higher degree for a given concept from all the potentially redundant relationships discovered. For example, in Figure 5.6 $O_1\#Professor \sqsubseteq O_2\#Student$ is discarded because $O_1\#Professor \sqsubseteq O_2\#Person$ has a higher subsumption degree and $O_2\#Student$ is subconcept of $O_2\#Person$ ($O_2\#Student \sqsubseteq O_2\#Person$).

The final list of subsumption relationships discovered along with the original axioms from the source ontologies is materialized to create an integrated ontology. A challenge that should be addressed when integrating two ontologies from different sources is to create a consistent ontology as a result (i.e., an ontology which does not contain contradictory facts). Some definitions of concepts can be contradictory so we advocate inserting the subsumption axioms discovered one by one, in descending subsumption degree order, and using a DL reasoner to check the consistency of the ontology after each insertion. If the ontology is inconsistent the last inserted axiom should be removed before inserting other axioms. Also, it would be useful to remove concepts which are classified by the reasoner as unsatisfiable, which means that they are classified as equivalent to *Nothing* and thus cannot have instances.

⁴In DLs, $C_s \equiv C_S \Leftrightarrow (C_s \sqsubseteq C_S) \wedge (C_S \sqsubseteq C_s)$.

5.3 Related Work

In this section we present works related to the two main contributions of this chapter: the discovery of subsumption relationships for the integration of knowledge and the enrichment of the information about the context of the user using the context of users around.

5.3.1 Works on Discovery of Subsumption Relationships

Ontology alignment systems aim to extract semantic relationships among entities from different ontologies to obtain an integrated view of the ontologies [Sow99]. Many research works have focused on ontology alignment [SE13] due to the growing use of ontologies as a formal specification for modeling knowledge. However, most of the effort has been made in the alignment of ontologies through the extraction of synonyms. Systems like CIDER [GA13] and ASMOV [JMSK10] extract these synonymy relationships between pairs of entities from two ontologies using their ontological context as in our approach. There exist only few ontology alignment systems focused on the discovery of subsumption relationships. From them, we comment in the following systems that have a similar goal and use similar information to ours: MOMIS [BBCGMMV00], SCARLET [SdM08], and CSR [SVV08]. Therefore, we do not include here systems that base the extraction of subsumption relationships on shared instances (such as [KLXWL05; TPRT11]) as our goal is the extraction of the relationships at the schema level.

Both MOMIS and SCARLET are based on the use of *Background knowledge* [SM06] about relationships already defined in other resources such as ontologies or lexical databases. Given two concepts from two ontologies, MOMIS [BBCGMMV00] tries to find their semantic relationships looking up their names in WordNet [RS07], and SCARLET [SdM08] tries to find this information in other ontologies where the subsumption relationships have been previously defined. In other words, the relationships that they find must exist in third-party sources. Our approach also exploits some external sources to compare the labels of concepts however, in addition we consider the ontological context of the concepts in the input ontologies.

Classification-based learning of Subsumption Relations (CSR) [SVV08] is an approach for knowledge integration which, similarly to ours, leverages the information contained in the source ontologies to discover subsumption relationships. This approach uses concepts' features (roles and terms extracted from labels, comments, and instances) that provide evidence for the subsumption relationships among these concepts. CSR is based on the use of machine learning

techniques, such as classifiers, and therefore, it needs a training phase which exploits as training examples known subsumption and synonymy relationships from each source ontology individually. However, not all the ontologies are adequate for the training phase as noted by the authors. Our approach does not use machine learning (and thus, do not require training), instead we present some generic rules that capture the existence of subsumption relationships between concepts.

5.3.2 Works on Context Enrichment

Context awareness is a very active field and so, there is an extensive literature available [BDR07]. Context extraction, generating context information for users by using their sensors or other information, has received great attention in the field. The techniques proposed can be broadly classified into two: those that rely on machine learning models to learn about features from sensor data to predict the user context [LMLPCC10]; and those that define context using ontologies and rules and use a reasoner to infer associations between sensor data and user context [GWPZ04].

However, in this work we do not deal with context extraction from low level data, as we rely on external context synthesizers for this task. Instead, we focus on context enrichment by using information shared by others, which has received less attention from the context-awareness community. Indeed, we use P2P networks of devices sharing high level context information in a context enrichment process. So, we want to mention first the recent work by Wibisono et al. [WZL13] that also leverages P2P networks of devices in context-awareness computing. In their approach, whose goal is different from ours, devices in a specific location (e.g., a room) are used to detect the “situation” there (the situation concept they use is similar to the activity concept used in this paper). For that, they integrate their low-level sensor information and use machine learning techniques to reason the most probable situation from the previously defined list of situations for the room. In our approach, we consider high-level context information shared by the devices and base our integration on semantic techniques (ontologies and a semantic reasoner). In addition, we do not start with a set of possible situations for a location and we also consider that more than one could occur at the same time and the same place.

Cooperative Ambiance Monitoring Platform (CoMon) [LJMKHS12] also uses participatory sensing of mobile users by allowing querying and sharing of contexts of interest. The goal of CoMon is different from ours as it focuses on potential energy savings from sharing of context and do not deal with

inconsistencies in the context. Collaboration for achieving an application's goal was studied in [CTHH13]. However, their focus was only on location based collaboration. In this paper we have demonstrated that in a small group of devices one can generate collaborative and shared semantic contextual information for a variety of contextual information pieces. Our system acts like a cross-device middle ware capable of generating consistent and enriched context information for multiple devices.

5.4 Summary of the Chapter

In this chapter, we have presented the agents in charge of keeping the knowledge in a SHERLOCK device updated. We explained the Context Updater agent (CU) whose main goal is to keep the user context information updated. We focused on its ability to improve the user context information inferred locally from the device's sensors by leveraging the context of other devices/users around. This technique is based on the exchange of the high-level context information, which has been inferred by the context synthesizers on the device. With the information received, the CUI agent creates an integrated context model by detecting and resolving possible inconsistencies. Finally, the CU agent uses the integrated context to enrich the information inferred for its user. Then, we explained the Ontology Updater agent (OU) whose main goal is to update the local ontology on the device by exchanging it with other devices. We detailed the knowledge alignment task of the OU agent which performs the integration of the knowledge received based on the extraction of subsumption relationships. The technique presented compares the ontological context of the concepts in the ontologies to integrate to discover such relationships. The OU agent utilizes some generic rules to capture the existence of a subsumption relationship and performs a filtering phase to discard the relationships discovered that are less probable.

Chapter 6

Request Management

The main goal of SHERLOCK is to provide users with interesting services. By selecting these services, users express their information needs in what we refer to a *user information request* and the system is able to obtain such information from different sources. Therefore, the most important part of the knowledge exchanged among SHERLOCK devices, explained in the previous chapter, is related to services. The system performs two steps to handle a user information request (or simply *user request*): generating the formal request and processing it. First, the system helps the user to select an appropriate service and to fill in parameters associated with it. Then, the system generates the formal request with this information, and processes it. Depending on the type of request, the system deploys a network of mobile agents to find the information wherever it is, executes a call to a third-party service, or invokes the service provided by another SHERLOCK device.

6.1 Request Generation

A first step in order to fulfill the user information needs is to capture those needs and formalize them into a request in order to avoid ambiguities. As explained in Section 4.2.1, to provide SHERLOCK with enough expressivity and flexibility, we have designed a SPARQL-like query language that is used by the system making it possible to express semantic location and non-location based queries. This language requires knowledge about SPARQL and SHERLOCK's ontology, so it might be too complicated to be used by non-advanced users. Therefore, our approach helps users to define their interests, guiding them, and capturing their requests (which might require building queries using SHERLOCK's language). To do so, the system relies on the *User Request Manager* agent (URM from

now on) to guide users when defining their information needs. The URM agent performs three main tasks (see Figure 6.1), summarized in the following:

1. *Service Selection*: The URM agent is in charge of obtaining the services (based on a location or not) that are relevant for a user in a particular situation. As we will see in the following, what relevant is comes defined by the user's input/interaction and context. The result of this task is a list of services to be shown to the user for her to select one.
2. *Parameter Provision*: Given a service, the URM agent retrieves the information needed to invoke it (its formal parameters) from the ontology, and, if needed, handles the interaction with the user required to obtain the actual values for the parameters. These values can be simple, such as booleans and numbers, or complex, such as the specification of the type of image to find, and the user can define which ones are more important for her.
3. *Service Handling*: Given a service and a set of parameters with the selected values, the URM agent generates the appropriate service request/invocation. This could mean calling the accessing mechanism of the provider of such service, generating a formal query, or selecting an execution plan for the service.

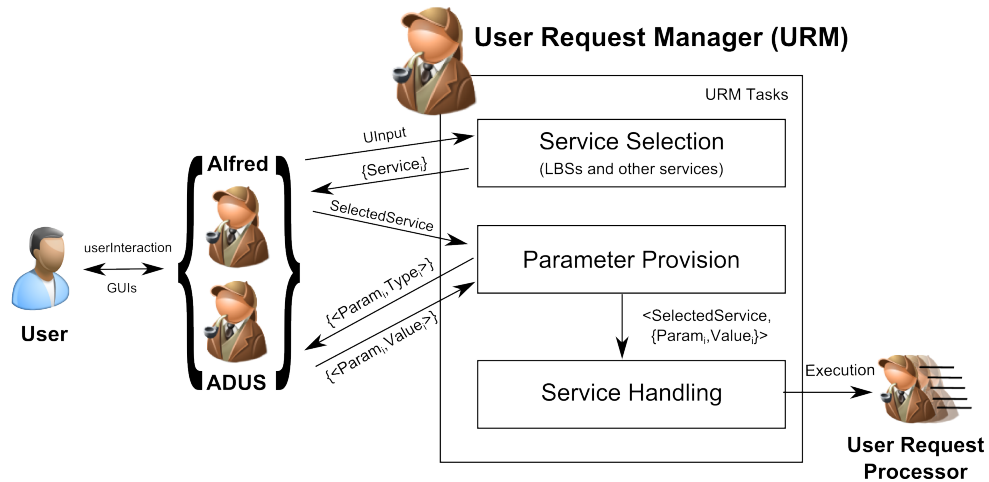


Figure 6.1: User Request Manager agent (URM) tasks.

This way, the user has not to be aware neither of the details of the query language, nor the schema and available services. In the rest of the section, we

detail these steps which are focused on the generation of the request whose processing we will explain in Section 6.2.

6.1.1 Service Selection

Alfred captures a user interaction with SHERLOCK started by her selection of an entity on the map (which can be a particular GPS coordinate that the user is interested in¹, or objects shown as the result of a previous or ongoing request) or by her request of the list of available services by tapping on the “Services” tab. Then, Alfred creates a new URM agent and starts its “Service Selection” task with the information captured from the user (Figure 6.2 shows the different steps of this task which begins with the information that the user inputs and ends with a list of possible interesting services from which the user can select one).

In the following, we detail the process of obtaining the list of relevant services for the user’s input and context (through the “serviceSelection” method in Algorithm 1). First, if the user showed interest on an entity, the URM has to obtain services related to it. For that, the URM obtains services related to such entity as well as to its class and superclasses. For example, if the user selected the MoMa museum the URM will obtain services related to it but also to *Museum* (because *MoMa is-a Museum*) and *Tourist Building* (because *Museum is-a Tourist Building*). Also, the URM obtains services related to entities which geographically contain the selected entity (e.g., New York contains the MoMa museum). This extension is done to expand the list of possible interesting services (lines 4 to 10 in Algorithm 1). If the user selected a coordinate (by just tapping on a point of the map) the URM obtains services related to entities which geographically contain the coordinate (e.g., if the user tapped on a point of Central Park, the URM will obtain services related to Central Park and New York. The URM agent uses the following query processed locally by the KE agent to obtain such information:

```
SELECT ?entity, ?type, ?superClass
WHERE{
  OD{
    FILTER(geof:sfWithin(?entity,<selected entity>)),
    OPTIONAL(StrictSubClassOf(?entity, ?superClass)),
    OPTIONAL(Type(?entity,?type))
  }
}
```

For each entity obtained before (e.g., New York, MoMa, museum) the URM obtains all the services which are related to it. An additional context constraint

¹It might be the current user’s location.

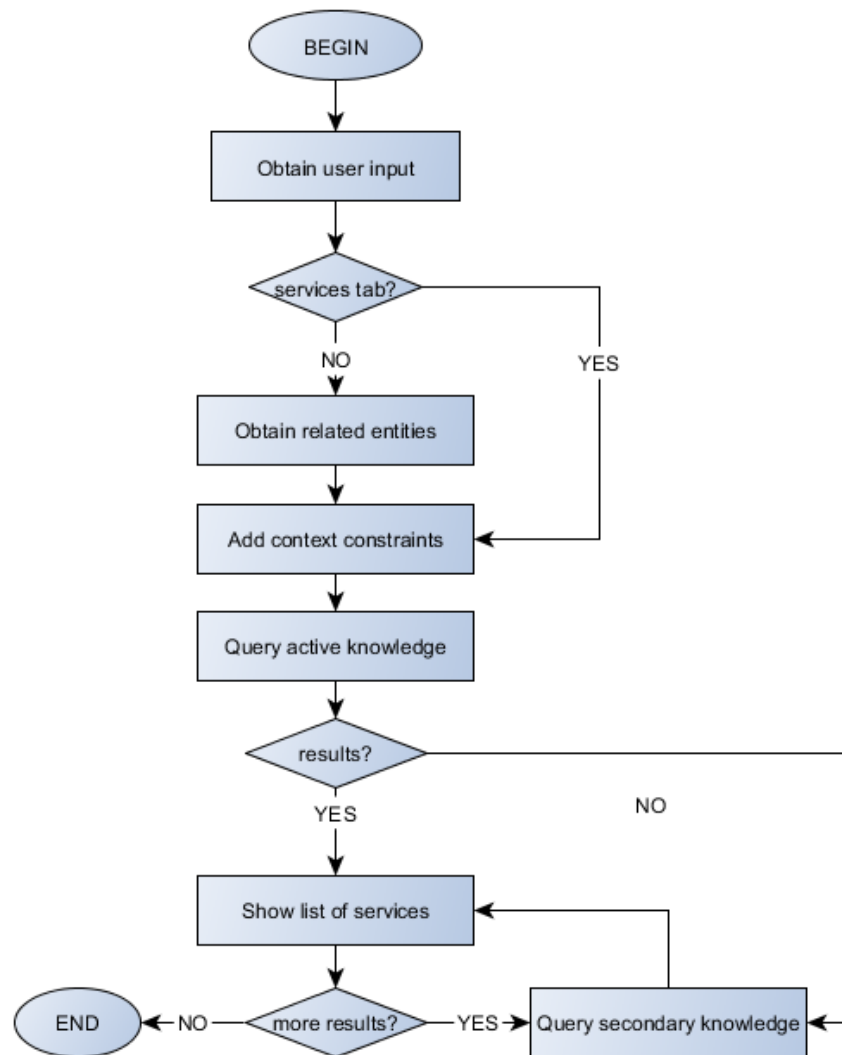


Figure 6.2: Steps to obtain relevant services for a user.

is added to promote those services which are relevant to the current context of the user. This is done by the URM agent through the following query:

```

SELECT ?nameService, ?property, ?context
WHERE{
  OD{
    Type(?service, sherlock:Service),

```

Algorithm 1 serviceSelection(UInput)

Input: *UInput* is the kind of input provided by the user, if it is an *entitySelection*, then it includes the selected entity and information about it to further query the KE. The CU and ADUS agents are also available to the URM, to use context information as well as handle user interaction.

Output: Obtains a list of services relevant for the user's input and context.

```

1: serviceList  $\leftarrow \emptyset$ 
2: entityConstraints  $\leftarrow \emptyset$ 
3: ctxtOptionalConstraints  $\leftarrow$  CU.getUserCtxtConstraints()
4: if UInput.isEntitySelection() then
5:   // the user can select an object or a location, first obtain locations and objects
   // which contain the selected entity
6:   entityConstraints  $\leftarrow$  KE.getConstraints(UInput.entity)
7:   if UInput.entity isA object then
8:     entityConstraints  $\leftarrow$  UInput.entity // the user has selected an object
9:   end if
10: end if
11: // the entity constraints are mandatory (if any), while context ones are just
   // optional
12: serviceList  $\leftarrow$  KE.queryActOntology(entityConstraints, ctxtOptionalConstraints)
13: // the list is ranked to promoting services related user's context
14: rankAccordingContext(serviceList)
15: // ADUS shows the list of services and asks whether further services should be
   // retrieved
16: searchSecondary  $\leftarrow$  ADUS.showServiceListQuerying(serviceList)
17: if serviceList.empty() or searchSecondary then
18:   serviceList.append(KE.querySecondaryModules(entityConstraints, ctxtOptional-
     // alConstraints))
19:   ADUS.showServiceList(serviceList) // ADUS shows the list of services
20: end if

```

```

CASE(PropertyValue(?service, ?property, <entity_i>),
  SubPropertyOf(?property, sherlock:provides)),
CASE(PropertyValue(?service, ?property, <entity_i>),
  SubPropertyOf(?property, sherlock:returns)),
CASE(PropertyValue(?service, ?property, <entity_i>),
  SubPropertyOf(?property, sherlock:interestingFor)),
OPTIONAL(PropertyValue(?service, sherlock:interestingFor, ?context),
  SameAs(?context, <userContext>))
},
PropertyValue(?service, sherlock:name, ?nameService)
}

```

Notice that this query is used for entities which are instances in the ontology (e.g., New York and MoMa). For entities which are concepts (e.g., museum)

the query would be similar but adapted using the “Range” and “Domain” functions instead of the “PropertyValue” function.

The result of the previous query executed for each entity is a set of tuples containing the entity, the service related to it, and the property that links them ($\langle \text{entity}_i, \text{service}, \text{property} \rangle$). For example, if the selected entity was the MoMa museum, $\langle \text{MoMa}, \text{BuyTicketMoMa}, \text{provides} \rangle$ and $\langle \text{Museum}, \text{FindMuseums}, \text{returns} \rangle$ will be returned, among others. This set is ranked (i.e., services related to the selected entity go first, services related to its direct classes second, and so on) and passed to ADUS to generate an interface to show a list of services. The user will select one of the services and this information will be used in the next step.

6.1.2 Parameter Provision

The result of the previous interaction with the user is the selection of a family of services (a subclass of the concept *Service* in the ontology) or a particular service (an instance). For example, the user could select the family of “Find Transportation” services or the specific “Find NY Transports” service (which is the service provided by the local transport office in New York). In fact, a user that wants to find transports regardless of the provider of this information would use the former whereas a user that wants the information offered by a specific provider would select the latter.

In the parameter provision step, firstly the URM obtains the parameters of such selected services, if any. These parameters, which have been defined when the service was modeled, are the formal parameters that the service requires to be invoked or that can be used to restrict the information returned by the service. Thus, depending on the selected entity, the URM obtains the set of parameters to be fulfilled as follows:

- *Service family*: The user has selected a family of services that share a set of formal parameters needed and a set of returned objects. We will denote such a family of services as $SServ$. The URM consults the ontology to obtain all the constraints of the type $SServ \sqsubseteq \text{parameter} : ?x$, which define the minimum set of parameters that such a service has to receive. The result is a set $\{fp_1, \dots, fp_n\}$ of parameters that are applicable to that service.
- *Particular service*: The user has selected a particular instance of service. In this case, firstly, the URM consults the service family which

such an instance belongs to. Then, the URM obtains all the parameters that correspond to this family of services (as in the previous item) as the instance inherits them. Finally, as this service might have extra or constant value parameters, the URM extends (and overrides) the previous result set $\{fp_1, \dots, fp_n\}$ with the parameters applicable to such a particular instance obtained by consulting the Service ontology via *parameter* property. This would be retrieved using the clause *PropertyValue(< serviceSelected >, parameter, ?ip)*.

In both cases, the result is a set of parameters which have to be assigned a value to in order to be able to invoke the service or to filter its results out. The parameters that have a predefined value are not required from the user and are automatically filled for the final request. For the rest, to obtain the actual values of the parameters, the URM relies on ADUS and Alfred. Each parameter comes along with information about their expected value to be entered (e.g., a location, a boolean, or even an instance of a concept defined in the ontology).

Notice that there are three types of “parameters” shown to the user: 1) parameters defined in the ontology through the *parameter* property; 2) location for LBS; and 3) provider in the case of services provided by several providers. A service might not have any parameter at all, although typically services will have parameters of the first type used to filter out the information returned. LBS will need, by definition, a location which might have to be requested to the user. Finally, external services (e.g., provided by SHERLOCK objects as explained in Section 4.1.2) might need the user to select the specific provider. For example, if the user selected the external service provided by SHERLOCK taxis to “pick her up”, she will have to select the specific taxi (or any or even all as explained before).

6.1.3 Service Handling

With the service which the user selected (remember that the user could select a particular service or a family of services) and the values that the user introduced for the parameters associated with such service, the goal of this task is to decide how to execute it. Depending on the type of service a different process has to be followed (see Algorithm 2):

- Lines 3-5 If the service contains an execution plan (modeled through the *execute* property in the ontology as we explained in Section 4.1.2), the URM creates an *User Request Processor* agent (URP) to execute the plan,

Algorithm 2 serviceHandling($\{service\}$, $\{< parameter, value >\}$)

Input: $\{service\}$ is a set of services selected by the user, $\{< parameter, value >\}$ is a set of tuples containing a parameter and the value selected by the user.

```

1: for each service in  $\{service\}$  do
2:   // the user could have selected more than one service
3:   if service.execute  $\neq$  null then
4:     // create a new URP to execute the service plan
5:     URP.execute(service.execute,  $\{< parameter, value >\}$ )
6:   else
7:     if service.call  $\neq$  null then
8:       // create a new URP agent to handle the call
9:       URPi.call(service.call,  $\{< parameter, value >\}$ )
10:    else
11:      // the service is considered to be a search service
12:      formalQuery  $\leftarrow$  GenerateFormalQuery(service,  $\{< parameter, value >\}$ )
13:      URP.execute(formalQuery)
14:    end if
15:  end if
16: end for

```

which will in turn create a URM agent to handle each of the different services involved in such a plan.

- Lines 7-8 If the service is provided by a SHERLOCK controlled object (e.g., a specific taxi or camera or even an external web service) or a third party provider, there is an access mechanism defined (modeled through the *call* property in the ontology), then URM creates a URP to call the service.
- Lines 10-14 Finally, if there is neither an execution plan attached to the service nor a call mechanism defined, then the URM translates the request into a query expressed in SHERLOCK's language and then creates a URP to execute it.

The generation of a formal query for a search service with its associated parameters and values is shown in Algorithm 3:

- Lines 1-18 First, the URM translates the definition of target object(s), the entities that the service returns, by including: (Lines 2-9) 1) the specific target(s) of such service modeled in the local ontology, and the ontological definition of the target of such service (remember that different SHERLOCK devices could have different information in their local ontologies and thus, the ontological definition might be needed to understand the request); and

Algorithm 3 GenerateFormalQuery(*service*, {< *parameter*, *value* >})

Input: *service* is a SHERLOCK service and {< *parameter*, *value* >} is a set of tuples containing a parameter and the value selected by the user each.

Output: Generates a formal query expressing the user request in SHERLOCK's language

```

1: query ← "SELECT ?resultObj"
2: // Generate the Target Object Definition part of the query
3: ODpart ← "OD {"
4: // Add the object(s) returned by the service, as defined in the ontology
5: for each target in service.{target} do
6:   ODpart ← "CASE( Type(?resultObj,target)),"
7: end for
8: // Add the definition of object returned by the service
9: ODpart ← "CASE( PropertyValue(sherlock:service,sherlock:returns,?resultObj)),"

10: // Add the constraints defined by the user
11: for each < parameter, value > in {< parameter, value >} do
12:   //The property that links the object to the parameter could be unknown
13:   ODpart ← "OPTIONAL( PropertyValue(?resultObj,?propertyi,?parami),
14:     PropertyValue(?parami, propNamei, parameter.name),
15:     PropertyValue(?parami, propValuei, value))"
16: end for
17: ODpart ← "}"
18: //Generate the Location of Interest Definition
19: LIpart ← ""
20: for each loc in {< parameter, value >} do
21:   LIpart ← "WHERE{ LI{FILTER(geof:sfWithin(?resultObj,"
22:   if loc.distance!=null then
23:     // the user defined and area with a buffer
24:     LIpart ← "geof:buffer(loc,loc.distance.value,loc.distance.unit))}"
25:   else
26:     LIpart ← "loc)"
27:   end if
28:   // Include the Target Object Definition in the WHERE
29:   LIpart ← "ODpart }"
30:   if more locations remaining then
31:     LIpart ← ", OR "
32:   end if
33: end for
34: //Include the rest of the content in the query
35: query ← LIpart

```

(Lines 10-17) 2) constraints to fulfill the parameters and values that the user selected (which are included as optional to maximize the chances of obtaining results for the user even if they do not fulfill completely her demands).

Lines 19-34 Then, the URM translates the definition of the location of interest using the special location parameters included by the user, in the case of a LBS. As explained in Section 4.2.1, a **WHERE** clause is generated for each different location selected by the user or modeled in the service definition and the previous definition of the target objects are included in them.

The next step is to process the user request which might be: 1) a SHERLOCK query, 2) a call to an external service provided by an external source, or 3) an execution plan of a composed service. We detail the processing of a user request by the User Request Processor agent in the next section.

6.2 Request Processing

In this section, we detail the tasks carried out by the User Request Processor agent (URP from now on) introduced in the previous section. In particular, we explain how a URP agent deals with the three different types of user requests.

As we have seen in the previous section, a URM agent creates a URP agent to delegate the management of three different types of requests (see Figure 6.3), namely:

1. *Processing of a SHERLOCK query* against the local knowledge on the device and external sources.
2. *Invocation of external services* including services provided by third party providers external to the system and SHERLOCK objects.
3. *Execution of service plans* composed of a workflow of atomic services.

The rest of the section is dedicated to explain the processing of each of these three types of user requests. First, we will introduce the processing of SHERLOCK queries which involves the creation of a network of mobile agents which find the information wherever it is (we will detail this process in Chapter 7). Second, we will explain how SHERLOCK processes simpler requests involving the invocation of external services or execution plans.

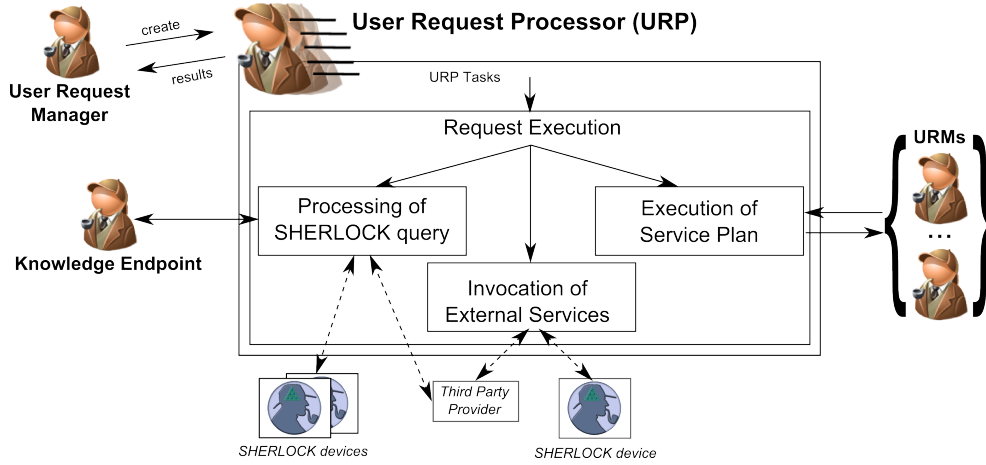


Figure 6.3: User Request Processor tasks.

6.2.1 Processing of SHERLOCK Queries

When it comes to processing a SHERLOCK query, the URP is able to consider three different sources (i.e., the device's local knowledge, the known third-party services, and the network of SHERLOCK devices) to maximize the chances of retrieving a valid answer. To do so, the URP follows the flow diagram shown in Figure 6.4:

- Querying the Local Ontology:** First, the URP executes the query against the local ontology on the device (through the Knowledge Endpoint agent) to try to answer it with the already gathered knowledge. Apart from retrieving the data that comprise the possible answer, the URP also needs to check their temporal validity as they might not be up to date. This checking is done using the timestamps associated to the data, and what is valid differs depending on the properties of queried properties and the result². For example, if the result has been defined as static in the ontology (e.g., the location of the MoMa museum in New York), the gathered information is probably still valid; while if the result is more dynamic, (e.g., the position of available taxis around the user), the gathered information has to be quickly considered out of date, and therefore, not valid.

²In our ongoing prototype, the temporal threshold is a fixed number, but more sophisticated approaches might be applied.

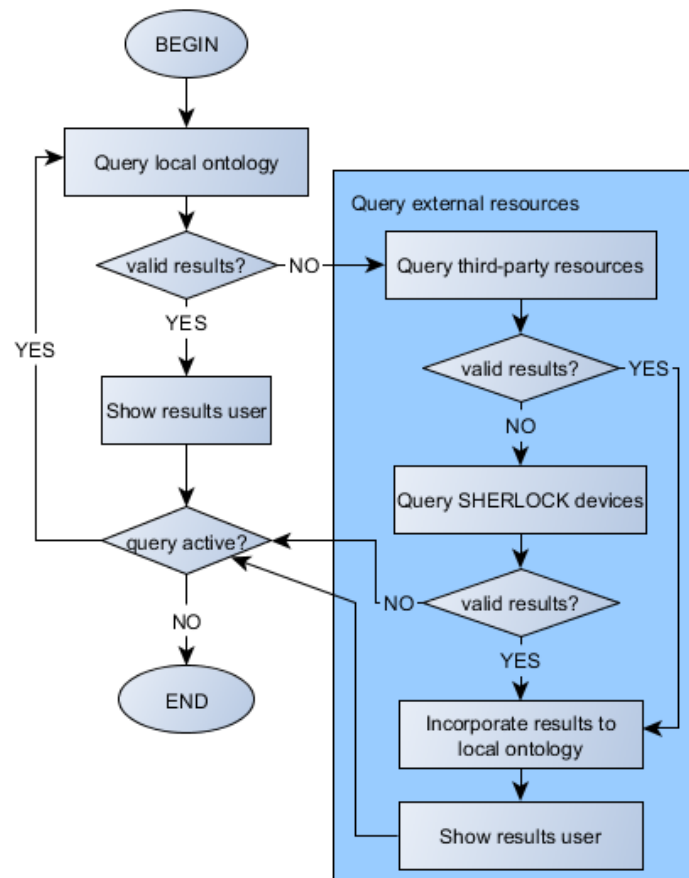


Figure 6.4: Flow diagram of the execution of a SHERLOCK query.

- Querying Third-Party Sources:* If the local ontology does not contain any result or the results are outdated, the URP tries to get this information from external sources. SHERLOCK's next option is to use external third-party services which are capable of obtaining the requested information. Thus, the URP searches services in the local ontology which belong to the same service family and thus, return similar information (i.e., the information returned has the same type than the information returned by the family). This is done using the following query with the service selected by the user:

```
SELECT ?service, ?provider
```

```

WHERE{
  OD{
    Type(?service, <selected service>),
    PropertyValue(?service, provider, ?provider),
    Type(?provider, ThirdParty)
  }
}

```

Note that this step is performed only if the user selected an atomic service which involves a SHERLOCK query in the first place. If the user posed a query directly to the system, trying to find external services to fulfill the user requirements might not be trivial³. Therefore, in our current approach, this step is not supported for queries posed directly to SHERLOCK.

If the service search is successful (i.e., SHERLOCK knows about services in the same family as the user's selected one), the URP executes the service(s) obtained one by one until it gets results back. For that, a URM is created to handle each service. As in the previous case, the URP checks the validity of the results returned by the third party service.

- *Querying SHERLOCK devices:* Finally, if the information could not be obtained from the local ontology nor external sources, the URP has a last mechanism to obtain results: deploying a network of mobile agents which will try to find the information from other SHERLOCK-enabled devices. For that, the URP creates helping agents which autonomously move toward the sources of information defined in the query. These agents are able to monitor an area associated with a request, if any, and find devices in it to communicate with them. Through this communication, agents execute the query against the devices' knowledge, and can even deduce extensions of the request and ask them to execute the extended service.

Querying other SHERLOCK devices is a complex task which we detail, including the creation and maintenance of the network of mobile agents, as well as the tasks performed by each agent in it, in the following chapter (see Chapter 7). In the following section, we explain how SHERLOCK queries external sources and deals with requests defined as a plan of services.

³We do not aim at addressing the problem of service equivalence and matching, which has been thoroughly studied in the literature.

6.2.2 Invocation of External Services

How SHERLOCK processes a service invocation comes determined by its provider. Innerly, the system distinguishes two different types of service invocations: 1) invocation of services provided by entities which are outside SHERLOCK itself (we call such services *third party services* as they are not executed within SHERLOCK devices), and 2) invocation of services provided by devices that have SHERLOCK installed on them (see Section 4.1.2 for examples of such services). They are defined by the accessing mechanism included using the *call* property of each service, and the differences in the nature of the providers make the system handle them differently.

- *Third Party Services*: The access information is codified within the value of the *call* property, and it is dependent on the different APIs that external services might expose. In the current prototype, REST Web Services can be accessed by adding a parameterized URL to the ontology where SHERLOCK includes the actual parameter values. Also, SPARQL queries are admitted in our current implementation against third party knowledge bases where SHERLOCK fills in the values that the user selected. Notice that these services can be continuously invoked if they have been defined as continuous in the ontology.

In the presence of connectivity issues, the URP agent follows a best-effort policy to be able to invoke the defined service. Thus, it is able to create and send *Remote Request Execution* agents (RREs) to SHERLOCK devices in its range looking for someone with connectivity and willing to provide access to such a service. These RRE agents will move to appropriate devices autonomously (as we will explain in Chapter 7) and go on communicating the retrieved results to its associated URP until the request lifetime is expired (or canceled).

- *SHERLOCK Device Services*: They are to be invoked at the target devices, which are the actual ones that expose the services on behalf of the objects. To define such services, the *call* property adopts special URIs:

$$\textit{sherlock} : // < instance_name > / < service_name > \\ ['?'[ParamName : ParamValue]+]?$$

where $< instance_name >$ is the identifier of the object which is being invoked, $< service_name >$ is the name of the service being invoked,

and $[ParamName : ParamValue]$ are the parameters to be passed to such a service. In this case, if the SHERLOCK device has direct connectivity with the target device, the invocation is similar to the previous case. However, if it does not, as the URP is trying to contact a known SHERLOCK device the agent platform will take care of delivering the message as this is one of the main tasks of an agent platform.

Invoking services provided by SHERLOCK objects/devices might require of an interaction with the owner of the device. For example, a service to take a picture provided by devices equipped with cameras might require the user to point the camera in a certain direction and then click the photo. This interaction is modeled in the ontology and to fulfill it the URP creates a *Human Interaction Manager* agent (HIM). HIM agents are mobile agents that move to a device carrying the specification of the different interfaces that the ADUS agent on the user device will show to the user. After getting the information from the user, a HIM agent communicates it to its corresponding URP agent.

6.2.3 Execution of Service Plans

The URP is also able to orchestrate a service plan involving one or more atomic services (which can be SHERLOCK queries or invocations of external services). To handle this kind of requests, the URP checks the execution plan of the service modeled in the ontology. This plan is an XPDL specification which is processed with the help of a BPMN workflow module. During its execution, the URP creates URM agents to handle each atomic service, as each atomic service might require further information from the user.

In Figure 6.5, we can see an example of an execution plan S where URM_s created URP_s to process it. Let's imagine that this plan represents a service to alert the firefighters and volunteers suppressing a wild fire about a possible danger. The execution plan would involve to find all the firefighters and volunteers, and then invoking the service to alert each of them:

- The execution plan demands the execution of atomic services S_1 and S_2 in parallel, as defined in the BPMN workflow attached. Therefore, URP_s creates URM_1 and URM_2 to execute these services, respectively.
- After generating the request (which might involve user interaction), each URM creates a new URP to handle the service (i.e., URP_1 and URP_2 are created).

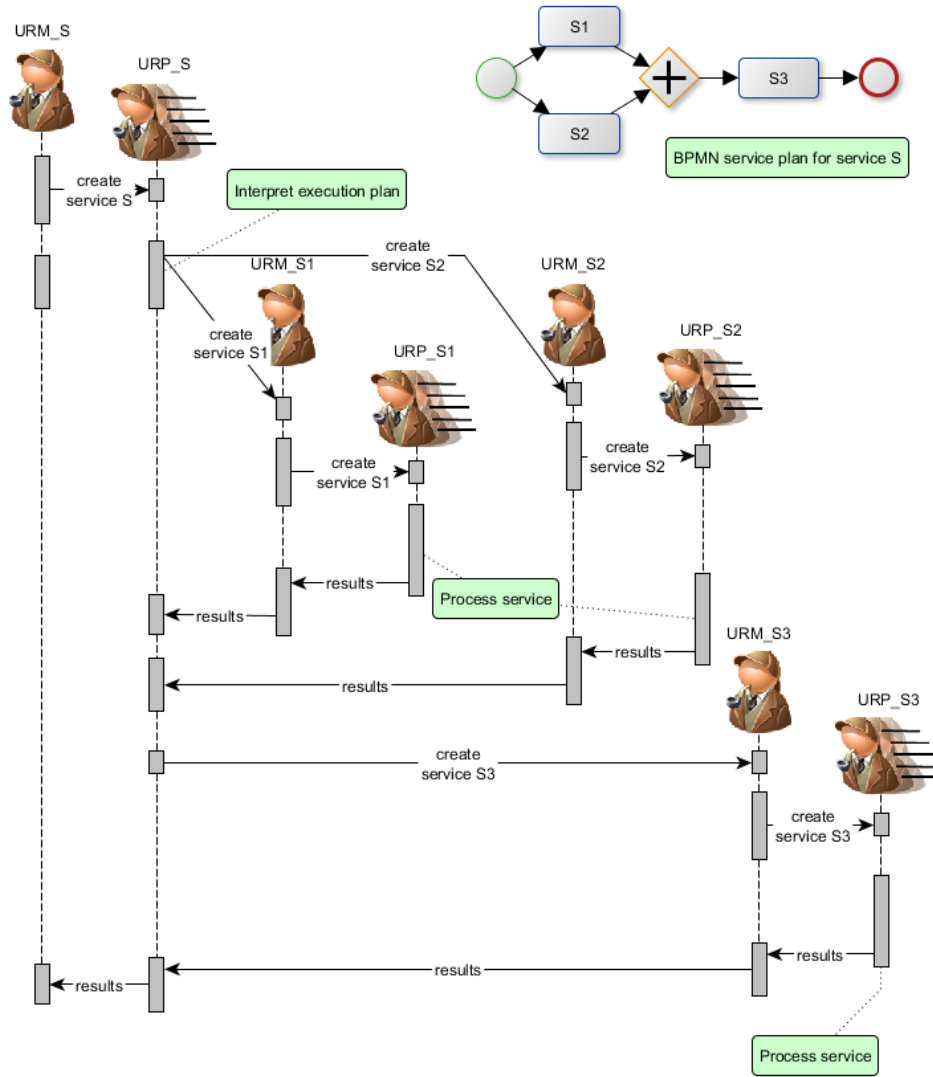


Figure 6.5: Sequence diagram of the execution of a service plan.

- The newly created URP agents process their requests and communicate the results up in the hierarchy to reach URP_S which is in charge of the execution plan.
- Finally, after getting the results from URM_1 and URM_2 , URP_s creates

a new URM to execute the last sequential service in the plan, S_3 .

6.3 Related Work

SHERLOCK's main goal is to provide users with multiple services, based on their context, helping them to express their information needs and keeping the knowledge about available services updated. Up to the authors' knowledge, no other work has proposed a general and flexible system based on semantics to capture user information needs and process them against different sources. Therefore, we will provide an overview of contributions to some specific research areas related to our proposal. First, we present works focused on providing LBS, which SHERLOCK offers too. Then, we present Service-Oriented Architectures, which are focused on providing services, which are similar to the concept of external services in SHERLOCK.

6.3.1 Location-Based Services

Location-Based Services (LBS) have been defined before as “services that integrate a mobile device's location or position with other information so as to provide added value to a user” [SV04]. There are plenty of applications in the literature to provide users with specific location-based services [RGKR07]. For example, taxi searching [SCC10], helping firefighting [JCHWTL04], detecting nearby friends [AEMPW07], or multimedia retrieval in sport events [IMIYLM12], among many others. Also, there are some proposals of architectures to provide LBS. For example, in [BMJ07] an architecture to support LBS applications is presented. The Base Stations (BS) serving cells in a cellular network contain a geolocation server and database that gathers information about mobile devices in the area and their requests. This way, when a mobile device connects with a BS and executes a service registered in the local registry, the server can execute the service using the information in the database and return the result to the mobile device. In [DA11] a LBS system is presented with a similar decentralized architecture. In this system, a local registry is placed in each cell of the cellular network system which enables providers to register their services. The system running on each Base Transceiver Station (BTS) which serves a specific cell broadcast the information from its local registry to devices connected to the BTS. Then, mobile devices can execute a call to specific services. The main difference between these approaches and SHERLOCK is that their decentralized architecture is based on a set of BS which maintain information about objects and services in their cell whereas SHERLOCK do

not assume the existence of a fixed infrastructure. Also, in this approaches the mobile devices relies on the BS for the execution of the service whereas in our approach the device itself handles it. Finally, SHERLOCK also helps the user to express her information needs and integrates new information about services using semantic techniques for interoperability.

6.3.2 Service-Oriented Architectures

Context-aware frameworks simplify the development of context-aware applications/services (see [BDR07] for a survey on context-aware systems). For example, the Service-Oriented Context-Aware Middleware (SOCAM) architecture [GPZ04] supports the building of context-aware mobile services. SOCAM is based on a centralized server which gathers context data from context providers and offers it to clients. Context-aware services can be built by defining rules which specify under which circumstances an action has be performed. SOCAM uses a set of OWL ontologies for modeling the context information that context-aware services can use.

6.4 Summary of the Chapter

In this chapter, we presented the handling of user information requests by SHERLOCK. First, we focused on the translation of the user information needs into a formal request. In this step, we explained how the system obtains services that might be interesting for the user regarding her input (e.g., the selection of a location or an object) and her context. Then, we showed how the system helps the user to fill in parameters associated to the selected service that will be used as constraints over the information to obtain. Finally, we explained how the system generates the appropriate service request/invoke which might involve the generation of a formal query in the case of, for example, location-based queries. After the generation of the request, we explained the steps involved in its processing. We showed how the User Request Processor agent handles the different types of requests supported by SHERLOCK. Thus, for the processing of SHERLOCK queries, we introduced our approach to try to obtain results from different sources including the local knowledge on the device, third-party sources, and finally other SHERLOCK devices. We also explained how the system processes requests which involve the invocation of external services provided by third-parties (e.g., web services) or even by SHERLOCK devices (e.g., a service to take a picture), and the execution of a service plan composed of one or more atomic services.

Chapter 7

Processing of SHERLOCK Queries

When processing a user request in the form of a SHERLOCK query (see Section 4.1.2) the URP agent executes it against the local knowledge on the device and external third-party sources as explained in the previous section. In the event of not obtaining information from such sources the URP tries to obtain the information from other SHERLOCK-enabled devices. This implies finding devices around the user or in a location of interest (depending on whether the query is based on a location or not) and executing the query against their local ontologies. For this task the URP delegates in a network of mobile agents, which are programs that execute in contexts called *places* and can autonomously travel among devices in the scenario resuming their execution on the destination. Considering any device in the scenario as a potential place where an agent can execute their code has many benefits as explained in [LO99]. This adaptive behavior of the hierarchical mobile agent network deployed by a URP, where each mobile agent executes on the device that minimizes the computing and communication delays while trying to accomplish its goals, is specially important in highly-dynamic environments where new devices can appear/disappear or change their capabilities (e.g., a laptop can use Wi-Fi and then change to a wired connection).

The URP agent, which is in charge of processing the query, obtaining results, and sending them back to its URM agent, creates two types of mobile agent networks to help it regarding the type of query to process (see Figure 7.1):

- *Non-location-based queries*: *RRE* agents are created by a URP agent on devices in communication range to execute the query against the local

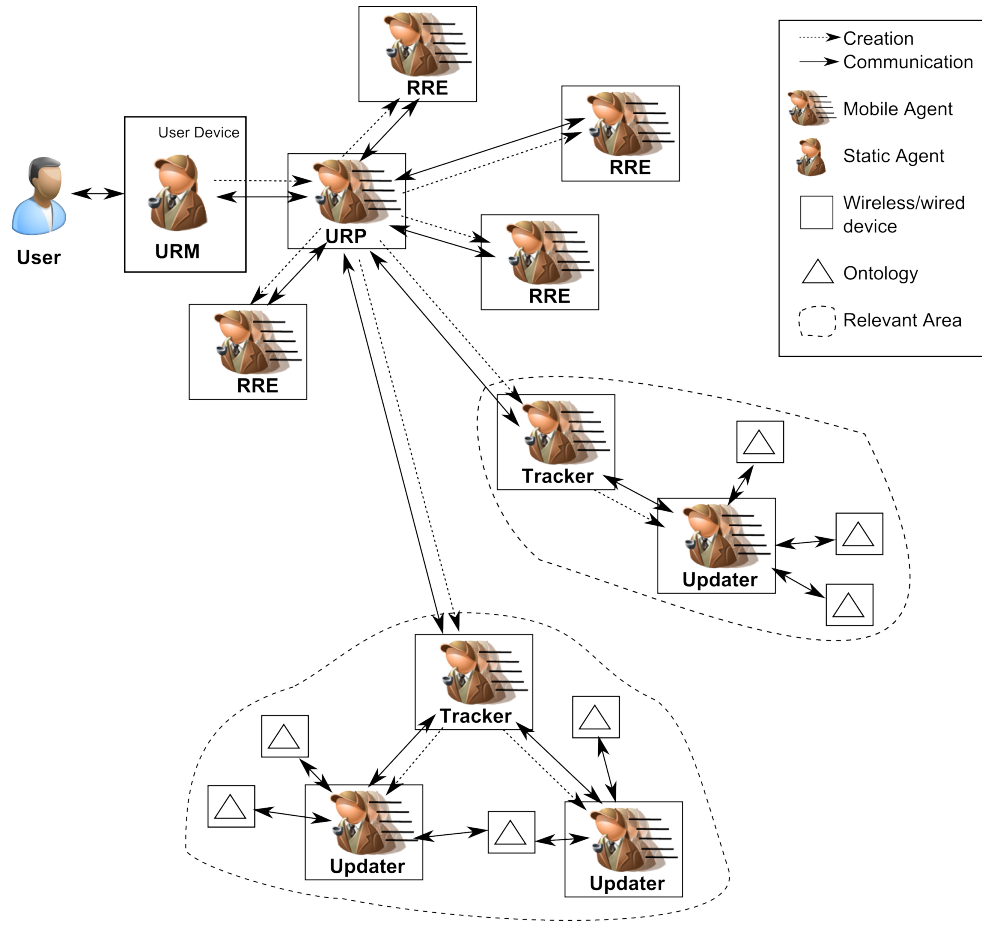


Figure 7.1: Mobile agents to process a user query and obtain results.

ontology of such devices, recursively if needed (RRE agents move from there to other near devices).

- *Location-based queries:* *Tracker* agents are created by a *URP* agent to *monitor* locations of interest for the query. These *Trackers* move toward the location of interest, jumping from device to device, and once there, they create *Updater* agents on devices in range with devices in the location of interest. *Updater* agents continuously execute the query against devices in their communication range, if needed, and obtain results that communicate to their *Trackers*.

In the following sections we detail the tasks¹ assigned to each agent to fulfill their goals.

7.1 User Request Processor Agent: Coordinating the Network of Agents

In addition to the tasks presented in the previous section, the URP agent performs the following tasks when processing a SHERLOCK query against other devices (see Figure 7.2):

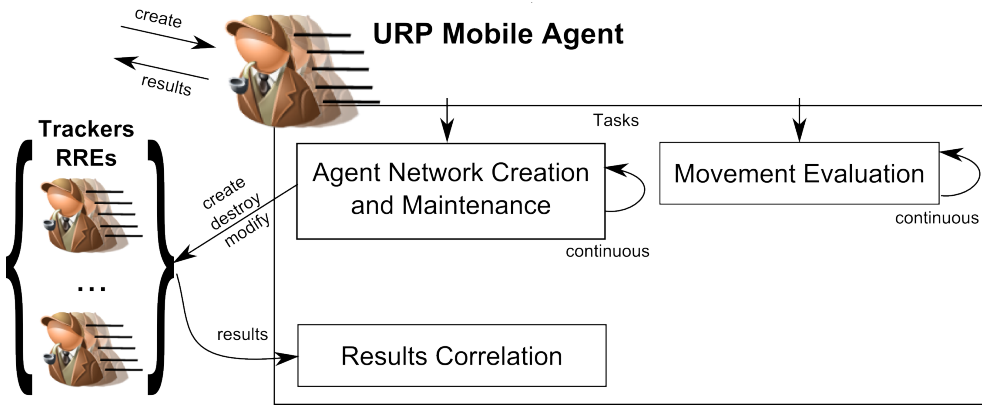


Figure 7.2: User Request Processor tasks.

1. *Creation and Maintenance of a Helping Agent Network:* As mentioned before the URP creates RRE agents for queries which are not related to a location and a more complex network of Tracker agents for location-dependent queries.
2. *Results Correlation:* The information returned by the helping agents has to be correlated in order to detect redundant information.
3. *Movement Evaluation:* The URP evaluates whether other devices in range provide better features to execute its tasks.

In the following sections we details the previous tasks.

¹The tasks are ordered according to their importance.

7.1.1 Creation of a Network of Helping Agents

There are two types of queries processed by the URP agent: location-based queries and queries without an explicit location. In the following we explain how the network of helping agents is deployed to process each type of query.

Executing queries without a location of interest In queries where there is not a explicit location of interest defined (e.g., “What is the age of Barack Obama?”) there is no evidence of which device could contain the information that the user needs. Therefore, we advocate to ask devices in range. So, the URP creates RRE agents in devices in range with the user’s device to execute the query against their ontologies:

1. The URP accesses the local knowledge about devices around maintained by the CU agent. In a greedy fashion, the URP agent creates an RRE agent on each of these devices and assigns them a deadline to obtain the results. Notice that, the URP limits the number of active agents it can handle to avoid creating a very extended network of RRE agents.
2. Once the RRE agents are created on the devices around the URP, they execute the query against their local ontologies by communicating with their local KE agents. Then, each RRE agent returns the results obtained to its URP by the given deadline (we will explain how the deadlines are assigned in the following section).
3. Finally, if the results obtained by the network of RRE agents are not enough, the URP can ask RRE agents to move to other devices. For this, the URP obtains the list of devices in range with the devices considered previously. Then, the URP asks each RRE to move to a discovered device and creates new RRE agents if there are more new devices discovered than existing RRE agents (taking into account the limit mentioned before).

Executing location-dependent queries As location-dependent queries are related to a target location, it would be more probable that devices nearby such a target location have the requested information. Therefore, the URP executes the query against devices near such location with the help of Tracker agents which will monitor the location(s) of interest defined in the LI (Location of Interest definition) part(s) of the query. Therefore, the first step is to obtain information about the coordinates of such a location of interest so Trackers could be sent there (see Algorithm 4), if possible.

A location-dependent query might contain more than one location of interest as explained in Section 4.2.1. Therefore, for each *WHERE* and for each *LI* clause the URP obtains its coordinates so a Tracker agent can be sent there for its monitoring. The URP obtains this information as follows:

- Executing a query against the local ontology which could contain the information (Lines 4–6 in Algorithm 4).
- If the local ontology does not contain the information, creating a URM agent to obtain the information from other sources (Lines 7–11 in Algorithm 4).
- If neither the local ontology nor other sources contain the information, processing the query as non location-dependent (Lines 12–15 in Algorithm 4).

Algorithm 4 Algorithm followed by a URP agent to create Trackers to monitor the location(s) of interest of a query.

Input: *QueryArea* is the query to process.

```

1: // A query can have more than one location of interest
2: for each WHERE in QueryArea do
3:   for each LI in WHERE do
4:     // Execute a query against the local ontology to find its coordinates
5:     QueryLIi = extractQueryLI(QueryArea, LIi)
6:     coordinates = KE.obtainCoordinates(QueryLIi)
7:     if coordinates == null then
8:       // The coordinates are not in the local ontology
9:       // Create a URM to find the coordinates in other sources
10:      coordinates = URMi.executeQuery(QueryLIi)
11:    end if
12:    if coordinates == null then
13:      // The URM could not find the coordinates
14:      // Process the query as a non location-based query
15:      createRREs(QueryLIi)
16:    else
17:      // The coordinates could be found
18:      // Create Tracker to monitor the area
19:      createTracker(coordinates, QueryLIi)
20:    end if
21:  end for
22: end for

```

First, to obtain the coordinates of a location of interest from the local ontology the URP executes a query against the local knowledge through the KE agent. If no coordinates are obtained for the location of interest from the local ontology, the URP tries to obtain them from other sources. For this task, SHERLOCK uses itself so the URP creates a new URM agent in charge of processing the previous request against other devices. After this step it could happen that the coordinates are not yet found after trying other external sources, then the last option of the URP is to execute the query as if it was a non location-dependent query using RRE agents as explained before. This could mean that no results for the query might be found as it is less probable that devices around the URP would have the requested information. Nevertheless, queries are reevaluated continuously, until the user cancels them, which means that in a next iteration the URP might obtain the coordinates from one of the sources. In this case, a Tracker will be sent to the location and the RRE agents will be destroyed.

If the coordinates of the location of interest are found, then the URP creates a Tracker to monitor such location and provides it with the coordinates, as well as the query to execute against devices in such location. It might happen that the coordinates of the location of interest are outdated (e.g., if the area changes if it is associated to a moving object). Again, as queries are reevaluated continuously, if a new location is found by the URP in a iteration of the execution, it communicates the new location to the Tracker so it can change its destination. In addition, a Tracker might find the interesting information near to the last known coordinates of the location of interest.

7.1.2 Maintenance of the Network of Helping Agents

The URP agent uses the autosynchronization technique explained in [IMI08] for establishing deadlines for its helping agents (RREs or Trackers). This technique enables agents to assign deadlines to their helping agents which, in the case of continuous queries that have to be reevaluated, are dynamically adapted to the current situation by taking into account the delays experienced by their cooperative agents and the environmental delays (e.g., network delays). The autosynchronization technique ensures that the agents will return information at the frequency requested and in situations where this is not possible due to challenging environmental conditions, the technique offers several approaches to deal with it regarding the status of the agents. The details of this process can be found at [IMI08]. Also, when creating a helping agent, a time to live is assigned to ensure that in the case of losing the connection with it for a long

time the agent will destroy itself.

When managing a network of agents to process a location-based query, it might happen that a Tracker could be overwhelmed trying to fulfill its task. This can happen in scenarios where the location of interest to cover is too large or heavily populated with SHERLOCK devices. Therefore, such a Tracker agent might not be enough to correlate the information obtained by its helping agents and start missing deadlines assigned by the URP agent. In these situations, if the URP detects that a Tracker is overwhelmed (e.g., if the Tracker is not achieving its assigned deadlines), it will create a new Tracker to help it and divide the area into two parts. The URP will modify the target destination of the existing Tracker (a centroid of one of the new subareas). Also, the URP can destroy Trackers if they are not needed anymore (e.g., if they are obtaining only redundant information).

7.1.3 Results Correlation

The URP correlates the results obtained by its helping agents as they might obtain the same information from different devices. In the correlation process, the URP detects this redundant information obtained by more than one helping agent and discards the information that might be outdated using the timestamp associated with it. We consider that devices share real information about objects and therefore, in the case of receiving the information about the same object, all the information obtained is valid although some (or even all) could be outdated due to the dynamically changing scenario. Also, in the correlation process an agent can discard information retrieved by its helping agents according to its similarity with the expected result. For example, if the user wants to get a limited set of results (e.g., five pictures which fulfill certain parameters) the URP can filter out these results that are more similar to the expected result before sending them back to its URM agent.

7.1.4 Movement Evaluation

As all the mobile agents in SHERLOCK, a URP agent locates itself in the best device possible to perform its goals, among the different possibilities in range. For that, it considers other devices as possible execution places following Algorithm 5:

1. The URP obtains the list of devices that can be communicated by the device it is residing in. This is done by executing a query against the local ontology of the devices (lines 2–3 in Algorithm 5).

Algorithm 5 Algorithm to evaluate if moving to other device is needed

Input: *TargetLocation* is the target location and *currentDevice* is the device where the agent is executing currently.

```

1: while alive do
2:   // obtain the list of devices in communication with the host
3:   devicesComm  $\leftarrow$  KEcurrentdevice.obtainDevicesAround(TargetArea)
4:   candidateMovement = currentDevice
5:   candidateDestDegree = computeDestDegree(currentDevice)
6:   for each device in {devicesComm} do
7:     // check if moving to the device is interesting
8:     destDegree  $\leftarrow$  computeDestDegree(device)
9:     if destDegree > candidateDestDegree then
10:      candidateDestDegree = destDegree
11:      candidateMovement = device
12:     end if
13:   end for
14:   // move if needed
15:   if candidateMovement != currentDevice then
16:     move(candidateMovement)
17:   end if
18: end while

```

2. For each device the URP computes a “destination degree” which models the appropriateness of the device for the URP goals (lines 4–13 in Algorithm 5).
3. If any device is more appropriate than the device where the URP is currently residing in, it moves there (lines 14–17 in Algorithm 5). If the communication with that device is not possible then the next most interesting device will be considered (in the worst scenario the URP will not move to any device in this iteration but this process is performed continuously).

Regarding the method to compute the destination degree, the URP considers the following parameters of the device:

1. Current available resources of the device: Powerful devices are preferred for most of the tasks (such as correlation of results) but the current load (e.g., in terms of available CPU, memory, and battery) is an important factor to take into account. As the capabilities of a device (e.g., processor load, remaining battery time, communication range, etc.) will

be considered when choosing a destination, fixed devices with wired communication will be preferred if available.

2. Agent-specific goals: In the case of the URP it has to maintain itself as close to the device that posed the query as possible to communicate it the results obtained.

The movement formula in our prototype obtains a value that models the appropriateness “destination degree” of a device by taking into account these two parameters and assigning them a weight (relative importance attached):

$$Dest(d) = w_c * Dest_{features}(d) + w_g * Dest_{goals}(d) \quad (7.1)$$

where $Dest_{features}(d)$ represents the first parameter and takes into account the current features of the device as follows:

$$Dest_{features}(d) = w_{perf} * d_{perf} + w_{CPU} * d_{CPU} + w_{mem} * d_{mem} + w_{battery} * d_{battery}$$

where d_{perf} represents the performance of the device according to a benchmark (such as, for example, Antutu or Passmark), d_{CPU} represent the load of the processor(s), d_{mem} the available memory in MB, and $d_{battery}$ the remaining battery as a measure of time left. Also, each factor has a weight, w_{perf} , w_{CPU} , w_{mem} , and $w_{battery}$ (with $w_{perf} + w_{CPU} + w_{mem} + w_{battery} = 1$), which can be modified depending on the situation (e.g., if the agent needs more CPU for a complex operation). Notice that in the case of missing information about a parameter (e.g., available CPU) the device could obtain a score lower than other devices.

Regarding $Dest_{goals}(d)$, which represents the second parameter of Formula 7.1, the URP agent computes this factor using the distance (in our prototype the euclidean distance) to the location of the user device: $w_{dist}(1 - distance(userDev, loc_d))^2$. Notice that other parameters could be also considered such as the speed and direction of the device. They would have to be included in the formula which can also be replaced by more sophisticated approaches to evaluate the best destination. The important aspect is that the system is independent of the movement formula.

To obtain the information about devices which can be considered by the previous formula the URP executes a query against the local ontology of the

²The distance function returns a value between 0 and 1, being 1 the exact target location and 0 a location farther than a threshold.

device in which it resides at the moment (see Figure 7.3). Remember that the Context Updater (CU) agent on the device is in charge of updating this information in the local ontology by querying devices around continuously (see Section 5.1). The URP agent considers as possible execution places devices around the device it is residing in and around the device that posed the query. Also, a range is used to limit the number of results of the query and this value is increased automatically if no results are obtained.

```

SELECT ?name, ?IP, ?lat, ?lon, ?CPU, ?memory, ?battery
WHERE{
  LI{
    FILTER(geof:sfWithin(?thing, geof:buffer(<UserDeviceCoordinates>, 300, 'm'))
  },
  OD{
    Type(?device, sherlock:SHERLOCKdevice)
  },
  PropertyValue(?device, geo:lat, ?lat),
  PropertyValue(?device, geo:lon, ?lon),
  PropertyValue(?device, sherlock:name, ?name),
  PropertyValue(?device, sherlock:IP, ?IP),
  PropertyValue(?device, sherlock:Available_CPU, ?CPU),
  PropertyValue(?device, sherlock:Available_Memory, ?memory),
  PropertyValue(?device, sherlock:Available_Battery, ?battery)
}

```

Figure 7.3: Query to obtain the features of devices around a coordinate.

7.2 Tracker Agent: Monitoring a Location of Interest

A Tracker agent is in charge of monitoring a location of interest. This means trying to find as many devices as possible inside it (ideally all the devices inside) and executing the user query against their local ontologies. To achieve this goal, and similarly to the URP agent, a Tracker performs the following tasks:

1. *Movement Evaluation*: The Tracker agent has to move itself near to the location of interest assigned by the URP.
2. *Helping Agent Network Creation and Maintenance*: In the case of the Tracker it creates a network of Updater agents which will communicate with devices inside the location of interest and execute the query.

3. *Results Correlation*: The information returned by the Updater agents is correlated by the Tracker in order to reduce the information sent to its URP.

7.2.1 Movement Evaluation

First, the Tracker moves toward the location assigned using the same approach presented for the URP agent (see Algorithm 5). However, the Tracker agent considers as possible execution places devices around the location of interest instead of around the device that posed the query. Therefore, it specializes the movement formula presented before (Formula 7.1) by taking into account two elements: 1) the distance to the centroid of the area to cover, and 2) the communication with its helping agents, as follows:

$$Dest_{goal}(d) = w_{dist} * (1 - distance(target, loc_d)) + w_{comm} * (1 - \sum delay(Updater_i))$$

where *target* are the coordinates of the centroid of the location to monitor, *delay* models the communication delay with a helping agent, and w_{dist} and w_{comm} are the weights assigned for each factor ($w_{dist} + w_{comm} = 1$). When the Tracker is created the location to monitor is the location of interest and thus *target* is the location of its centroid. However, whenever the Tracker agent creates Updater agents and they start communicating with devices in the location of interest, *target* becomes the centroid of the “uncovered area” which is the part of the location of interest that the current Updater agents cannot cover because of the communication range of the devices they are residing in. Thus, the Tracker will try to remain as close as possible to its helping agents while trying to be close to the uncovered area. Notice that, the devices considered as possible execution places could be connected through ad hoc MANETs (e.g., devices around its location) or even wider area networks (e.g., imagine an Tracker interested in moving to another city, the local ontology might contain information about SHERLOCK-enabled devices available there).

7.2.2 Creation of the Network of Updater Agents

The Tracker agent has to find and communicate with as many devices in the location of interest as possible to increase the chances of returning a complete answer to the URP (and thus to the user). For this, the Tracker finds devices inside the location whose communication mechanisms enable them to communicate with as many devices as possible. A network of *Updater* agents

is created on these devices to maximize the area of the location of interest monitored or *covered*. Figure 7.4 shows an example of a situation where the Tracker in charge of monitoring a location of interest created three Updater agents. Notice that because of the communication range of the devices where the Updater agents have been created, the Tracker covers most of the location and thus, receives information from all the devices inside it.

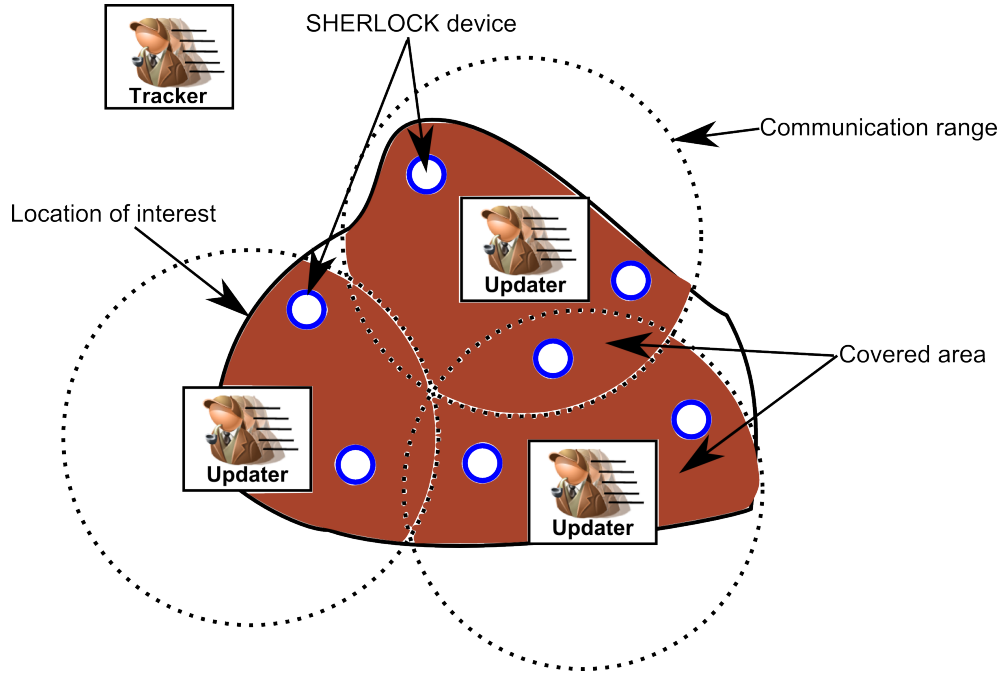


Figure 7.4: Example of the monitoring of a location of interest.

At the same time that the Tracker agent is communicating with the Knowledge Endpoint in SHERLOCK devices in range to evaluate if moving to them is needed, the Tracker evaluates if a helping agent, from now on Updater agent, is needed on that device. The task to start the creation of the network of helping agents is evaluated continuously until the area is completely covered, eventually, or no new devices able to cover new parts of the area are discovered. Algorithm 6 explains the behavior of the Tracker which can be summarized in the following steps:

1. Obtain the list of devices that can be communicated from the device the Tracker agent resides in (lines 2–4 in Algorithm 6). This includes devices

Algorithm 6 Algorithm followed by a Tracker to fulfill its goal of covering a certain target area

Input: *TargetArea* is the target area to cover, *QueryArea* is the query to process in such an area, *currentDevice* is the device where the Tracker is executing currently.

```

1: while alive do
2:   // the Tracker is executing in a device and obtains the list
3:   // of devices in communication with the host
4:   devicesComm  $\leftarrow$  KEcurrentdevice.obtainDevicesAround(TargetArea)
5:   // if the Tracker has Updaters created gets devices in
6:   // communication through them
7:   for each updater in {updaters} do
8:     devicesComm  $\leftarrow$  updater.obtainDevicesComm()
9:   end for
10:  // compute the current coverage of the target area by the Tracker
11:  coveredArea  $\leftarrow$  computeAreaCoverage({updaters})
12:  for each device in {devicesComm} do
13:    // check if creating an Updater in the device is needed
14:    // estimate the amount of uncovered area that the device could cover
15:    coverageDevice = estimateCoverage(device, coveredArea)
16:    if device has no updaters AND coverageDevice > threshold then
17:      createUpdater(device, QueryArea)
18:    end if
19:  end for
20: end while

```

which can be reached from the devices where Updater agents are residing it, if any (lines 5–9 in Algorithm 6).

2. Estimate the part of the location of interest which is currently covered by its network of Updater agents taking into account the features of the devices they reside in (lines 10–11 in Algorithm 6).
3. Estimate if any device from the previous list could cover part of the location of interest which is not currently covered by the network of Updaters. This part should be greater than a threshold used to avoid the effort of trying to cover very small parts of the location of interest. In that case, create new Updater agents in such devices (lines 12–19 in Algorithm 6).

To illustrate this process consider the example in Figure 7.5 where a URP agent created a Tracker to monitor a location of interest for a query. In Figure 7.5(b) the Tracker agent has detected several SHERLOCK devices and

the estimated coverage area of one of them covers part of the still uncovered location of interest. So, the Tracker creates an Updater agent on the device. In the next iteration (Figure 7.5(c)) the previously created Updater sends information to the Tracker about new SHERLOCK devices discovered. So, the Tracker evaluates if they are covering part of the uncovered area, this is true for two of them, and the Tracker creates more Updater agents on them.

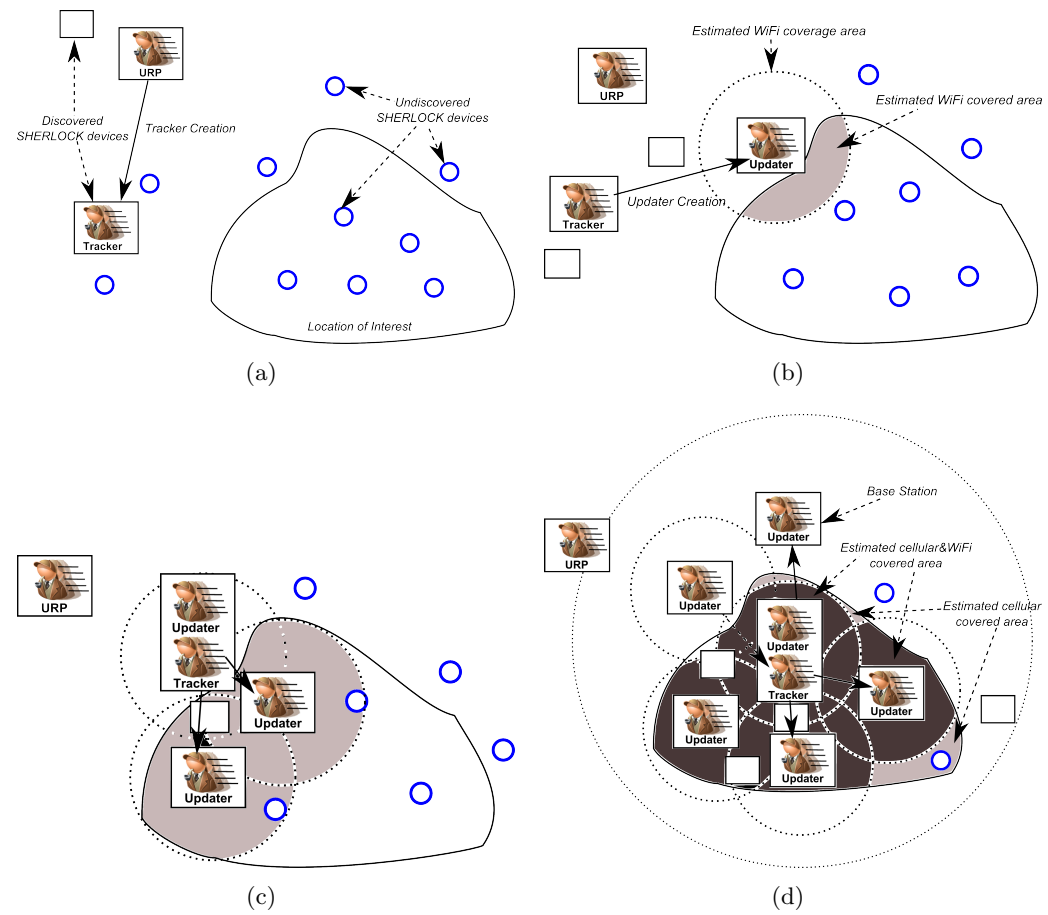


Figure 7.5: Creation of Updater agents by a Tracker agent to cover a location of interest.

Notice that the previous protocol is independent of the communication technology used (Bluetooth, WiFi, cellular, etc.). However these technologies have some differences regarding the communication with devices and their estimated coverage area (see Table 7.1 for a summary). For example, how to estimate the coverage area of a technology differs from cellular to WiFi/Bluetooth (where the manufacturer provides us with a possible estimation). Protocols to improve the estimation of the coverage area are outside of the scope of this work but they could be easily incorporated to the system. While SHERLOCK devices can have all these technologies enabled, it could happen that some of them only have one of them enabled at the same time (e.g., a device might have only WiFi and so only other devices with this mechanism enabled could communicate with them in an ad hoc manner). Therefore, Trackers try to cover the location of interest with all these technologies in mind. For that, Trackers keep information of the amount of the location of interest covered with each technology. For example, in Figure 7.5(b) notice that the estimated covered area is covered using WiFi only because the devices in it only have this mechanism enabled.

Technology	How to discover devices?	How to communicate with devices?	How to estimate coverage area?
Bluetooth	Find devices nearby in discoverable mode	Direct communication	Communication range provided by manufacturer
WiFi	Find networks created by other devices/access points	Direct communication and communication through intermediary access point	Communication range provided by manufacturer
3G/4G	Obtain information from base station	Communication through intermediary access point	Information from the base station, information from cell

Table 7.1: Information of the different communication technologies considered by a Tracker to cover an interesting area.

In Figure 7.5(d) a new device has been detected by the Tracker which is a cellular Base Station (BS). A BS is able to communicate with mobile devices under their coverage area (e.g., by using 3G) and therefore the Tracker estimates its cellular coverage area. In the example, the Tracker creates an Updater in the BS device and therefore parts of the location of interest area are now covered by WiFi and 3G which means that devices inside them with these technologies would be detected.

Finally, notice that it might happen that a SHERLOCK-enabled device is not discovered by Updater agents and therefore it is not considered by the Tracker. This happens if the device is not currently visible by any device (e.g., if its communication mechanisms are disabled). However, as this process of monitoring the location of interest is performed continuously, the device would be detected if at a given moment it becomes visible to others.

7.2.3 Updater Agents Network Maintenance

Apart from creating new Updaters to cover the area assigned, the Tracker maintains its network of Updaters regarding their load (similarly to how the URP maintains its network of Tracker agents). If an Updater is currently overwhelmed because it has to communicate with many devices it will start missing deadlines and therefore the delay in the communication with the Tracker will be increased. In this situation the Tracker creates another Updater near the previous Updater to help it. Moreover, the Tracker commands an Updater to stop communicating with a certain object, and add it to its “black list”, when other Updaters have been providing better communication with such an object in the past (using a configurable window over the past communications). For example, in Figure 7.5(d) the first Updater that was created in Figure 7.5(b) is covering the same devices that other Updaters but its communication with them is slower. Therefore the Tracker will ask that Updater to include such devices in its black list. Notice that an Updater finishes its execution when it has no objects to monitor (all the objects that can be communicated from the device are in the black list). This way, each Tracker maintains its network of Updater agents in a dynamic way.

7.3 Updater Agent: Obtaining Results

Updater agents, which are in charge of discovering SHERLOCK-devices in the location of interest and executing the query against them, perform four tasks:

1. *Query Execution*: The Updater poses the query against the local knowledge of a device through its KE agent.
2. *Results Correlation*: Similarly to the previous agents, the Updater correlates the results obtained from the different devices.
3. *Query Extension*: The Updater is able to autonomously decide to extend the user request if it is not returning enough results.
4. *Movement Evaluation*: Updater agents try to execute as close as possible to the assigned coordinate/device.

7.3.1 Query Execution and Results Correlation

Once an Updater agent reaches a device it starts their protocol to fulfill their main goal: to obtain results for the given query (see Algorithm 7). An Updater starts executing the query against the local ontology of the device it resides in since the moment it is created (as it is created in a device that might cover the target area). Also, the Updater obtains the list of devices in communication range with the device it is residing in from its local ontology and executes the query against their ontologies (through the KE agent on each device). Finally, when the results from the different devices are obtained the Updater correlates them (to eliminate duplicates or outdated information) and sends it back to its Tracker agent along with the new discovered devices.

7.3.2 Query Extension

Independently of the type of request, Updater agents are able to consider alternative ways of retrieving the required information. After receiving results from a query executed against devices in range and correlating them, the Updater agent autonomously can decide to extend a current request which is not providing enough results. When so, the service extension process (see Algorithm 8) is performed.

Basically, Updater agents exploit three different semantic relationships (i.e., services in the same family, services which return same results, and services which return objects that provide related services) to select new services which could obtain the results needed by the user (see Figure 7.6):

Lines 3-6 If the service that the user selected, which was translated into the location-based query, belongs to a family of services, the Updater selects other services which belong to the same family, retrieving all the instances of

Algorithm 7 Algorithm followed by an Updater to fulfill its goal of obtaining results for the given query

Input: *TargetArea* is the target area to cover, *QueryArea* is the query to process in such an area, *currentDevice* is the device where the Updater is executing currently, and *Tracker* is the Tracker agent which created it.

```

1: while alive do
2:   // the Updater executes the query against the local ontology
3:   // of the device where it resides
4:   results  $\leftarrow$  KEcurrentdevice.execute(QueryArea)
5:   // obtain the list of devices in communication with the host
6:   devicesComm  $\leftarrow$  KEcurrentdevice.obtainDevicesAround(TargetArea)
7:   for each device in {devicesComm} do
8:     // execute the query against each device if they are not in the black list
9:     if device is not in blacklist then
10:      results  $\leftarrow$  KEdevice.execute(QueryArea)
11:    end if
12:  end for
13:  // correlate results obtained and communicate them to the Tracker
14:  results  $\leftarrow$  correlateResults(results)
15:  sendResults(Tracker, results)
16: end while

```

such family (see 1 in Figure 7.6). Thus, if the user selected a particular service which is not returning results, the Updater executes more instances from the same family of services in the device it is residing on. For example, if the user selected the service to find buses and it is not returning any result after some time, the Updater launches other services from the “Find Bus” family (e.g., an external service from the transport organization of the city).

Lines 10-21 Then, the Updater executes services belonging to service families which return the same results than the selected service family. To do so, it consults the ontology to obtain information about the returned class of the selected service, and uses it to obtain service concepts that are constrained to return objects of such a class (see 2 in Figure 7.6). For example, if the user selected the family of services to find pictures and no information is returned, the URM will find other families returning pictures such as, for example, the service to take a picture.

Lines 22-29 Finally, Updater agents also exploit the *provides/returns* chain of proper-

Algorithm 8 ExtendService(*service*)**Input:** *service* is a SHERLOCK service**Output:** Extends the current service to obtain further results.

```

1: suggestedServices  $\leftarrow \emptyset$ 
2: // Check whether the service selected by the user is an instance or a family
3: if service isA instance then
4:   // Search other services from the same family
5:   serviceFamily  $\leftarrow$  KE.DirectType(service)
6:   suggestedServices  $\leftarrow$  KE.RetrieveInstances(serviceFamily)
7: else
8:   serviceFamily  $\leftarrow$  service
9: end if
10: // Search other family of services which could obtain the similar results
11: retObjectClass  $\leftarrow$  KE.RetrieveReturnedClass(serviceFamily)
12: {otherServiceFamilies}  $\leftarrow$  KE.Range(return, serviceFamily.return.range)
13: for each serviceFamilyi in {otherServiceFamilies} do
14:   suggestedServices  $\leftarrow$  suggestedServices  $\cup$  KE.Retrieve(serviceFamilyi)
15: end for
16: // Finally, exploit the provider/return chain
17: // we get first the concept of the objects that provide the family of this services
18: tgtObjectClass  $\leftarrow$  KE.Range(provides, serviceFamily)
19: // we obtain the service families that return this kind of objects
20: serviceFamilies  $\leftarrow$  KE.Range(return, tgtObjectClass)
21: // the user selects a particular service interacting with a HIM agent
22: for each serviceFamilyi in {serviceFamilies} do
23:   suggestedServices  $\leftarrow$  suggestedServices  $\cup$  KE.RetrieveInstances(serviceFamily)
24: end for
25: return suggestedServices

```

ties to obtain services that returns objects which provide related services³ (see 3 in Figure 7.6). In this case, if the service to find pictures is not retrieving any information, the Updater executes a service to find users with mobile cameras, as they are providers of the service take a picture (which could involve user's interaction through a HIM agent as explained in Section 6.2.2).

With this process the Updater selects an extension service that could obtain the same information that the user is requesting. This way, Updater agents autonomously react when they are not able to achieve their goal (obtain the information the user needs) and execute the extension services through the

³Note that, despite we only explore this up to one level, a controlled composition mechanism could be devised to follow this path thoroughly.

(that the user must know) is assumed instead. Even though there are interesting proposals that have considered some semantic aspects (e.g., [YCPSA11] proposes the management of semantic trajectories and [ZL01] presents the concept of semantic caching of location-dependent data), they do not aim at developing a general semantics-based query processing architecture.

The LOQOMOTION system [IMI06] presents a general architecture to process location-based queries. Their approach is also based on the use of a layered hierarchy of mobile agents that move autonomously over the network to track moving objects, correlate partial results, and finally, present and keep updated the answer to the user's query. LOQOMOTION relies in an underlying infrastructure composed of proxies that manage location data about moving objects within their coverage areas. Therefore, in order to monitor a location of interest, their agents traveled to the (already known) proxy that covered it through the fixed network and obtained the results from the database in the proxy (that contained the information about all the objects inside its communication range). The scenario that SHERLOCK considers is more dynamic as proxies might not exist and discovering SHERLOCK-enabled devices and creating ad hoc P2P networks might be needed to process the query. Therefore, we based SHERLOCK's hierarchy of agents to process location-based queries in LOQOMOTION agents (URP –Monitor Tracker in [IMI06], Tracker, and Updater) and adapted it to our requirements. So, we extended these agents with semantic capabilities, such as the ability to understand the location to monitor or results that could be inferred as interesting for the query, and with the capability of adapting themselves to a dynamic ad hoc scenario.

7.5 Summary of the Chapter

In this chapter we explained how SHERLOCK deploys a network of mobile agents to process user queries. We started by introducing the different tasks performed by the User Request Processor (URP) agent which coordinates the complete network. This agent decides which helping agents would be needed to process the query depending on whether it is related to a location or not. Once the network of appropriate helping agents have been created the URP maintains it by increasing/decreasing their number to adapt the network to the current status of the scenario. For location-dependent queries, which are a fundamental building block of Location-Based Services, the URP creates Tracker agents. For these agents, which are in charge of monitoring a location of interest, we explained how they autonomously move toward the assigned location and start collecting data. For the collection task, Trackers create

Updater agents trying to maximize the coverage of their assigned location. Finally, we explained how Updater agents communicate with devices around and inside the location of interest to pose the user query against their local knowledge. We showed how, Updaters autonomously react to situations when these devices do not return results by extending the user query.

Chapter 8

Multimedia Information Management

A relevant feature of a system to obtain information for users would be to support the possibility of managing multimedia information (i.e., text, still images, video footage, audio, etc.). In fact, multimedia information has become omnipresent in our society and, for example, every minute users upload 48 hours of new video to Youtube¹, 3,125 new photos to Flickr², and share 3,600 photos on Instagram³, among others (information extracted from [Dat]).

Two main challenges have to be addressed to attend user requests involving multimedia information. First, the system should be able to process the multimedia information to extract high-level features in real-time. Second, the high-level features extracted have to be matched against the user request to filter out information which is not interesting, and to rank the results according to their similarity to what the user requested. Also, as multimedia information (especially pictures and videos) might contain sensitive information of people (e.g., their faces can appear in the photo), it would be desirable to take their privacy preferences into account when using multimedia information which users captured to answer requests from others.

As an example of a device able to capture multimedia information we focus on cameras. In this chapter we show how SHERLOCK processes what a camera is viewing to determine whether the camera could capture the image that a user requested. First, we introduce our approach to efficiently process camera views in real-time and extract high-level features from them without analyzing

¹<https://www.youtube.com>

²<https://www.flickr.com>

³<https://instagram.com>

real images. Then, we explain the mechanism developed to compare camera views in order to measure their similarity. This way, the user can provide a virtual reconstruction of the image he is interested in and the system is able to compute the similarity of the camera views retrieved to it. Therefore, we present our approaches to address the two main challenges for a system based on querying-by-example on images [SU11]: the development of an appropriate similarity measure and an efficient method to compute it in real-time. Finally, as SHERLOCK retrieves content obtained from users, we present our approach to preserve their privacy preferences when being part of pictures taken by others.

8.1 Motivation

Processing what cameras can capture is essential for the last two of our motivating scenarios. In the third scenario the Technical Director (TD) wants to obtain cameras that can provide specific shots of the rowing race, and in the fifth authorities are interested in obtaining images of the traffic accident. For this task, a popular approach is to process the real images provided by cameras. However, using real image processing techniques to extract *high-level features* related to the semantics of the scene, such as the kind of objects or the specific identity of the object, is a challenge (and even more in real-time). This is related to the problem of the well known “semantic gap” that exists between low-level features and high-level semantics, which has attracted considerable research attention (e.g., see [CSLC12] and [YYY13]). For example, consider the camera shot in Figure 8.1(a), which a camera in our third motivating scenario could obtain, where all the rowing boats participating in the race are shown from a large distance to allow the viewer to have a general overview of the race. Real image processing techniques would face two main problems:

1. Along with the rowing boats there exist multiple moving objects (judges, support team, etc.) very close to them (in Figure 8.1(b) we have highlighted the rowing boats and the other moving objects with red dotted and yellow circles, respectively). So, it would be difficult to distinguish the *objects-of-interest* (rowing boats) from the other objects in the scenario based on their visual features and moving patterns.
2. Even if the objects that are rowing boats could be identified, the Technical Director (TD) could be interested in a specific boat (e.g., “Kaiku” in Figure 8.1(b), highlighted with an arrow). Identifying this boat automatically among the others would be very difficult.



(a)



(b)

Figure 8.1: Real camera footage (a) and interesting and other objects in the scene (b).

To overcome these difficulties, we present a different approach: Instead of on line analyzing the real images provided by cameras, such real camera views are recreated by using the information contained in a 3D model of the scenario. SHERLOCK manages and keeps this 3D model up-to-date in real-

time according to the information of objects-of-interest (identification, location, direction, approximate extent, etc.) and cameras (location, direction, Field of View –FOV–, etc.) in the scenario through the exchange of information between devices, as explained in Chapter 4, or the processing of SHERLOCK queries, as explained in Chapter 7. This way, our approach uses geometric computations (implemented by a 3D engine) over the 3D reconstruction of the camera view to extract high-level semantic features of the real camera view. So, SHERLOCK is able to automatically detect the specific objects that are viewed by a camera (e.g., “the Kaiku rowing boat” vs. simply “a boat”). Moreover, our approach obtains other high-level features of each object detected in a camera view, such as: the percentage of the object visible (taking occlusions into account), the viewpoint of the camera concerning each object (e.g., it could view the front and top of the object), the amount of the shot occupied by them (that determines the space available for uninteresting objects), etc. We present in this chapter the efficient methods that we have developed to obtain this information continuously and in real-time (in our tests in Section B.4.3 we show that extracting the high-level features of a camera view can be done in tenths of a second with our algorithms).

So, as long as the locations of the objects-of-interest and cameras (and an approximation of the extent of the objects) can be obtained in real-time, our approach can be applied to any context, as no assumption is made regarding the number of cameras in the scenario (each SHERLOCK-enabled device can analyze the view of its attached camera), the kind of scenario (the system can be used in scenarios involving moving objects, cameras, and queries about them), and the positioning mechanism used to obtain the locations of the objects and cameras. In some situations it could be challenging to obtain this information for certain objects (e.g., it could be difficult to obtain the real-time precise location of a ball or the extent of soccer players that move their limbs while running). However, our approach does not rely on a specific technology to obtain this information nor requires 100% precision of these data to effectively distinguish between cameras that are interesting or not for a given query.

8.2 Processing of Camera Views

In this section, we explain how SHERLOCK analyzes a camera view to obtain high-level features. First, we show how the viewpoint of the target that the camera is capturing is obtained. Then, we describe how the percentage of the target object viewed by a camera is computed taking occlusions into account. Finally, we explain a combination of the two previous processes that allows

obtaining the percentage of a specific viewpoint of an object that a camera is providing.

8.2.1 Kind of View Obtained

Being able to classify the views provided by cameras according to the kind of view obtained enables SHERLOCK to answer specific requests of users (e.g., *cameras viewing the front, top and right side of a certain object*). As explained in Section 4.1, SHERLOCK uses the top and front vectors of the extent of an object to process requests involving the following views of the object: top/bottom, front/rear, and left/right side. As the extent of the objects in the scene could be complex and we need to perform the calculations automatically and quickly (e.g., the TD in the fourth scenario needs results as fast as possible), we propose the use of light sources and the illumination they produce to calculate the kind of view that a camera is providing of an object (see Algorithm 9).

The first step is to recreate the view of the camera in the 3D engine (we used in our prototype *JMonkeyEngine*⁴) by setting the virtual camera with the same location, direction, and Field of View (FOV) than the real one. Then, different colors are assigned to the target and other objects in the scenario so when the scene is illuminated only the parts of the target object that are not occluded by other objects will be visible. *Directional light sources* (which have no position –only a direction–, are considered “infinitely” far away, and send out parallel beams of light) are used to illuminate the parts of the object that belong to each requested view using different colors for each light source. This way, the system checks several kinds of views with a single pass and efficiently decreases the number of renderings needed (obtaining a rendering is one of the most time-consuming tasks). However, the algorithm is limited to checking three views per render as each view uses a different *rgb* color channel. For example, to check if the camera of Figure 8.2(a) is viewing the top and rear of the boat, the system “selects” these parts of the object by using a red and a blue light source, respectively (see Figure 8.2(b)). Finally, the algorithm checks the color of each pixel of the 2D projection of the 3D scene and if its equal to one of the colors used for the light sources that means that the camera is viewing, at least, some part of the target object belonging to the kind of view considered⁵.

⁴<http://jmonkeyengine.org>

⁵Notice that, due to the illumination used, the value in the color channel of a pixel might vary but as long as it is somehow illuminated we should take it into account. This is done by checking whether the color channel is equal to zero or not in the algorithm.

Algorithm 9 Calculate the kind of view obtained of a target object

Input: *target*, *cam*, *<views>***Output:** *<visible views>*

```

1: scene = recreate cam's view in the 3D engine
2: remove all the illumination sources of scene
3: for each object in the scene do
4:   if object == target then
5:     paint object with reflective texture
6:   else
7:     paint object in black (background color)
8:   end if
9: end for
10: for each view in <views> do
11:   create light source in view's direction
12:   set light source's color to an unused one from <red, blue, green>
13: end for
14: projection = obtain 2D projection of the scene
15: for each pixel in projection do
16:   if pixel's red, blue, or green channels  $\neq 0$  then
17:     set true in <visible views> for each view in <views> whose light
        source's color is  $\neq 0$  in pixel
18:   end if
19: end for
20: return <visible views>

```

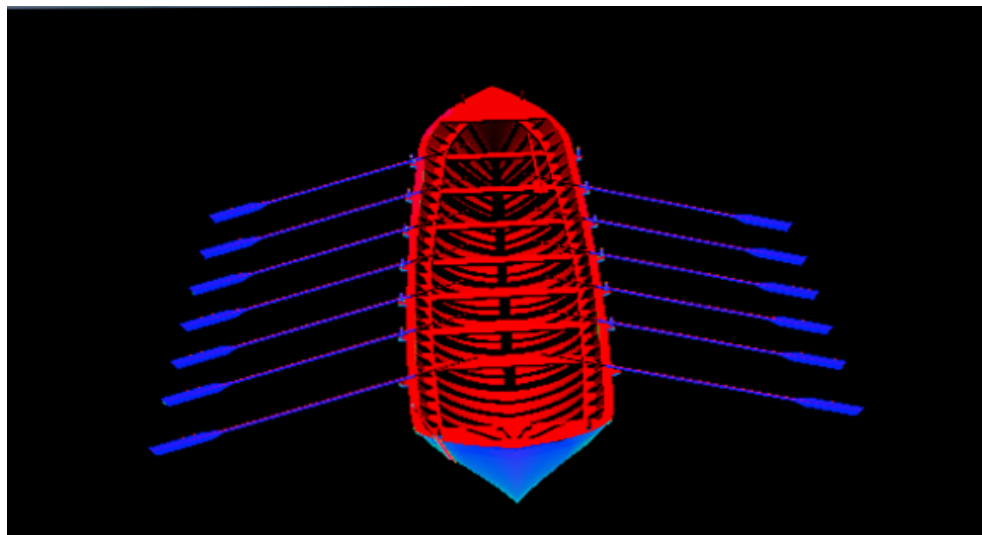
8.2.2 Percentage Viewed of an Object

SHERLOCK supports queries that ask for cameras that view a certain minimum percentage of an object. There exist two situations where a camera could have an incomplete view of an object: 1) when the target is partially or fully occluded by another object, and 2) when the target does not fit the FOV of the camera or it is outside the FOV. Algorithm 10 calculates the percentage of a target object that a camera is viewing taking occlusions into account.

As in Algorithm 9, our approach assigns different colors to the objects (red for the target object and green along with a transparent material for the other objects in the scenario), in order to show in the same rendering the hidden and visible parts of the target (see Figure 8.3(b)). Then, the current FOV is painted in transparent blue (which means creating an object that represents the FOV with a transparent material and blue color) to select what the camera



(a)



(b)

Figure 8.2: A real camera shot of a rowing boat (a) and the recreation of the view in our system with the top (red) and rear (blue) of the boat highlighted (b).

Algorithm 10 Calculate the percentage viewed of a target object

Input: *target, cam***Output:** *percentage viewed*

```

1: scene = recreate cam's view in the 3D engine
2: for each object in the scene do
3:   if object == target then
4:     paint object in red
5:   else
6:     assign object a transparent material and paint it green
7:   end if
8: end for
9: assign current FOV a transparent material and paint it blue
10: while target does not fit completely the FOV do
11:   move virtual camera backwards
12: end while
13: projection = obtain 2D projection of the scene
14: for each pixel in projection do
15:   if pixel's red channel  $\neq 0$  then
16:     increase #pixels of the target object
17:   end if
18:   if pixel's red and blue channels  $\neq 0$  and green channel == 0 then
19:     increase #pixels not occluded
20:   end if
21: end for
22: return  $\frac{\text{\#pixels not occluded}}{\text{\#pixels of the target object}}$ 

```

is currently viewing (see Figure 8.3(c)). If the target does not fit completely the FOV, the virtual camera is moved backwards in the focal axis of the camera until it views the target object completely. This movement allows our algorithm to obtain a rendering covering the full object while it does not affect the perspective of the scene (see Figure 8.3(d)). Finally, the algorithm obtains the total number of pixels of the target (*#pixels of the target object*) and the pixels of the target visible and not occluded (*#pixels not occluded*) and computes the percentage visible ($\frac{\text{\#pixels not occluded}}{\text{\#pixels of the target object}}$). For example, using the image of Figure 8.3(d), the algorithm obtains that the camera views 41% of the target object (the second boat).

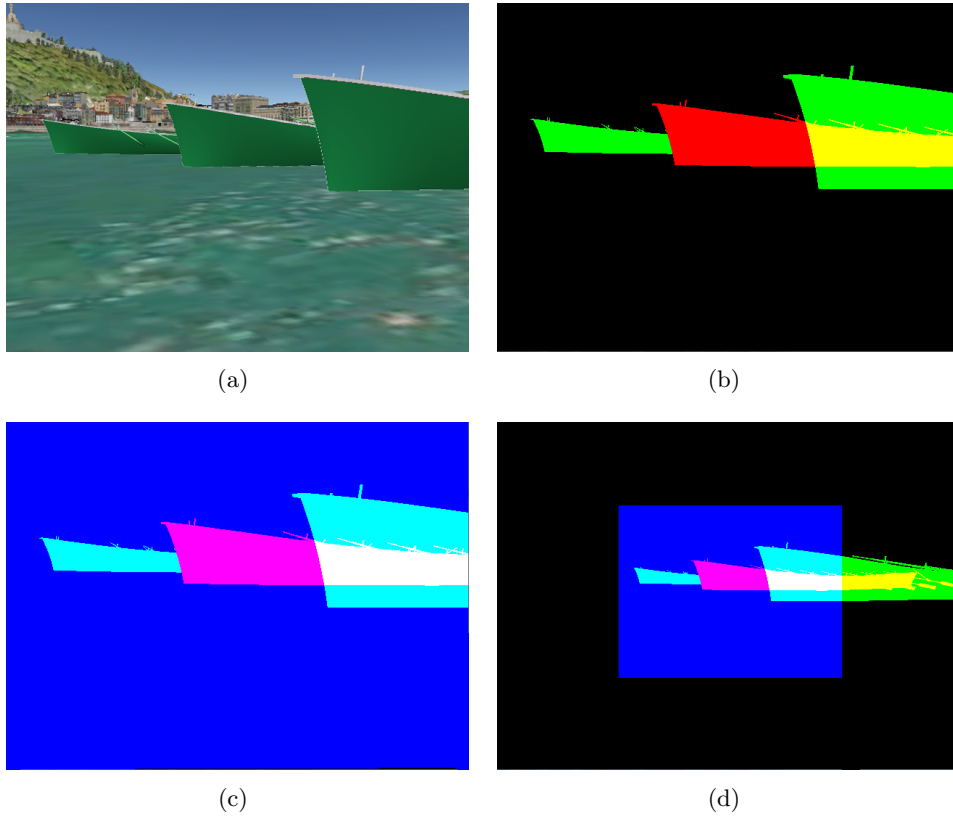


Figure 8.3: Computing the percentage of a target object in a shot: scene in *Google Earth* (a), selecting the target (b), painting the FOV (c), and covering the target completely (d).

8.2.3 Percentage of a Part of an Object

For a user, it could be interesting also to retrieve cameras viewing a percentage of a certain part (i.e., top/bottom, front/rear, left/right) of an object (e.g., *cameras viewing at least 50% of the front of the object*). The method explained before needs an additional step to support this kind of queries (see Algorithm 11), to obtain first which shot would cover 100% of that target viewpoint in order to calculate the actual percentage viewed. This way, the algorithm sets the virtual camera of the 3D engine in the direction of the target viewpoint and at the same distance of the object as the real camera, and counts the number of illuminated pixels (*#pixels belonging to viewpoint*).

This information will be used along with the number of pixels of the viewpoint that the camera is viewing ($\#pixels\ viewed\ of\ viewpoint$) to calculate the percentage of the viewpoint viewed ($\frac{\#pixels\ viewed\ of\ viewpoint}{\#pixels\ belonging\ to\ viewpoint}$). Notice that, if the target object did not fit the FOV in Algorithm 11, the approach moves the virtual camera backwards a distance d to compute the total amount of pixels visible for a shot that covers 100% of the target view. This way, the algorithm will move the camera backwards the same distance d before calculating the amount of pixels of the view that the camera covers. The example of Figure 8.2(b) shows a 2D image rendered by the algorithm for the current view of a camera, where the algorithm obtains that the camera views 95% of the top and 92% of the rear of the target object. Notice that there are different intensities of red and blue in the image used by the algorithm, as depending on the normal of the corresponding polygon the illumination method (*Phong* is used in JMonkeyEngine) makes it look darker or brighter. This is not a problem for our approach because it only counts pixels that have a nonzero value for that specific channel.

Algorithm 11 Calculate the number of visible pixels of the object in a shot that covers 100% of the target viewpoint of the object

Input: *target, view, cam*

Output: *pixels belonging to viewpoint, d*

- 1: recreate *cam*'s view in the 3D engine
 - 2: paint *target* with reflective texture
 - 3: set virtual camera's direction to *view*'s direction
 - 4: create light source in *view*'s direction
 - 5: **if** *target* does not fit completely the FOV **then**
 - 6: d =move virtual camera backwards
 - 7: **end if**
 - 8: *projection*=obtain 2D projection of the scene
 - 9: $\#pixels\ belonging\ to\ viewpoint$ =count pixels in *projection*
 - 10: **return** $\#pixels\ belonging\ to\ viewpoint, d$
-

8.2.4 Estimating Future Views

It could be interesting for a user to show information about cameras that are not currently viewing a target object but they could in a near future. For example, it could be useful to estimate if a camera is going to be able to view the target if rotated (and the rotation and time needed, if so). Therefore, if a

certain camera is not currently viewing a target object, our system can obtain the rotation (pan, tilt, or both) and the time needed for the camera to view it. For this purpose, the system needs to take into account the current location, speed, and direction of the target object and other objects in the scenario (because they could partially or fully hide the target), and the features of the camera being considered (maximum pan and tilt allowed, and rotation speeds). Once the system estimates the time needed for a certain camera to view a target object, it will recreate the state of the scenario and obtain high-level features of the view that the camera would provide at that moment.

One can think that answering whether a camera could view an object or not is easy; for example, if a camera has the object to its right then it should be able to view it if rotated to the right). However, calculating this estimation is not so easy when considering that objects in the scenario, and so also the cameras that are attached to them, can move. In the previous example, if the object keeps moving around the camera with a speed higher than the rotation speed of the camera, the camera will never view the target if rotated to the right. Thus, in the following we present in detail our approach to estimate the time and rotation needed by a camera to focus a target object considering that both objects and cameras can move.

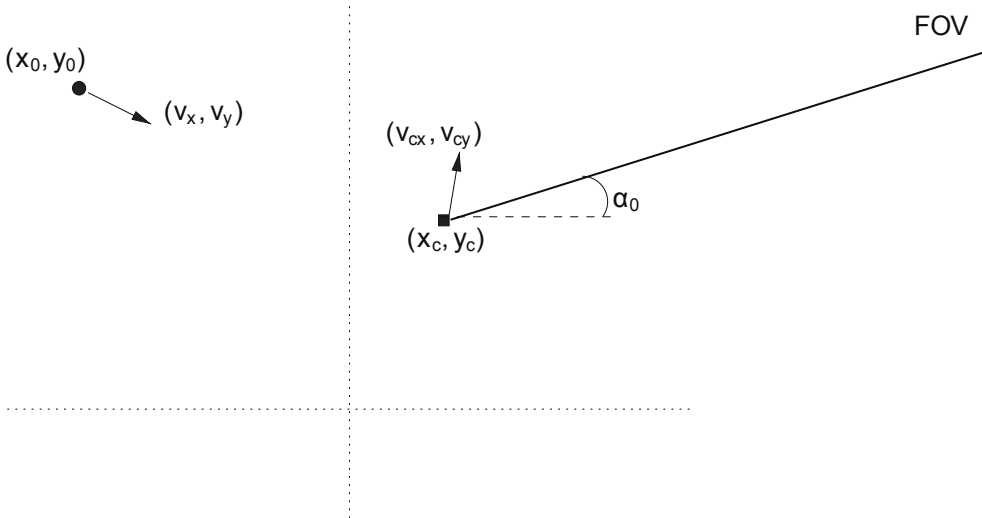


Figure 8.4: Initial state of a target object in (x_0, y_0) and a camera in (x_c, y_c)

On the one hand, we have to model the movement of the objects (that for simplicity can be represented here as points in the plane –their center of mass–)

with a motion function depending on time $S(t) = (x(t), y(t))$ (see Figure 8.4). For our scenarios, we consider that the movement of the objects is linear, but the method presented in this section would work with any analytic functions $x(t)$, $y(t)$ (such as interpolation polynomials) or with other approaches to model the movement of objects (e.g., [TFPL04; HM12]). So, the motion function is:

$$S(t) = (x_0 + v_x t, y_0 + v_y t)$$

where (x_0, y_0) and (v_x, v_y) are the initial position and the speed vector of the object, respectively.

On the other hand, a camera is modeled as a semiline (defined by the bisector of its FOV) that can move and rotate (see Figure 8.4). The motion function for a camera is the semiline formed by the values of X and Y of the line $C(t) \equiv (X - x_c(t)) \sin(\alpha(t)) - (Y - y_c(t)) \cos(\alpha(t)) = 0$, such that $\text{sign}(X - x_c(t)) = \text{sign}(\sin(\alpha(t)))$, $\text{sign}(Y - y_c(t)) = \text{sign}(\cos(\alpha(t)))$, where $(x_c(t), y_c(t))$ is the translation motion function of the camera and $\alpha(t)$ is the rotation function.

We consider that a camera has a linear translation motion with a uniform angular speed. Therefore, the motion equations of the semiline representing a camera depending on time are:

$$\begin{aligned} (X - (x_c + v_{xc}t)) \sin(\omega_c t + \alpha_0) - (Y - (y_c + v_{yc}t)) \cos(\omega_c t + \alpha_0) &= 0 \\ \text{sign}(X - (x_c + v_{xc}t)) &= \text{sign}(\sin(\omega_c t + \alpha_0)) \\ \text{sign}(Y - (y_c + v_{yc}t)) &= \text{sign}(\cos(\omega_c t + \alpha_0)), \end{aligned}$$

where (x_c, y_c) and (v_{xc}, v_{yc}) are the initial position and the speed vector of the camera, respectively, ω_c is the pan speed of the camera, and α_0 is the initial pan of the camera.

We want to obtain the minimum time instant when the camera focuses the object, considering that the camera can rotate to the left side (a positive pan speed) or to the right side (a negative pan speed). Thus, we have to obtain:

- which pan speed (positive or negative) leads to a faster movement to focus the object,
- for that pan speed, the time instant when the camera and the object intersect.

In order to compute these values, first we have to find the solution t_{f+} for the equation

$$\begin{aligned} E(t) &= (x_0 + v_x t - (x_c + v_{xc} t)) \sin(\omega_c t + \alpha_0) \\ &\quad - (y_0 + v_y t - (y_c + v_{yc} t)) \cos(\omega_c t + \alpha_0) = 0 \end{aligned}$$

that holds

$$\begin{aligned} t_{f+} &= \min\{t_r | t_r \geq 0, E(t_r) = 0, \\ &\quad \text{sign}(x_0 + v_x t_r - (x_c + v_{xc} t_r)) = \text{sign}(\sin(\omega_c t_r + \alpha_0)), \\ &\quad \text{sign}(y_0 + v_y t_r - (y_c + v_{yc} t_r)) = \text{sign}(\cos(\omega_c t_r + \alpha_0))\} \end{aligned}$$

Second, we have to find the solution t_{f-} changing ω_c by $-\omega_c$ in the above equation. The minimum value of $\{t_{f+}, t_{f-}\}$ gives us the sign of the pan speed and the time instant that we are looking for (see Figure 8.5).

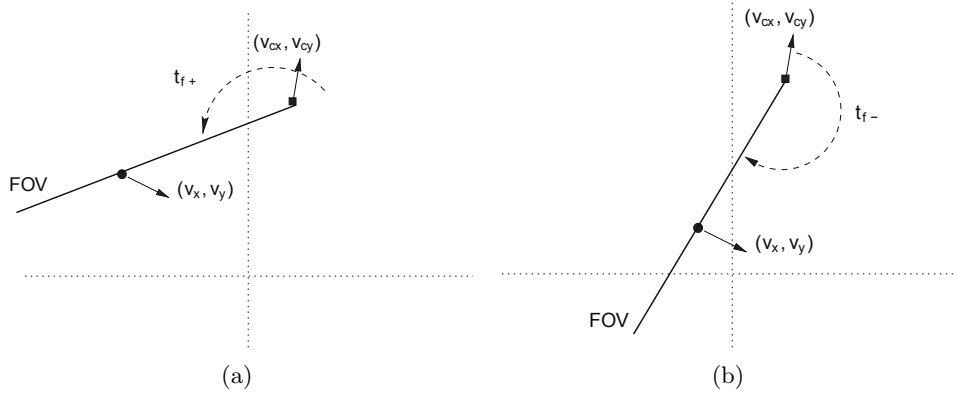


Figure 8.5: t_{f+} (t_{f-}) time to focus the target object rotating the camera to the left (right) side.

The trajectories of the objects are estimated by using linear extrapolation based on their speed vectors. The speed vector of an object is computed by considering three previous reference locations of the object (in our prototype, the locations of the objects during the last three seconds). To solve the above equation we reduce the problem to finding the zeros of a polynomial. We use an approximation of \sin and \cos using Taylor polynomials and solve numerically the equation using Laguerre's method, a root-finding algorithm tailored to

polynomials. As an example, Figure 8.6 shows the estimation of the time needed to focus horizontally a target object by considering the movements of the objects, the current pan, and the pan speed of the camera.

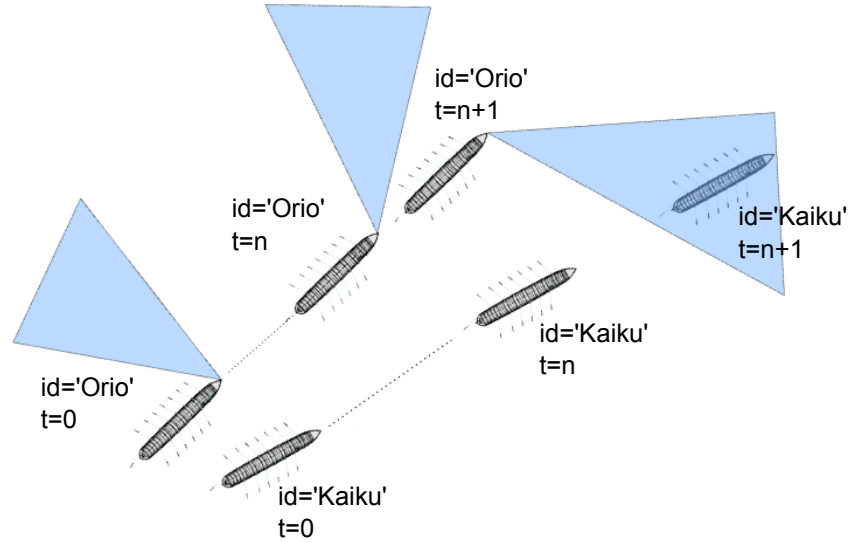


Figure 8.6: Estimation of trajectories and time needed to focus a rowing boat, "Kaiku", from another boat, "Orio".

As we are dealing with scenarios where the objects can move in 3D, we need to obtain the time needed to focus the target both horizontally (pan) and vertically (tilt). As pan and tilt movements can be done in parallel, we obtain the maximum of the time needed to pan and the time needed to tilt and use that value as the time needed to focus the object. So, the equation above is used for both movements using the horizontal plane for the pan (equation above) and the plane defined by the trajectory of the object and the z-axis for the tilt.

Once the estimation has been completed, the algorithm generates the scene to calculate if the requirements of the query (e.g., type of view, percentage of the target object or part shown, etc.) would be satisfied. Notice that, as the objects are moving, when the camera is able to focus the target it could happen to be occluded by another object. Therefore, the trajectories of all the objects have to be taken into account too, not only the trajectory of the camera and the target object.

8.3 Measuring the Similarity of Camera Views

SHERLOCK enables users to define the kind of camera view they want to obtain by providing an example created by using a 3D query-by-example (3DQBE) interface, as we explained in Section 6.1. The 3DQBE interface enables the user to select the objects that should be in the FOV of the camera and move/rotate them to select their specific location in it. Also, SHERLOCK ranks the camera views obtained as an answer to a user request regarding their similarity to the one defined by the user. To measure the similarity between the query image I_q defined by the user and a candidate image I , we propose the following formula, that makes use of the high-level features extracted from both images to take into account the similarity of each specific object appearing in the two pictures:

$$S(I_q, I) = \sum_{i=1}^n \gamma_{o_i} S_{object}(o_i, I_q, I) \quad (8.1)$$

$$\gamma_o = \frac{pct_{img}(o, I_q)}{\sum_{i=1}^n pct_{img}(o_i, I_q)} \quad (8.2)$$

where n is the number of objects in I_q , $S_{object}(o, I_q, I)$ is the similarity between the two images if only the object o is considered, and γ_o is the weight (relative importance) assigned to this object in the query image I_q defined by the user.

To avoid overwhelming the user by requesting him/her to enter the weight for each object, we advocate using an objective value extracted from the scene. So, we consider that the user defines indirectly the importance of each object in the scene by adjusting the percentage of the image occupied by each of them. So, as shown in Formula 8.2, γ_o represents the percentage of the image filled by object o in the query image I_q ($pct_{img}(o, I_q)$) relative to the part of the image filled with objects ($\sum_{i=1}^n pct_{img}(o_i, I_q)$); this percentage is computed regarding only the part of the image filled with objects (as we are interested here in identifying the relative importance of the objects, rather than the amount of space they occupy in the image). For example, if we consider that the user defined the image of Figure 8.3 as the query image, the objects close to the camera are clearly expected to be more important for the user than the other object because they fill a greater amount of the shot.

In the rest of this section, we explain the different factors that define the computation of $S_{object}(o, I_q, I)$ for a given object o , a query image I_q , and an image I .

8.3.1 Percentage Difference

The first aspect that we consider to obtain the similarity of an object in two images is the percentage of it that is being covered and the percentage of the image that it fills. In this way, we compare the percentage of the object o visible in the query image I_q defined by the user with the percentage visible in the candidate image I , as well as the percentage of each image (I_q and I) occupied by o :

$$S_{pctObj}(o, I_q, I) = 1 - (\omega_{oi} \frac{\Delta pct_{obj}(o, I_q, I)}{pct_{obj}(o, I_q)} + \omega_{oi} \frac{\Delta pct_{img}(o, I_q, I)}{pct_{img}(o, I_q)}) \quad (8.3)$$

$$\Delta pct_x(o, I_q, I) = \min(pct_x(o, I_q), |pct_x(o, I) - pct_x(o, I_q)|), \text{ with } x = \{obj, img\} \quad (8.4)$$

where in relation to the first addend $pct_{obj}(o, I)$ and $pct_{obj}(o, I_q)$ obtain the percentage of the object visible, taking occlusions into account, in image I and I_q , respectively; in relation to the second addend, $pct_{img}(o, I)$ and $pct_{img}(o, I_q)$ obtain the percentage of the corresponding images filled by the object. In addition, we normalize the difference between these percentages ($\Delta pct_{img}(o, I_q, I)$ and $\Delta pct_{obj}(o, I_q, I)$) to obtain a value between 0 and 1 that measures their similarity. Notice that, as we want to obtain an objective measurement, we consider that, for example, given the percentage visible of an object in the query image $x\%$, an image that shows $(x + y)\%$ is as similar as an image that shows $(x - y)\%$. In this way, an image that does not show the object (i.e., it shows $(x - x)\%$ of it) is considered as similar as one that shows $(x + x')\%$ with $x \leq x' \leq 1.0$ (percentages are expressed here as values between 0 and 1); this is the motivation for the use of the min operator in Formula 8.4. Moreover, we assign each factor a weight ω_{oi} and ω_{oi} , and to preserve the objectivity we consider that $\omega_{oi} = \omega_{oi} = 1/2$.

8.3.2 Viewpoint Difference

The second aspect that we consider is the kind of view obtained of the object in the two images. As part of the high-level features of a scene we consider that the following views of an object can be obtained: top/bottom, front/rear, and left/right side. So, the following formula defines the similarity of an object in two images according to the kind of view obtained:

$$S_{views}(o, I_q, I) = \sum_{i=1}^n \gamma_{v_i} \left(1 - \left(\omega_{vi} \frac{\Delta pct_{view}(v_i, I_q, I)}{pct_{view}(v_i, I_q)} + \omega_{vii} \frac{\Delta pct_{img}(v_i, I_q, I)}{pct_{img}(v_i, I_q)} \right) \right) \quad (8.5)$$

$$\gamma_v = \frac{pct_{img}(v, I_q)}{\sum_{i=1}^n pct_{img}(v_i, I_q)} \quad (8.6)$$

$$\Delta pct_x(v, I_q, I) = \min(pct_x(v, I_q), |pct_x(v, I) - pct_x(v, I_q)|), \text{ with } x = \{view, img\} \quad (8.7)$$

where v is the vector that contains the views to check, whose components belong to the set {front, rear, top, bottom, left, right}, v_i represents the view in position i of v , and n is the number of elements in v . For each view of an object (front, rear, top, bottom, right side, and left side) we compute the similarity according to the percentage of the view obtained ($pct_{view}(v_i, I)$) and the percentage of the image filled with this view ($pct_{img}(v_i, I)$). We normalize the difference between these percentages ($\Delta pct_{view}(v_i, I_q, I)$ and $\Delta pct_{img}(v_i, I_q, I)$) to obtain a value between 0 and 1. We consider that the percentage of the view obtained and the percentage of the image filled with this view have the same importance to compute the similarity of an object (so, in our prototype the weights used for them are $\omega_{vi} = \omega_{vii} = 1/2$). Moreover, we assign each view a weight γ_v according to their importance in the image. For example, in Figure 8.3, regarding the boats, the user seems to be interested in a lateral view of such boats.

8.3.3 Location Difference

The location of each object within the image is an important parameter to take into account when computing the similarity. We consider the location of the camera and the object in the 3D scene to measure this factor. In particular, we use the angle defined by the bisector of the FOV of the camera and the vector defined by the location of the camera and the centroid (of the volume) of the object. For the sake of clarity, we further decompose the vector in terms of its horizontal and vertical components. To illustrate this, Figure 8.7 shows the different angles involved in the horizontal plane of two scenes (it would be similar for the vertical plane).

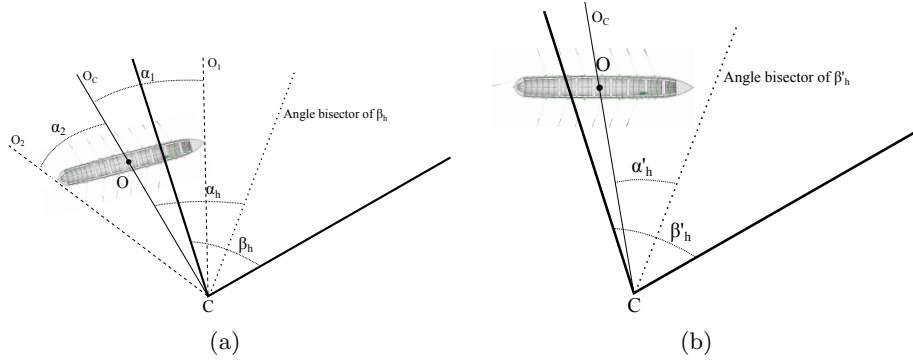


Figure 8.7: Horizontal angles involved in the computation of the similarity between two images (a) and (b).

So, we define the similarity of an object in two images according to its location as:

$$S_{location}(o, I_q, I) = 1 - (\omega_{li} \frac{\Delta\alpha_h}{max_h} + \omega_{lii} \frac{\Delta\alpha_v}{max_v}) \quad (8.8)$$

$$max_h = \max(\frac{\beta_h}{2} + |\alpha_1|, \frac{\beta_h}{2} + |\alpha_2|) - \epsilon \quad (8.9)$$

$$max_v = \max(\frac{\beta_v}{2} + |\alpha_3|, \frac{\beta_v}{2} + |\alpha_4|) - \epsilon \quad (8.10)$$

$$\Delta\alpha_x = \min(max_x, |\alpha'_x - \alpha_x|), \text{ with } x = \{h, v\} \quad (8.11)$$

where α_h and α_v are the angles that define the location of the object in the horizontal and vertical plane of the query image I_q defined by the user, respectively. Similarly, α'_h and α'_v measure the location for the scene we are comparing I_q with (i.e., image I). Besides, $\Delta\alpha_h$ and $\Delta\alpha_v$ stand for the difference in the location of the object in both scenes in the horizontal and vertical plane, respectively. As before, we consider the absolute value of these differences and we normalize them by using max_h and max_v (the maximum value of α_h and α_v for an object to be inside the FOV of a camera). In Formula 8.9 and 8.10, ϵ is a small value greater than 0, which must be subtracted from $\max(\frac{\beta_h}{2} + |\alpha_1|, \frac{\beta_h}{2} + |\alpha_2|)$ and $\max(\frac{\beta_v}{2} + |\alpha_3|, \frac{\beta_v}{2} + |\alpha_4|)$ (used in the computation of max_h and max_v) in order to have the target object within the view of the camera. We consider that the similarity of an object's location in the horizontal and vertical planes have the same importance (so, in our prototype the weights used for them are $\omega_{li} = \omega_{lii} = 1/2$).

8.3.4 Summing Up: Differences in Each Object

To compute the similarity between two images according to object o we take into account the percentage of the object visible, the viewpoint of the camera, and the location of the object in the image, which are computed as explained in the previous subsections. First, we check if object o (that appears in the image I_q) is visible in the image I ; if this is not the case, then the similarity is 0. Otherwise, if the object is visible in the image I (partially or completely), the similarity between the two images regarding that object is:

$$S_{object}(o, I_q, I) = \omega_o S_{pctObj}(o, I_q, I) + \omega_v S_{views}(o, I_q, I) + \omega_l S_{location}(o, I_q, I) \quad (8.12)$$

where $S_{pctObj}(o, I_q, I)$ stands for the similarity according to the percentage of the object covered in the image and the percentage of the image filled by the object (see Section 8.3.1); $S_{views}(o, I_q, I)$ represents the similarity according to the different views of the object (see Section 8.3.2); and finally, $S_{location}(o, I_q, I)$ is the similarity according to the location of the object in both images (see Section 8.3.3). In the formula, ω_o , ω_v , and ω_l are the weights assigned to each factor and by default $\omega_o = \omega_v = \omega_l = 1/3$. So, in our proposal the three weights are equal, as there is no objective criterion to assign different weights to them. Indeed, this is completely subjective. For example, for a user two images could be more similar if they show the same percentage of the object (regardless of the percentage of the object or the kind of view obtained), and for another user two images could be more similar if they show the object in the same locations.

8.4 Preserving User Privacy in Photos

Whenever a SHERLOCK user takes a picture, which might be shared with other devices as an answer to requests of their users, it first turns it into a *privacy-aware picture*. We understand for privacy-aware picture a photo that preserves the privacy preferences of the entities (i.e., people, buildings, etc.) captured. SHERLOCK allows users to state their policy about being photographed (i.e., “I don’t want my picture to be taken”) by other people the same way they define their policies about the information that can be shared (see Chapter 4). To start with, SHERLOCK generates an *eigenface* [SK87], a mathematical representation which we call a *face identifier* (see Figure 8.8(b)) using a picture of the user’s face (see Figure 8.8(a)). Whenever a device is in the vicinity of the user, his SHERLOCK sends it the face identifier along

with the policy (i.e., obscure my face in your pictures). In order to enforce the policy, SHERLOCK running on the stranger device uses the face identifier to detect if the user who shared the policy is part of the pictures taken by the device (see Figure 8.8(c)). It then selectively obscures the face of all the people who have sent such a policy to the device (see Figure 8.8(d)). Using eigenfaces helps to preserve the privacy of the sender even if the transmission is intercepted and at the same time the enforcement of the policy rule ensures the privacy in any pictures taken.

8.4.1 Creating Privacy-Aware Pictures

As explained in Chapter 4, the user’s context is represented using an OWL ontology, privacy policies are described using SWRL rules, and a DL reasoner is used to infer if the current context of the user matches with any of the privacy policies defined. Also, the user device holds the ontology and reasoner and the Knowledge Endpoint agent is in charge of checking the policies that should be applied. Figure 8.9 shows an example of privacy policy for a user: “do not allow my social network colleagues group to take pictures of me at parties held on weekends at the beach house”. Where *FaceBlockPictures(?p, True)* indicates the system that a picture taken under such circumstances should be converted into a privacy-aware picture.

We describe the cross device process for generating privacy-aware pictures as shown in Figure 8.10. We use an example with two users, *Primal* and *Roberto*, to explain how our method to generate privacy-aware pictures works. Primal is the user who wishes to protect his privacy and Roberto is the user with the device for taking pictures, in this case a Google Glass. Initially, Primal takes a picture of himself to complete his profile in the system and SHERLOCK generates a face identifier (*step 1* in Figure 8.10). He also specifies the context constraints for his pictures. At the Beach House, Primal’s SHERLOCK detects and shares the face identifier with Roberto’s Google Glass (*step 2*) along with a unique identification. Roberto’s device receives this information, stores it and sends back his UID and an acknowledgment of the previous message (*steps 3 and 4*).

Afterwards, SHERLOCK on Primal’s device continuously collects information about his context and checks if any rule should be triggered by using the reasoner (*step 5*). In this case, the context has changed (the party started) and the rule presented in Figure 8.9 gets triggered requesting Roberto to preserve his privacy in pictures where Primal appears (*step 6*). The corresponding privacy policy (*disallow pictures*) for Primal is shared with Roberto’s device



(a)



(c)



(b)



(d)

Figure 8.8: Images involved in the process of obtaining a privacy-aware picture: (a) picture of a user; (b) face identifier of the user generated by the system; (c) picture taken by a SHERLOCK user; and (d) privacy-aware picture generated by SHERLOCK.

(*step 7*). Each privacy policy has a Time To Live (TTL) associated with it during which the policy should be applied to the pictures of the user. Currently, we are using a uniform TTL for every policy. Roberto's device accepts the privacy policy from Primal's device (*step 8*) and whenever he takes a photo SHERLOCK converts it into a privacy-aware picture (*step 9*). For that, if faces are detected in the recently taken picture, the system checks if Primal

```

Person(?p) ∧ Colleague(?p) ∧ Context(?c) ∧
hasTime(?c,?t) ∧ hasDay(?t,?day) ∧ WeekendDay(?day) ∧
hasLocation(?c,?loc) ∧ BeachHouse(?loc) ∧
hasActivity(?c,?act) ∧ Party(?act)
→ FaceBlockPictures(?p,True)

```

Figure 8.9: Example of a context-aware privacy policy to create privacy-aware pictures.

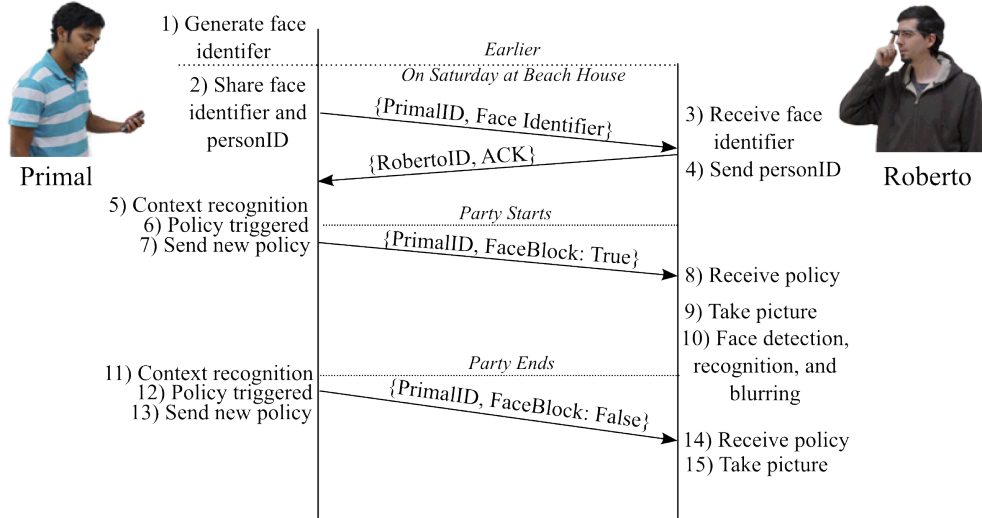


Figure 8.10: Handshake diagram for the creation of privacy-aware pictures.

is present in the picture by comparing the detected face with Primal's face identifier and obscures it (*step 10*). Thus, SHERLOCK creates a privacy-aware picture for Roberto and protects the privacy of Primal.

8.5 Related Work

In this section, we present related works on the two main topics explained in this chapter. On the one hand, we present works dealing with the selection of cameras views in real-time. On the other hand, we presents works focusing on preserving user privacy preferences in pictures taken by cameras.

8.5.1 Works on Selection of Camera Views

Up to the authors' knowledge, no other work has focused on the real-time selection of camera views based on the extraction of high-level features of images provided by multiple moving cameras using a 3D model of the scenario. However, there is extensive research on the analysis of real images to obtain what a camera is viewing. These studies can be classified according to the kind of processing performed on the video streams (on-line or offline). Proposals to process the videos on-line, such as [LIS01; XCDS02], usually take into account the *cinematic* features of the views provided by the cameras, such as the shot types (e.g., a long shot, a close-up shot, etc.). Considering high-level features related to the semantics of the scene, such as the specific object, the visible amount of the object, or *object-based* features, such as the color and shape of the objects, their interactions, etc., may be computationally too costly for on-line processing, even though it would provide richer semantic details.

Real-Time Camera Selection for Sport Events

We can mention [WXCLT08; CLY09], that share the goal of our proposal of selecting camera views in real-time for TV broadcasting, even though their approaches are based on analyzing the real images provided by the cameras.

The context of the system presented in [WXCLT08] is soccer games. It relies on the well-defined structure of a soccer broadcast to alternate the selection between cameras that provide a far view and cameras that provide a medium/close-up view. The view switching method proposed in that paper does not analyze high-level features of the camera views and the authors assume that all the cameras are following the game action (and hence they have a similar content). So, their problem is to select those cameras providing a clear view (they discard blurry images). The main differences between our work and [WXCLT08] is that we allow the TD to define the criteria to be considered to select a camera view, and moreover our proposal is able to obtain a good number of high-level features of a camera view in live. Besides, we make no assumptions about the current views of the cameras.

In [CLY09] a system is also presented to automatically select in live the camera to broadcast in a soccer game. As we do, the authors also consider low-cost cameras as a way to reduce the costs of a sport broadcasting. They assume that there exist four cameras located along the field and they process their views to obtain the size of the ball in each one. With this information, they propose to select the camera whose view shows the largest area of the projected ball. Therefore, their approach is focused on ball sports and under

the assumption that the best views are those that provide a better view of the ball, while ours can be applied to other contexts where selecting among many camera views is needed.

The goal of these two works is to select automatically the best camera to broadcast in soccer events based on parameters as image quality. However, our approach, that is not focused in any specific sport, enables the TD to define the kind of camera he/she wants to broadcast based on the objects that this camera views.

Assistants for Sport Videos Summarization

A number of works have focused their efforts on the specific problem of helping producers of sport videos. For example, as part of the *APIDIS* project, in [CDV10] a system that helps the video production and video summarization in the context of basketball games is presented. The work presented in [ETM03] tackles the problem of summarizing videos of soccer games by applying different image processing algorithms to analyze the input videos extracting cinematic and object-based features. In [NTB09] the problem of video summarization, based on metadata describing the semantic content of MPEG-7 videos, is considered in the context of baseball games. Cricket videos, as well as soccer videos, are used to validate the work in [Kol11], which exploits audio features (such as an increase in the audio level of the voice of the commentators or the cheers of the audience) to extract excitement clips from sport videos. Along the same line, the work in [CH06] benefits from audio and motion cues to extract highlights from baseball videos. Textual overlays appearing in images are exploited in [BKOK04] to create personalized summaries of American football videos (i.e., video abstracts that take into account the user preferences); similarly, works such as [XZZRLH08; XWLZ08] use webcast text associated to the video for event detection. In [ZHXXGY07] the authors focus on the problem of ranking, structuring, and summarizing highlights to match a user's personalized query, within the context of racket games (tennis and badminton). Like [ZHXXGY07], most works in this area emphasize the importance of taking into account the user preferences and/or expectations [CDV11]. A detailed survey of soccer video analysis systems can be found at [DL10].

All these works are concerned about facilitating the production of sport events, like the proposal in this paper, but they have a different purpose. Thus, the purpose of these works is usually to perform an automatic video production in an offline setting (so, for example, achieving a good performance for real-time processing is not an issue) by using real image processing techniques to extract

low-level features (e.g., color, texture, shapes, etc.) that will be processed to obtain cinematic features (i.e., shot classification).

Camera Management for Broadcasting

Several works in the literature have considered the problem of automatic camera management for recording and broadcasting lectures and talks. For example, in *AutoAuditorium* [Bia98] two cameras and microphones are used to obtain information about what is happening on the stage and perform an automatic audio mixing, tracking of the people on stage, and camera selection. Another interesting work is [RHGL01], which implements several production rules, inspired by the way professional video producers work, in order to take the appropriate recording decisions. The system *FlySPEC* [LKFWB02] combines a PTZ (Pan-Tilt-Zoom) camera and a panoramic camera and benefits from the involvement of the audience, participating through explicit requests, to reduce the probability of unsatisfactory recordings. The *Microsoft Research LecCasting System (MSRLCS)* [ZRCH08] supports a scripting language to facilitate the customization of production rules for different room configurations and production styles. As a final example, the *Virtual Videography* [HWG07] advocates an offline processing to have more time and information to perform the video production.

Although the context and purpose of these works is different from the ones considered in this paper, they highlight the interest of the development of automatic video production techniques to save production costs and enable fast access to multimedia information. Several other works focus on multi-camera management (e.g., [AKK06; PLMC09]). However, they usually consider only cameras that are static (i.e., at fixed locations), whereas the cameras considered in our proposal can move.

8.5.2 Works on Privacy on Pictures

The technique we developed to preserve user privacy in photos is general and can be applied to any device equipped with a camera. However, we designed this technique with wearable cameras in mind. Wearable computing devices like Google Glass are at the forefront of technological evolution in smart devices. One of the best capabilities of such devices is that they allow spontaneous and effortless photo taking. Instead of pulling out your camera, turning it on, aiming, taking a picture and putting it away, Google Glass makes it as easy as saying Okay Glass, take a picture. The ubiquitous and oblivious nature

of photography using these devices has made people concerned about their privacy in private and public settings.

Although eyewear devices have been introduced to the public very recently there are other approaches in the literature to privacy enhancing for wearable computing (see [KDSW15] for a study on the topic). For example, in the Respectful Cameras approach presented in [SMMSG07], people wear colored hats and scarfs to let cameras know that their face should be made unrecognizable in pictures taken. The P3F approach presented in [DWE13] follows a similar approach where users wear a piece of wardrobe to let cameras know their preferences. In this case, the piece of wardrobe can encode more complex privacy preferences such as *do not search* (to specify that the user do not want the photo to be retrieved in queries using her identification) or *do not publish* (to specify that the user do not want the picture to be published). However, in these two approaches the user is required to wear something whereas in our approach is the face identifier what is used to detect the user and her devices shares her privacy preferences. Other approaches, such as SnapMe [HSS13], use location information to prevent pictures taken from being shared. In that approach, users register into the SnapMe platform and upload the pictures they take to it. Whenever a picture is uploaded, using the metadata information about the location, users in the same location are notified. In our approach, users can define their context-aware privacy policies (which include other aspects in addition to the location, for example, the activity) that are applied regarding the situation. Also, by sharing the face identifier we can detect if a user is in a picture or not and thus, apply her privacy preferences automatically.

8.6 Summary of the Chapter

In this chapter we have explained the mechanisms developed to enable SHERLOCK to manage cameras which are providers of multimedia information. We have focused on the processing of camera views to be able to determine whether a camera could provide the image that the user requested. We presented our algorithms to efficiently determine what a camera is viewing in real-time without the need to analyze real images (which can be too costly for mobile devices). Our algorithms are based on the recreation of the scene that a camera is viewing in a 3D engine. To this end, our system relies on a 3D model of the scene generated based on information about the interesting objects and cameras in the scenario (location, direction, and approximate 3D extent) which is shared with each SHERLOCK device. The approach presented obtains

high-level features of cameras views, such as: the specific objects viewed, the percentage of them covered, the percentage of the shot filled by a specific object, the kind of view of the object obtained (e.g., front, top, side), etc.

Also, a similarity measurement is presented to obtain an objective value to compare the high-level features extracted by SHERLOCK for two camera views. This value is used by the system to provide a ranking list of similar images for a user request query. High-level features (such as the specific objects in a shot, their visible percentage, their viewpoint, etc.) are extracted from the example shot and the current camera views efficiently and used in the similarity formula presented. Moreover, the approach can be fine-tuned to the preferences of specific users.

Finally, we have presented our approach to preserve the privacy of users that are part of pictures taken by other SHERLOCK users. Our approach allows a user's mobile device to share privacy policies with nearby devices using the dissemination mechanisms that SHERLOCK implements. Along with the policy, users share information that helps identify them in a picture and obscure their faces as necessary.

Chapter 9

Dealing with the Motivating Scenarios

In Section 3.1 we presented four motivating scenarios that we designed to highlight the most important challenges that a system to provide LBS would have to address. So far we have explained our proposal to address these challenges and to provide users with LBS in wireless environments. In this chapter we revisit the four motivating scenarios explaining how SHERLOCK deals with them. For each of these scenarios, we will show the knowledge that the system needs to manage it, and the most important steps involved in its processing focusing on the most important challenge in such a scenario. In addition, to further explain the flexibility of SHERLOCK, we show other uses cases that were not directly considered when we designed the architecture, but that are processed by SHERLOCK by just providing knowledge about them. These scenarios include a tourist in a foreign city who wants recommendations for sightseeing, a PhD student attending a conference who wants to find other researchers in his field, and finally a community health-care worker in rural India who needs assistance treating patients.

9.1 First Scenario: SHERLOCK for Looking for Transportation

In our first scenario, a person arrived in a foreign city and needs to find transportation (see Section 3.1.1). Let's imagine that John is visiting Zaragoza (Spain) for the main festivities (*Fiestas del Pilar*). He just arrived at the railway station of Zaragoza and wants to find transportation that could carry

him to his hotel (“Hotel Palafox”). It is the first time that John visits Spain and he does not know anything about transportation there but he would prefer a private transport that can carry him directly to the destination.

9.1.1 Knowledge for the Scenario

To be able to handle this scenario SHERLOCK needs knowledge about transportation services in the area as well as the surroundings (e.g., about the previously mentioned hotel). Figure 9.1 shows an excerpt of the ontology which models a definition of the different services to find transports. On the one hand, the general *Find Transportation* service returns any type of transportation and could be part of the local knowledge of the device prior to arriving in Zaragoza. This service will be processed as a SHERLOCK query as explained in Section 6.2. On the other hand, the *Find Bus Tuzsa* service is a particular service that operates in the city and returns the location of buses belonging to the local bus corporation of Zaragoza through a web service (which was unknown for John). Notice also that we define other services to obtain the location of bus stops and information about buses from them.

Let’s imagine that the knowledge about Zaragoza and the different transportation means which operate in it has been shared by a device in the tourist information center at the railway station. This knowledge has been integrated into the local ontology on John’s device. Therefore, the moment John runs SHERLOCK on his smartphone at the railway station, his device learns this knowledge by autonomously communicating with the information center.

9.1.2 Steps Followed

We show the most important steps that explain how SHERLOCK deals with this scenario focusing on the request generation part (i.e., translating the user information needs into a SHERLOCK request). For some of them we will use screenshots of the SHERLOCK prototype we developed and which we explain in Section B.1.2.

1. John types in *Hotel Palafox* in SHERLOCK’s search bar (see Figure 9.2(a)). A User Request Manager (URM) agent is created which finds an instance of the hotel class whose name corresponds to that string, and therefore understands that the user is interested in a hotel.
2. The URM deduces, after querying the local ontology on the user device, that there are two LBS related to hotels in its local ontology: *Find*

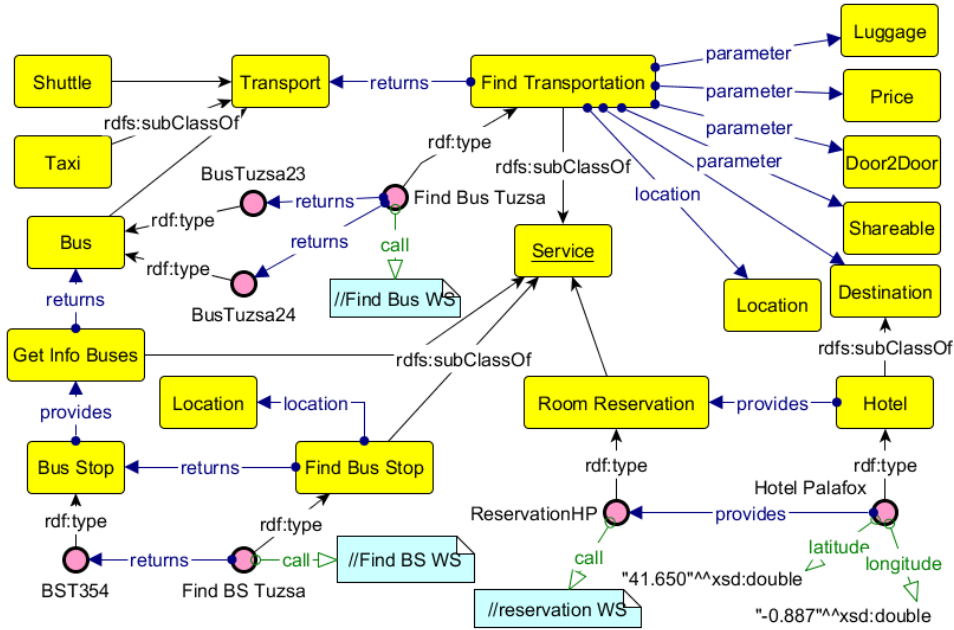


Figure 9.1: Excerpt of the ontology for the “Looking for Transportation” scenario.

Transportation and *Room Reservation*. Remember that SHERLOCK looks for services that are somehow related to the concept *Hotel* whatever the name of the property that references such a concept is (we do not assume any predefined schema in the definition of services). In this case, the properties are *parameter* (because *Hotel* is a subclass of *Destination*, which is a parameter of the *Find Transportation* service) and *provides* (because *Hotel* provides the *Room Reservation* service), respectively.

3. The user selects the *Find Transportation* service and the URM obtains from the local ontology the parameters of such a service, (*Price*, *Shareable*, *Door2Door*, and *Luggage*), to allow the user to specify his preferences. Then ADUS generates a GUI to fill in these parameters as shown in Figure 9.2(b).
4. The user shows his interest in a transport *Door2Door* (indicating that this is mandatory) that admits *Luggage*, if possible. Then, the system infers that objects belonging to the *Taxi*, *Bus*, and *Shuttle* classes fulfill the user preferences and provide transport services. In addition, SHERLOCK

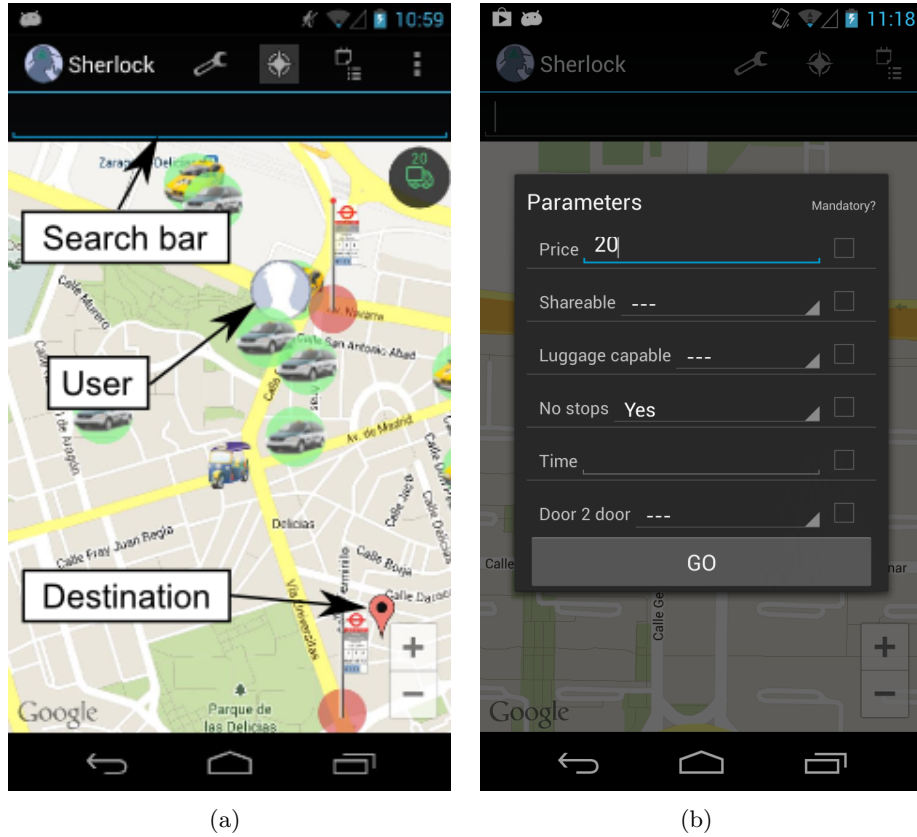


Figure 9.2: Screenshots of the SHERLOCK prototype executing the first scenario.

devices surrounding the user share that *Find Bus Tuzsa* is an instance of the *Find Transportation* service available for that specific geographic area (Zaragoza) and time, which returns information about buses in the city obtained from a web service. The URM creates two user requests to handle the two services. In particular, for the former service it translates the user information needs into the following SHERLOCK query:

```
PREFIX sherlock: <http://sid.cps.unizar.es/ontology/sherlock/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX geof: <http://www.opengis.net/def/geosparql/function/>

SELECT ?name, ?lat, ?lon
WHERE{
  LI{
```

```

    FILTER(geof:sfWithin(?thing, geof:buffer(<userloc>, 1, km))
  },
  OD{
    CASE(Type(?thing, sherlock:Transport),
      PropertyValue(?thing, sherlock:parameter, ?door2door),
      PropertyValue(?door2door, sherlock:name, 'door2door'),
      PropertyValue(?door2door, sherlock:value, true)),
    PropertyValue(?thing, sherlock:parameter, ?luggage),
    PropertyValue(?luggage, sherlock:name, 'luggage'),
    PropertyValue(?luggage, sherlock:value, true)),
    CASE(Type(?thing, sherlock:Taxi)),
    CASE(Type(?thing, sherlock:Shuttle)),
    CASE(Type(?thing, sherlock:Bus)),
  },
  PropertyValue(?thing, sherlock:name, ?name),
  PropertyValue(?thing, sherlock:latitude, ?lat),
  PropertyValue(?thing, sherlock:longitude, ?lon),
}

```

Notice that the query includes the inferred interesting transports (*Taxi*, *Shuttle*, and *Bus*) as well as the general definition of interesting transport that the user selected (a *Transport* that is *door2door* and admits *luggage*). As explained in Section 6.1, the URM includes transports that do not fulfill completely the requirements of the user (i.e., *Bus*) to maximize the chances of obtaining results. Nevertheless, it will display them on the map with a different color to highlight that they are not completely aligned to the user needs.

5. The URM agent creates two URP agents to manage both requests. On the one hand, a URP agent processes the *Find Bus Tuzsa* service by calling the web service. On the other hand, a second URP processes the SHERLOCK query to obtain taxis, shuttles, and buses located nearby with a relevant area of 1 km around the user (the area was selected by the user in the previous step) by deploying a network of mobile agents. In the meanwhile, the Ontology Updater (OU) agent discovers that there exist moving objects classified as *Bikecab* (an unknown subclass of *Taxi* for the ontology of the user). This new knowledge enriches the user device knowledge and enables the URM to infer that bikecabs also fulfill the user preferences and the query being processed is edited to reflect it in the next reevaluation of the query.
6. One of the Updater agents executing the query against other devices is not obtaining results that fulfill the user requirements at the moment. Then, this Updater decides to extend the user request. According to the knowledge modeled in Figure 9.1, buses are also returned by a service

Get Info Buses which is provided by *bus stops* which in turn are returned by the *Find Bus Stop* service. Exploiting this information the Updater deduces it can execute the service to obtain bus stops to show information which might be interesting for the user. In this case, the execution of this service triggers the call to a web service of the bus organization in Zaragoza (the *Find BS Tuzsa* service in Figure 9.1).

7. SHERLOCK presents on the GUI the interesting objects in different colors (see Figure 9.2(a)): in green, those fulfilling all the mandatory and optional user preferences (i.e., taxis and shuttles); in red, those fulfilling some optional preferences but not all the mandatory ones (i.e., buses and bus stops); the rest of moving objects displayed fulfill all the mandatory preferences but not all the optional ones (i.e., bikecabs).
8. The user could click on a bus stop icon to trigger a request to obtain the remaining time for the next bus arrival. As the user does not want to wait too much, he finally decides to click on a taxi (through its displayed icon) and selects its *Call Taxi* service to get to “Hotel Palafox”.

Notice that the information provided by the user (a click on a map, selecting the *Transportation Service*, and filling a user-friendly form) is enough for SHERLOCK to retrieve interesting transportation for that geographic area and time. John did not know anything about specific transportation means in the city (e.g., buses in Zaragoza and *bikecabs*). SHERLOCK managed all this knowledge for him and even deduced that bus stops could also be interesting when it did not find results. We have shown also how the system integrates data obtained directly from querying the moving objects in the scenario with third-party data sources (e.g., web services) specified in ontology descriptions of the services providers. This way, if the query processed against SHERLOCK-enabled devices do not return results, his SHERLOCK app might find them by using a web service as long as Internet connectivity is available.

9.2 Second Scenario: SHERLOCK for Helping Fire-fighting

In our second scenario the coordinator of a firefighting team in charge of the suppression of a wildfire needs information about his team and the scenario (see Section 3.1.2). Let’s imagine that John is the coordinator of a wildfire suppression team in Yellowstone National Park and is interested in obtaining information about fire outbreaks and the firefighter team under his command

(which consists of five firefighters, two firefighting trucks, and a helicopter). In particular, John needs information about the location of each of the members of the team as well as their sensors readings. Also, he needs the approximate location of the fire outbreaks to have information about the affected area.

9.2.1 Knowledge for the Scenario

To manage this scenario we define the knowledge in Figure 9.3. First, we model a definition of a service to monitor wild fires which returns the location of fire outbreaks as well as the location of any personnel or vehicle involved in the fire suppression (the *Fire Monitoring* service). Notice that we have modeled that the service returns also the location of people inside a *Dangerous Area*, defined as *High Temperature Area* (*hasTemperature* > 50) and *High Level of CO2 Area* (*hasCO2* > 400)¹. Second, we define information about John and his team, including the members and equipment.

As in the previous scenario, this knowledge could have been defined by a knowledge engineer working for the firefighting unit and shared with the SHERLOCK on John's device.

9.2.2 Steps Followed

In the previous scenario we focused on how SHERLOCK assists users in selecting services and filling in their parameters. Also, we showed how SHERLOCK obtains information from a discovered web service. In this scenario, where 3G connectivity is not guaranteed because of the location and the fire, we will focus on the processing of a user request which involves the creation of a network of mobile agents. In this case, we will illustrate this scenario by using the PC prototype of SHERLOCK (which we explain in Section B.1.1). The prototype includes a simulator that enables us to randomly create and move fire outbreaks and change the location of the firefighters (see Figure 9.4(b)).

1. The user taps on the service tab where a URM agent displays a list of service which can be interesting for him. In this case, the *Fire Monitoring* service appears in this list because the user selected *firefighter* as his profile and it matches the profile linked to the service through the *interestingFor* property. The URM obtains from the local ontology the parameters of this service, which in this case is the location to monitor. The URM offers John a map through ADUS to select such location of interest. With

¹Temperatures are measured in Celsius degrees and CO₂ in ppm (parts per million).

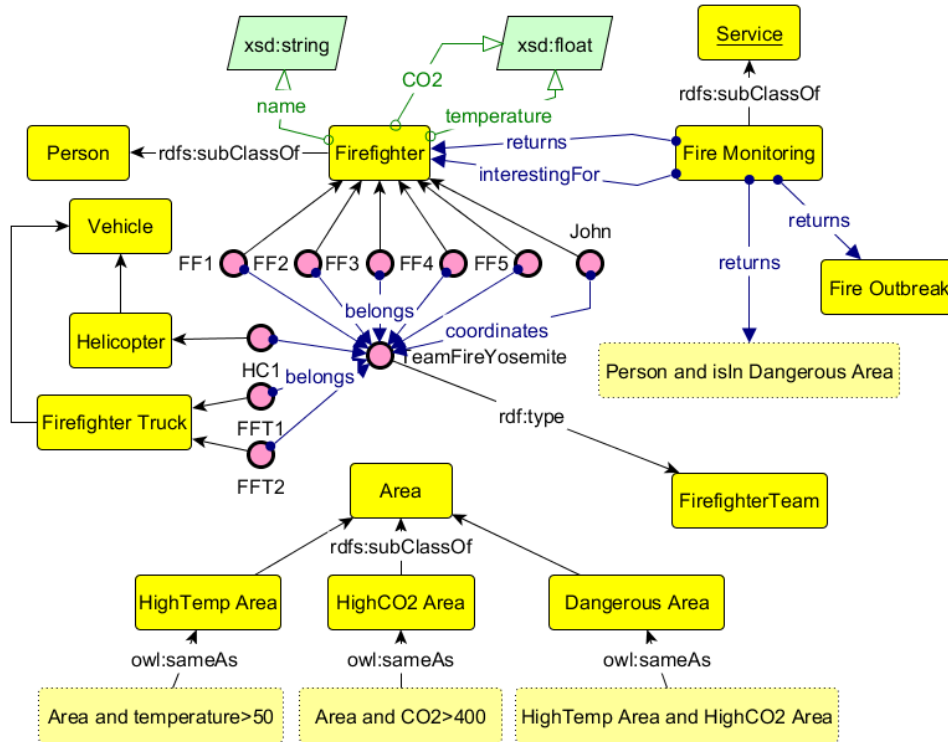


Figure 9.3: Excerpt of the ontology for the “Helping Firefighting” scenario.

the information defined by the user the URM generates a SHERLOCK query and creates a URP agent to process it.

2. The URP processes the query and creates a single Tracker agent to handle the defined location of interest. The Tracker starts executing its protocol and decides to move to a truck that provides firefighters with water (see Figure 9.5(a)) because its device has a powerful CPU and a high capacity battery and is closer to its target destination.
3. From the truck, the Tracker discovers two devices (belonging to firefighters *FF2* and *FF4*) which the Tracker estimates can cover part of the location of interest with their WiFi communication mechanism. Then, the Tracker creates two Updater agents, *Updater₁* and *Updater₂*, one on each discovered device (see Figure 9.5(b)). Then, *Updater₁* executes the query against the local knowledge on the device (through its KE

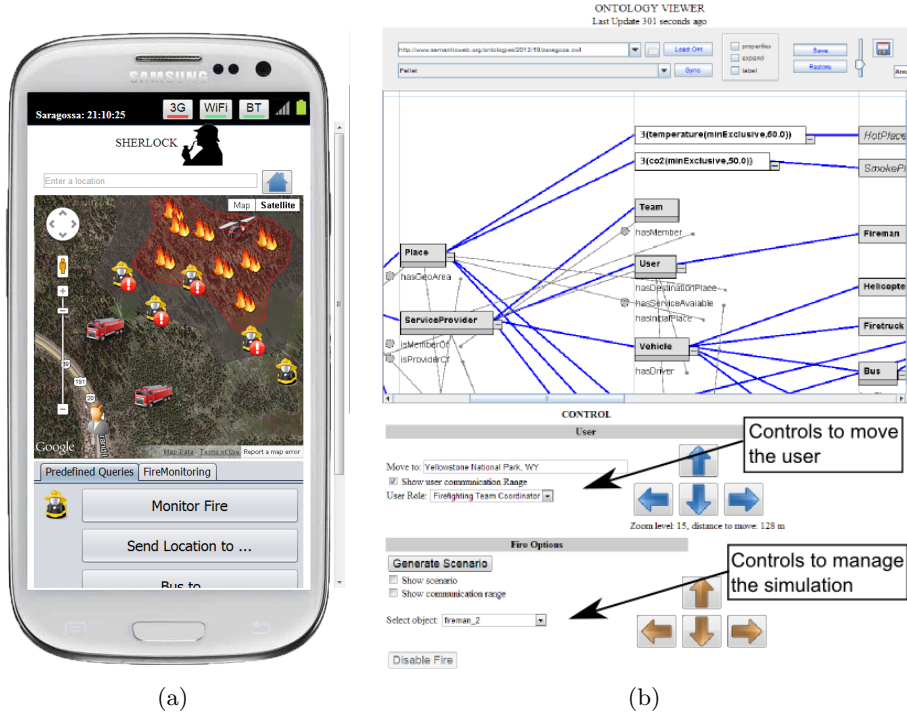


Figure 9.4: GUI of the PC prototype for the fire monitoring scenario.

agent) and obtains information about four firefighters ($FF1$, $FF2$, $FF3$, and $FF4$). At the same time, $Updater_2$ obtains information about two firefighters ($FF3$ and $FF4$). The Updater agents send this information to the Tracker which evaluates it and determines that their estimated coverage area is included in the currently covered area and therefore, no more Updaters are created (see Figure 9.5(c)).

4. The Updater agents keep querying the local knowledge on the device they are residing in and they also query the knowledge on the other devices discovered (which belong to the rest of firefighters). They retrieve all the information modeled in the ontology as properties of the firefighter (which in this case includes the readings of their CO_2 and temperature sensors) and send it continuously upwards through the network; the location of the firefighter units is presented to John along with the dangerous areas computed with the firefighter sensor measures (see Figure 9.6 in orange).
5. The Updater agents also send to the Tracker the communication delay

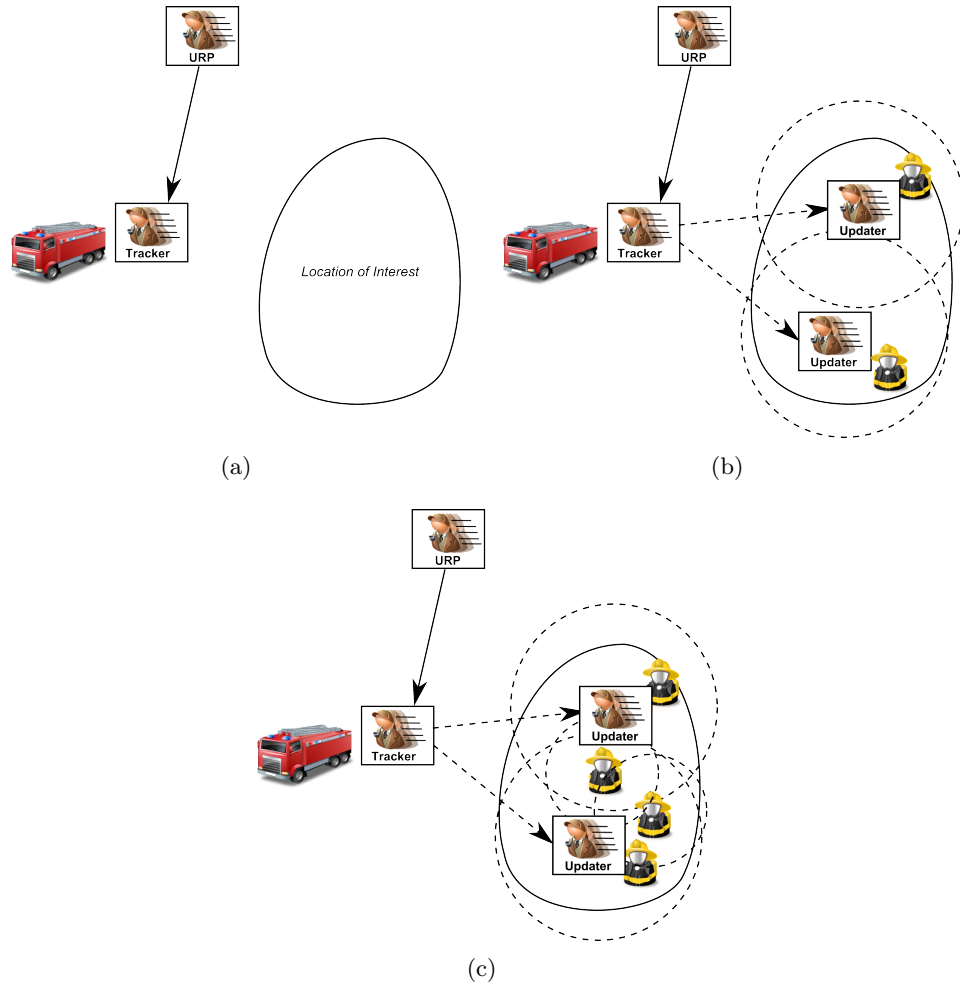


Figure 9.5: Deployment of the agent network in the fire monitoring scenario.

with the objects (see Table 9.1 for a simulation of the possible values they would send). The Tracker agent uses this information to evaluate if the updater agents are overwhelmed. The Tracker commands *Updater₂* to continue monitoring *FF3* and *FF4* because it has a better communication with them, whereas it commands *Updater₁* to stop monitoring these two devices as it seems to be overwhelmed. This way, as *Updater₁* is continuously reevaluating which is the best device to execute its task, in the next reevaluation it will try to stay close to *FF1* and *FF2* by

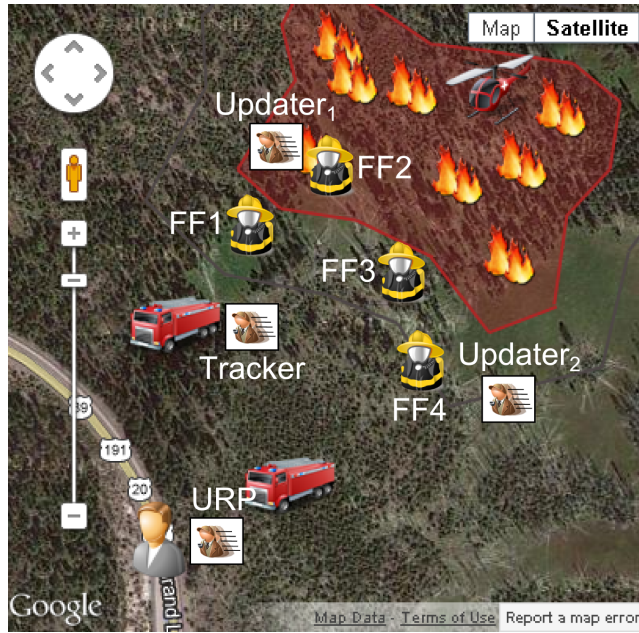


Figure 9.6: Results shown to the user in the fire monitoring scenario with the location of the mobile agents deployed.

moving to $FF2$'s device which has more battery at the moment.

Reference Object	Updater	Comm. Delay (s)	Time Stamp
$FF1$	$Updater_1$	0.64, 0.62, 0.68, 0.67	19:17:12
	$Updater_2$	1.05, 1.12	19:17:10
$FF2$	$Updater_1$	0.56, 0.0, 0.0, 0.0	19:17:12
	$Updater_2$	0.85, 0.81	19:17:10
$FF3$	$Updater_1$	0.71, 0.94	19:17:11
	$Updater_2$	0.0, 0.0, 0.0	19:17:15
$FF4$	$Updater_1$	1.32, 1.11	19:17:11
	$Updater_2$	0.43, 0.38, 0.39	19:17:15

Table 9.1: Table used by a Tracker agent to dynamically maintain its network of Updaters.

6. Then, the firefighter $FF2$ moves towards the fire and an increment on

the communication delay between *Updater*₁ and the firefighter's device occurs. The Tracker, which is analyzing the communication delays, detects that the firefighter might move out from the area covered by the current Updater agents. Thus, the Tracker creates a new Updater and sends it towards the location of *FF2*.

7. Finally, John taps on *FF2*, who has been classified as inside a dangerous area with the information the URM received, and a new URM show him the information modeled in the ontology as properties of a firefighter (i.e., name and sensor readings). Along with the information, the URM shows services related to him, for instance, a service related to a person in a dangerous area to command her to move to a location (*Move To* service). John taps on such service to require *FF2* to get closer to the rest of the team and avoid the danger.

Thus, in scenarios where the system cannot rely on a fixed infrastructure (e.g., because of the fire damaging communication infrastructures) SHERLOCK is able to leverage P2P communications to process a user request. In this scenario, we have seen how SHERLOCK agents move towards the location of interest and discover devices in it. Also, we have shown how SHERLOCK is able to manage its network of agents to dynamically adapt itself to changes such as new devices being discovered or delays in the communication with the devices.

9.3 Third Scenario: SHERLOCK for Handling Sport Events Broadcasting

In the third scenario, a Technical Director (TD) needs assistance in the live broadcasting of a sport event (see Section 3.1.3). In this case, let's imagine that John is the TD in charge of the live broadcasting of *La Bandera de la Concha 2015*, a famous rowing race celebrated annually in San Sebastian (Spain). Among the many tasks that John has to perform, the most important task is to select the camera to broadcast at each moment. John is an experienced TD and has some specific shots in his mind to broadcast. So, he would like to define them and then obtain the list of cameras (from the broadcasting company and even from the audience) that could provide them.

9.3.1 Knowledge for the Scenario

This scenario, as well as many others, needs a service to obtain cameras that could provide a specific view of different objects. We defined the general service *Get Camera* as an implementation of such services (see Figure 9.7 for an excerpt of the ontology). This service returns entities of type *Camera*, which could be even attached to a mobile devices such as an smartphone, and has parameters such as the distance from the camera to objects inside and the visibility of such objects in the field-of-view of the camera. Notice that, for the latter we have modeled that such parameter can be obtained through a specific GUI (a 3D Query-by-example interface). Also, we have defined two services provided by these cameras to ask them to take pictures and videos and share them with the requester, the *Take Picture* and *Take Video* services, respectively.

We have defined also a service to manage the broadcasting of a sport event (see Figure 9.8 for an excerpt of the ontology) and a specific instance of this service for the rowing race in the example (*BroadcastingBanderaConcha2015*). Notice that, this service obtains information about the rowing boats participating in the race and the cameras of the broadcaster. Also, we have defined the knowledge related to the rowing race which is used in combination with the previous definition of the service. The knowledge about the race includes the participants of the rowing race and cameras managed by the broadcasting company, as well as possible interesting locations for the broadcast (such as the *ciaboga* area which is the turning point for the boats). All this knowledge would be defined by the broadcasting company and its information system would share it with the SHERLOCK on the TD device.

9.3.2 Steps Followed

For this scenario, we will focus on the steps related with the multimedia information component, that is, specifying the kind of views the user is interested in obtaining, and processing the views provided by cameras in the scenario to retrieve the cameras that can provide interesting shots.

1. First, the TD selects the *Manage Broadcasting* service to obtain the real-time location of the rowing boats and the cameras under his control. The URM in charge of handling this request creates a URP agent that processes the SHERLOCK query generated. After deploying a Tracker agent to cover the location of interest (in this case the bay), similarly to the previous scenario, results are displayed on the map of the SHERLOCK-enabled device of the TD.



2. Then, the TD is interested in broadcasting a shot of the local team rowing boat (*Donostiarra*). For that he first wants to obtain a list of cameras that could provide such a shot to select one of them and broadcast its feed.

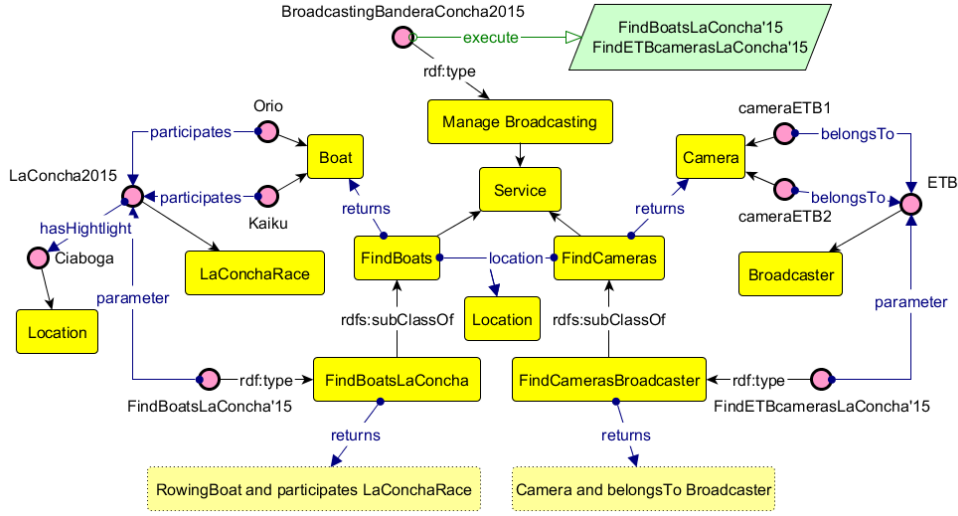


Figure 9.8: Excerpt of the ontology for the “Handling Sport Events Broadcasting” scenario.

Therefore, the TD selects the *Get Cameras* service and a URM agent obtains the parameters associated to such a service in the ontology. As one of the parameters is the definition of a sample shot, ADUS displays the 3DQBE (3D Query-By-Example) interface (linked to the parameter through the *interface* property) to define the specific shot of the rowing boat to retrieve (see Figure 9.9). On that interface, the user defines the kind of shot to obtain by rotating the camera view and even including other objects in the scene. The interface translates this sample shot into: “An image showing 50% of the front view and 70% of the right side view of Donostiarra, and 40% of the front view and 15% of the right side view of any other rowing boat”².

3. The URM translates the information obtained from the 3DQBE interface to a formal query expressed in SHERLOCK’s language. Then, it creates a URP agent to process it which first executes the query against the local knowledge on the device. Notice that the other user request (associated to the *Manage Broadcasting* service previously selected by him) is still active and the agents processing it are continuously retrieving results.

²Notice that the features regarding to the location of the boats in the image and the percentage of the shot filled with them have been omitted for readability.

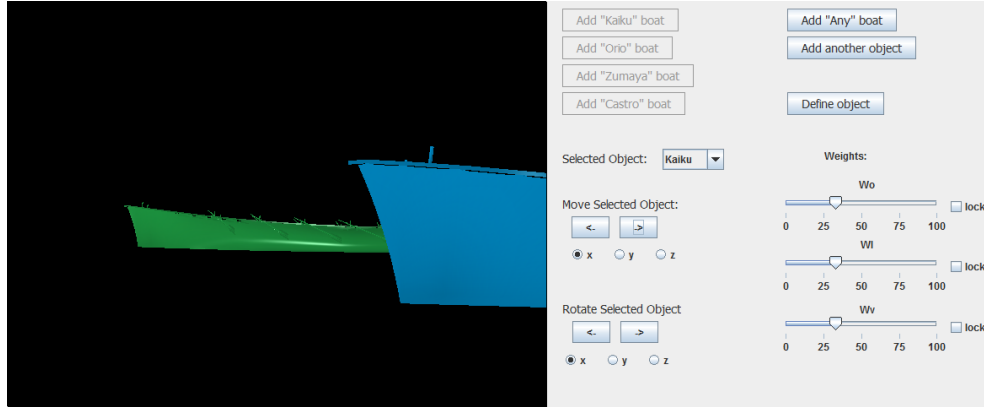


Figure 9.9: Definition of the shot that the technical director wants to broadcast.

Therefore, the local knowledge is being updated with information about the rowing boats (such as their location, direction, extent, and top/front vectors) and the broadcaster cameras (such as their location, direction, FOV, current pan and tilt). So, when the URP created to handle the *Get Cameras* service executes the query against the local knowledge on the device, it retrieves such information.

4. With this information about the rowing boats and cameras in the scenario, the URP module in charge of obtaining high-level features of the view of a camera recreates the scene in the 3D engine. Then, for each camera retrieved from the local knowledge, it computes what the camera is viewing (as explained in Section 8.2) and obtains the similarity degree when compared with the requested shot (as explained in Section 8.3). For example, for one of the cameras in the scenario the system recreates its view in the 3D engine, as Figure 9.10 shows.
5. The URP obtains high-level features for the rest of cameras and compare them to the view defined by the TD. Then, the URP agent sends the ranked list of camera views to the URM which requires ADUS an appropriate interface to display the results (which in this case are renders of camera views, and therefore, pictures). ADUS displays the images that the cameras could take as shown in Figure 9.11.
6. From these results the TD selects the first one, which he thinks looks similar to the shot he want to broadcast. Then, the TD orders the



Figure 9.10: Camera view recreated in a 3D engine by the system.

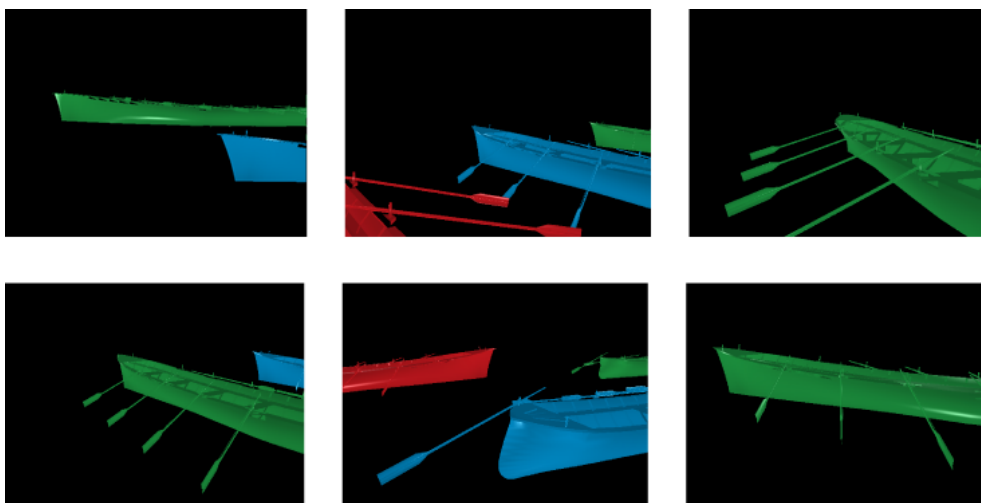


Figure 9.11: Some of the camera views returned to the TD.

corresponding camera feed (see Figure 9.12) to be broadcasted (the selection and streaming of the camera feed is outside of SHERLOCK).

As we have seen, SHERLOCK is able to manage complex information requests involved the specification of the multimedia information to retrieve. This way, the system enables users to use a 3DQBE interface to visually define



Figure 9.12: Camera feed broadcasted.

the kind of camera shot they want to obtain. Then, the system is able to retrieve information about cameras and objects in the scenario and use this information to compute high-level features of the views of such cameras. Finally, it can use these features to compute a similarity degree between the view of each of the cameras and the shot that the user defined in order to present the cameras which could provide him with the best content.

9.4 Fourth Scenario: SHERLOCK for Emergency Management

In the fourth scenario, the user is the coordinator of an emergency response team who has to manage the a request for assistance to help the people affected by an emergency (see Section 3.1.4). Let's suppose that John is the coordinator of the emergency teams assigned to a traffic accident. The accident occurred in the US160 highway mile 32 and there are two vehicles involved.

9.4.1 Knowledge for the Scenario

For this use case, we have defined two main services (see Figure 9.13 for an excerpt of the ontology). On the one hand, the *Emergency Call* service, which is meant to be used for people in distress (like the drivers involved in the accident of the use case), is composed of two subservices *Search Assistance* and

Request Assistance. The *Emergency Call* service includes a workflow (linked through the *execute* property) which describes how these subservices should be executed. On the other hand, we defined the *Manage Emergency* service which is interesting for emergency coordinators. They can use this service to automatically find emergency units (*Find Emergency Team* service), ask them to move to the location of the accident (*Move To*), and obtain and send them appropriate pictures of the area (*Obtain and Send Picture*). Notice that we have defined emergency units as police, medical, and firefighting units. Also, the *Obtain and Send Picture* service uses the *Get Cameras* and *Take Picture* services of the previous scenario to find them cameras that can take pictures of the accident and ask them to take pictures. Then, the service uses the *Send Picture* service provided by the device to send the photos obtained to a set of emergency units.

9.4.2 Steps Followed

In this case we will focus on the processing of complex services, which involve the execution of other services transparently to the user. As we will mention services and the subservices which compose them, we will use Figure 9.14 to show the general picture of the scenario with the flow of events.

1. An emergency call is issued by the driver of a car involved in the accident (step 1 in Figure 9.14). For that, the user selected the *Emergency Call* service³, which is defined in the ontology as a composed service. This service triggers the *Search Assistance* service to find help and for each emergency unit detected it calls its *Request Assistance* service.
2. First, the URM handling the selected *Emergency Call* service, from now on URM_1 , determines that the service is composed of others (because the *execute* property is included) and creates a URP, URP_1 , to handle it. Then, URP_1 , using the module to process BPMN workflows, takes the XPDL code and starts executing it by creating another URM, URM_2 , to handle the *Search Assistance* service.
3. The new URM_2 agent, attending to the definition of the service in the ontology, generates a SHERLOCK query and creates a URP agent, URP_2 , to process it. As explained in previous scenarios, URP_2 processes the query creating the network of mobile agents to find emergency units

³Notice that in the future this service could be even automatically selected by the car itself.

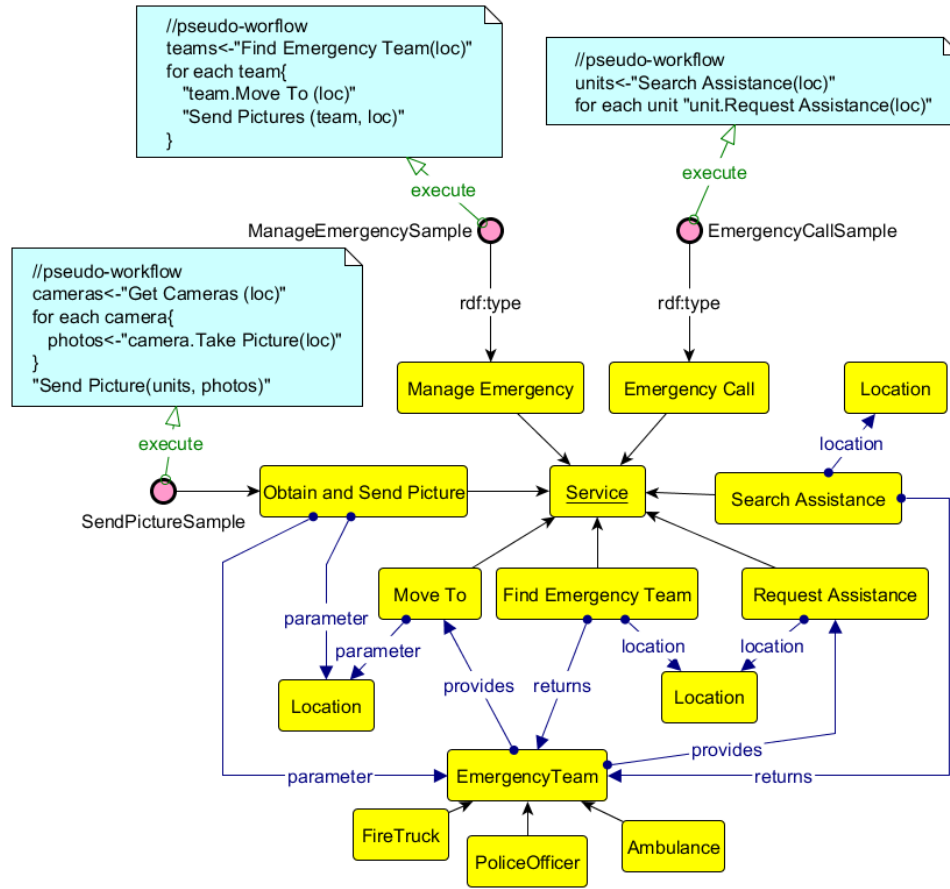


Figure 9.13: Excerpt of the ontology for the “Emergency Management” scenario.

around. Once the results are returned by its Tracker agent, URM_2 send the results back to URP_1 , the agent processing the service selected by the user. According to the workflow, for each emergency unit found, its *Request Assistance* service has to be called with the location of the user as parameter. In this case, URM_2 found John and thus, a URM agent is created to handle the petition for assistance, which in turn creates a URP to execute the service on his device.

4. Once John receives the petition through his SHERLOCK device, he selects the *Manage Emergency* service (step 2 in Figure 9.14). The URM that

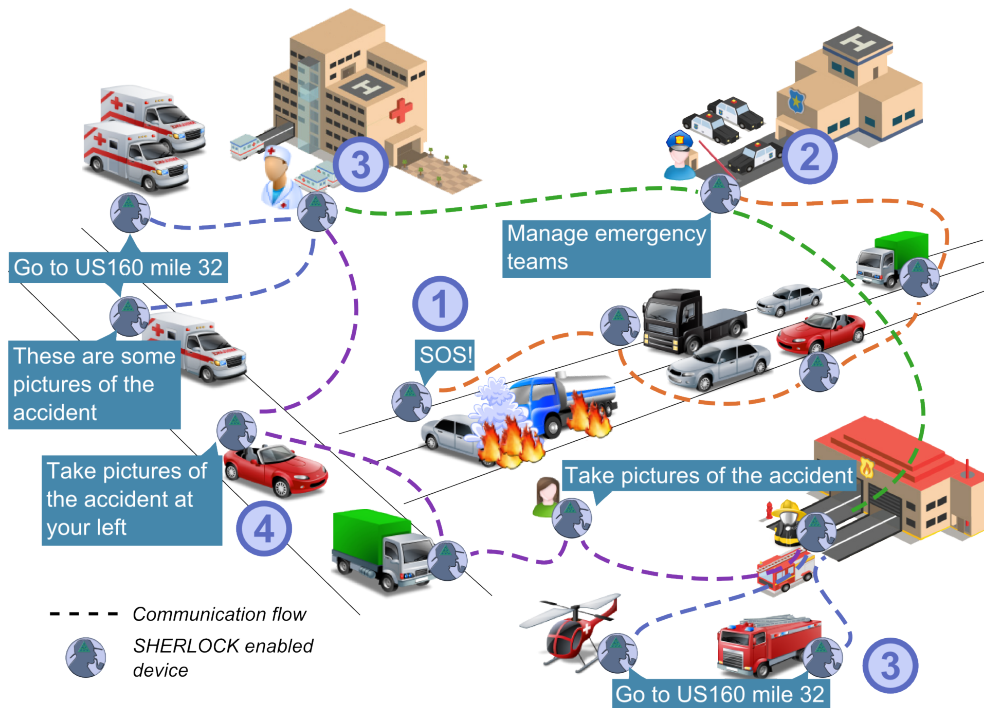


Figure 9.14: Emergency management scenario.

gets created to handle this service requires John to input the parameters (in this case, the location of the accident that he received with the petition of assistance). Then, the URM agent creates a URP agent to process the request. Again, this request is composed of three other services (the *execute* property links it to the workflow): 1) The *Find Emergency Team* service to find emergency teams near the area; 2) The *Move To* service to require them to move to the location of the accident; and 3) The *Obtain and Send Picture* service to find and send them pictures of the accident. The service is processed similarly to the *Emergency Call* service explained before and in this case it first finds ambulances, fire trucks and a helicopter in a nearby hospital and fire station and ask them to move towards the accident (step 3 in Figure 9.14).

5. The URM agent created to handle the service to find and send pictures determines that it is a composed service which first finds cameras that could obtain pictures of the accident, then asks them to take pictures, and finally, sends the pictures to the units. The service to find cameras

is the *Get Cameras* service explained before in the broadcasting scenario and it is executed in the same way. However, in this situation John does not define any specific shot to obtain as any image of the accident would be helpful. Once cameras have been found, the Updater agents created ask users in the area to take photos of the accident (step 4 in Figure 9.14) if they are not getting results by applying their request extension capability.

6. Finally, the pictures are sent to the emergency units which are on their way to the accident. At the same time, the URP agent processing the *Manage Emergency* service is obtaining the real-time location of the emergency units and communicating this information to the URM which show them on a map.

We have shown how SHERLOCK is able to handle complex services like the *Emergency Call* and *Manage Emergency* in this scenario. For the former service, the user simply selected a service after the accident and the system triggered a search for help that eventually reached a police officer. For the latter service, the officer, which had to act quickly, selected a service and used as parameter the information which he received about the accident. Then, the system triggered a search for emergency units nearby and require them to move towards the accident. Also, notice how the modeling of the service included sending real-time pictures of the accident to emergency units to let them be ready to help, and SHERLOCK found a way to obtain such images by asking users equipped with cameras to take them.

9.5 Other Extra Scenarios

The main goal of this thesis was to design a general and flexible system to provide many different LBS. The previous four scenarios have shown that the designed system is able to address the heterogeneous challenges that arise in the context of the provision of LBS in wireless environments. We have seen how SHERLOCK helps users to define their information needs (which might involve obtaining multimedia information such as images provided by cameras), then it processes the user request against different sources (local knowledge on the device, external services, and other SHERLOCK devices in the scenario), and finally it shows the results to the user. These functionalities are possible thanks to the knowledge that the system integrates and which defines services and scenarios as well as the mobile agents that find the information wherever

it is. Thus, the system: 1) is decoupled from the contextual knowledge of the scenario; and 2) is capable of adapting itself automatically to different situations.

To further highlight that SHERLOCK is flexible enough to provide different services, we present in the following another three scenarios where we have used SHERLOCK. These scenarios do not present new challenges, as the previous four already covered a great variety of challenges for a system like SHERLOCK, but help to show the generality of our approach. Indeed, we will further show in the following that just by defining the knowledge related to the LBS and scenario as an ontology, and providing a SHERLOCK device with that, it is able to offer the functionality to the user. First, we explain two simple scenarios where the user is interested in obtaining tourist information and finding fellow researchers in a conference. Then, we explain a more complex scenario where the system can help community health-care workers in underserved areas. Bear in mind that SHERLOCK was not specifically designed for any scenario, nor the previous four nor the following extra scenarios, but nevertheless, the system is able to support them.

9.5.1 SHERLOCK for Obtaining Tourist Information

John is a tourist that just arrived in Rio de Janeiro. After using SHERLOCK to find a taxi to his hotel, he is in his room planning what to see the next day. John is interested in visiting different monuments and museums and therefore wants to locate them on the map and obtain information such as their price and schedule.

Knowledge for the Scenario

Along with the knowledge of the area that SHERLOCK found at the airport about transportation in the city, a tourist information stand shared a *Find Tourist POIs* service with information about some of the points of interest of the city (see Figure 9.15). The ontological definition of such a service relates it with different “static” objects, such as *monuments*, *cathedrals*, *parks*, etc. Moreover, the service has been also related (in the ontology) with some “moving” objects, such as *tourist buses*, *tourist guides*, etc. It also shows that there exists a related Web service that provides the location of some of these objects (*Google Places*).

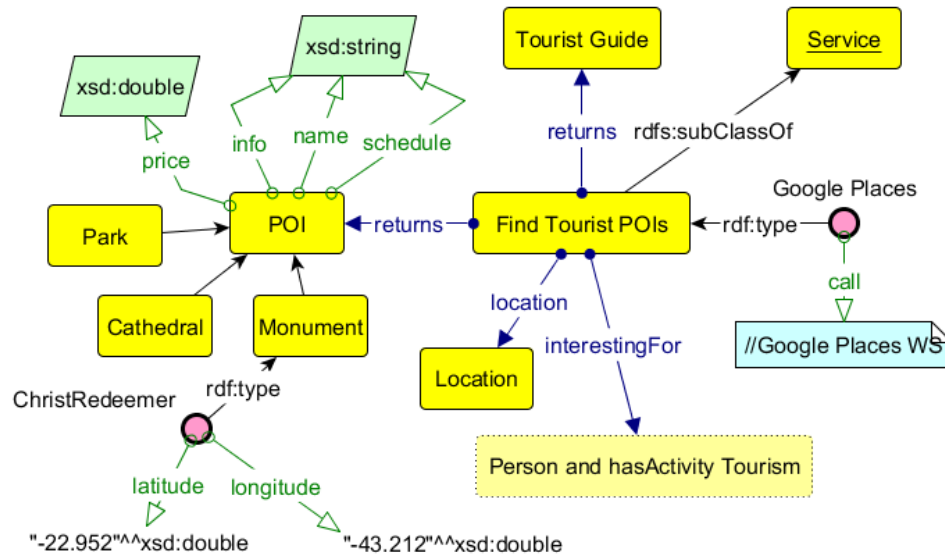


Figure 9.15: Excerpt of the ontology for the “Obtaining Tourist Information” scenario.

Steps Followed

John first taps on the services tab on his SHERLOCK app and a URM agent offers him the *Find Tourist POIs* service, defined as interesting for tourists. After selecting the service, the URM creates a URP to handle the call to the external web service and another URP to handle a SHERLOCK query to find the objects related to the service (linked through the *returns* property). After getting results from the URP agents (for instance, the former URP finds the *statue of Christ Redeemer* and the latter a tourist guide), the URM agent shows them on the map (see Figure 9.16(a)). Finally, John taps on the icon representing the statue of Christ Redeemer on the map to retrieve more information. SHERLOCK generates a URM that shows the local knowledge collected about this point of interest: properties of the object (e.g., price and schedule) and related services such as the *Find Transportation* service explained for the first motivating scenario.

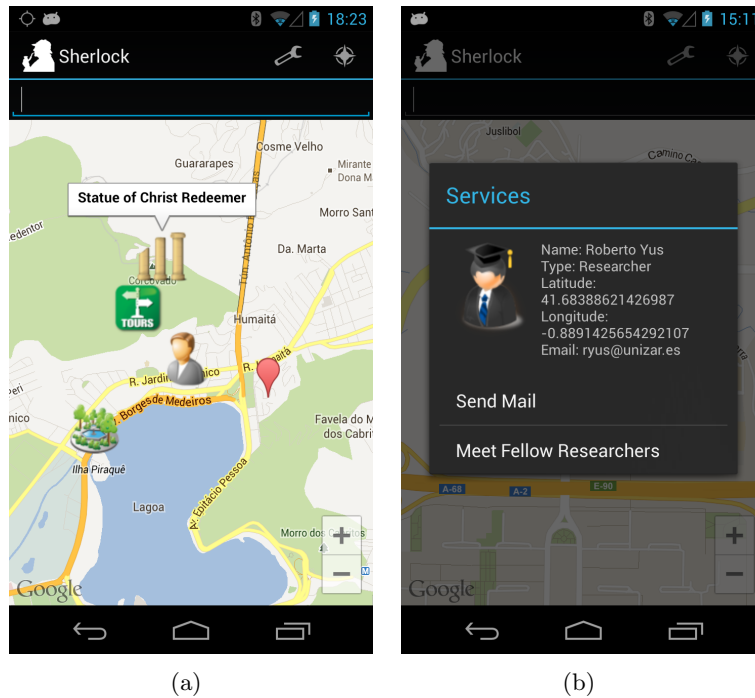


Figure 9.16: SHERLOCK obtaining: (a) interesting touristic points and (b) information about a researcher.

9.5.2 SHERLOCK for Meeting Fellow Researchers

After sightseeing in Rio, John attends the WWW'13 conference (which is happening in a hotel in the city) to present his work. John is a PhD student attending his first research conference and a common problem in this situation is to find other researchers with similar interests to talk with them. John could start talking to people to find their research fields, but he is shy, or check the names on their badges on DBLP, but that takes effort and time.

Knowledge for the Scenario

Let's imagine that the conference organizers made some knowledge available in the hotel for customers registered in the conference (see Figure 9.17). In this case, they created an ontology describing the attendees and defining a *Find Researchers* service. John is interested in locating fellow researchers from his same research field to talk to them, so the recently found service could

be very handy. The ontological definition of the service establishes that it is *interestingFor* WWW'13 attendees and the knowledge that the organizers shared described the user as such.

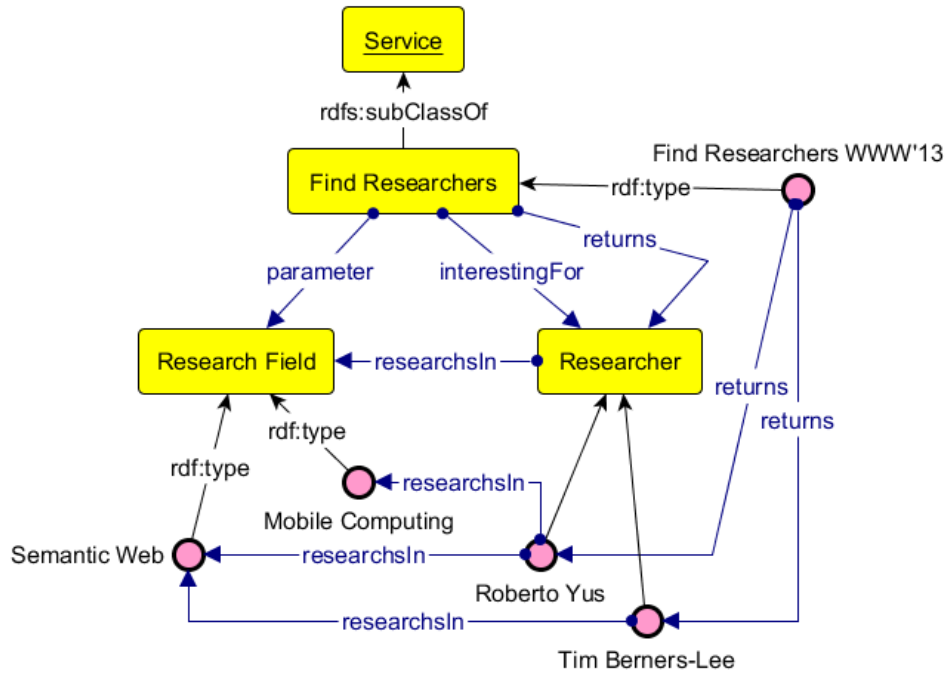


Figure 9.17: Excerpt of the ontology for the “Meeting Fellow Researchers” scenario.

Steps Followed

John taps on *Find Researchers* and a URM agent ask him for parameters associated (in this case, the research field). When John taps on the interface to fill in such parameter, the URM shows a list of instances of research fields in its local knowledge (shared along with the service by the organizers). With this information the URM generates a SHERLOCK query and creates a URP agent to process it. Then, the URP executes the query against the local knowledge and retrieves the researchers shared by the organization. As the timestamp of the information shared by the organizers is outdated for dynamic information such as the location of the results, the URP also creates a network of RRE

agents to find this information around. Recall that SHERLOCK queries that do not depend on a location are processed by these agents (see Section 7.1.1) and the service is not location-dependent as the *location* property is not defined in the ontology for it. These RRE agents move to devices of other attendees and return their location (whenever their privacy preferences allow it) to the URM agent that shows them on the map. Finally, John taps on a researcher (*Roberto Yus*) and the URM handling this request shows the information it has about him, as well as other services related to a researcher, such as the *Send Mail* service (see Figure 9.16(b)).

9.5.3 SHERLOCK for Helping Health-Care Workers

The lack of access to adequate care in medically underserved areas is one of the biggest challenges in health-care in both developing and developed countries [AEJJIM13]. Community Health Workers (CHW), who act as liaisons between patients and health-care providers in these areas, have been able to address this problem to certain extent. CHWs are typically high school educated and use simple forms and manuals for collecting information about the patient by filling in information about symptoms and patient demographics as well as diagnosing the patient.

Let us look at a possible scenario developed on the basis of the ASHA⁴ manual [Ind], which points out inefficiencies of CHW program mentioned in [SS12] and how SHERLOCK could help to deal with such issues. Imagine a CHW who arrives at a house in Sirpur, a small village in the Chattisgarh state of India, after being informed of a child presenting some problems. Kumar, a six-year old boy, has been experiencing vomiting according to his parents. The CHW, following the manual, starts collecting details about Kumar and his problem using specific forms. Finally, the child is identified as having diarrhea and the CHW provides the parents with oral rehydration solution (ORS) and advises them on food intake. Even though the CHW indicated that Kumar suffers from severe dehydration, a dangerous and potentially life-threatening condition, she did not feel the need to refer the child. SHERLOCK could help in this situation by managing the knowledge about diseases and questions to ask, alerting the CHW about possible emergency situations, and keeping her updated with the latest information related to diagnosis.

⁴Accredited Social Health Activist (ASHA) is a CHW organization in India.

Knowledge for the Scenario

The knowledge needed to support the Community Health-care scenario can be split into three main modules:

- *Diagnosis module*, which models the basic knowledge related to diagnosis such as diseases, symptoms, and questions (see Figure 9.18).

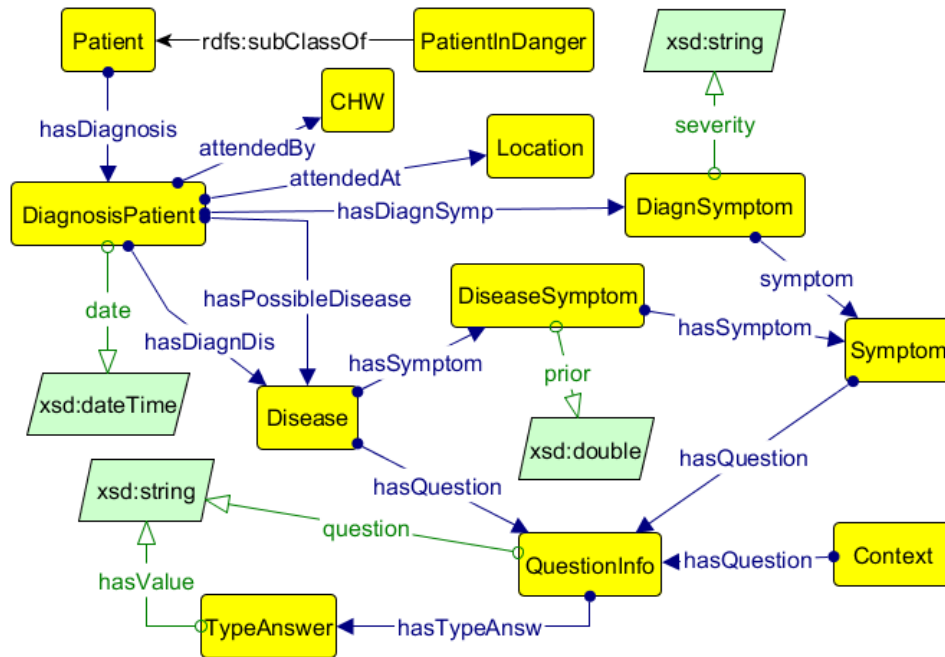


Figure 9.18: Excerpt of SHERLOCK's ontology for the CHW scenario: diagnosis module.

- *Patient module*, which includes all the information related to the patient which might be of relevance for diagnosis of the disease (see Figure 9.19). The primary patient information considered by SHERLOCK are location and patient demographic.
- *Stats module*, which models the aggregated health-care information related to patients and diseases in various regions (see Figure 9.20).

Also, we defined some SWRL rules which are used along the previous knowledge to define, for instance, possible diseases for a patient (we will show

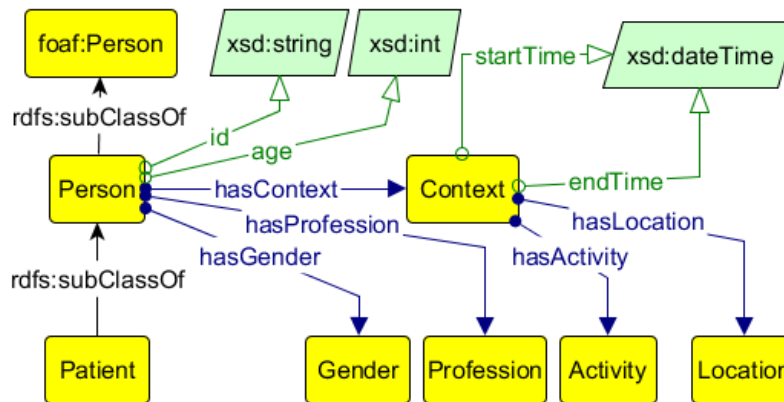


Figure 9.19: Excerpt of SHERLOCK's ontology for the CHW scenario: patient context module.

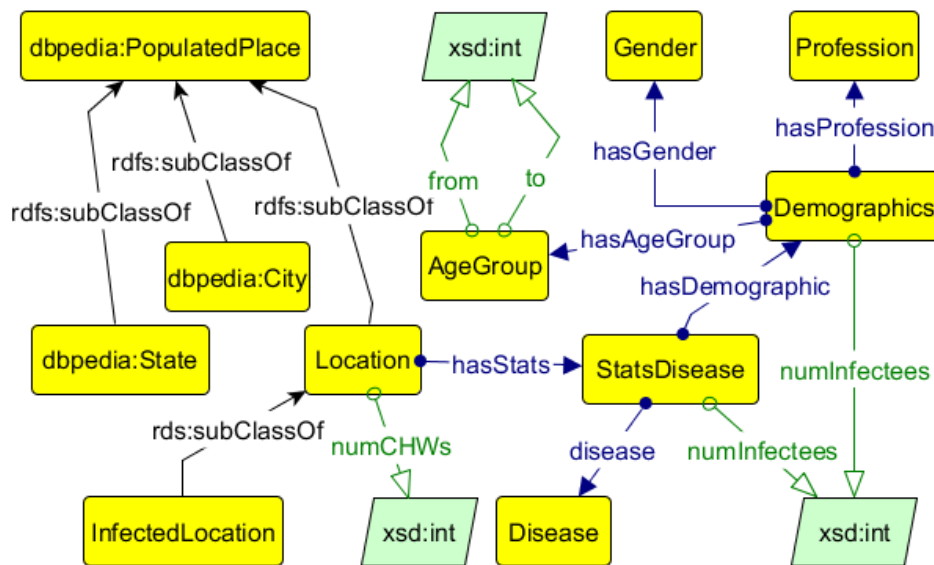


Figure 9.20: Excerpt of SHERLOCK's ontology for the CHW scenario: stats module.

these rules in the following section when they are applied). In addition, the definition of a service for the diagnosis of patients is needed. In this case,

Rafiki is an external diagnosis service which has been installed in the device. *Rafiki*'s definition in the ontology links it to the code to execute, as in the example of the *TakePicturesCamera* service in Figure 4.7. Let's consider that the CHW device receives the previous knowledge about diseases and statistical information about diseases in her current region (Sirpur), as well as the *Rafiki* service.

Steps Followed

In the following we explain the most important steps for the managing of this scenario by SHERLOCK:

1. The CHW runs the *Rafiki* service on SHERLOCK to diagnose the patient. This service, which has been shared with her device, is offered by SHERLOCK but provided by the device itself (the *call* property links it to the execution method) and uses the local knowledge on the device about diseases and symptoms. First, the URM handling the service asks the CHW for parameters such as the patient context information, that is location and demography.
2. SHERLOCK on the CHW device obtains the location of the patient (Sirpur) automatically using the GPS of the device and Kumar's parents will be asked about the age and gender of the kid. Therefore, the following information will be entered into the system: *Patient: name=Kumar, age=6, gender=male, location=Sirpur*.
3. The information gathered about the patient context, along with the stats module of the ontology, is used by *Rafiki* to infer the list of possible diseases based on others with similar context (e.g., neighbors from the same age group). For that, *Rafiki* queries the local knowledge, through the KE agent, to retrieve diseases linked to the patient through the *hasPossibleDisease* property. As, explained before, The knowledge shared with SHERLOCK includes some rules, and one of them classifies diseases as possible for a patient given his context (in this example, age group):

```

Person(?p) ∧ hasLocation(?p,?l) ∧ age(?p,?ageP) ∧
InfectedLocation(?l) ∧ hasStats(?l,?s) ∧ disease(?s, ?d) ∧
Disease(?d) ∧ hasDemographics(?s,?dem) ∧ hasAgeGroup(?dem,?ageG) ∧
from(?ageG,?minAge) ∧ to(?ageG,?maxAge) ∧
swrlb:lessThanOrEqual(?ageP,?maxAge) ∧
swrlb:greaterThanOrEqual(?ageP,?minAge)
→ hasPossibleDisease(?p,?d)

```

Several cases of diarrhea were detected in Sirpur and this knowledge was shared in the *stats module*. So, the reasoner classified the region as *InfectedLocation*. In addition, stomach flu was diagnosed to other kids in Sirpur so, it could be a diarrhea outbreak and the disease is added as a possible one for Kumar (i.e., *hasPossibleDisease(Kumar,StomachFlu)*).

4. The URM asked the CHW for another parameters, for instance, for other reported symptoms that can be used to add more diseases to the list. Reported symptoms can be common symptoms like fever, vomiting, or aches for which the patient might have got in touch with a CHW. The following rule is applied with the information shared about Kumar and adds more diseases to the results retrieved by the query executed by *Rafiki*:

$$\begin{aligned} & \text{Person}(\text{?p}) \wedge \text{hasSymptom}(\text{?p}, \text{?s}) \wedge \text{Disease}(\text{?d}) \wedge \text{hasSymptom}(\text{?d}, \text{?s}) \\ & \rightarrow \text{hasPossibleDisease}(\text{?p}, \text{?d}) \end{aligned}$$

Kumar was suffering from vomiting and therefore, viral diarrhea (gastroenteritis) is added to the list of possible diseases.

5. The query that *Rafiki* executed ranks the list of possible diseases according to the information gathered about reported symptoms and patient context:
 - 1) diseases are ranked according to the number of their reported symptoms;
 - 2) for diseases with the same number of reported symptoms, the number of infectees for each disease with similar patient context (e.g., location and age group) is used. Therefore, the final result of the query for Kumar includes viral diarrhea as the first disease to check, as he suffers from a known symptom of this disease and it is common among other children in the area.
6. Now *Rafiki* requires the CHW to ask questions related to the symptoms of the possible diseases in order to detect the most likely disease. These questions are obtained by querying the KE agent about the diagnosis module of the ontology based on the disease and symptoms. The CHW inputs the patient response (“severe dehydration”) by selecting the answer from the available options.
7. Severe dehydration is a sign of worry in patients, especially in children, with diarrhea as a possible disease and should be immediately referred to the health-care provider. The authorities have shared a rule capturing this guideline with the device of the CHW:

```

Person(?p) ∧ hasPossibleDisease(?p,Diarrhea) ∧ hasDiagnosis(?p,?dp) ∧
hasDiagnSymp(?dp,?s1) ∧ symptom(?s1,DryMouth) ∧ severity(?s1,"Severe") ∧
hasDiagnSymp(?dp,?s2) ∧ symptom(?s2,Fever)
→ hasEmergencySymptom(?p,SevereDehydration)

```

At each step of diagnosis, *Rafiki* verifies if the patient has been classified under the *UrgentCareNeededPatient* (*Patient and hasEmergencySymptom some DiagnSymptom*) class in the local knowledge managed by SHERLOCK by querying the KE agent. In this case, the previous rule generates a fact stating that *Kumar is-a UrgentCareNeededPatient*.

8. Finally, *Rafiki* suggest the CHW the service to report the kid to the nearest health center which starts a call with a health-care provider.

As shown, the *Rafiki* service benefits from the management of knowledge performed by SHERLOCK. Thus, the system removes a burden from the CHW by: 1) keeping the knowledge about diseases updated and 2) using the reasoner to evaluate the rules that the CHW would have to do manually.

9.6 Summary of the Chapter

In this chapter, we have revisited the motivating scenarios described in Section 3.1. We have explained how SHERLOCK deals with them to show that the system designed is able to tackle the challenges we identified in these representative scenarios. For each use case, we have shown a possible modeling example of the services involved and the scenario itself, which is shared and integrated by the system to handle it. Also, we have described the steps involved in its processing focusing on different parts of the process depending on the scenario. This way, for the first scenario, where the system retrieves transportation means for the user, we focused on how SHERLOCK captures the user information needs and translates them into a formal request. In the second scenario, where the system helps the coordinator of a firefighting unit suppressing a wildfire, we focused on how SHERLOCK deploys the network of mobile agents to obtain information from other devices using their wireless communication mechanisms. In the third scenario, where the system assists a technical director broadcasting a rowing race, we focused on how SHERLOCK enables the definition of complex information needs (in this case a camera shot of the rowing boats) and how it processes camera views to obtain high-level features of them. Finally, in the fourth scenario, where the system assists the coordinator of an emergency unit, we focused on the processing of complex

services which involves the execution of different services in a certain order and by combining the information they retrieve. Also, we have shown how SHERLOCK deals with other extra scenarios, which do not add new challenges but show how new services and scenarios can be modeled and shared with the system. The scenarios presented in this chapter show the flexibility of the system, which is able to handle them by receiving an ontology which models such scenarios.

Chapter 10

Conclusions

In this chapter, we show some conclusions about the work presented in this dissertation. First, we present the main contributions of SHERLOCK as a system and then, we present contributions to different fields such as ontology alignment, context-awareness, high-level feature extraction in multimedia information, and the use of semantic technologies on mobile devices. After this, we present the publications related to our work, analyzing their quality according to different quality index rankings. Finally, we indicate some future lines of work opened.

10.1 SHERLOCK: Main Contributions

In this thesis, we have presented SHERLOCK, a general system that provides support for Location-Based Services (LBS) that depend on highly-dynamic information and infrastructures. SHERLOCK-enabled devices collaborate by exchanging their local knowledge and can also become processing nodes when managing information requests from their users. Besides, we have introduced four different sample motivating scenarios that can be tackled by our system; any other use case where a user is interested in obtaining information about moving objects or in asking them to perform actions in highly-dynamic distributed scenarios can also be processed by SHERLOCK. As a summary, the original contributions of SHERLOCK are the following:

- It enables devices to exchange knowledge related to services which might be interesting for their users. This knowledge is modeled as ontologies to avoid imposing the users with a global schema. Through their interactions, SHERLOCK devices exchange ontologies modeling services and scenarios

which the system integrates into their local knowledge using ontology alignment techniques. Therefore, SHERLOCK devices can learn about their surroundings from the interaction over time. Also, it maintains the local knowledge on the device updated while taking into account its size to enable efficient reasoning. This is achieved by detecting the modules of the local knowledge that might be useful in the current context of the user.

- It offers interesting LBS to the user at each moment and helps in expressing her information needs. This way, it relieves the user from managing specific knowledge about LBS. It translates the user requirements, which might involve multimedia information provided by cameras, into formal requests in SHERLOCK's query language, based on SPARQL with extensions to consider geospatial data and DL ontologies. This language decouples SHERLOCK from the specific scenario and service increasing its flexibility.
- It is able to deploy a network of mobile agents to process a user request and find the information needed wherever it is. These agents can autonomously decide to move to specific areas where the information might be available and contact directly devices that are producing such information and even with their users. Therefore, they can monitor specific areas and communicate with the devices inside to obtain information from their local knowledge. This way, the system is flexible regarding the underlying network infrastructure, enabling the use of static and mobile networks. Also, the use of the agent network ensures that the processing is carried to the most appropriate nodes in order to balance the processing load and communication tasks.

SHERLOCK is a complex system that deals with many different challenges from different research fields. We have delved into some of them and this has allowed us to make contributions to different fields as we present in the following section.

10.2 Other Contributions

During the development of the SHERLOCK system we made contributions to different research aspects. They are mainly related to the use of semantic technologies on mobile devices, ontology alignment, context-awareness, and multimedia management:

- We have empirically shown that using semantic technologies on current mobile devices is feasible. Focusing on Android-based devices, we have been able to use most of the available semantic reasoners. We have detailed the changes needed to make some reasoners work, hoping that this will make porting future versions easier. Also, we have evaluated the performance of these reasoners on current smartphones and tablets using a standard dataset with more than 300 ontologies and different reasoning tasks.
- We have presented an approach for the extraction of subsumption relationships among concepts from different ontologies. Our approach uses the ontological context of concepts (i.e., names, roles, and hierarchical relationships), external information (if available), and some novel generic rules that we designed to capture the existence of a subsumption relationship.
- We have presented an approach to improve the information about the context of the user that the mobile device manages. Mobile devices exchange the context of their users and integrate this information to create a shared context model. The shared context model is leveraged by each device to extend or even correct the local information about the context of its user they had.
- We have presented a proposal to help users to visually define an interesting example shot (with a certain view of one or more objects) and to obtain cameras which could provide it. To our knowledge, this is the first contribution in the literature to extract high-level features of the defined shot and the cameras views without analyzing real images. Therefore, our technique is efficient and scalable enough to be performed in real-time with multiple cameras.
- We have proposed a technique to preserve the privacy preferences of users in pictures taken by others. The approach is based on the exchange of policies as well as a mathematical representation of the user's face to devices around so they can obscure the user accordingly if her face is detected in the picture.

Working on different topics related to the main architecture of SHERLOCK helped me to have a broad view on knowledge management. We dealt with relevant topics such as the integration of information modeled in ontologies and as contextual information of users, the management of multimedia information

which has interesting challenges in comparison with texts (specially when it has to be performed in real-time), and even privacy issues associated with the exchange of information. Finally, I want to highlight that our work on the use of semantic technologies on mobile devices contributed in reinvigorating this line of research. In fact, we co organized a workshop on the topic at the 14th International Semantic Web Conference (ISWC 2015).

10.3 Research Results

The results of the thesis presented in this document have been published in relevant international journals, conferences, and workshops. In the following, we briefly describe these publications in chronological order, grouped by issue, and providing several quality measures for each of them.

Publications related strictly to the SHERLOCK architecture

- In [YMII13] we presented an early prototype of SHERLOCK focused on the interaction with the user (without any agent). The prototype showed how different smartphones equipped with SHERLOCK autonomously exchanged ontologies and integrated them (with a simple string matching techniques). Also, the user could select a service to find other devices in the same WiFi network. The paper was published at the 22nd International World Wide Web Conference (WWW 2013) which is ranked as CORE A* (CORE 2014) and its Google h5 index and position are 75 and #1/20 (Databases & Information Systems category), respectively (Google Metrics 2015).
- In [YMII14] we introduced the basic architecture of SHERLOCK. We presented a preliminary set of agents involved in the different tasks and a brief explanation of their goals. Also, we introduced the first two motivating use cases in Section 3.1. The paper was published in the Pervasive and Mobile Computing journal whose Impact Factor is 2.079 (JCR 2014) - Q1 Computer Science, Information Systems, Q1 Telecommunications, also its Google h5 index is 31.
- In [YM15c] we introduced the use of SHERLOCK to manage emergencies (the fourth of our motivating scenarios). The paper was published in the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2015) which is ranked as CORE B (CORE 2014) and its Google h5 index is 49.

- In [YM15d] we presented SHERLOCK's query language which is based on SPARQL and its two extensions GeoSPARQL and SPARQL-DL. The paper also introduces the processing of queries against other SHERLOCK devices. The paper was published in the 14th International Semantic Web Conference (ISWC 2015) which is ranked as CORE A (CORE 2014) and its Google h5 index and position are 40 and #12/20 (Databases & Information Systems).
- In [YMSB15] we presented our approach to the discovery of subsumption relationships between concepts of different ontologies for the integration of knowledge. We explained the complete architecture of our approach focusing on the rules we designed to capture the existence of a subsumption relationship between two concepts regarding their roles. The paper was published in the 2015 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2015) which is ranked as B (CORE 2014) and its Google h5 index is 18.
- In [YM15b] we introduced the network of mobile agents in charge of the processing of SHERLOCK queries. We explained how the network is created to process location-based queries. The paper was also published in the WI 2015 conference.
- In [YM15a] we introduced the multimedia information processing features of SHERLOCK focusing on the last two motivating use cases. We explained the complete process starting with the user selecting one of the services and finishing with her obtaining results. We focused on the management of multimedia information by integrating the ideas we had previously developed to manage camera views into the architecture. The paper was published in the 13th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2015) which is ranked as B (CORE 2014) and its Google h5 index is 11.
- In [BYBIBMTLG15] we presented the SHERLOCK architecture as an example of semantic-based application as part of a chapter of a book on "Semantic Web: Implications for Technologies and Business Practices".

Currently, we are working on two journal papers which will be the most complete works about the SHERLOCK system. The first paper will focus on describing the modeling of knowledge in the SHERLOCK system, the query language, and the interaction with the user to generate a user request (therefore, it will cover Chapters 4 and 6). The second paper will focus on the processing

of SHERLOCK queries by detailing the network of agents involved and the tasks of each agent (as shown in Chapter 7).

Apart from the previous publications, that are strictly related to the SHERLOCK architecture, we show in the following the publications related to different aspects of the thesis.

Other publications related to different modules of SHERLOCK

In [YBEBM13; BBYEM14; YBBM15] we presented our early work on the use of semantic technologies on mobile devices. We showed the steps to port some semantic reasoners to Android and preliminary tests with a dozen of well-known ontologies. We published these papers at the OWL Reasoner Evaluation Workshop. In [BYBM15] we presented our detailed experiments on the performance of semantic reasoners on current mobile devices. The experiments were performed with more than 300 ontologies from the ORE 2013 dataset and 10 semantic reasoners. This last paper was published in the Journal of Web Semantics whose Impact Factor is 2.550 (JCR 2014) - Q1 Computer Science, Artificial Intelligence, Q1 Computer Science, Information Systems, Q1 Computer Science, Software Engineering, also its Google h5 index and position are 36 and #14/20 (Databases & Information Systems category), respectively (Google Metrics 2015).

In [IMIYLM12] we presented our preliminary ideas on a system to assist technical directors in the broadcasting of a sport event. We focused on explaining how the system would process location-based queries with constraints about the views provided by different cameras. The paper was published in the Mobile Information Systems journal whose Impact Factor is 1.789 (JCR 2013) - Q1 Computer Science, Information Systems, Q1 Telecommunications, also its Google h5 index is 14 (Google Metrics 2015). In [YMBII11] we explained our approach to extract high-level features of camera views in real-time by using a 3D model of the scenario. We presented our algorithms to efficiently and in real-time compute what a camera is viewing by using 3D operations. The paper was published in the 19th ACM International Conference on Multimedia (ACMMM 2011) which is ranked as CORE A* (CORE 2014) and its Google h5 index and position are 45 and #2/20 (Multimedia category), respectively (Google Metrics 2015). In [YAMII11] we presented a first prototype of the system to assist technical directors implementing the ideas presented in [IMIYLM12; YMBII11]. The prototype enabled users to define queries through a web interface and showed the camera feed of cameras fulfilling their requirements. The paper was published in the 8th International Conference

on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2011) which is ranked as CORE A (CORE 2014) and its Google h5 index is 12 (Google Metrics 2015). In [YIM15] we presented our approach to compute the similarity between camera shots regarding the high-level features extracted. The paper included the 3DQBE interface to enable users to define an interesting shot easily and the formula designed to obtain cameras which could provide a similar shot. Finally, in [YMIIB15] we presented the most complete work that we have published about our system to help technical directors in the live broadcasting of sport events in live. The paper presents the complete architecture and details the process from the technical director defining the required shot and the system showing the cameras that can, or could in the near future, provide it. These two papers ([YIM15; YMIIB15]) were published in the Multimedia Tools and Applications journal whose Impact Factor is 1.346 (JCR 2014) - Q2 Computer Science, Information Systems, Q2 Computer Science, Software Engineering, Q2 Computer Science, Theory & Methods, Q2 Engineering, Electrical & Electronic, also its Google h5 index and position are 33 and #5/20 (Multimedia category), respectively (Google Metrics 2015).

In [YPDMJF14] we presented our approach to preserve user privacy on pictures taken by others. The paper introduces the architecture of the approach and the early prototype of the system presented. The paper was published in the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2014) which is ranked as CORE B (CORE 2014) and its Google h5 index is 49. In [PYDFMJ14] we showed the semantic definition of privacy rules to express the context under which the user wants her privacy to be preserved on pictures. The paper was published in the PrivOn workshop co-located with the 13th International Semantic Web Conference (ISWC 2014). The work presented in these papers was referenced in a recent article in Nature (“What could derail the wearables revolution?”¹).

In [PYJF14] we introduced the use of SHERLOCK’s knowledge management capabilities in wireless environments to help community health-care workers in underserved areas. We explained how the devices carried by community health-care workers exchange ontologies with information about diseases in the area and help them to detect symptoms by showing questions to ask to the patients. The paper was published in the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2014) which is ranked as CORE C (CORE 2014) and its

¹<http://www.nature.com/news/what-could-derail-the-wearables-revolution-1.18263>

Google h5 index is 15.

10.4 Future Work

The design of a complex system like SHERLOCK opens up many research questions. We have focused on the design of the general architecture and tackled the major challenges for it. Also, we have delved into some of the challenges related to specific functionalities of the system but others could be further studied. In the following, we explain possible lines of future work related to the work performed in this thesis focusing in the three main parts of the architecture.

Regarding Knowledge Management on Mobile Devices

- We should study mechanisms to efficiently perform the exchange and integration of knowledge on mobile devices. For example, our current approach in which devices exchange part of their knowledge and then perform local integration of the information received could be extended. In the scenario where the devices have limited capabilities, the integration task could be distributed among the different devices. Also, it would be possible to define a protocol to take into account factors like: the capabilities of each device (e.g., in terms of available resources), their local knowledge, their context (e.g., related to their movement), and even the privacy policies of their users to determine an optimal way to minimize the total number of resources consumed in the process of updating the knowledge of all the devices while maximizing the amount of new knowledge each device will obtain.
- We plan to continue our work on porting and evaluating semantic reasoners on mobile devices. In particular, we want to explore the use of reasoners for OWL 2 QL and OWL 2 RL profiles, as sacrificing expressivity and possible inferences to obtain a better performance could be interesting according to the results we obtained for other profiles. Moreover, we are interested on evaluating other reasoning tasks such as query answering or realization, as they involve ABox reasoning and are interesting in many mobile scenarios.

Regarding the Management of User Requests

- We should further study the interaction with users to capture their information needs. Previous interactions of the user with respect to her context, as well as previous interactions of other users with a similar context, could be leveraged to improve our suggestion of interesting services. Also, this information could be used to enable the system to autonomously and preemptively select services which could be interesting for the user and even execute them.
- We plan to study the design of strategies to optimize the network of mobile agents deployed to answer a user request. Users have similar needs and therefore, a network deployed for one of them could be used for more users with minor modifications. Of course, in this scenario security and privacy issues arise as agents created by a user would act on behalf of others. Also, in our current approach the goal of the agents is to satisfy the needs of the owner of the device where they were created. In the new scenario envisioned agents could have to deal with slightly different needs (and maybe even contradictory) of the different users and the owner of the device where they were created.

Regarding the Management of Multimedia Information

- We should consider other forms of multimedia information (e.g., sound captured by microphones). The techniques we presented are focused on images provided by cameras and they are specific to them. However, the fundamentals about using information about the context of the microphone and objects around to avoid analyzing the real data could be interesting in this scenario too.
- We plan to further explore our approach to preserve privacy in photography. For instance, the mechanism could be generalized to other multimedia sources (e.g., video and audio).

In general the goal of our future lines of research will be to continue exploring the challenges related to the semantic management of data in mobile computing scenarios.

Relevant Publications Related to the Thesis

International Journals

- [BYBM15] Carlos Bobed, Roberto Yus, Fernando Bobillo, and Eduardo Mena. “Semantic Reasoning on Mobile Devices: Do Androids Dream of Efficient Reasoners?” In: *Journal of Web Semantics* 35 (2015), pp. 167–183.
- [IMIYLM12] Sergio Ilarri, Eduardo Mena, Arantza Illarramendi, Roberto Yus, Maider Laka, and Gorka Marcos. “A Friendly Location-Aware System to Facilitate the Work of Technical Directors When Broadcasting Sport Events”. In: *Mobile Information Systems* 8.1 (2012), pp. 17–43.
- [YIM15] Roberto Yus, Sergio Ilarri, and Eduardo Mena. “Real-time Selection of Video Streams for Live TV Broadcasting Based on Query-by-Example Using a 3D Model”. In: *Multimedia Tools and Applications* 74.8 (2015), pp. 2659–2685.
- [YMII14] Roberto Yus, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. “SHERLOCK: Semantic Management of Location-Based Services in Wireless Environments”. In: *Pervasive and Mobile Computing* 15 (2014), pp. 87–99.
- [YMIIB15] Roberto Yus, Eduardo Mena, Sergio Ilarri, Arantza Illarramendi, and Jorge Bernad. “MultiCAMBA: A System for Selecting Camera Views in Live Broadcasting of Sport Events Using a Dynamic 3D Model”. In: *Multimedia Tools and Applications* 74.11 (2015), pp. 4059–4090.

International Conferences

- [PYJF14] Primal Pappachan, Roberto Yus, Anupam Joshi, and Tim Finin. “Rafiki: A Semantic and Collaborative Approach to Community Health-care in Underserved Areas”. In: *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2014)*. 2014, pp. 322–331.
- [YAMII11] Roberto Yus, David Anton, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. “MultiCAMBA: A System to Assist in the Broadcasting of Sport Events”. In: *8th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQ-uitous 2011)*. Vol. 104. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (LNICST). Springer, 2011, pp. 238–242.
- [YM15a] Roberto Yus and Eduardo Mena. “Continuous Processing of Real-Time Multimedia Requests Using Semantic Techniques”. In: *13th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2015)*. ACM, 2015, pp. 216–220.
- [YM15b] Roberto Yus and Eduardo Mena. “Cooperative Network of Mobile Agents to Remotely Process User Information Requests”. In: *2015 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2015)*. IEEE, 2015, pp. 227–228.
- [YM15c] Roberto Yus and Eduardo Mena. “Emergency Management Using SHERLOCK”. In: *13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2015)*. ACM, 2015, pp. 495–495.
- [YM15d] Roberto Yus and Eduardo Mena. “Mobile Endpoints: Accessing Dynamic Information from Mobile Devices”. In: *14th International Semantic Web Conference (ISWC 2015)*. Vol. 1486. CEUR-WS, 2015, p. 4.

- [YMBII11] Roberto Yus, Eduardo Mena, Jorge Bernad, Sergio Ilarri, and Arantza Illarramendi. “Location-Aware System Based on a Dynamic 3D Model to Help in Live Broadcasting of Sport Events”. In: *19th ACM International Conference on Multimedia (ACMMM 2011)*. ACM, 2011, pp. 1005–1008.
- [YMI13] Roberto Yus, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. “SHERLOCK: A System for Location-Based Services in Wireless Environments Using Semantics”. In: *22nd International World Wide Web Conference (WWW 2013)*. ACM, 2013, pp. 301–304.
- [YMSB15] Roberto Yus, Eduardo Mena, and Enrique Solano-Bes. “Generic Rules for the Discovery of Subsumption Relationships based on Ontological Contexts”. In: *2015 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2015)*. IEEE, 2015, pp. 309–312.
- [YPDMJF14] Roberto Yus, Primal Pappachan, Prajit Kumar Das, Eduardo Mena, Anupam Joshi, and Tim Finin. “FaceBlock: Privacy-Aware Pictures for Google Glass”. In: *12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*. 2014, pp. 366–366.

International Workshops

- [BBYEM14] Carlos Bobed, Fernando Bobillo, Roberto Yus, Guillermo Esteban, and Eduardo Mena. “Android Went Semantic: Time for Evaluation”. In: *3rd International Workshop on OWL Reasoner Evaluation (ORE 2014)*. Vol. 1207. CEUR-WS, 2014, pp. 23–29.
- [PYDFMJ14] Primal Pappachan, Roberto Yus, Prajit Kumar Das, Tim Finin, Eduardo Mena, and Anupam Joshi. “A Semantic Context-Aware Privacy Model for FaceBlock”. In: *2nd International Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014), co-located with the 13th International Semantic Web Conference (ISWC 2014)*. Vol. 1316. CEUR-WS, 2014.

- [YBBM15] Roberto Yus, Fernando Bobillo, Carlos Bobed, and Eduardo Mena. “The OWL Reasoner Evaluation Goes Mobile”. In: *4th International Workshop on OWL Reasoner Evaluation (ORE 2015)*. Vol. 1387. CEUR-WS, 2015, pp. 38–45.
- [YBEBM13] Roberto Yus, Carlos Bobed, Guillermo Esteban, Fernando Bobillo, and Eduardo Mena. “Android goes Semantic: DL Reasoners on Smartphones”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. CEUR-WS, 2013, pp. 46–52.
- [YP15] Roberto Yus and Primal Pappachan. “Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications”. In: *1st International Workshop on Mobile Deployment of Semantic Technologies (MoDeST 2015), co-located with the 14th International Semantic Web Conference (ISWC 2015), Bethlehem, PA (USA)*. Vol. 1506. CEUR-WS, 2015, pp. 2–13.
- [YPDFJM14] Roberto Yus, Primal Pappachan, Prajit Kumar Das, Tim Finin, Anupam Joshi, and Eduardo Mena. “Semantics for Privacy and Shared Context”. In: *2nd International Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014), co-located with the 13th International Semantic Web Conference (ISWC 2014)*. Vol. 1316. CEUR-WS, 2014.

Book Chapter

- [BYBIBMTLG15] Carlos Bobed, Roberto Yus, Fernando Bobillo, Sergio Ilarri, Jorge Bernad, Eduardo Mena, Raquel Trillo-Lado, and Angel Luis Garrido. “Chapter 4: Emerging Semantic-Based Applications”. In: *Semantic Web: Implications for Technologies and Business Practices*. Ed. by Michael Workman. Springer International Publishing, 2015, pp. 39–83.

Appendix A

Semantic Technologies on Mobile Devices

In the current ubiquitous and mobile scenario, there are lots of interesting applications that take advantage of the context of the users (for instance, their geographical locations). Despite of their usefulness, it seems promising to incorporate semantic technologies to enhance such applications by combining ontological information with extensional data obtained from the mobile sensors. This way it would be possible to take advantage of the benefits of using ontologies, such as the improvement of knowledge sharing, reusing and maintenance, the decoupling of the knowledge from the application, or the possibility of discovering implicit knowledge by using semantic reasoners.

Therefore, semantic mobile applications need to access to a semantic reasoner. In many cases, mobile devices could use the services of a semantic reasoning service located in the cloud. However, users or developers could prefer using a semantic reasoner locally installed on the mobile device for several reasons, such as privacy preserving (performing the reasoning locally can minimize the exposure of user information), or connectivity problems (local reasoners can be helpful for applications where Internet connectivity could not be available or where the number of network connections have to be minimized). We are specially interested in these cases where a local reasoner on the mobile device is required.

However, the use of semantic technologies on mobile apps has not (yet) spread due, in part, to the fact that there are currently no remarkable efforts to enable mobile devices with semantic reasoning capabilities. A first possibility is developing new reasoners specifically designed for mobile devices. Examples of this alternative include mTableau [MHLN06], Pocket KR Hyper [SK05],

Delta [MHK12], and Mini-ME [RSSGL12] reasoners. In order to reuse as much as possible the work to the optimize current Description Logics (DL) reasoners, we are more interested in another choice: reusing existing semantic reasoners on mobile devices. In particular, we have focused on porting and using existing DL reasoners on Android, and our experience tells us that porting them to Android is less costly than developing new reasoners from scratch.

We have focused our research on devices using Android operating system [Bur10] due to several reasons: its diffusion (85% of devices use this operative system according to a recent estimation by Strategy Analytics¹), its openness, and the existence of a Java-like native virtual machine (Dalvik) that makes it easier to reuse existing Java applications, something very important since most of the semantic APIs and reasoners have been developed in this language.

A.1 Porting Semantic Technologies to Android

Most of current popular semantic reasoners are implemented using Java (e.g., Pellet and HermiT) and are usually used along with semantic APIs (e.g., OWL API and Jena). Android is a Linux-based operating system whose middleware, libraries, and APIs are written in C. Since its version 2.2, Android uses a Java-like virtual machine called Dalvik [OKCM12] that makes it possible to support Java code. In fact, Dalvik runs “dex-code” (Dalvik Executable), and Java bytecodes can be converted to Dalvik-compatible .dex files to be executed on Android. However, Dalvik does not completely align to Java SE and so it does not support J2ME classes, AWT or Swing. Thus, running semantic APIs and reasoners on Android could require some rewriting efforts.

Table A.1 summarizes the current Android support for the main semantic APIs and DL reasoners. We tested all of them, and found out that *OWL API 3.4.10*, and the *jcel 0.19.1*, *JFact 0.9.1*, *TReasoner revision 22*, and *TrOWL 1.4* reasoners can be imported directly in Android projects. However, as the table shows, most of them (16 out of 21) are not directly compatible with Android. In the next section, we explain our experience trying to port these latter reasoners to Android.

As we have shown in Table A.1, there are APIs and reasoners that do not work directly on Android. Thus, we tried to port them (or use alternative ones) to make them work in our Android projects. In the following, we detail how we have ported these different technologies, and we highlight some of the

¹<http://goo.gl/8KCJXs>, last accessed 2014-12-12.

possible problems for those we did not successfully port.

As a summary, the main causes that we found for reasoners not to be directly imported in Android projects can be broadly classified as:

- Direct use of Java classes not supported in Android.
- Use of external libraries that use unsupported Java classes.

Most of these problems can only be detected in run time. So, the process we followed consisted on importing each reasoner (downloaded from their websites) in an Android application that we developed to automate the testing, and running it. Then, for those reasoners that failed to run, we tried to detect the problematic classes/libraries by studying its code (whenever it was possible, as not all the developers made available the code of their reasoners). Finally, we tried to replace problematic classes/libraries by their equivalent ones for the Android platform. In the following we explain the experience trying to port the technologies that failed to run on Android. Further and detailed information about the specific methods and classes changed for each reasoner can be found at the webpage of the project [And] together with, if the licenses make it possible, a download link.

We will firstly present the case of *Jena* API, then the successfully ported reasoners and then the unsuccessfully ones. In both cases, reasoners will be presented in alphabetical order.

Jena cannot be directly imported into an Android project but there exists a project called Androjena² to port it to the Android platform. The latest version of Androjena 0.5, which was used in our tests, contains all the original code of Jena 2.6.2 and *can be used in Android projects*.

CB is implemented in OCaml and Android does not support it natively. There are some projects to develop OCaml interpreters for the platform, such as the OCaml Toplevel³; however, we chose a different approach: compiling the reasoner to native Android code. For that, we used the Android Native Development Kit (NDK)⁴ to cross-compile the code for the ARM processor. *The resulting native code can be executed on Android using the command line tool Android Debug Bridge (adb)*. To import this native code into an Android project, we could use the Java Native Interface (JNI) and Android NDK. However, for the purpose of this paper we tested the native code directly.

²<https://code.google.com/p/androjena>

³<https://bitbucket.org/keigoi/ocaml-toplevel-android>

⁴<http://developer.android.com/tools/sdk/ndk>

ELK presents a problem with the only external library that the reasoner imports. *Log4j* is an open source (Apache License 2.0) logging utility that uses classes of the Java package `java.beans` but the full package is not completely supported in Android. There exists a port for this library⁵ but in its current version presents some problems. Therefore, the recommended process is to replace the *Log4j* library by the *SLF4J*⁶ that is supported in Android. In this case, we use the *log4j-over-slf4j* library that allows *log4j* applications to be migrated to *SLF4J* without changing the code. *With this replacement, ELK 0.4.0 can be used in Android projects.*

HermiT references unsupported Java classes (both in its source code and in the imported external library *JAutomata*). On the one hand, we detected problems with the debug, Protégé, and command line packages. Specifically, the references to `java.awt.point` and other Java AWT classes must be replaced as Android has its own graphical libraries. Due to their nature, we thought that these packages would not be required by a developer who uses the reasoner in an Android application, and therefore, we removed all of them from our port. On the other hand, *JAutomata*⁷, a library for creating, manipulating, and displaying finite-state automata, presents two problems: 1) it references the aforementioned `java.awt.point` class in some hashing functions, and 2) it references two unsupported libraries, *JUnit* and `dk.brics.automaton`. To address the first problem, we just modified the hashing functions. For the second problem, on the one hand, we removed the *JUnit*⁸ library and its references (as it is a library used for development); and, on the other hand, we reimplemented part of `dk.brics.automaton`⁹. This library is required as it contains a DFA/NFA (finite-state automata) implementation that is used to process datatypes of the ontology. It uses some files that contain automata that cannot be unmarshalled in Android (as they were marshalled using Java). Thus, to solve this problem, we reimplemented the marshalling/unmarshalling methods to create a new set of automata files compatible with Android. *With this changes HermiT 1.3.8 can be used in Android projects.*

MORé uses the *HermiT*, *JFact*, and *ELK* reasoners and, as explained previously, the original version of *HermiT* and *ELK* are not compatible with Android. Replacing the two reasoner by the ported versions fixes this problem. *Therefore, MORé 0.1.5 can be used in Android projects.*

⁵<https://code.google.com/p/android-logging-log4j>

⁶<http://www.slf4j.org>

⁷<http://jautomata.sourceforge.net>

⁸<http://junit.org>

⁹<http://www.brics.dk/automaton>

Pellet presented problems with unsupported Java classes being referenced from its tests packages (that can be removed) and three external libraries: Jena (which can be replaced by Androjena as explained before), OWL API 2.2.0 (which can be removed from the final compiled version), and JAXB¹⁰, a library to map Java classes to XML. JAXB uses the `javax.xml.bind` package and the *Xerces*¹¹ parser libraries which are not supported in Android. This latter problem can be solved by removing the JAXB .jar file and adding the source code of both `javax.xml.bind` and *Xerces* to our Android project. However, Dalvik has a limit of 65536 methods references per .dex file and it gets exceeded when applying this solution. To solve this, we removed the JAXB library and copied only the nine classes that Pellet needs from both the `java.xml.bind` package and the *Xerces* library to our Android project. *With this changes Pellet 2.3.1 can be used in Android projects.*

Apart from the above mentioned ones, we also tried other reasoners that we could not port to Android. Some reasoners were not available at the time when this work was performed, such as *SHER*, *SOR*, or *WSClassifier*.

KAON2 presented problems when imported in an Android project. The source code of this reasoner is not available, so we have been able to detect possible problems only by analyzing the libraries it imports. Among them, the reasoner imports Java Remote Method Invocation (RMI) which is not supported in Android. There are two projects to port this library to Android, *LipeRMI*¹² and *RipeRMI*¹³, which we have used in other projects and are a possible replacement for RMI. However, they do not align completely with the API of RMI, so, modifying the code would be necessary. The reasoner might also need further code rewritings that could only be detected by accessing the source code. *Therefore, the last version of KAON2 cannot be used in Android projects.*

The fuzzy ontology reasoner *fuzzyDL* uses the *Gurobi*¹⁴ library, an optimization programming solver. This library is not supported in Android and, up to the authors' knowledge, has not a supported replacement. In addition, the library has a proprietary license and so, we could not explore the steps needed to port it to Android. *Therefore, fuzzyDL build 60 cannot be used in Android projects.*

Finally, other reasoners are developed in languages different from Java that

¹⁰<https://jaxb.java.net>

¹¹<http://xerces.apache.org>

¹²<http://lipermi.sourceforge.net>

¹³<https://code.google.com/p/ripermi>

¹⁴<http://www.gurobi.com>

Android does not support natively. In these cases, one could try the same approach presented for *CB*: compiling the code for ARM with the help of the Android NDK and using JNI to import the code from an Android project. For example, there are a lot of reasoners implemented in C++, such as *Chainsaw*, *ConDOR*, *FaCT++*, *ELepHant*, *Konclude*, and *WSClassifier*. Porting these C++ reasoners would also make it easier to support some metareasoners written in Java but using reasoners implemented in C++. For example, *Chainsaw* uses *FaCT++* and *WSClassifier* uses *Condor*. There are also some reasoners implemented in Lisp, such as *Racer* and *CEL*. Note that there is also a *Racer* server version that could be used on an external device and used from clients. However, that would require a connection between the mobile device (client) and the server which defeats the purpose of this study.

A.2 Evaluating the Use of Semantic Web Technologies on Mobile Devices

In this section we describe the evaluation of the behavior of the current semantic reasoners on mobile devices studied, and some of them ported to Android by us, in Section A.1.

A.2.1 Experimental Setup

We had to make different choices to perform the experiments including the selection of: the ontology dataset, devices, reasoning tasks, and reasoners.

Selecting the Ontology Dataset

To evaluate the performance of the studied reasoners, we selected the ORE 2013 ontology set [GBJRMPGK13] which contains 200 ontologies per profile (i.e., OWL 2 EL, OWL 2 RL, and OWL 2 DL) from the NCBO BioPortal¹⁵, the Oxford Ontology Library¹⁶, and the Manchester Ontology Repository¹⁷. Every ontology has at least 100 logical axioms and 10 named concepts, and they are classified according to their number of logical axioms as *small* (≤ 500), *medium* (between 500 and 4999), and *large* ontologies (≥ 5000). The ORE 2014 ontology set, with 16555 ontologies, is too large for our purposes [BGJRMPS13].

¹⁵<http://bioportal.bioontology.org>

¹⁶<http://www.cs.ox.ac.uk/isg/ontologies>

¹⁷<http://rpc295.cs.man.ac.uk:8080/repository>

In our case, we focused on the OWL 2 DL and OWL 2 EL ontology sets. We did not select the OWL 2 RL and OWL 2 QL profiles as we were not able to port any reasoner specifically for these profiles. Besides, we had to take into account the restrictions that mobile devices suffer from, especially the limited CPU, memory, and battery. For this reason, we selected a subset of the ORE 2013 ontology set carrying out the following steps:

1. We ordered the ontologies according to the size of the file as, when evaluating mobile devices, we should not only take into account the number of logical axioms, but also the file size (which is directly related to the memory needed to load such ontology). We could not ignore annotation axioms as they were problematic in our scenario: they also need to be processed by the ontology parser and, thus, they might consume extra memory temporally (e.g., by the OWL API parsers).

Please note that the file size is just a heuristic, because it depends on the OWL 2 syntax, the length of URIs, and other non-logical aspects. In our experiments, we have considered OWL/XML syntax¹⁸.

2. The maximum heap size per application provided by the Android version in the test devices is 256 MB (after setting the variable to get the maximum heap size for a mobile application¹⁹). This could be the theoretical maximum size of an ontology loaded on Android, but we must build instances of the OWL API class `OWL-Reasoner` in the device's memory. So, we filtered the ontology sets to take out the ontologies whose files occupied more than 128 MB.

This resulted in 186 OWL 2 DL and 194 OWL 2 EL ontologies. However, four of the OWL 2 DL ontologies and one OWL 2 EL ontology were not admitted by our testing application because of their URIs. Therefore, our final *DL ontology set* used in our experiments contains 182 OWL 2 DL ontologies, distributed as follows: 43 small, 103 medium, and 36 large ones; whereas our *EL ontology set* has 193 ontologies: 72 small, 82 medium, and 39 large ones. Our full ontology set and some ontology stats (such as their size, number of axioms, and expressivity) can be found at [And].

Selecting the Devices

We considered two mobile devices for the tests: a smartphone and a tablet. The smartphone selected was a Galaxy Nexus (Android 4.2.1, 1.2 GHz dual-core,

¹⁸<http://www.w3.org/TR/owl-xmlsyntax>

¹⁹`android:largeHeap='true'`.

1 GB RAM, released in 2011, in the following denoted as A1), and the tablet was a Galaxy Tab 2 7.0 (Android 4.1.2, 1 GHz dual-core, 1 GB RAM, released in 2012, denoted A2). In order to avoid battery shortage, both devices were plugged in during all the tests.

We also performed some tests with a Galaxy Tab tablet (Android 2.3.3, 1.0 GHz single-core, 512MB RAM, released in 2010, denoted as A3) using five popular ontologies: *Pizza*²⁰ and *Wine*²¹, which are two expressive ontologies; *DBpedia* 3.8²² (TBox), which can be useful for mobile application developers to access the structured content of DBpedia (a semantic entry point to Wikipedia) [BLKABCH09]; and the Gene Ontology (*GO*)²³ and the US National Cancer Institute (*NCI*)²⁴ ontologies, which contain a high number of concepts. Table A.2 shows a summary of the results obtained when comparing the performance of the devices A1 and A3. A1 outperformed A3 up to a 30% of increment on the performance in some situations, and thus, we left it aside for the rest of our experiments.

According to the official Android report²⁵, and as of July 2015, the use of the Jelly Bean version of Android (from 4.1.x to 4.3) represents around 37.4% of current Android devices (76.6% considering also 4.4 devices as the core of the OS is almost the same). Also, as of 2015, most of the devices on the market have similar or even better capabilities than the Galaxy Nexus and the Galaxy Tab 2. Thus, with the Galaxy Nexus and the Galaxy Tab 2 we are representing the average current Android device in terms of capabilities and Android version.

Also, we considered a desktop computer to determine how slow is reasoning on Android compared to this baseline (taking into account that the desktop computer hardware outperforms Android devices, and that their virtual machines are optimized for different purposes). In this case the desktop computer selected (denoted PC) was a Windows 64-bits, i5-2320 3.00 GHz, 16 GB RAM (12 GB were allocated for the JVM in the tests).

Selecting the Tasks

We analyzed the behavior of the reasoners for two standard Description Logic inference services that were part of the ORE 2013 competition:

²⁰<http://www.co-ode.org/ontologies/pizza/pizza.owl>

²¹<http://www.w3.org/TR/owl-guide/wine.rdf>

²²<http://dbpedia.org/Ontology>

²³<http://www.geneontology.org>

²⁴<http://ncit.nci.nih.gov>

²⁵<http://developer.android.com/about/dashboards/index.html>

- *Ontology classification*: computing the complete class hierarchy based on the subsumption relation between the ontology classes.
- *Ontology consistency*: checking whether an ontology contains any contradictions or not.

For the moment, we did not consider other tasks such as concept satisfiability²⁶, query answering, or realization²⁷ because the results obtained are strongly dependent on the particular choice of the selected concept, query, or individual, respectively. To obtain significant results, each test would have to be repeated for a considerable amount of different elements on each ontology. Moreover, the selected tasks are being currently used in our prototypes. For example, SHERLOCK [YMII14], FaceBlock [YPDMJF14], and Rafiki [PYJF14] use classification, whereas Triveni [YPDFJM14] also checks consistency.

We measured the performance of the different reasoners and devices for these tasks in terms of finished tasks and computation time. Due to the high variance of processing time on Android devices observed in our preliminary experiments [YBEBM13; BBYEM14], we repeated every test three times and computed the average and variance of the processing time. We considered a task as finished if it was processed without throwing any error and within a defined timeout, which was set to 25 minutes for Android devices and 5 minutes for the desktop computer.

Regarding the consumed memory, on Android it is difficult to obtain a precise measure of the memory consumed (the most accurate value would be the so called Proportional Set Size, which includes the private memory and divides the shared pages between all the processes that share them, but it is just an estimation); thus, in our experiments, we made the biggest amount of memory available (setting the heap value), and focused on the amount of tasks finished with that heap size as limit for all the reasoners. Moreover, we considered measuring battery consumption as well, but we found several difficulties to do that accurately on Android 4.x devices. As shown in [PM14], where the power consumption for some reasoners on Android 4.x is analyzed, measuring this factor requires external hardware connected to the battery of the device.

Finally, note that we are not checking if the result returned by the reasoner is correct, we only guarantee that the results are the same on Android devices

²⁶Concept satisfiability is subsumed by classification, so one should test concept satisfiability on non-classified ontologies.

²⁷As defined in [BCMNP03], here, we consider realization as finding the most specific concepts a given individual is an instance of.

and on the desktop computer. The ORE 2013 competition estimates the correctness of the DL reasoners by a majority vote and publishes their results, so the interested reader is referred to [GBJRMPGK13].

Selecting the Reasoners

In our preliminary experiments, we detected that reasoning on Android devices was up to 100 times slower than on a desktop computer for certain ontologies and reasoners [BBYEM14]. Therefore, testing all the reasoners would require hundreds of computation hours, and so, we selected a representative subset of popular reasoners for our tests: *ELK*, *HermiT*, *jcel*, *JFact*, *Pellet*, and *TROWL*. *MORe* was not included in our tests because it uses *ELK*, *HermiT*, and *JFact* that are already being analyzed.

We have not included CB reasoner in the experiments with the complete ontology set, despite having tested it in our preliminary work [YBEBM13]. Although CB has the advantage of running outside the virtual machine using native code, and thus not being subject to the restrictions imposed by the runtime environment, its expressivity and ease of use are problematic. CB requires to work with Horn-*SHIF* ontologies, but only 12% of our DL ontology set are Horn-*SHIF* ontologies. Furthermore, using CB in a mobile application might be quite complex due to different steps needed to make the port work. This might be a barrier to its use in mobile applications, especially when there are other reasoners than can be imported and used almost directly on Android. Indeed, we have not been able yet to access CB on Android using the OWL API, which has become a de facto standard. Thus, we decided to restrict the tests to the reasoners accessible using the OWL API, as they are more prone to be used in practice.

Since CB executes native code outside the virtual machine of Android, it is not subject to the restrictions imposed by the virtual machine, while the other reasoners run within the virtual machine in a more constrained framework. It is worth to include some results showing the effect of this. Table A.3 compares the classification times (in seconds) of three popular Horn ontologies (DBpedia, GO, and NCI) on the desktop computer and the smartphone using CB and ELK. In this case, since CB is not accessed using the OWL API, ELK is accessed using its own API to avoid a possible overhead caused by the OWL API. DBpedia includes datatypes and thus is not fully supported by CB, so we do not consider the results of the classification²⁸. Notice how, on PC, CB is 2.3

²⁸Note that [YBEBM13] considers the results because the results of the classification happen to be correct.

times faster than ELK for the GO ontology, but on the Android device CB is 3.5 faster than ELK. Furthermore, on PC, ELK is slightly faster than CB for the NCI ontology, but on A1 CB is 8 times faster. Therefore, running native code directly without using the virtual machine on Android devices seems to make a difference. A more detailed study of this fact is left as future work.

Regarding the reasoners designed for mobile devices (see Section 2.1.2), we were forced to discard *Mini-ME*, the only reasoner that is available and compatible with Android. Before starting the experiments, we compared the expressivity of the ontologies in both ontology sets with the expressivity that Mini-ME supported, and, at first, 160 ontologies from the DL ontology set and 42 from the EL ontology set lay out from the expressivity supported (\mathcal{ALCN}). Anyway, we computed the classification of the EL ontology set, but 96 ontologies did not finish due to several problems (such as class exceptions), and 97 ontologies reached the established timeout for the tests. Thus, the results of 9 ontologies (5 %) are not significant enough to compare *Mini-ME* with the other reasoners.

Verifying the Android Versions

Before leaping into the main experiments, we performed a set of tests on the selected reasoners to test whether they produced exactly the same results on Android devices as on desktop computers. The aim of these tests was twofold: on the one hand, we wanted to check the behavior of the Android-compatible libraries we used to replace the unsupported ones; and on the other hand, we wanted to check the behavior of the reasoners that can be directly imported in Android projects.

We focused on the classification task because the consistency checking is just a yes-no question. First, as baseline, we obtained the sets of subsumption axioms computed by the original version of the reasoners running on the desktop computer. Then, we checked that the results obtained by the reasoners running on Android devices contained exactly the same axioms (of course, this was done separately per ontology). The only mismatches we found on our selected reasoners were due to two different reasons: sometimes the reasoning task on Android did not end before the timeout (and therefore, no comparison could be made), or there were problems with some ontologies due to the encoding of the ontology files, which led to malformed URIs (these problems disappeared once we aligned the file encodings, as detailed in Section A.2.7).

Note that verifying the Android versions is indeed necessary: we found some examples where different versions of the ported reasoners gave different

results on Android devices and in the desktop computer. In JFact 1.2.1, the results of the classification and the consistency checking are different on PC and Android:

- `aee636cb-4238-41af-a3d6-541d30f2e7ed_spills.owl` is correctly identified as consistent by the PC version, but is inconsistent according to the Android version.
- `52cf3ab5-1662-4296-835e-b22ac92339e7..2_DUL.owl` misses two misses two subclasses of the class `Role`, namely `Entity` and `E1.CRM.Entity` in Android.

This problem does not happen in JFact 0.9.1. Thus, once we tested that we had not introduced any problem when porting the reasoners to Android, we moved on to the performance experiments.

In the following sections, we present our analysis of the performance of reasoners on mobile devices grouped by ontology profile. First, we present the results obtained for the OWL 2 DL profile and then for the OWL 2 EL profile (the same order used in the ORE 2013 report [GBJRM PGK13]) in Sections A.2.2 and A.2.3, respectively. For each profile, we present the results obtained when comparing the reasoners in terms of finished tasks and average computation time needed per task. Then, we compare the time consumption of the reasoners with the minimum set of ontologies that every reasoner and device was able to process. Next, in Section A.2.4 we study the role of the limitation of memory and the virtual machine on the reasoning on Android devices.

A.2.2 Comparing the Reasoners for the OWL 2 DL Profile

In this section, we detail the results of our performance experiments for the OWL 2 DL profile. For our DL ontology set (with 182 ontologies), we tested the following reasoners: *JFact 0.9.1*, *HermiT 1.3.8*, *Pellet 2.3.1*, and *TrOWL 1.4*. As already mentioned, we first detail the number of finished tasks per reasoner and device, to then move on to their performance.

Comparing the Number of Finished Tasks

The results for the classification task for the desktop computer (PC), the Galaxy Nexus (A1), and the Galaxy Tab 2 (A2) are shown in Figure A.1(a), Figure A.1(c), and Figure A.1(e), respectively; for the consistency checking results for PC, A1, and A2 see Figure A.1(b), Figure A.1(d), and Figure A.1(f).

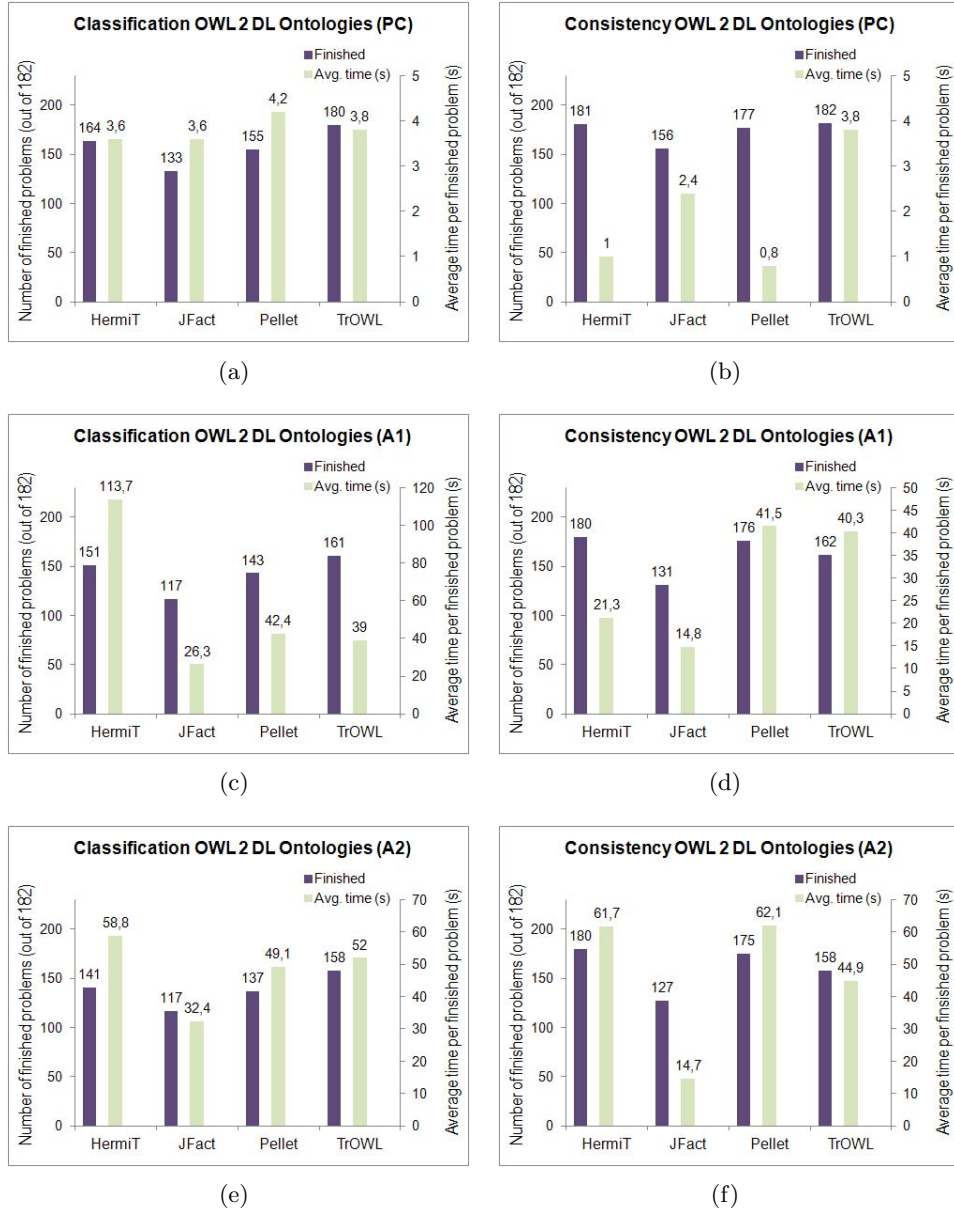


Figure A.1: Results (finished tasks/average time) for the complete OWL 2 DL ontology set.

First of all, analyzing the results in terms of finished tasks, we can observe that, as expected, the reasoners on PC finished more tasks than on Android due to the more powerful hardware. Only *HermiT* and *Pellet* finished a similar number of tasks on the desktop computer and on Android devices when checking the consistency (181 vs 180 and 177 vs 175-176). Moreover, the reasoners on A1 finished more tasks than the same reasoners running on A2 for most of the tests. There are only two situations where both devices finished the same number of tasks: computing the classification with *JFact* (117 out of 182 tasks on both devices), and checking the consistency with *HermiT* (180 out of 182 tasks on both devices).

Although the number of finished tasks on the desktop computer and on the mobile devices are different, both Android devices follow the PC trend for the classification task: *TrOWL* is the reasoner that finished a higher number of tasks, followed by *Hermit*, *Pellet*, and *JFact*. For the consistency checking, the PC trend is not followed by the Android devices: on the Android devices both *HermiT* and *Pellet* finished more tasks than *TrOWL* while *TrOWL* was the reasoner that finished more tasks on PC.

Analyzing the results of *TrOWL* for the consistency checking on A1, we noticed that the timeout period elapsed for 13 tasks, while it never elapsed for any task on PC. Also, 7 large ontologies that were processed by *TrOWL* on PC threw out of memory errors on the smartphone. Table A.4 shows the number of tasks that could not be finished by the reasoners on each device and by each reasoner. We classify the reasons for not finishing as elapsed time out (“T/O” column in the table), and others (“Other” column). Notice that, as PC had enough memory to perform the reasoning, the errors on the “Other” column for PC are mostly due to unsupported data types. On the Android devices, the same errors always occur and some additional problems appear: usually, out of memory issues when processing large ontologies. Notice also that, in some situations, timeout problems on PC are translated into out of memory problems on Android devices. This happens because the timeout of PC was set to 5 minutes while on Android it was set to 25 minutes. Therefore, for some complicated tasks the reasoner run out of memory before the timeout elapsed (for example, *HermiT* in the consistency checking).

We would like to highlight that, although the results obtained for this test (Figure A.1) enable us to compare the reasoners in terms of number of finished tasks, one should be cautious when comparing their processing times. The most difficult ontologies (i.e., the most expressive or large ones) require more time to be classified and so, completing more tasks could imply increasing the average time per finished problem. For example, notice that *HermiT* running

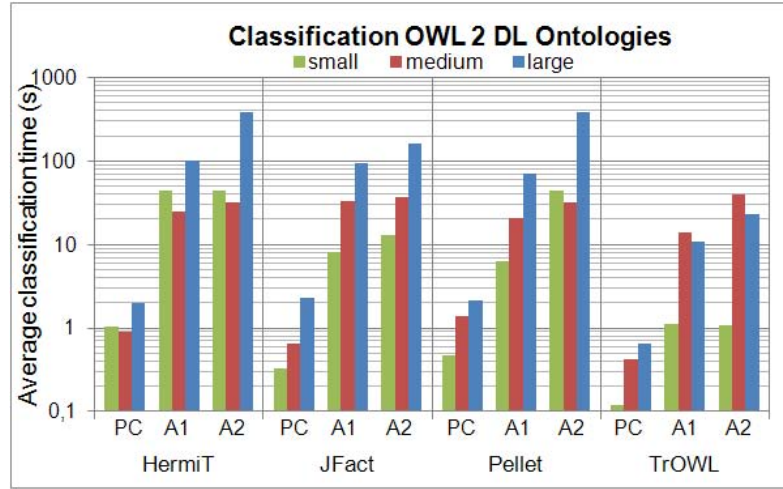
on A1 completes 151 classification tasks with an average time per task of 113.7s while the same reasoner running on A2 completes 141 tasks with an average time of 58.8s (the average time increases about 50 seconds because of these 10 more challenging tasks). Therefore, we performed the following test to be able to compare the processing time of the reasoners.

Comparing the Processing Time

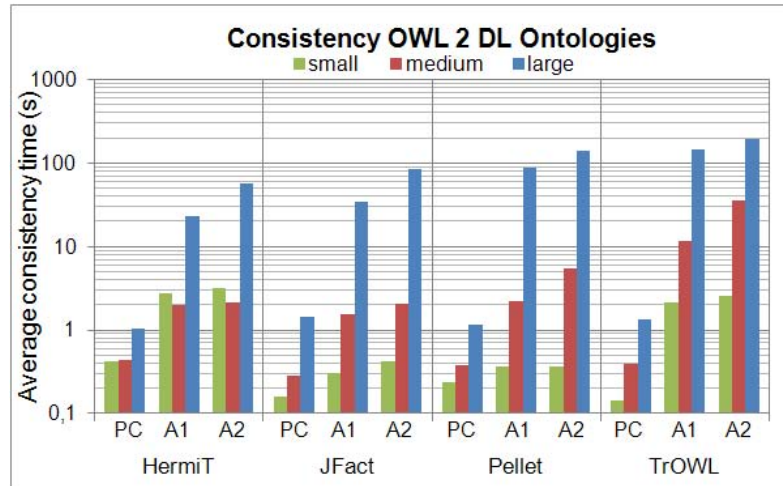
To fairly compare the reasoners regarding the average processing time needed per ontology, we selected the minimum set of DL ontologies that all the devices and all the reasoners were able to process. We also split the ontologies with respect to their number of axioms into three subsets (small, medium, and large) obtaining a *minimum DL set* of 93 ontologies for the classification task (29 small, 62 medium, and 2 large ontologies), and 124 ontologies for the ontology consistency checking task (37 small, 81 medium, and 6 large ontologies).

Comparing Trends Figure A.2(a) and Figure A.2(b) show the results obtained for the reasoners computing the classification and consistency, respectively, on the three devices with the minimum set of DL ontologies. First of all, as in our previous test, the mobile devices follow the PC general trend. There are two exceptions: on the one hand, *TrOWL* was slightly faster on PC for the classification of the medium set of ontologies than for the large one (about 0.2s), but on the mobile devices it was slightly slower (around 3s on the A1 device); on the other hand, *HermiT* was slightly faster on PC for checking the consistency of the small set than for the medium one (around 0.02s), but, on the mobile devices, it was slightly slower (around 0.7s on the A1 device). However, in the case of the classification in *HermiT*, the difference between the small and medium sets is also small but the trend on the desktop computer and the mobile devices is similar.

Comparing Performance Table A.5 shows the difference on the performance of the desktop computer, the smartphone, and the tablet in terms of the number of times of PC being faster than the Android devices. In general, PC outperformed A1 and A2 for all the reasoners. The greatest differences happen in the large ontology set, where, for example, the desktop computer is almost 110 times faster than A1 when checking consistency using *TrOWL*, and almost 200 times faster than A2 when computing classification with *HermiT*. However, the consistency checking of the small set of ontologies in the PC



(a)



(b)

Figure A.2: Average computing time for each ontology group in the minimum set of OWL 2 DL ontologies processed by all the devices and reasoners.

was “only” 2 times faster than A1 for *JFact* (from 0.16s to 0.31s) and *Pellet* (from 0.24s to 0.37s).

TrOWL was the fastest reasoner in both devices for the classification of all the ontology sets, but it uses approximate reasoning in OWL 2 DL. Note also that the difference between the different reasoners is smaller on PC than on

mobile devices, for both reasoning tasks. In the mobile devices, the order of the reasoners according to their reasoning times is usually the same as in the desktop computer, although this is not always the case due to the variance of the results obtained for each test repetition.

A.2.3 Comparing the Reasoners for the OWL 2 EL Profile

In this section, we detail the results of our performance experiments for the OWL 2 EL profile. For the OWL 2 EL ontology set (with 193 ontologies), we tested the following reasoners: *ELK 0.4.0*, *HermiT 1.3.8*, *jcel 0.19.1*, *JFact 0.9.1*, *Pellet 2.3.1*, and *TrOWL 1.4*.

Comparing the Number of Finished Tasks

The results for the classification task for the desktop computer (PC), the Galaxy Nexus (A1), and the Galaxy Tab 2 (A2) are shown in Figure A.3(a), Figure A.3(c), and Figure A.3(e), respectively; for the consistency checking results for PC, A1, and A2 see Figure A.3(b), Figure A.3(d), and Figure A.3(f).

As above mentioned for the DL ontology set, we can observe that the reasoners on PC finished more tasks than on Android. However, the difference on the number of finished tasks is smaller than for the DL ontology set. On the one hand, in the DL ontology set, the difference of finished tasks on PC and A1 is: 16 (for classification) and 25 (for consistency checking) for *JFact*; 19 and 20 for *TrOWL*; 13 and 1 for *HermiT*; and 12 and 1 for *Pellet*. On the other hand, in the EL ontology set the difference of finished tasks on PC and A1 is: 12 and 10 for *JFact*; 2 and 2 for *TrOWL*; 4 and 4 for *jcel*; 4 and 1 for *HermiT*; 4 and 1 for *Pellet*; and 1 and 0 for *ELK*. This can be explained because of the difference of expressivity of the two ontology sets. As the EL profile is less expressive, performing reasoning tasks within this profile is less costly. Therefore, more tasks can be finished on the mobile devices.

Also, in general, A1 finished more tasks than A2. There are only two situations where both devices finished the same number of tasks: first, *ELK* finished 190 out of 193 classifications on both devices; and second, *HermiT* finished 192 out of 193 consistency checkings on both devices.

Table A.6 shows the results for the analysis of the tasks that were not finished. As for the DL ontology set, the errors of the “Other” column for the PC are mostly due to unsupported datatypes or ontologies that could not be processed by the reasoner. On the Android devices, the increment on the number of “Other” errors is due to out of memory errors. Notice also that

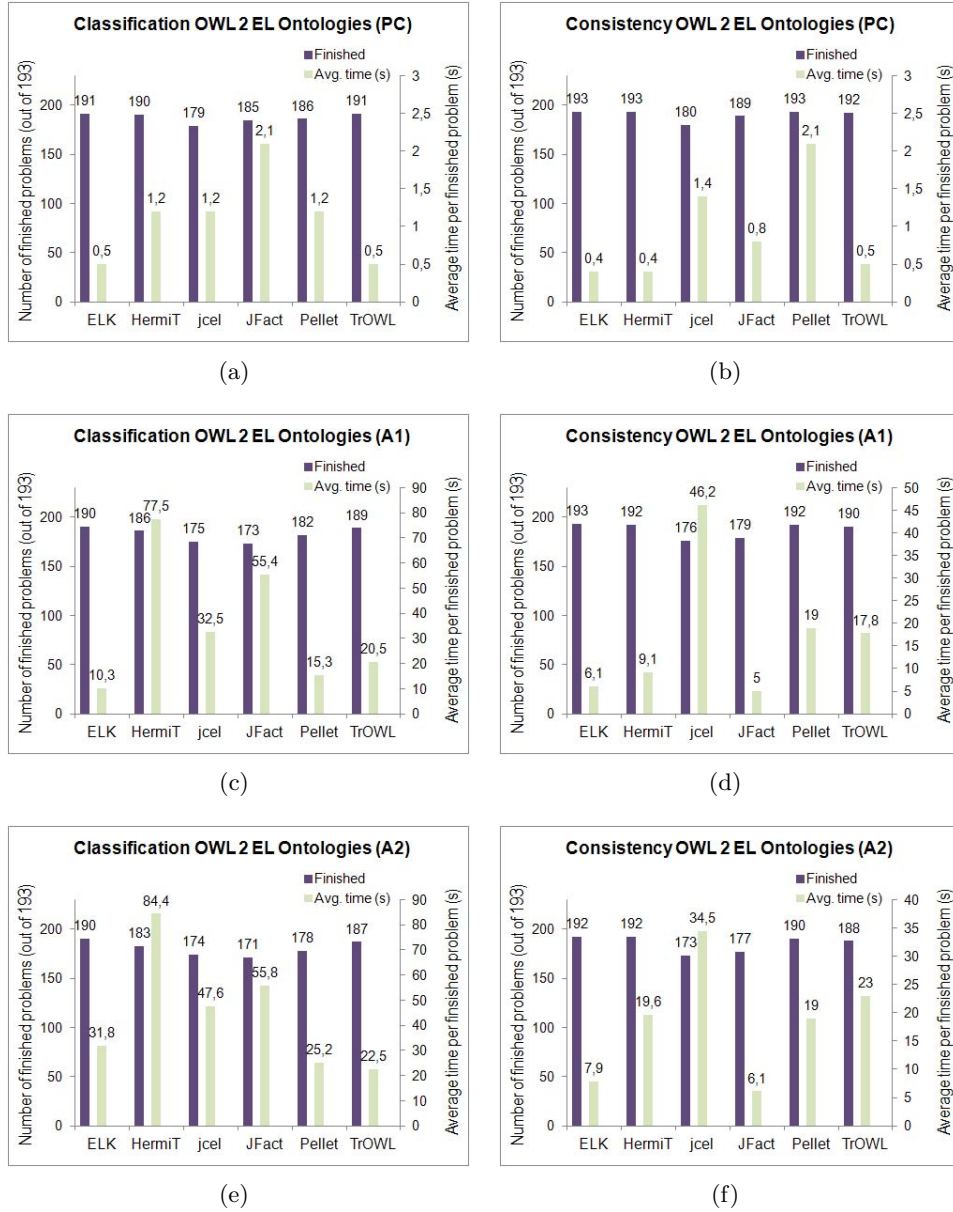


Figure A.3: Results (finished tasks/average time) for the complete OWL 2 EL ontology set.

the number of “Other” errors for the consistency checking is zero in *HermiT*, *Pellet*, and *ELK*, for the three tested devices.

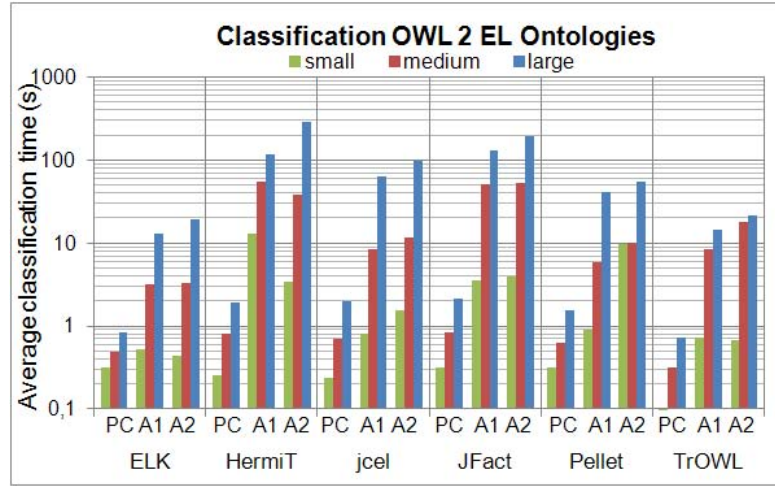
Comparing the Processing Time

As we did for the DL ontology set, we selected the minimum set of EL ontologies that all the devices and all the reasoners were able to process to be able to fairly compare the performance of the reasoners regarding the average processing time per ontology. We also split the ontologies with respect to their number of axioms (small, medium, and large) obtaining a *minimum EL set* of 152 ontologies for classification (62 small, 67 medium, and 23 large ontologies) and 166 ontologies for consistency checking (65 small, 72 medium, and 29 large ontologies).

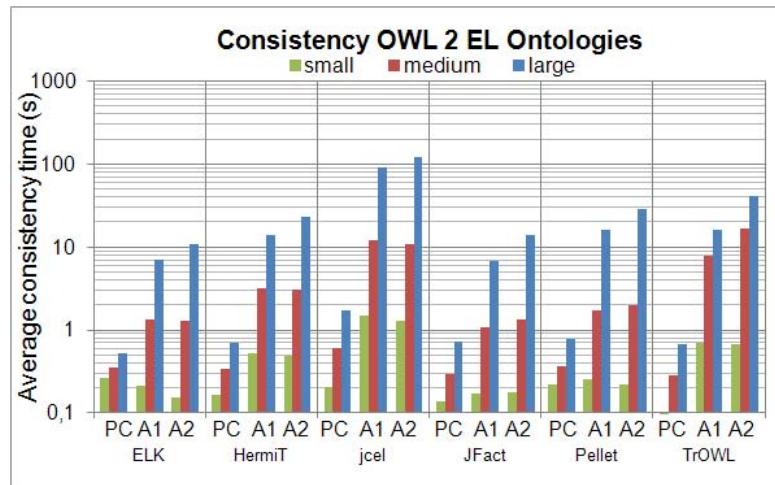
Comparing Trends Figure A.4(a) and Figure A.4(b) show the results obtained for classification and consistency, respectively. Notice that both mobile devices follow the general trend obtained for the PC. For all the reasoners and devices, the average time for the small set is smaller than the time for the medium one, which in turn is smaller than for the large set. There are two situations that should require further explanation: first, *HermiT* was faster on A2 than on A1 for the classification of the small and medium sets of ontologies; and second, *Pellet* was as fast on A2 for the classification of the small set as for the medium one (while it was faster on the PC and A1). Analyzing the results per ontology of these ontology sets, we observed that there were some ontologies where the variance of the measured times for the three repetitions of each test was high, which led to these situations.

Comparing Performance Table A.7 shows the difference of performance between the PC and the two mobile devices in terms of number of times of the PC executions being faster than the Android ones. As in the DL ontology set, in general, the desktop computer outperformed the mobile devices for all the reasoners. The greatest differences are again achieved on the large and medium ontology sets. For example, for the classification of these sets using *HermiT*, PC is almost 70 times faster than A1, and almost 150 times faster than A2. In general, the differences between PC and the mobile devices are smaller for consistency checking. Notice that there is a situation where the Android devices were faster than the desktop computer: in the consistency checking with *ELK*, PC was 0.8 (from 0.26s to 0.21s) and 0.6 (from 0.26s to 0.15s) times “faster” than A1 and A2 devices, respectively (these values are coherent with the observed variance).

As it happens in the DL ontology set, the difference between the different reasoners on PC is smaller than on the mobile devices. Furthermore, the order of the reasoners according to their reasoning times is usually the same as in the desktop computer.



(a)



(b)

Figure A.4: Average computing time for each ontology group in the minimum set of OWL 2 EL ontologies processed by all the devices and reasoners.

A.2.4 Other Experiments

In this section we summarize some additional experiments measuring other interesting features of the mobile devices, namely the impact of the memory and the virtual machine.

A.2.5 Analyzing the Impact of Memory

After comparing the results obtained for PC and Android, we wanted to check how the limitation of memory that Android imposes on applications and its management by the OS affects the results. Our goal was to check whether the processing time would be affected if the maximum memory for the application was limited.

Firstly, we restricted the maximum memory for the desktop computer to 256 MB RAM (the maximum size that Android assigns to the applications in our test devices) and computed the classification of the DL and the EL ontology sets. We observed that there are significant differences on the number of finished tasks but not on the reasoning time. Figures A.5 and A.6 compare the number of finished classifications in PC, PC with the memory limitation (denoted PCmem), and A1 for the small (S), medium (M), and large (L) OWL 2 DL ontology set. In particular, we represent the differences between the number of finished tasks in the devices: PC vs. PCmem, PC vs. A1, and PCmem vs. A1.

We can see that the number of finished tasks over small ontologies is the same in all the cases. For medium ontologies, PC and PCmem only differ in one OWL 2 DL ontology, although A1 does not finish 14 OWL 2 DL and 5 OWL 2 EL ontologies. The only OWL 2 DL reasoner that does not finish a smaller number of tasks on A1 is TrOWL, which computes approximated reasoning in this profile. In large ontologies, the number of unfinished tasks on PCmem and A1 is significant: more than 10% in the EL ontology set and more than 50% in the DL ontology set. Overall, the number of finished tasks in PCmem and A1 is comparable: 586 vs 572 OWL 2 DL ontologies, and 1099 vs 1095 OWL 2 EL ontologies. Note that some tasks finished on A1 but not on PCmem.

After investigating the role of memory on the desktop computer, we also limited the memory on the Android devices. To do that, we restricted the maximum memory heap for the application to the “standard memory heap” by setting the variable `android:largeHeap=‘false’`. In particular, we restricted the memory to 96 MB. Since the experiments on Android devices are more costly, we restricted to the classification (the most challenging task)

of the DL ontology set using Pellet and the Galaxy Nexus smartphone (A1, as it was the fastest device). For this experiment, we did not want to consider TrOWL (it only offers approximate reasoning in the DL ontology set) and JFact (because of the smaller number of finished classifications on A1); between HermiT and Pellet we chose the latter one to decrease the total computation time because, as shown in Figure A.2 (a), it was usually faster.

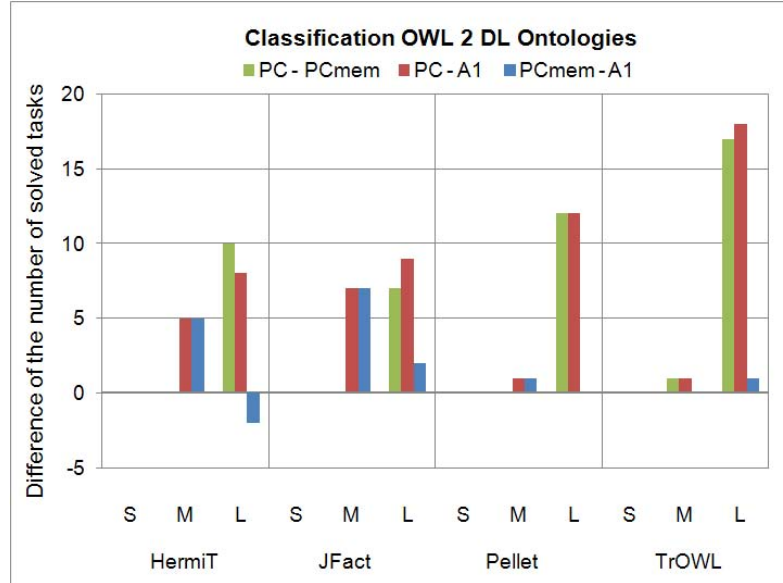


Figure A.5: Comparison of the number of finished classifications of OWL 2 DL ontologies.

The first aspect to highlight from the results is that, as it happened when limiting the memory on Android, limiting the memory decreased the number of finished tasks. The reasoner finished 143 tasks before and 135 after the memory limitation. These 8 tasks that could not be finished by the “limited version” include 2 medium and 6 large ontologies. Figure A.7 shows the comparison of the time needed for every ontology that the reasoner was able to classify in the two tests. In the graph, we plot the processing times difference as a percentage of the time required by the non-limited version (y-axis) and the time needed by the non-limited version in seconds (x-axis). The first thing we can highlight is that there are some negative values which mean that the limited version was faster than the non-limited version. This can be explained because measuring time consumption of the same application on the same device can

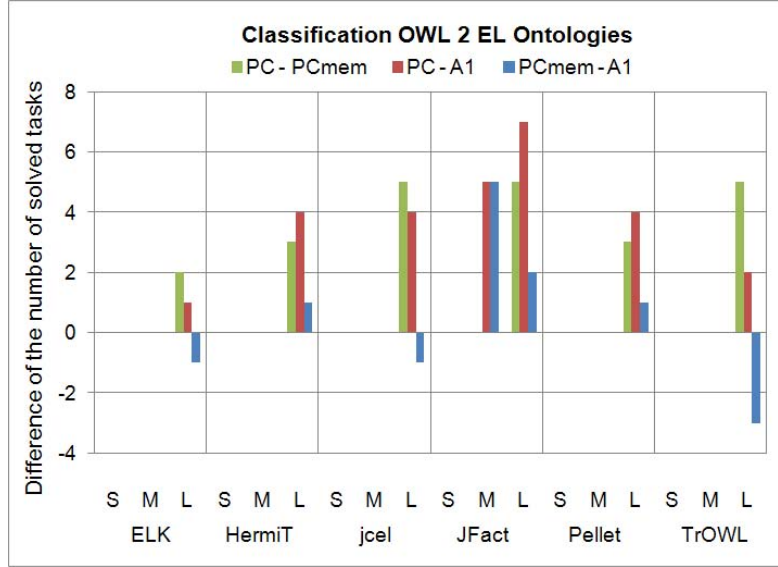


Figure A.6: Comparison of the number of finished classifications of OWL 2 EL ontologies.

have a small variance due to the management of the running applications done by the OS. Moreover, for ontologies that can be classified quickly (under 5s), the difference is around 40%, and reaches even 60% in some cases. In these cases, the ontologies require around 1s to be classified, so the actual difference in seconds is around 0.5s. Regarding the values obtained, notice that the difference between the two versions for those ontologies that need 15s or more to be classified is less than 2%. This includes ontologies that needed 80s-250s, where the difference is less than 1s. Therefore, the main conclusion of this test is that limiting the memory on the devices do not significantly modifies the time consumption, but it does affect the number of accomplished tasks.

A.2.6 Analyzing the Impact of the Virtual Machine

Very recently, as of November 2014, Google released the new Android 5.0 version that includes a new runtime environment called Android Runtime (ART) to replace the Dalvik virtual machine. While the previous virtual machine Dalvik uses just-in-time compilation every time an application is launched, ART uses a more sophisticated ahead-of-time compilation that can be performed just once during the installation of the application. This way,

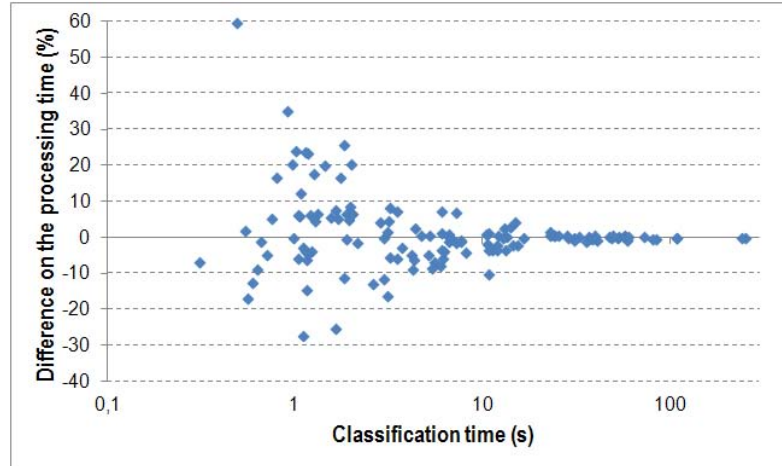


Figure A.7: Comparison of the Pellet reasoner on Android with limited and “unlimited” memory.

processor and battery usage are optimized.

Therefore, the performance of mobile applications running on the new virtual machine is expected to increase. However, attending to historic information, it is expected that previous versions of the OS would continue maintaining their popularity. In fact, Android 4.X required a bit more than a year to reach 50% of the Android devices and, as of July 2015, previous versions were present on almost a 6% of the global devices²⁹.

We performed a final test with this new runtime environment to show the expected tendency in the future. We used a Google Nexus 5 smartphone (equipped with a 2.26 GHz quad-core processor and 2 GB of RAM) for this test. We computed the classification of 10 ontologies extracted from the DL ontology set which we used in our tests in [BBYEM14] using Pellet. We ran this test twice, one with the current Android version (4.4) using the Dalvik virtual machine, and another one with the new Android version (5.0) using ART.

Figure A.8 shows the comparison of the classification time on both virtual machines and for each of the 7 ontologies that finished the test (2 of them failed because of unsupported datatypes, and another one elapsed a time out). Notice that the new Android version and its virtual machine outperformed the previous one for all the ontologies using the same hardware. In fact, Pellet

²⁹<http://developer.android.com/about/dashboards/index.html>, last accessed 2015-07-04.

on the ART virtual machine was 2.5 times faster on average. As the tests were only performed once for each device we should take this number with a pinch of salt due to the variance of the times measured in our previous experiments. However, the improvement is similar with large, medium, and small ontologies so we can highlight that *reasoners on the new ART virtual machine could be around 2 times faster than in Dalvik*. Therefore, although reasoning on a desktop computer clearly continues outperforming reasoning on mobile devices, the processing times on mobile devices will go on decreasing as more powerful hardware and OS and software optimizations will be available, making reasoning on mobile devices even more feasible.

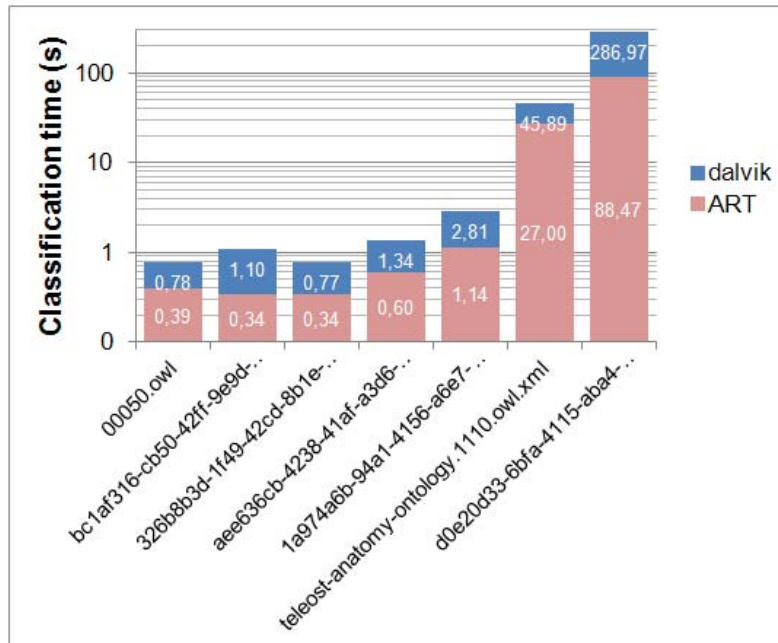


Figure A.8: Comparison of the Pellet reasoner on the Dalvik and ART virtual machines.

A.2.7 Discussion

In this section, we present some key ideas extracted from our experience within these tests. We believe that developers of existing and future semantic APIs and reasoners can benefit from taking into account these pieces of advice to enable mobile application (and specifically Android) developers to use their

technologies. We also hope that developers of mobile applications that are considering using semantic technologies find these advices interesting when creating their applications.

Regarding the coding of semantic APIs and reasoners, we highlight the following points:

- Some of the reasoners we have studied and/or ported include many functionalities that might not be required for mobile application developers (e.g., graphical packages or remote access). We advocate for modular designs with a micro-kernel supporting only the core reasoning tasks, and with the rest of functionalities (e.g. parsers, query processing, etc.) included in separated modules. This would greatly help porting and using them, as the deployed version could be tailored and thus, some of the limitations of mobile devices could be avoided.
- Using native code to develop a reasoner can help the developer to avoid many of the limitations that the virtual machine imposes, achieving better performance. However, the difficulty of use of a reasoner within a mobile application/an Android project might be an unbearable barrier that could reduce the popularity of the reasoner. For this reason, there should be a simple, and if possible standard, mechanism to support its use from mobile applications. For that, we advocate providing always interfaces (such as OWL API or Jena) for mobile application developers.
- Beware of using some common standard Java libraries. Current Android/Dalvik versions do not perfectly align to a standard Java environment (e.g., all the graphical packages are not supported), and, even worse, there are libraries which are not completely included on current versions of Android and do not throw compilation or execution errors (e.g., JAXB and Xerces). Therefore, it is possible to import some reasoners on an Android project but the results of the reasoning are not the same as on desktop computers (as it happens in JFact 1.2.1).
- Keep in mind resource limitations. Regarding memory, in Android, we can reduce the memory imprint by using shared libraries. However, one might end up exceeding the maximal number of methods that can be invoked from a single dex file³⁰. Thus, we advocate trying to use only

³⁰Although multidex mobile applications can be build since Android 4.0 (SDK 14), there are many documented problems with it, and we discourage its use, see <https://developer.android.com/tools/building/multidex.html> for details.

those packages/classes needed and building a customized version of the library.

- As a final suggestion, using open-source licenses and distributing the code (as, for example, HermiT and Pellet) can help developers to find programmers willing to help in porting existing reasoners (as we have done) to other mobile OS, such as iOS.

Regarding the use and performance of semantic technologies on current Android devices:

- The performance of all the semantic reasoners we tested on Android is lower than on a PC. Indeed, the PC is from 1.5 to 150 times faster in our tests depending on the task and the ontology. Therefore, complicated tasks such as classification of large ontologies should be reduced to the minimum.
- The reasoners tested on Android behave similarly to the same version on a desktop computer: reasoners that are faster on desktop computer are generally faster on Android too.
- The variance of the reasoning time is higher on Android devices than on desktop computers. In the case of Android devices, the time variance is almost negligible for small ontologies, moderate for medium ontologies, and significant for larger ontologies. Ontologies in the OWL 2 DL ontology set usually produce higher variances than OWL 2 EL ontologies. Note that the main priority in Android is the responsiveness of the device, and, thus, its scheduling policies seem to penalize applications which require intensive use of resources (e.g., CPU time and memory). In general, DL ontologies require more computation time than EL ontologies, and thus, they are more prone to be affected by the variance introduced by the possible context changes.
- When developing a mobile semantic application, it is a good idea to separate the reasoning thread in an isolated process whenever possible, and not relying on Android to relaunch it if it gets killed (e.g., due to memory issues). If a task is killed for being too resource greedy, Android seems to penalize it and does not resume it immediately.
- To use the full potential of mobile devices, reasoners could compile the core of their code as Android native code and avoid this way the overhead of the Android virtual machine.

- Android uses UTF-8 as encoding by default. Problems with the characters in the URIs can appear when working with ontologies that have been developed in an editor that stores them in any other encoding.

We also want to share some experiences with mobile application developers (researchers or not). First, with respect to the use of DL reasoners specifically designed for mobile devices, some authors of these reasoners argued that “current Semantic Web reasoners cannot be ported without a significant re-write effort” [RSSGL12]. After our work, this is no longer true, as we have made it possible to reuse existing semantic APIs and DL reasoners on Android and, in some cases, no rewriting was even needed. However, it is interesting to study if these reasoners outperform reused reasoners in mobile devices. In fact, most of the optimization techniques implemented by classical DL reasoners cannot easily be adopted in mobile systems, since they decrease running time but definitely increase the use of memory, which is limited in mobile devices. Thus, a study of the trade-off between expressivity and resource consumption for mobile devices could be useful.

In our experience, the use of semantic technologies on mobile devices allowed us to create smart mobile applications that do not require Internet connection and preserve the privacy of users by avoiding the use of cloud-based services. For example, our prototypes of SHERLOCK (see Section B.1), FaceBlock, Rafiki, and Triveni (see Section B.3 use Hermit). We only experienced problems when dealing with large ontologies (with hundreds of individuals) and complex SWRL rules. In those scenarios (large ontologies), the classification time exceeded the amount of time a user would be willing to wait for an answer.

As a summary, in this work we have tested semantic technologies on Android by designing some experiments with standard ontology sets used in the OWL Reasoner Evaluation 2013 workshop. First, we have tested that the ported versions on Android obtain the same results than the original versions on PCs. Then, we have tested the performance of the reasoners in terms of number of tasks finished and time consumed per task in two mobile devices, a smartphone and a tablet. Finally, we have tested the role of the amount of available memory and the virtual machine used on Android. The complete results of our experiments can be found on the webpage [And], together with a detailed description of all the changes needed to port the semantic reasoners and, if the licenses make it possible, download links.

From a practical point of view, the main limitation that reasoners will face on current smartphones/tablets concerns memory usage and processing time (and hence, battery consumption). Our experiments show that reasoners running on a PC are 1.5 to 150 times faster than on Android devices. Also,

the number of out of memory errors increase on Android devices compared with PCs, usually in the OWL 2 DL profile and in larger ontologies. We also noticed important differences in the performance of the three analyzed Android devices, showing that, although mobile devices are far from being desktop computers, they are increasing their capabilities quickly as needed by challenging tasks, such as semantic reasoning. We have recently done some tests with more modern devices that confirm this trend. In addition, we have shown some results with the future Android runtime, ART, that show that the same tasks on the same devices can be executed around 2 times faster.

We would like to finish this summary of the conclusions of the work with a statistical curiosity. We estimate that the complete empirical experimentation reported in this section required a total computing time of more than 1000 hours (which is more than 41 days only for the computation of the different tests).

A.3 Related Work

This section will be divided in two parts. Firstly, we will overview the previous work on supporting DL reasoners on mobile devices. Then, we will point to some relevant literature on empirical evaluations of DL reasoners on desktop computers.

A.3.1 Reusing and Evaluating DL Reasoners on Mobile Devices

To the best of our knowledge, our work is the first effort to offer a systematic support and evaluation of the performance of existing DL reasoners on mobile devices. However, there are some previous works in the field that are worth mentioning.

Using ELK on Android devices has been recently investigated [KK13]. In particular, the authors implemented some minor changes to make the reasoner work on an Android smartphone and performed some experiments on a Google Nexus 4 Android 4.2 phone. In particular, the authors measured the classification time of 5 \mathcal{EL} ontologies both in the Android device and in a desktop computer. The results show that reasoning times in the Android device are acceptable even if much slower (two orders of magnitude) than in the desktop version. We have also considered ELK in our experiments, measuring and comparing its reasoning times over more ontologies.

As previously mentioned, using Pellet on mobile (J2ME) devices has been

investigated [SK08; SKG09]. The modification of Pellet reasoner included some new optimization techniques and was called *mTableau*. The authors did some experiments on a desktop computer, proving that the optimization is useful to reduce the response time in situations of limited memory. They also performed some experiments (4 consistency tests over 2 ontologies) in a PDA showing that the reasoning times are acceptable. However, their approach is more oriented to proving the usefulness of the optimizations rather than performing a comparison between the performance of the reasoner in mobile and desktop devices.

Finally, there are three recent works that consider the use of reasoners on mobile devices complementing our work. Regarding battery consumption, [PM14] analyzes the performance per watt of Jena, Pellet, and Hermit over two ontologies. The authors found a nearly linear relationship between energy consumption and processing time, and studied the effects of some smartphone features (WiFi, 3G, and 4G radios) on battery consumption. More recently, [VNP15] studies the battery consumption on Android 5.x (API 21) of Pellet, Hermit, and Androjena, performing four different reasoning tasks over some datasets generated using the LUBM benchmark generator [GPH05]. Their software-based approach could be used to extend our study on the performance of DL reasoners on current and future versions of Android. Finally, [WHAA14] presents a benchmark framework for mobile semantic reasoners allowing them to be deployed on different platforms such as Android or iOS. So far, the authors have only considered 4 reasoners (AndroJena, Nools, RDFQuery, and RDFStore-JS), since their work is more focused on generability and extensibility, making it easier to add new mobile platforms and reasoners.

A.3.2 Evaluating DL Reasoners on Desktop Computers

The developers of some reasoners have performed evaluations of their systems with the main objective of showing that their new tools outperformed existing ones. There are also several (more or less) independent experimental comparisons in the literature that we will overview here.

In [Pan05], FaCT++, Pellet 1.1.0, and Racer 1.8.0 reasoners were compared for the classification of 135 OWL ontologies, with FaCT++ being slightly preferable (faster and more robust). Then, in [GTH06], FaCT++ 1.1.3, Pellet 1.3, and RacerPro 1.8.1, were compared along with KAON2 for the classification of 172 ontologies, without a clear winner due to the considerable difference of performance across ontologies. Hermit, KAON2, Pellet, RacerPro, Sesame, and SwiftOWLIM reasoners were compared for classification and conjunctive

query answering over 3303 ontologies in [BHJV08]. The authors concluded that SwiftOWLIM may be preferable in low expressive languages, RacerPro can be recommended in expressive ontologies with small ABoxes, and KAON2 is the best alternative in the other cases. The reasoners CB, CEL, DB, FaCT++, and HermiT were evaluated for the classification of 4 \mathcal{ELH} ontologies [DK09]. CB is the best option, although the results are focused on evaluating the new technique of computing classification using an SQL system. Dentler et al. [DCTK11] evaluated 8 reasoners (CB build 6, CEL 0.4.0, FaCT++ 1.5.0, HermiT 1.3.0, Pellet 2.2.2, RacerPro 2.0 preview, Snorocket 1.3.2, and TrOWL 0.5.1) over 3 OWL 2 EL ontologies for 4 reasoning tasks (classification, concept satisfiability, TBox consistency, and subsumption). As usual, there is not a clear winner. Besides, the authors also performed a very detailed comparison of other features of the reasoners. Another experiment measures the classification time of 358 real-world ontologies for 4 reasoners (FaCT++, HermiT, Pellet, and TrOWL) [KLK12]. The best reasoner depends on the criteria: Fact++ has the lowest median, HermiT has the lowest mean, TrOWL has the lowest number of errors, and Pellet has the lowest number of errors among the complete reasoners. Since DLs usually have a good performance in practice but high worst-case complexities, [GMPS13] investigates how often reasoning with existing ontologies requires an unreasonable time. The authors consider 4 reasoners (FaCT++ 1.6.1, HermiT 1.3.6, JFact 1.0, and Pellet 2.3.0) and 1071 ontologies, showing that most of the times there is some reasoner giving a quick response time, with Pellet being the most robust one. Hence, most of the existing ontologies on the Web are not inherently intractable but just hard for some particular DL reasoners.

Finally, it is worth to mention that the OWL Reasoner Evaluation Workshop (ORE) series organize DL reasoner competitions. The 2012 competition³¹ considered 143 ontologies and 5 reasoning tasks (classification, consistency, concept satisfiability, entailment, and instance retrieval) for 4 reasoners (FaCT++, HermiT, jcel, and WSReasoner). The 2013 competition [GBJRMPEGK13] considered 204 ontologies classified in 3 profiles (OWL 2 DL, OWL 2 EL, and OWL 2 RL) and 3 reasoning tasks (classification, consistency, and concept satisfiability) for 14 reasoners (BaseVISor, Chainsaw, ELepHant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORE, SnoRocket, Treasoner, TrOWL, and WSClassifier). The competition organizers gave priority to robustness of the systems rather than the reasoning times alone. The latest editions held at 2014 [BGJRMPS13] and 2015 [DGGHJMPSS15] considered more than 16500 unique ontologies divided in 2 profiles (OWL 2 DL, and OWL 2 EL), and 3 rea-

³¹<http://www.cs.ox.ac.uk/isg/conferences/ORE2012/evaluation/index.html>

soning tasks (classification, consistency checking, and realisation). The number of ontologies used out from the ontology set depended on the profile and the reasoning task, ranging from 200 ontologies used for realization in DL profile to 300 ontologies used for classification in EL profile. The participants in the 2014 edition were Chainsaw, ELepHant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORE, Treasoner, and TrOWL (11 reasoners), and the participants in the 2015 edition were Chainsaw, ELepHant, ELK, FaCT++, HermiT, jcel, JFact, Konclude, MORE, PAGOdA, Pellet, Racer, and TrOWL (13 reasoners). In all the competitions, the best reasoner depends on the reasoning task and the expressivity of the ontology.

Software	Version	Originally compatible	Currently compatible
Jena	2.12.0	×	✓
OWL API	3.4.10	✓	✓
CB	build 6	×	✓*
CEL	1.0	×	×
Chainsaw	1.0	×	×
ConDOR	revision 13	×	×
ELepHant	0.4.0	×	×
ELK	0.4.0	×	✓*
FaCT++	1.6.3	×	×
fuzzyDL	build 60	×	×
HermiT	1.3.8	×	✓*
jcel	0.19.1	✓	✓
JFact	0.9.1	✓	✓
KAON2	unknown	×	×
Konclude	0.6.0	×	×
MORe	0.1.5	×	✓*
Pellet	2.3.1	×	✓*
Racer	2.0	×	×
TReasoner	revision 22	✓	✓
TrOWL	1.4	✓	✓
WSClassifier	revision 1	×	×

*: It has been ported by us.

Table A.1: Android support for some semantic APIs and DL reasoners.

		HermiT	JFact	Pellet
DBpedia	A1	5.13	UDT	63.15
	A3	8.87	UDT	115.30
GO	A1	487.98	435.60	83.97
	A3	OOM	OOM	OOM
NCI	A1	2020.48	OOM	OOM
	A3	OOM	OOM	OOM
Pizza	A1	10.43	3.42	20.77
	A3	14.88	4.90	33.22
Wine	A1	361.38	1609.32	131.80
	A3	511.97	2196.05	194.12

Table A.2: Comparison of classification time (in seconds) for two Android devices. *OOM*: Out Of Memory; *UDT*: Unsupported Data Type.

		CB	ELK
DBpedia	PC	UDT	0.5
	A1	UDT	19.7
GO	PC	0.6	1.4
	A1	8.6	30.6
NCI	PC	2.6	2.5
	A1	35.4	200.7

Table A.3: Comparison of classification time (in seconds) for PC and Android. *UDT*: Unsupported Data Type.

Reasoner	Classification		Consistency	
	T/O	Other	T/O	Other
HermiT(PC)	13 (7.1%)	5 (2.7%)	1 (0.6%)	0 (0%)
HermiT(A1)	20 (11%)	11 (6%)	0 (0%)	2 (1.1%)
HermiT(A2)	28 (15.4%)	13 (7.1%)	0 (0%)	2 (1.1%)
JFact(PC)	18 (9.9%)	31 (17%)	8 (4.4%)	18 (9.9%)
JFact(A1)	26 (14.3%)	39 (21.4%)	18 (9.9%)	33 (18.1%)
JFact(A2)	23 (12.6%)	42 (23.1%)	24 (13.2%)	31 (17%)
Pellet(PC)	20 (11%)	7 (3.9%)	5 (2.7%)	0 (0%)
Pellet(A1)	26 (14.3%)	13 (7.1%)	6 (3.3%)	0 (0%)
Pellet(A2)	33 (18.1%)	12 (6.6%)	7 (4%)	0 (0%)
TrOWL(PC)	1 (0.6%)	1 (0.6%)	0 (0%)	0 (0%)
TrOWL(A1)	9 (5%)	12 (6.6%)	13 (7.1%)	7 (3.9%)
TrOWL(A2)	18 (9.9%)	6 (3.3%)	18 (9.9%)	6 (3.3%)

Table A.4: Errors for uncompleted tasks in the DL ontology set.

		Classification			Consistency		
		S	M	L	S	M	L
HermiT	A1	44	28	52	7	5	22
	A2	44	36	193	8	5	54
JFact	A1	25	51	41	2	5	24
	A2	40	57	71	3	7	59
Pellet	A1	14	15	34	2	6	76
	A2	21	46	38	2	14	123
TrOWL	A1	9	33	16	15	29	109
	A2	9	96	35	18	90	142

Table A.5: Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 DL ontology set.

Reasoner	Classification (193)		Consistency (193)	
	T/O	Other	T/O	Other
ELK(PC)	2 (1%)	0	0	0
ELK(A1)	1 (0.5%)	2 (1%)	0	0
ELK(A2)	2 (1%)	1 (0.5%)	1 (0.5%)	0
HermiT(PC)	3 (1.6%)	0	0	0
HermiT(A1)	7 (3.6%)	0	1 (0.5%)	0
HermiT(A2)	10 (5.2%)	0	1 (0.5%)	0
jcel(PC)	1 (0.54%)	13 (6.7%)	0	13 (6.7%)
jcel(A1)	0	18 (9.3%)	5 (2.6%)	12 (6.2%)
jcel(A2)	7 (3.6%)	12 (6.2%)	7 (3.6%)	13 (6.7%)
JFact(PC)	6 (3.1%)	2 (1%)	3 (1.6%)	1 (0.5%)
JFact(A1)	12 (6.2%)	8 (4.1%)	8 (4.1%)	6 (3.1%)
JFact(A2)	10 (5.2%)	12 (6.2%)	9 (4.7%)	7 (3.6%)
Pellet(PC)	3 (1.6%)	4 (2.1%)	0	0
Pellet(A1)	6 (3.1%)	5 (2.6%)	1 (0.5%)	0
Pellet(A2)	10 (5.2%)	5 (2.6%)	3 (1.6%)	0
TrOWL(PC)	2 (1%)	0	1 (0.5%)	0
TrOWL(A1)	2 (1%)	2 (1%)	2 (1%)	1 (0.5%)
TrOWL(A2)	5 (2.6%)	1 (0.5%)	4 (2.1%)	1 (0.5%)

Table A.6: Errors for uncompleted tasks in the EL ontology set.

		Classification			Consistency		
		S	M	L	S	M	L
ELK	A1	2	6	15	1	4	13
	A2	1	7	23	1	4	21
HermiT	A1	51	69	60	3	9	20
	A2	14	47	148	3	9	33
jcel	A1	3	12	33	7	20	54
	A2	7	17	49	6	18	71
JFact	A1	11	60	62	1	4	9
	A2	12	64	92	1	5	20
Pellet	A1	3	9	27	1	5	21
	A2	31	16	36	1	5	37
TrOWL	A1	8	27	20	8	28	24
	A2	7	58	29	8	59	62

Table A.7: Number of times (rounded to the closest integer) of the PC version being faster than the Android ones for the small (S), medium (M), and large (L) OWL 2 EL ontology set.

Appendix B

Prototypes for the Semantic Management of LBS

In this chapter we present the prototype of the SHERLOCK system for the semantic management of Location-Based Services in wireless environments. We show a SHERLOCK prototype which obtains knowledge from other devices and guides the user in the selection of an interesting LBS (as explained in Chapter 4) and then processes it (as explained in Chapter 6). Then we present the modules developed to address different challenges such as knowledge update and camera views processing (see Figure B.1). Firstly, we present the *DUCK* module to exchange ontologies and integrate them by discovering subsumption relationships between their concepts (as explained in Section 5.2). Secondly, we present the *Triveni* module which exchanges context information about the user and the device with other SHERLOCK devices and generates a shared context model to improve the locally inferred user context (as explained in Section 5.1). Finally, we present the *MultiCAMBA* module which obtains high-level features of the views of cameras, using the local knowledge on the device, and computes the similarity of such views with the shot in a user request.

B.1 SHERLOCK Prototype

We have developed a SHERLOCK prototype to test the underlying ideas of the system. Firstly, we developed a preliminary prototype of the system for PCs. This was motivated because both the mobile agent platform and the semantic reasoner we were planning to use in our system were not available

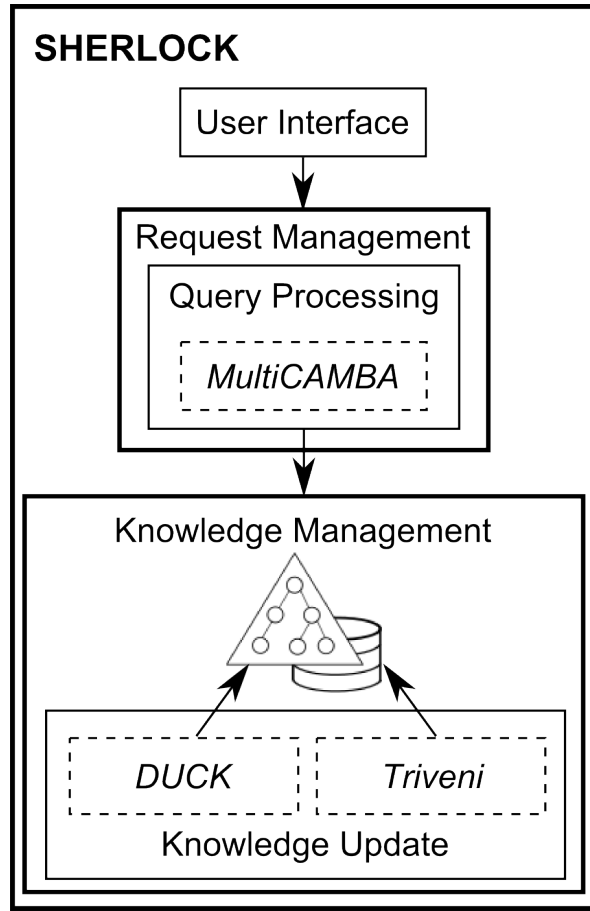


Figure B.1: Different modules of the SHERLOCK prototype.

for Android. Secondly, we developed an Android prototype of the system for smartphones and tablets. For that the technologies needed had to be ported to the Android system. In the following we explain both prototypes.

B.1.1 PC Prototype

The SHERLOCK prototype for PC¹ is a Java Applet that simulates how SHERLOCK would look like when running on a smartphone; this prototype uses the OWL API [HB11], the Pellet [SPCGKK07] reasoner, and the SPRINGS mobile agent platform [ITM06]. Figure B.2 shows four screenshots of the PC

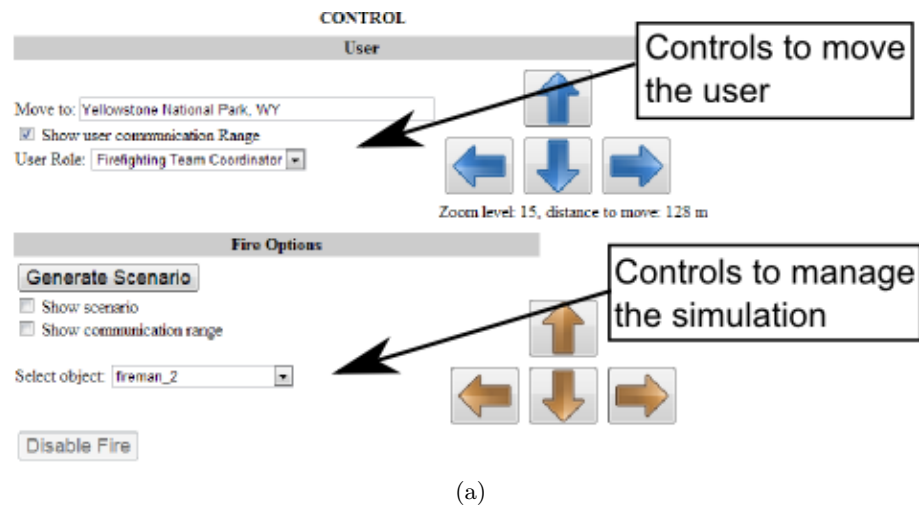
¹Available at <http://sid.cps.unizar.es/SHERLOCK/PC>



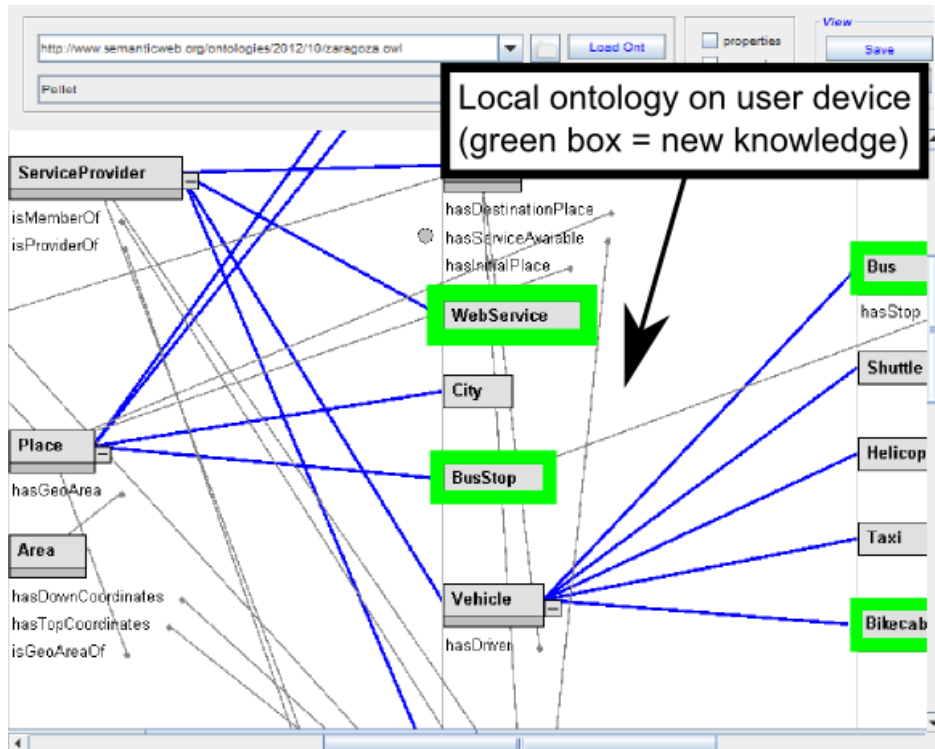
Figure B.2: Screenshots of the SHERLOCK prototype for PC.

prototype dealing with the first and second motivating scenarios. Notice that the prototype is running on a web browser and the smartphone part is simply an image.

To simulate other devices interacting with the system we developed a simple simulator that was incorporated into the Applet and generates moving objects (e.g., taxis, buses, firefighters, etc.) around the user and randomly moves them. The PC prototype also contains a mechanism that enables controlling the simulated objects (see Figure B.3(a)). Also, the simulator part of the prototype includes an ontology visualization tool that shows the real-time status of the ontology on the device. With this tool it is possible to see how new knowledge is added when the agents interact with other devices (see Figure B.3(b)). Notice that in this prototype we implemented a very simple approach for sharing and integrating knowledge. Therefore, devices only share their ontological definition (e.g., “the device is a taxi which is a type of transport”) and the information is integrated by directly adding the received OWL axioms into the local ontology.



(a)



(b)

Figure B.3: Screenshots of the simulator for the PC prototype.

B.1.2 Android Prototype

The SHERLOCK prototype for Android² was developed to enable us to test the system under real circumstances with real smartphones and tablets using their Bluetooth and WiFi communication mechanisms. The first step to develop the prototype was porting the technologies we used in the PC prototype to the Android system. As we explained in Appendix A we tried to use different Semantic Web technologies on Android devices and discover that, for example, the OWL API can be used directly on Android projects but the Pellet reasoner has to be ported. According to the results of our experiments with semantic reasoners on Android we decided to replace the Pellet reasoner by HermiT [GHMSW14] (the version we ported to Android). The HermiT reasoner obtained better results, in general, and could be directly incorporated to the app³.

Another technology that we needed and had to be ported to Android was the mobile agent platform. As in our PC prototype, we decided to use SPRINGS but, unfortunately, the platform is based on the use of the Java Remote Method Invocation (RMI) library for communications between agents. As explained in Appendix A, RMI is not supported in Android but there are two projects to port this library to the platform. After some tests we finally replaced the RMI library with the LipeRMI library and made some minor modifications regarding the management of sockets in Android, which differs slightly from the management by the virtual machine on a PC. After that, the SPRINGS mobile agent platform was successfully ported to Android and we performed some initial tests with a dozen of agents moving among three smartphones. SPRINGS was designed for a different environment with fixed computers and communications through wired connection. In this scenario, the platform has been tested with hundreds of agents from several devices moving every second and it overperformed other platforms [ITM06]. In our preliminary tests, we noticed that the performance when increasing the number of agents on the directly ported version for Android decreased rapidly. Therefore, we believe that the architecture of SPRINGS should be modified taking into consideration this new scenario with mobile devices and P2P wireless communications to enable the use of mobile agents on current mobile devices.

²Available at <http://sid.cps.unizar.es/SHERLOCK/Android>


³As we explained in Appendix A, the Pellet reasoner incorporates many classes and methods which makes it easy to exceed the limit imposed by Dalvik, Android's virtual machine, when used inside other projects.

B.1.3 Testing the Android Prototype

With the finished Android prototype of SHERLOCK we performed a test simulating the behaviour of a user that just installed the app. In the following we explain this test and show the different features of the prototype.

Selecting Settings and Profile

First, we tap SHERLOCK's launcher icon in our smartphone and this lead us to SHERLOCK's initial screen (see Figure B.4(a)) where we have access to the map, settings, and a quick access to our profile name and picture. The first time using SHERLOCK a dialog is prompted so we can enter our name and our picture (see Figure B.4(b)). Tapping on the image we can access the gallery of the device and select a picture for our profile. In the current version, our name and picture may be visible to other SHERLOCK users around us as part of the results to their requests. We can see our current picture and name at the bottom of the screen and change it anytime by tapping on the image or entering settings (where we can also edit other parameters that we will explain later).

In the settings screen we can define some information that SHERLOCK will take into account when processing our information requests. Accessing the settings can be done by tapping the settings button at the initial screen or the  button at the map screen. In addition to editing our name, email, and profile (which we will explain later), there are other options (see Figure B.5(a)) such as:

- Activate/deactivate the simulator (used to generate simulated moving objects around our location that will be part of the results obtained for user requests).
- Activate/deactivate the visibility of our local ontology (if active the ontology on the device will be shared with other SHERLOCK-enabled devices).

The prototype enables the user to define her profile (see Figure B.5(b)), which is her identification as a certain type of user (e.g., researcher or taxi). This information is useful for SHERLOCK to personalize the services that could be interesting for the user and also to connect her with other users. For example, the user can set her profile to taxi if she is driving a taxi and looking for customers, and SHERLOCK will use this information to show her location

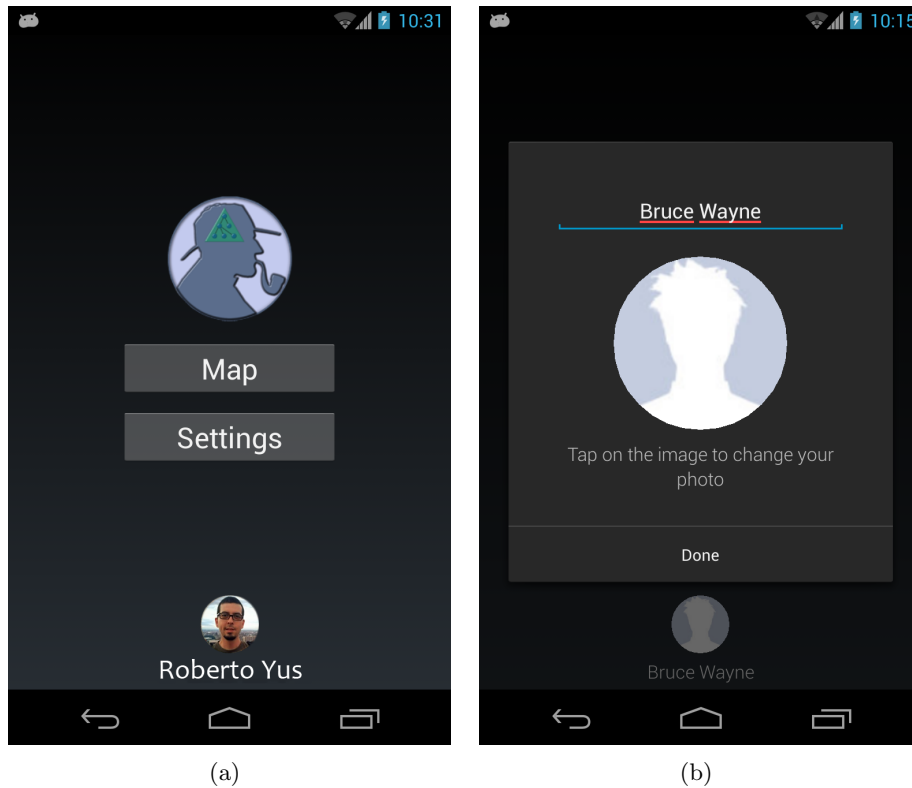


Figure B.4: SHERLOCK app: Home interface.

to users looking for transports. Also, a user may be at a conference and set her profile to researcher, so SHERLOCK will show her the location of other nearby researchers. Tapping on “profile” enables selecting the user profile by showing a list of all the profile kinds SHERLOCK knows at the moment. Meeting other SHERLOCK-enabled devices increases the local knowledge of our SHERLOCK and thus, new profile types can be found later.

Interacting with the System

Whenever the user enters the map screen, SHERLOCK shows her location on a Google map (see Figure B.6(a)). This map is the main mechanism to interact with the different Location-Based Services offered by the SHERLOCK app (remember that new services can be shared with the device by other SHERLOCK-enabled devices). There are three ways to start the interac-

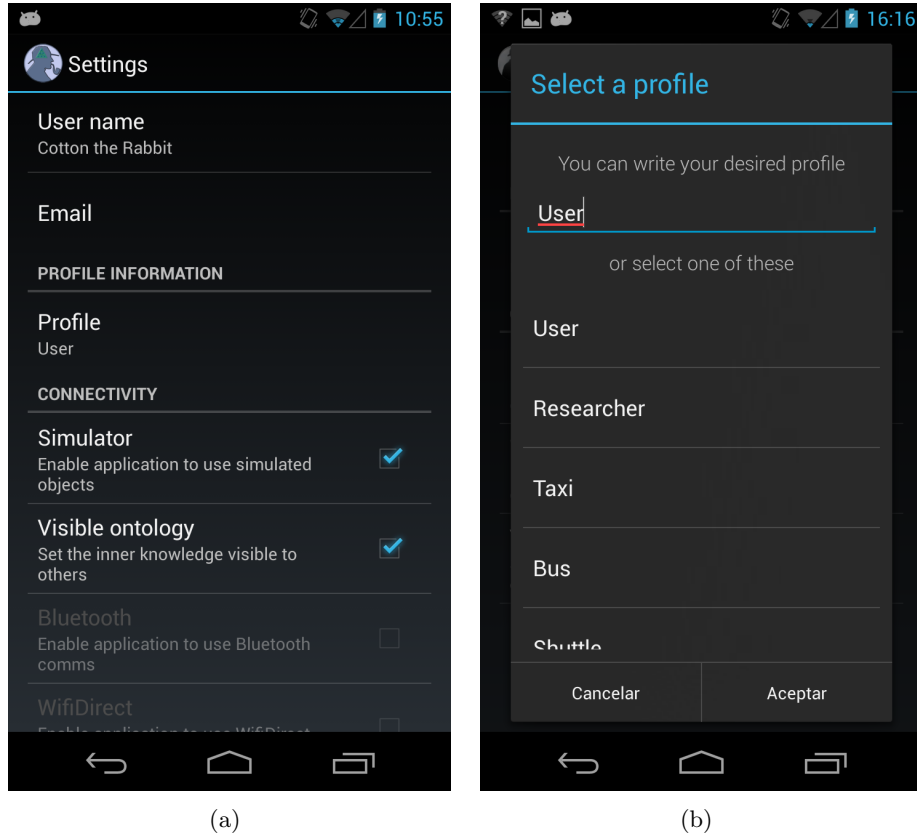


Figure B.5: SHERLOCK app: Settings.

tion to generate an information request: 1) Tapping on a location on the map, 2) tapping on an object, and 3) using SHERLOCK's search bar.

For example, to initiate a request for information we can tap on a location of the map for at least one second (see Figure B.6(b)). Then SHERLOCK reasons what services could be interesting for us using its knowledge, our context information, and the selected location.

After tapping on the map, SHERLOCK finds several interesting services related to that location and our context and shows them to us (see Figure B.7(a)). In this case, it shows two services: A service with tourist information and a service to find transports. If we tap on the "Transport service" SHERLOCK starts the LBS to find transports in the area. Some LBS could have been defined in the ontology as parameterizable. This means that they need some

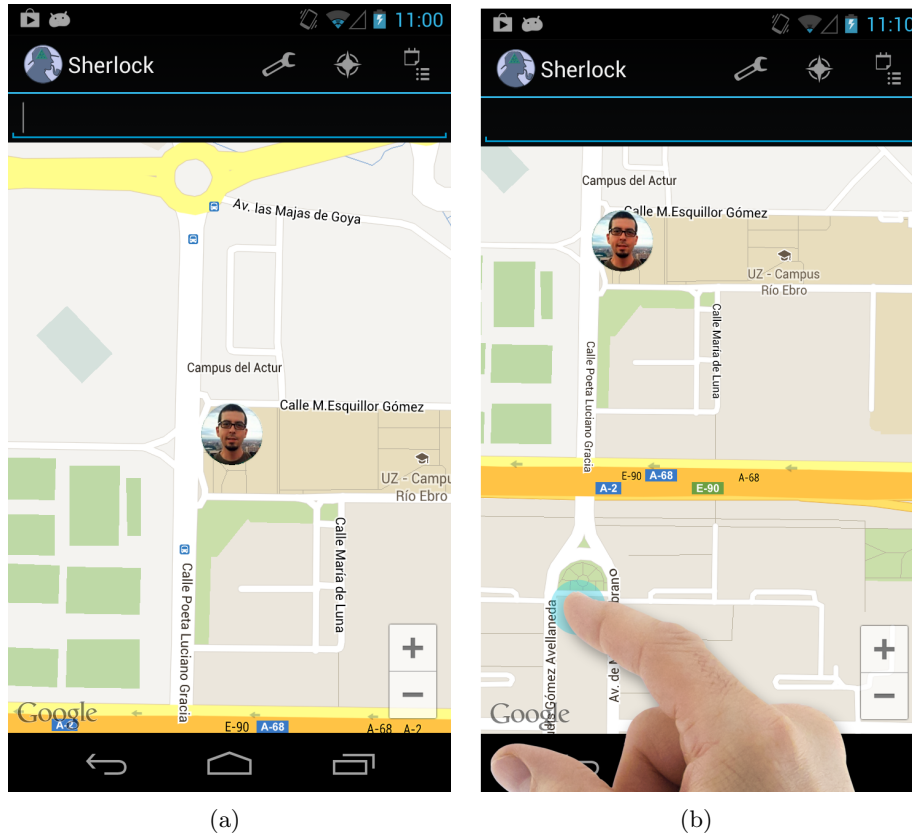


Figure B.6: SHERLOCK app: Map interface.

information about the user to impose constraints on the results to obtain (notice that some of this information is directly inferred by SHERLOCK). With this information SHERLOCK is able to reason what are the most appropriate providers for the service selected according to our needs. In our example, SHERLOCK asks the user about some information for the Transport service (see Figure B.7(b)). The user can select also which of these parameters are mandatory. A mandatory parameter means that it is important for the user that this condition is satisfied by the result. For example, the user could select that it is mandatory that the transport is “door to door” and then SHERLOCK will provide her with a more accurate response. After filling in the values we continue by tapping the “GO” button that launches the user request to obtain transports that match our preferences.

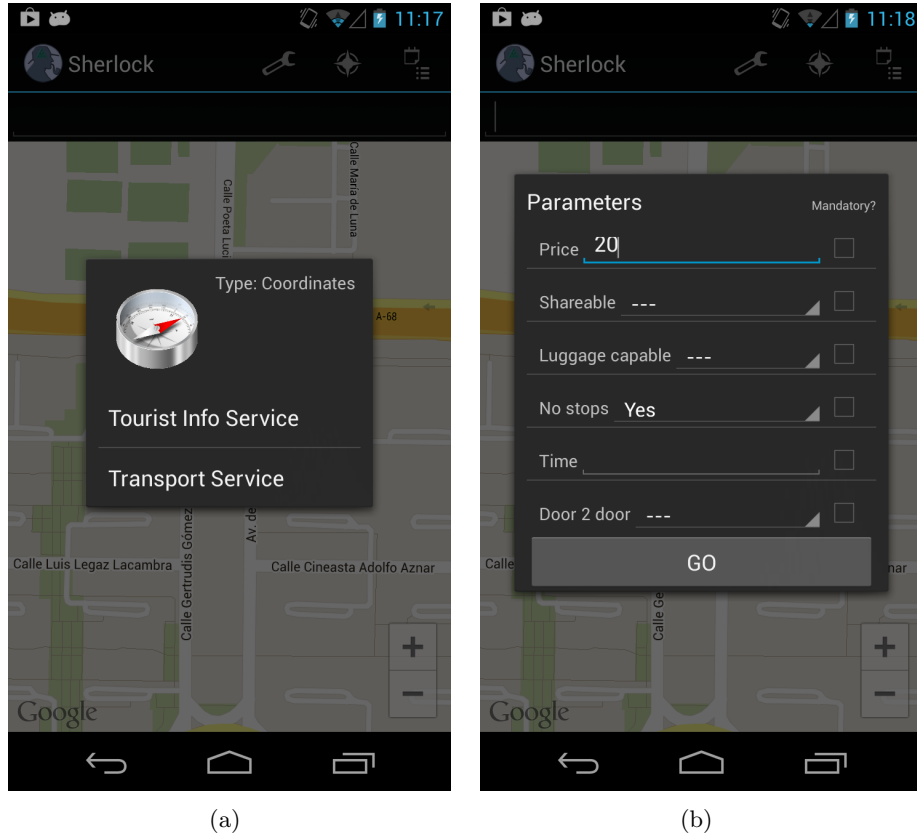


Figure B.7: SHERLOCK app: Selecting a service and its parameters.

Every user request may return results which will be showed on the map as objects whose location will be updated in real-time (in the case of continuous requests like the one selected). In our current case, we can see several objects moving around our map (see Figure B.1.3) that represent the different transports nearby (taxis, shuttles, and buses). These objects have been generated by the simulator included in the SHERLOCK Android app. The simulator creates different objects and move them around so they can be part of results for user requests in our prototype. The current prototype gets information from other devices within our same network (in this version the same WiFi network) but also can connect to third-party external services if the source and communication method has been defined in its local ontology. This is the case now, so we can also see objects representing bus stops whose info

have been retrieved from a local transport info web service that SHERLOCK knows (as someone has shared with it this knowledge). Last, notice that each result has a green circle attached, this means that this is a provider which best matches our search criteria. For other providers that could be interesting but do not match all our preferences SHERLOCK uses red circles.

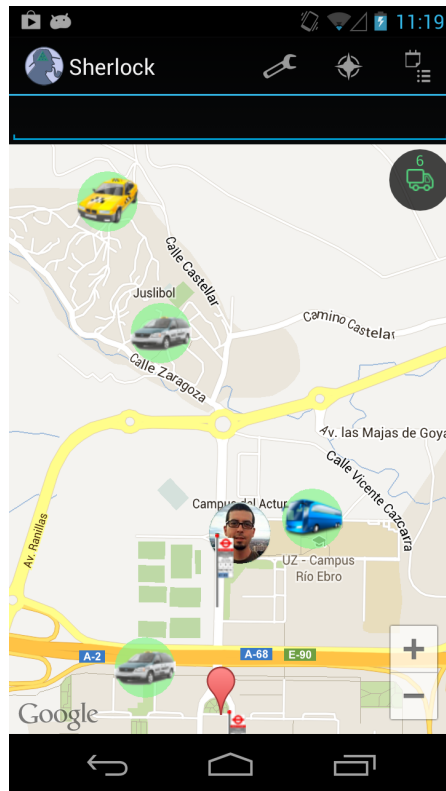


Figure B.8: SHERLOCK app: Displaying results.

At SHERLOCK you can interact with every object you watch on the screen by tapping on it getting information related to that object. When the user taps on an object, SHERLOCK reasons what kind of services are related to this specific object and could be interesting for the user. It is a similar process to tapping on a point on the map. For example, we tap on a bus stop (see Figure B.9(a)) and SHERLOCK obtains that the service that provides information about the buses that arrive to the stop is interesting (“Bus Stop Info Service”). Also, we change our profile to researcher and tap on the object representing us on the map. Then, SHERLOCK offers us a new service to

“Search Researchers”. We select this service and stop the simulator so only other real devices that belong to researchers will be part of the answer.

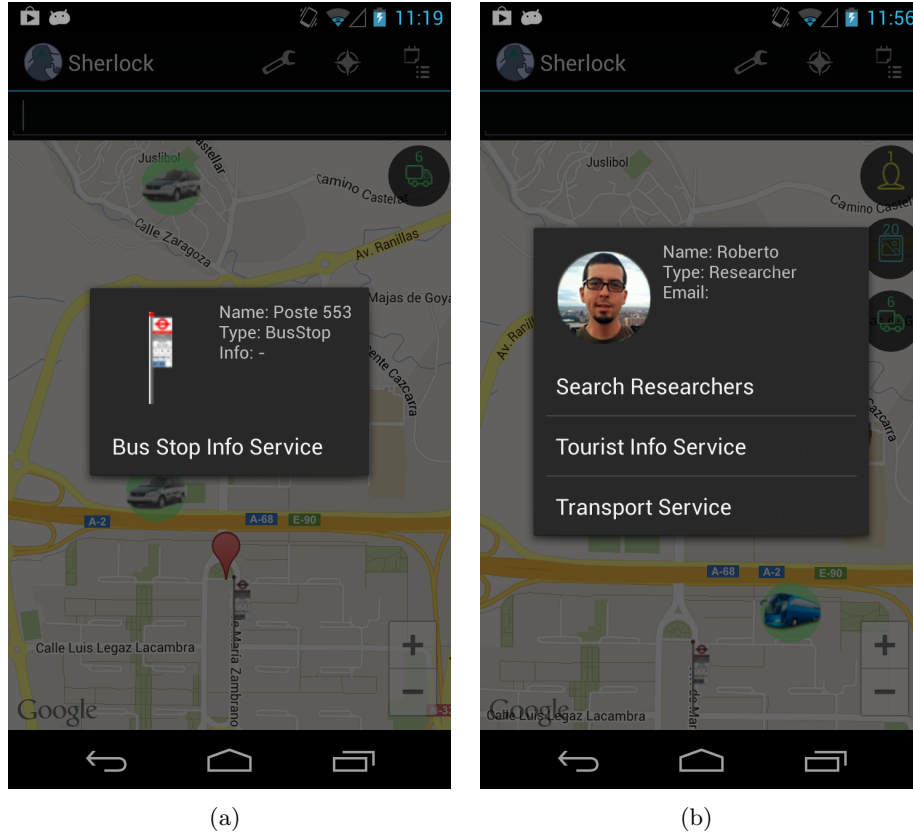


Figure B.9: SHERLOCK app: Interacting with objects.

At the moment, this is the only SHERLOCK device in the network so we do not obtain results for the executed service. Then, we connect another SHERLOCK device to the same network and set its profile to researcher. After that, we return to the device where we executed the service to find researchers and we can see the location of the second device on the map (see Figure B.10(a)). A researcher is also an object so we can tap on it to see a list of available services related to him (see Figure B.10(b)). In this case the “Send email service” has been defined in the local ontology on the device and it is retrieved by SHERLOCK.

SHERLOCK is able to process several request at the same time even if they

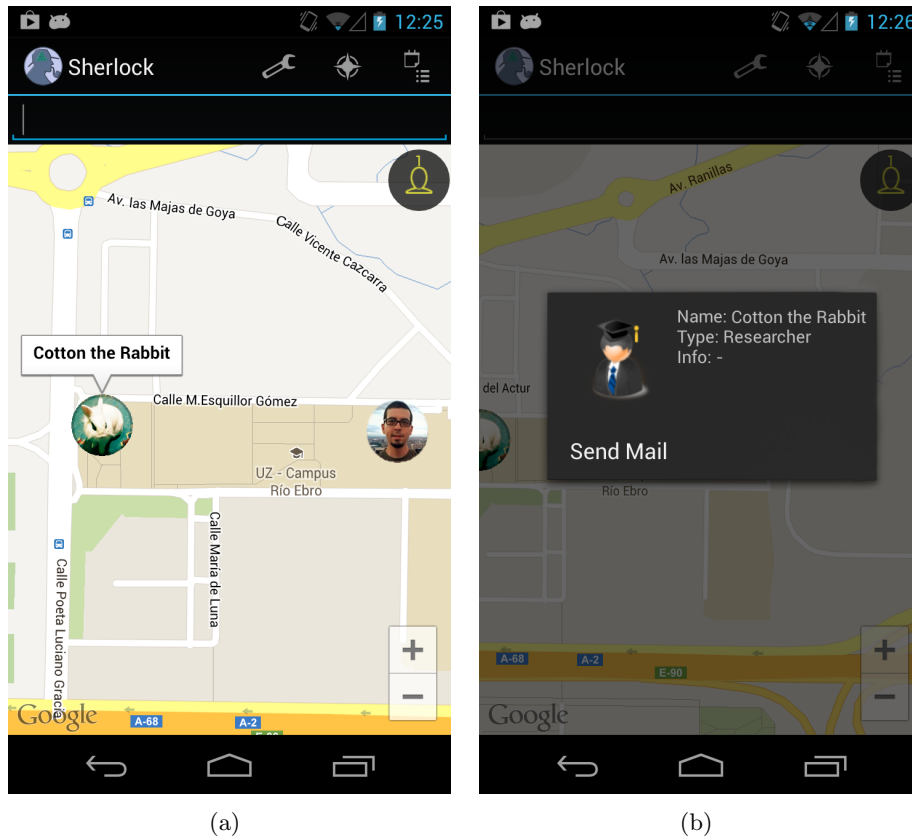


Figure B.10: SHERLOCK app: Interacting with other users.

are continuous requests. Notice that every new request is represented by a transparent black circle at the right top corner of the screen (see Figure B.11(a) where three requests are being processes at the same time). Requests are represented by: a color and an icon according to the kind of request, and a number that shows the amount of results currently returned (for requests which are performed continuously the number of results may change over time). We can access the request dialog, which enables to manage each of the requests, by tapping on any of the request circles or the right icon on the application bar. The request dialog (see Figure B.11(b)) enables us to interact with our current request. The available options (from left to right) are:

- Highlight the results: All the objects returned as a result of this request will be highlighted on the map.

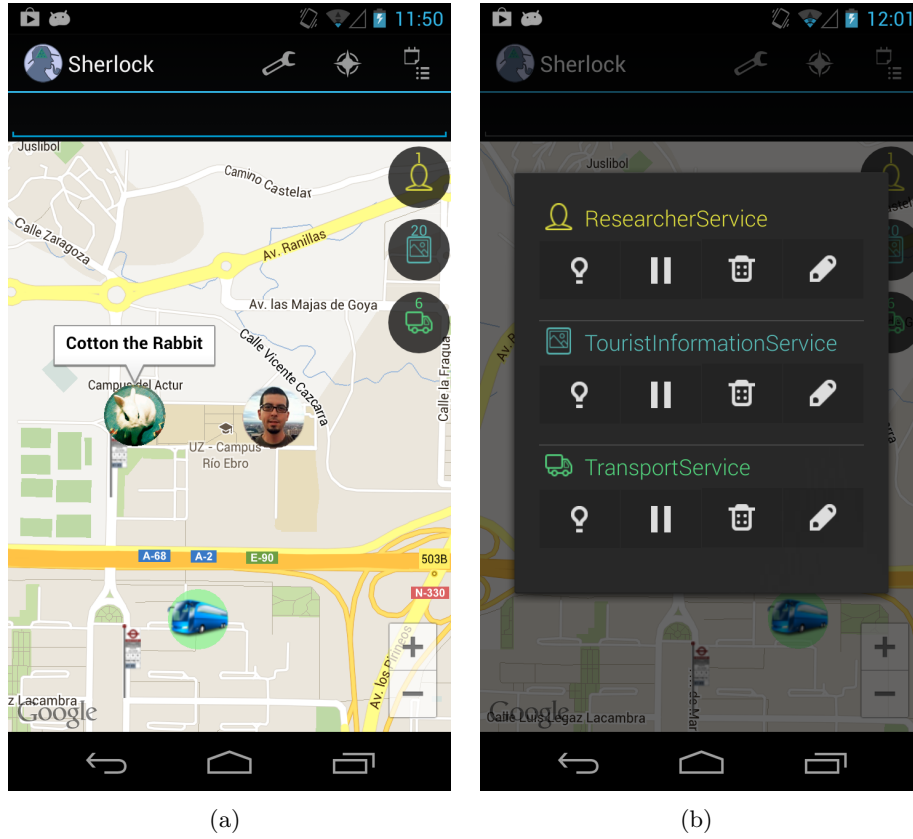


Figure B.11: SHERLOCK app: Handling multiple requests.

- Pause/resume: The results will stop being updated.
- Delete: Stops processing the request.
- Edit: Enables us to edit the parameters of the request.

With this test we have shown how the system would behave when used by a user on her smartphone. We have focused on the interaction with the system, the selection of services, and provision of parameters. Also, we showed how other real devices (in this case a tablet with the same prototype installed) can be part of the results returned to the user. In the next sections we will explain other standalone systems and prototypes developed to test our contributions to the problems tackled in the SHERLOCK architecture.

B.2 DUCK: Exchange and Integration of Knowledge in Wireless Environments

In this section we present the prototype of the SHERLOCK module in charge of the integration of shared ontologies in mobile scenarios. The prototype, which we have called DUCK (Deduction of Undefined Correlations in Knowledge) implements the mechanisms presented in this thesis to enable mobile devices to exchange knowledge and discover subsumption relationships between concepts from their local ontologies and the ontologies received.

B.2.1 Architecture of DUCK

The DUCK prototype enables two smartphones to exchange their local ontologies, discover subsumption relationships between their concepts, and generate an integrated ontology with the information discovered. In the prototype we implemented a simple exchange and integration procedure that assigns the extraction of subsumption relationships and alignment of the ontologies to the most powerful device (see Figure B.12). As commented in Chapter 10, we are planning to implement and test other more sophisticated procedures in the future. Thus, the current prototype performs the following steps:

1. *Device discovery*: A mobile device uses its communication mechanism (e.g., WiFi, Bluetooth, etc.) in order to discover other devices willing to exchange knowledge.
2. *Communication establishment*: The discovered devices establish a communication channel between them and each one sends information about their features (i.e., processor and its current load, memory and battery available) to the other one.
3. *Device information comparison*: The features from each device are compared in order to decide which one should perform the integration.
4. *Knowledge exchange*: Each device sends its active ontology to the other.
5. *Integration process*: One of the devices performs the integration and shares the result with the other.

Notice that we implemented a recovery mechanism in case the connection between the devices stops before the integrated result can be shared. So, the less powerful device, which is waiting to receive the integrated ontology, sets

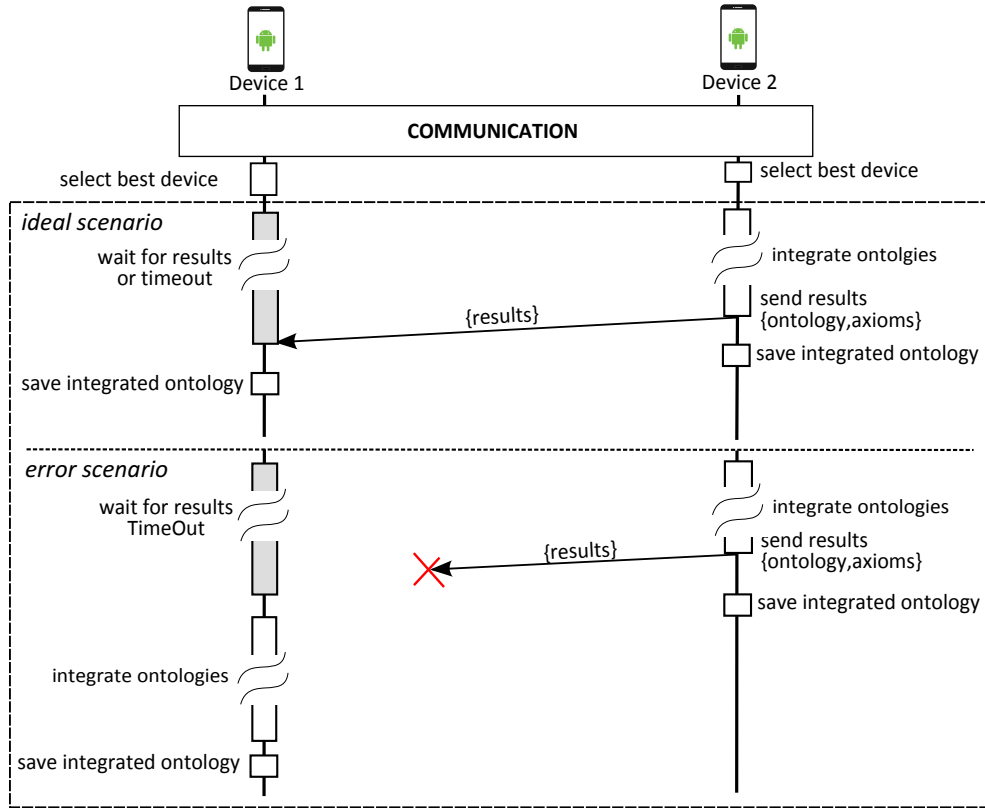


Figure B.12: Sequence diagram of the exchange and integration process in the DUCK prototype.

a timeout. In case that the connection is disconnected or the device which is performing the integration process takes too long to transfer the results, the device which is waiting will begin to integrate the ontologies after the timeout. Also, the steps for the integration process of the prototype are the ones shown in Figure 5.4.

B.2.2 Prototype of DUCK

We developed two applications to implement the DUCK prototype:

1. An Android app to implement the sharing of ontologies among devices and the integration of the information through the discovery of subsumption relationships.

2. A PC Java implementation of our subsumption relationships discovering technique. We developed the PC application as we planned to perform experiments with ontologies from standard datasets used in ontology alignment which would require too much processing time on mobile devices due to their limited capabilities.

In both prototypes we used the semantic DL reasoner Pellet [SPCGKK07] for the extraction of the implicit roles of each concept and the technique presented by Gracia et al. [GA13] to obtain a similarity measure among the roles. Also, we used Wordnik⁴, an on-line dictionary and language resource based on several sources (among them, *WordNet* [Mil95]), for the label analysis. Regarding the weights used in the different formulas of our approach (see Section 5.2.3), we assigned a higher value to: the role set analysis ($w_l = 0.15$, $w_r = 0.8$, and $w_{ch} = 0.05$, Formula 5.3); the information extracted from Wordnik (0.75); and to the roles shared with respect to the non-shared ($w_{sh} = 0.9$ and $w_{diff} = 0.1$, Formula 5.5). Also, we assigned $a = 1.5$ to model the slope of the logistic function of Formula 5.4. For the filtering of the subsumption relationships discovered we used the simple unsupervised k-means algorithm [Mac67] and a conservative approach to promote precision over recall (medium and low subsumption degree clusters are discarded). Nevertheless, these specific values, that were obtained after performing some initial tests, have been used to implement our generic rules to capture the existence of a subsumption relationship. Therefore, these weights and even the formulas could be modified, respecting the essence of the rules, for other ontology sets or scenarios.

On the one hand, the PC application gets the two source ontologies as a command line parameter and writes a file with the subsumption relationships discovered and their subsumption degree. This applications is not meant to be used by users and it has been developed to enable us to perform scripted tests with different ontologies in an easy way. On the other hand, the Android app enables a user to select some options through a user-friendly Graphical User Interface “GUI” that is also used to show the results to the user (see Figure B.13). The GUI is composed of three views:

1. *Main menu*: Shows some parameters which can be adjusted by the user such as:
 - A drop-down list that contains the ontologies on the device among

⁴<https://www.wordnik.com>

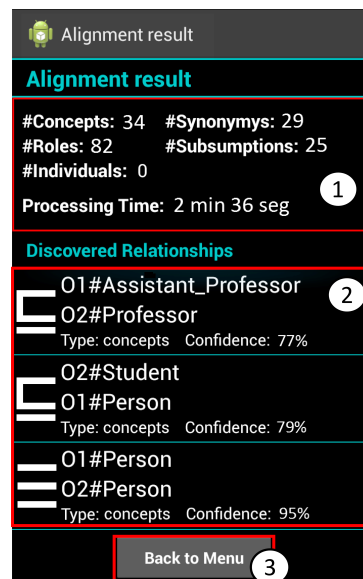
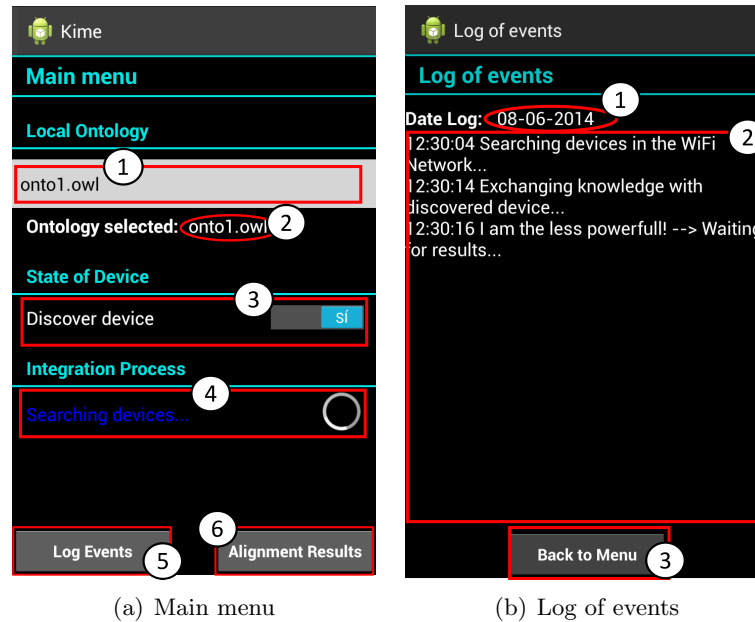


Figure B.13: DUCK app: Screenshots of the prototype.

which the user can select one to be shared with other devices ((1) in Figure B.13(a)).

- A switch to control the visibility of the device for other devices in the same network. When the device is made visible to others the process to search other devices and exchange knowledge with them starts ((3) in Figure B.13(a)).
2. *Log of events*: A dynamic log which registers each of the events occurring during the execution of the exchange and integration process.
 3. *Alignment results*: Shows the statistical results of the integration (including the number of concepts, roles, and individuals of the final integrated ontology, and the number of synonymy and subsumption relationships discovered) and a detailed list of each discovered relationship.

In the following we explain the experiments performed to evaluate our approach for the discovery of subsumption relationships with the previous prototype and different ontologies. We focused in testing the discovery of subsumption relationships and compared our approach with others in the literature whenever possible. As commented before, as future work we are working on developing an approach to efficiently perform the exchange and integration on mobile devices and therefore, the following experiments are performed with the PC prototype.

B.2.3 Evaluating the Extraction of Subsumption Relations

To evaluate our approach to discover subsumption relationships between concepts from different ontologies we carried out several preliminary tests⁵ using the PC prototype explained before.

Experimental Setup

We first tested our prototype with two well-defined ontologies as an example of an “ideal” scenario. Then, we tested some ontologies from the OAEI 2009 dataset. Finally, some experiments were performed with challenging real-world ontologies extracted from the Web. The results of these tests, in terms of precision and recall of the extracted subsumption relationships, can be found in Table B.1. Notice that, the comparison with other related works such as

⁵More information about the experiments and the ontologies used can be found at <http://sid.cps.unizar.es/SubsumptionExtraction>

MOMIS [SdM08] and SCARLET [BBCCGMMV00] is not possible as they did not publish enough information about their experimental evaluation. However, we do compare our results with CSR [SVV08], whenever possible.

Test	#relations	Precision	Recall	F-measure
O_1-O_2	26	0.96	0.89	0.93
101-222	32	0.96	0.73	0.83
101-223	82	0.65	0.41	0.50
101-223'	84	0.72	0.63	0.67
101-304	78	0.38	0.47	0.42
101-304'	48	0.83	0.44	0.58
univCs-univBench	74	0.77	0.63	0.69
confOf-sigkdd	24	0.58	0.67	0.62
confOf-sigkdd'	21	0.67	0.67	0.67
confOf-conference	41	0.61	0.58	0.60
confOf-conference'	27	0.74	0.47	0.57

Table B.1: Precision and recall of our prototype for different ontologies.

Testing the Running Scenario

First we tested the two well-defined ontologies presented in Section 2.1.1 which include from two to thirteen roles for each concept. The approach shows a good performance in terms of precision and recall for this test (see O_1-O_2 in Table B.1). Regarding precision, only one wrong subsumption is obtained ($Report \sqsubseteq Publication$) with a degree of 0.52, which is slightly above the dynamic threshold calculated (0.5). In fact, *Report* and *Publication* are co-hyponyms but they share most of their roles. Regarding recall, three relationships are not discovered by the system. For example, despite the calculated confidence for $Publication \sqsubseteq Piece_of_Work$ being above the threshold, it is discarded because *Article*, a descendant of *Publication*, is subsumed by *Piece_of_Work* with a slightly higher confidence.

Testing OAEI Ontologies

The Ontology Alignment Evaluation Initiative (OAEI)⁶ publishes datasets designed to test ontology alignment systems. However, these datasets are mainly designed to test the discovery of synonymy relationships. An “oriented alignment” track was presented in the 2009 edition of the OAEI competition⁷ to test systems to discover subsumption relationships⁸. Four systems participated in the competition but, except for CSR [SVV08], they based their subsumption extraction on the extraction of synonyms [EFHIJMMNPS+09]. For the second test, we used this dataset and compared our results with CSR.

The results for test *101-222* (see Table B.1) are promising with a precision of 0.96 and a recall of 0.73. In terms of precision this result is similar to the average of the tests performed by CSR (0.97) but in terms of recall CSR obtains better results (0.97). For the test *101-223* the performance of the system decreases as the domain of many roles have been incorrectly defined at the top of the hierarchy and this makes the concepts closely resemble each other. Trying to improve the results we selected a manual threshold increasing the precision up to 0.72 and the recall up to 0.63 (see *101-223'* in Table B.1). CSR also obtained a lower precision and recall for this second test (0.84 and 0.78). For the third test, *101-304*, our system obtains the lowest precision compared with the rest (0.38) whereas CSR obtained a precision of 0.66 and a recall of 0.72. This is due to an incorrect definition of the roles in one of the ontologies as *object properties* (e.g., *date* and *ISBN*) while their equivalents in the other ontology are *data properties*. In DL, object properties cannot be equivalent to a data properties and so, the approach used in our prototype to compute the similarity of roles does not consider this situation. After correcting the source ontology we obtained an improved precision of 0.83 and a recall of 0.44 (see *101-304'* in Table B.1).

Testing Ontologies from the Web

For the last test, we selected the most challenging scenario for our approach, real-world ontologies with a very poor definition of their concepts (e.g., with very few roles): a subset of ontologies from the OAEI benchmark, *confOf*, *Conference*, and *sigkdd*, and two ontologies from universities obtained using

⁶<http://oaei.ontologymatching.org>

⁷<http://oaei.ontologymatching.org/2009/oriented>

⁸Notice that the 2009 competition has been the only edition with an oriented alignment track, as of 2014.

the Semantic Web search engine Swoogle [DFJPCPRDS04], *univ-cs*⁹ and *univ-bench*¹⁰. In these ontologies, the subsumption extraction is heavily based on the label of the concepts. In general, the subsumption degree obtained for all the relationships is similar and the clustering technique for the dynamic threshold is slightly inaccurate in this scenario. We manually selected a threshold which improved the precision of *confOf-sigkdd* and *confOf-conference* up to 0.67 and 0.74, respectively. The tests *confOf-sigkdd* and *confOf-conference* were also very challenging for CSR which achieved a precision of 0.08 and 0.47, and a recall of 0.31 and 0.29, respectively. As mentioned in [SVK10], the decrease of performance of CSR is explained because these ontologies are not suitable to do the data training phase. Notice that a direct comparison of our results and CSR results for these two tests is not possible as the Gold Standard they used is not available and so, it could be slightly different to ours. Nevertheless, the performance of our system for these ontologies can be explained due to the combination of external sources of information with the ontological context of concepts (without the need of training data).

Discussion on Results

As a summary, these experiments have shown that our approach would be able to achieve good results with well-defined ontologies (with an average F-measure of 0.88 for the tested ontologies), and promising results with ontologies that caused problems to other systems (average F-measure of 0.64). Compared with CSR, a state-of-the-art system, our prototype obtained similar results in most of the tests (sometimes slightly better or worse) and better results in ontologies that are not suitable for their training phase. We have also detected two main problems with our prototype:

1. Some roles do not have a domain defined, so *Thing* becomes their domain and all the concepts of the ontology inherit them. Therefore, these roles are not characteristic enough and do not help much to discover possible subsumptions.
2. The clustering algorithm used in our prototype to obtain the dynamic filtering threshold, *k-means*, is good enough in most situations but sometimes is slightly inaccurate. This happens, especially, in the situation mentioned above where the subsumption degree obtained for each pair of concepts is similar.

⁹<http://www.cs.toronto.edu/semanticweb/maponto/MapontoExamples/univ-cs.owl>

¹⁰<http://swat.cse.lehigh.edu/onto/univ-bench.owl>

Even in extreme scenarios for our approach where the first problem occurs or not many roles are defined, our prototype obtained fairly good results in our tests with an average F-measure of 0.65. Take into account that in these situations even experts would have problems to discover relationships manually. While trying to fix the input ontologies is out of the scope of our approach (e.g., if someone defines the property *hasWheels* without a domain, according to that ontology a car, a person, a tree, and even a wheel can have wheels), it could be interesting to study the application of a preprocessing phase to refine the domain and range of roles [TKS12]. Regarding the second issue, the use of a more sophisticated clustering algorithm would help to improve the filtering phase.

Nevertheless, we want to highlight that the specific values achieved, although good, are not the most important part as they depend on the ontologies considered and the specific values and weights used in our prototype. The most important conclusion of these experiments is that we believe that our approach looks promising for being the first one to discover subsumption relationships using generic rules that capture the existence of such relationships.

B.3 Triveni: Shared, Semantic Context Models for Mobile Devices

In this section we present the prototype developed to test our approach to the enrichment of the information about the context of the user. The prototype, which we called Triveni, implements the mechanisms presented in this thesis to enable mobile devices to exchange their context information, create a shared semantic context model, and use this information to enrich (i.e., correct or increase) the information about the context of each user that their devices have.

B.3.1 Architecture of Triveni

The primary goal of Triveni is to enrich the information about a user's context, obtained by context synthesizers, by leveraging the context of other users nearby. This way, applications would be able to make use of the enriched context provided by the system. Triveni has a decentralized architecture where mobile devices communicate among themselves using wireless ad hoc networks and exchange their context (see Figure B.14 for the high-level architecture of each Triveni node). Therefore, Triveni:

1. Obtains the context information from the available *Context Synthesizer(s)* using the *Context Manager* module.
2. Communicates with devices discovered in the vicinity using the *Communication* module and shares context information with them.
3. Integrates the context information collected to generate the shared context model using the *Integration* module.
4. Verifies the information integrated in order to detect and resolve inconsistencies by the *Inconsistency Resolving* module.

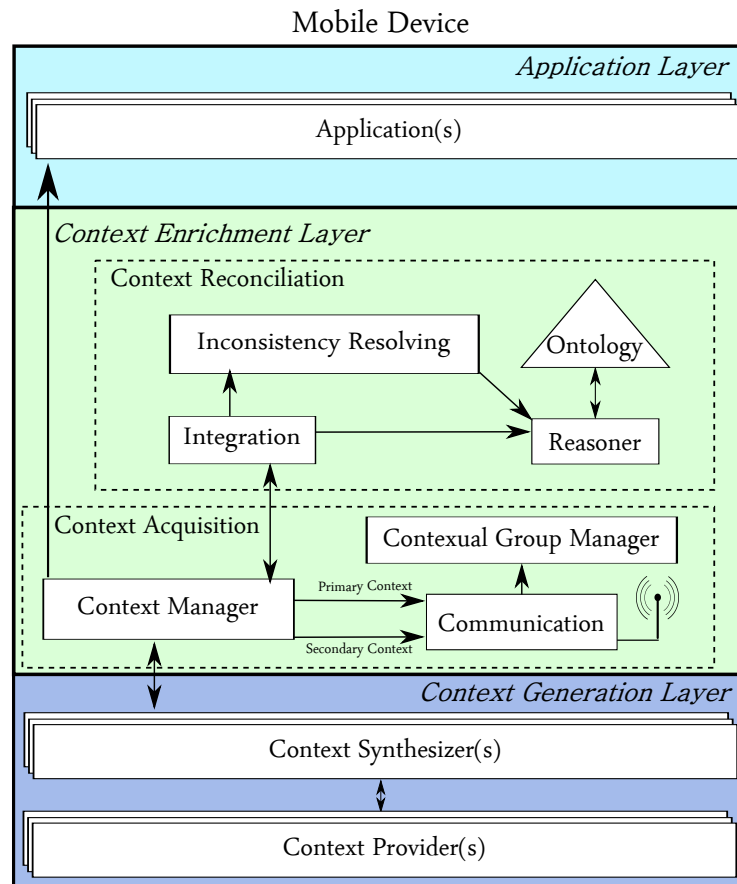


Figure B.14: High-level architecture of Triveni.

Triveni uses Semantic Web technologies (specifically ontologies and a semantic DL reasoner) for context modeling. This way, Triveni is able to detect inconsistent information and infer not explicitly defined facts by using the reasoner, a program that infers logical consequences from a set of asserted facts or axioms [DCTK11].

B.3.2 Prototype of Triveni

To test our approach we developed a prototype of the system¹¹ as an Android app where users can enrich their context information with the information of other users nearby by using Bluetooth communications. For managing the ontology, used in the enrichment process, on the device, we used the OWL API and the HermiT semantic DL reasoner. To simulate context scenarios we have used a Graphical User Interface (GUI) that enables us to select the current context of the device which mimics context synthesizers used to obtain high-level context of each mobile device (see some screenshots of the GUI in Figure B.15). This enables us to easily recreate different situations some of which involve conflicting information being shared. This GUI also enables us to modify the confidence in the correctness of the context information shared. Additionally, to test the scalability of our approach, we also developed a web service REST API to simulate multiple mobile devices sharing their context.

B.3.3 Evaluating the Extraction of a Shared Context

We tested our approach to extract a shared context model by using the Triveni prototype presented before.

Experimental Setup

For our experiments we recreated a scenario involving a study group in a room where there are four people, but only three of them are being part of the meeting (see Figure B.16). Notice that these mobile devices are equipped with different number of sensors and the information that users provide about their schedule is also different and in varying detail. In our specific scenario, Annie's and Abed's calendar entries give information about meeting scheduled for that day (e.g., duration, topic, participants, etc.) and Jeff's smartphone just used Foursquare to check in *Study room F*. With the information available on their devices, traditional context generation systems will create different high-level

¹¹See <http://sid.cps.unizar.es/ContextEnrichment> for more information about the prototype and tests.

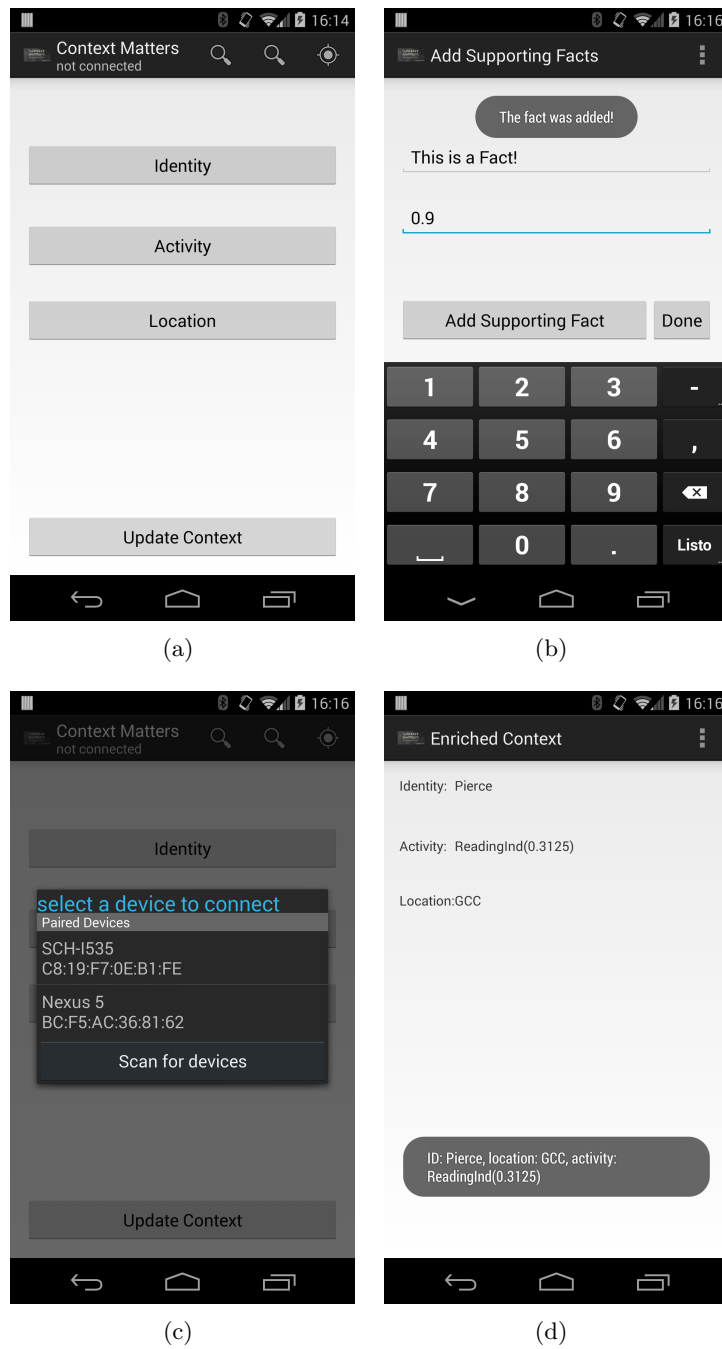


Figure B.15: Triveni app: Screenshots of the prototype.

context information for each device (see Figure B.16 where the current context of each user is shown in blue boxes). Notice that some of the contexts obtained by the devices are wrong. For example, Annie disabled the location gathering mechanism of her tablet, while at home, to save battery and so, her device thinks that the location is still “home”. Summarizing, the devices have some information about the context but most of them are not as rich in detail as it would be desired.

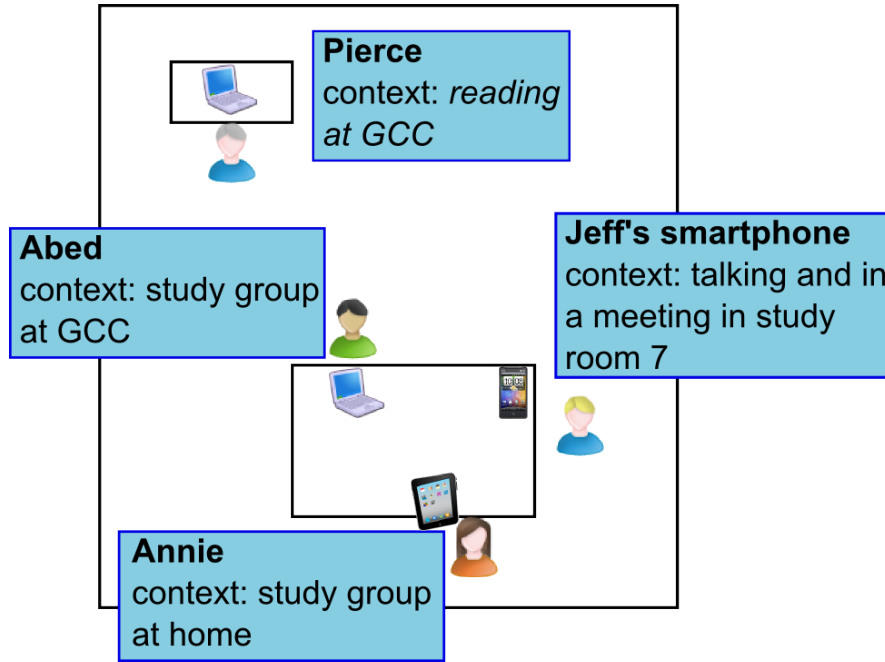


Figure B.16: Motivating use case for Triveni: Users being part of a study group.

As in the use case, we use four real smart phones to replicate the study room scenario: two *Google Nexus 5* (Quad core, 2260 MHz, 2048MB RAM, Android 4.4), one *Google Nexus 4* (Quad core, 1500 MHz, 2048MB RAM, Android 4.4), and one *Samsung Galaxy SIII* (Quad core, 1500 MHz, 2048MB RAM, Android 4.3). We considered the following primary and secondary context pieces for this scenario:

- $location = \{Greendale, GCC, Study_room_F, Home\}$
- $temperature\ of\ location = \{25^{\circ}C, 28^{\circ}C\}$

- *light of location* = {*lights on*}
- *activity* = {*Study_group*, *Meeting*, *Reading*}
- *topic of activity* = {*Spanish*}
- *duration of activity* = {*1h*, *2h*}

For the purpose of this experiment, we have focused on Annie’s device (in our tests, a Nexus 5) and generated every possible scenario for it by selecting either one or none of the values from each of primary and secondary context information (giving us a total of 720 scenarios). In 528 such scenarios generated, the context of the device included at least one incorrect piece of context (e.g., the location is set to “Home”); while in 191 other scenarios, which are correct, had at least one missing context piece (e.g., the location is “Greendale” and it should be “Study room F”); the remaining scenario has the most enriched context with no incorrect or missing pieces (i.e., *study group of Spanish with a duration of one hour at the Study Room F at 25° C and with the lights on*). For other three devices we randomly selected the context pieces that will be shared so as to minimize the redundancy in context scenarios. In addition, we also assigned a random confidence to each one of the context pieces shared.

Measuring Precision and Recall

For our experiment we have modeled the output of Triveni as a binary classification problem with correct and incorrect context pieces considered positive and negative. On the one hand, true positives are positive pieces which are included as part of the output of Triveni and false positives refer to negative pieces being included in final context. On the other hand, true negatives correspond to negative pieces correctly ignored by Triveni and false negatives refer to positive pieces incorrectly ignored by our system. Based on these definitions and the context scenarios mentioned before we computed the precision and recall of the context obtained by Triveni for the user by taking into account the three different approaches for integration explained in Section 5.1.1: conservative, optimistic, and semi-optimistic.

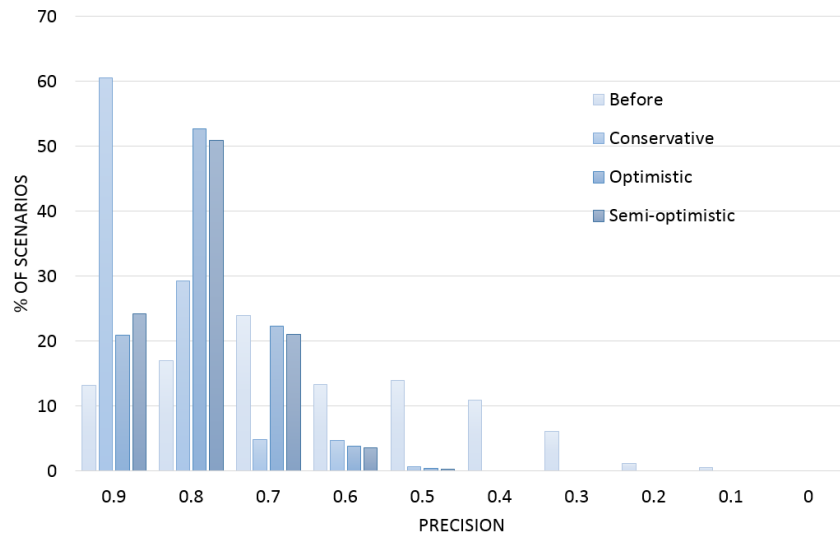
Figure B.17 shows the results obtained for the experiments which also includes the precision and recall based on the context synthesizer and before Triveni generated the enriched context. As expected the percentage of context scenarios with high precision is greater in a conservative approach than in an optimistic one (e.g., the precision was higher than 0.9 in 60% of the scenarios for the conservative approach and in 21% for the optimistic). This is because

the conservative approach only selects the most correct piece of context during integration. On the other hand, the recall is higher for optimistic approach as it considers all non-conflicting context pieces while enriching the context (e.g., the recall was higher than 0.9 in 0% of the scenarios for the conservative approach and in 25% for the optimistic). In the semi-optimistic method, we experimented with several different thresholds and finally we selected 0.15 which improves the precision of the optimistic approach in exchange of a slight decrease on the recall. While in 16% of the scenarios the semi-optimistic approach introduced some incorrect information (e.g., incorrectly added that the duration of the study group was two hours), overall it enriched the context of the user in 97% of the scenarios and corrected the context in 77%.

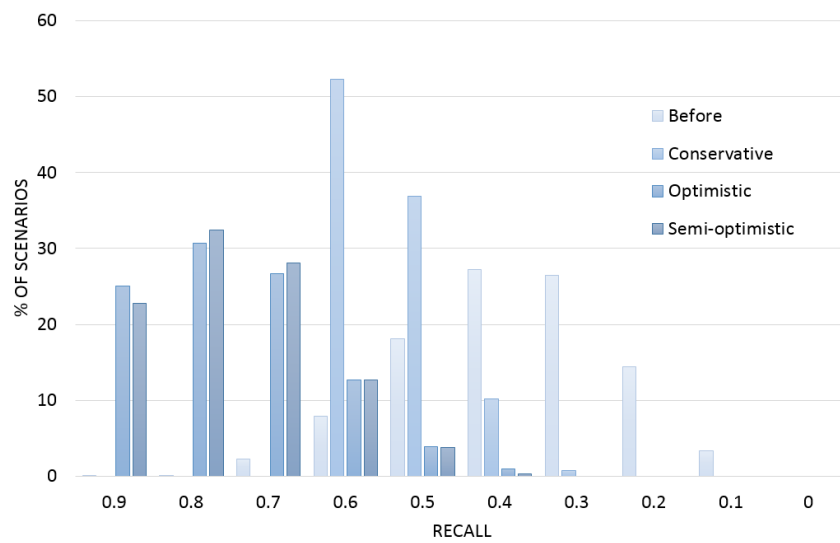
As an example of the experiments performed we show here three scenarios of context creation, enrichment, and correction. In the context creation step, we selected one of the scenarios where Annie’s device was devoid of any context information and after receiving same primary context pieces with different confidence values from other devices it learns about the context (see *Test1* in Table B.2). For an example of context enrichment scenario, all the devices had some information about the location and activity but with different confidence scores (see *Test2* in Table B.2). In this case, Annie’s location and activity were enriched. For the context correction step, conflicting information was shared (see *Test3* in Table B.2). The test device started with an incorrect location “Home” and an incorrect activity “reading”. After integrating the context from other devices, an inconsistency was detected and “Home” was removed. Notice that the activity “reading” is not inconsistent with “meeting” or “studyGroup” but the confidence obtained was lower than the threshold of the semi-optimistic integration and thus removed.

Measuring the Performance

Finally we ran some experiments that would allow us to test the scalability of our approach for context enrichment with respect to the execution time and memory consumption (two important parameters for mobile devices). For this experiment we used the four real mobile devices and the web service to simulate the presence of other devices sharing their context. We increased the number of simulated devices up to 512 and for each one of them we randomly generated four context pieces that they will share. Measuring computation time on Android presents some problems due to the variance of the time obtained due to the management of apps performed by the operative system. So, we ran every test five times and computed the average time and memory.



(a)



(b)

Figure B.17: Precision and recall of the context obtained for a user in different scenarios before and after using the three approaches of Triveni (conservative, optimistic, and semi-optimistic).

Device	Test1	
	Location	Activity
Nexus 5	<i>GCC(0.5)</i>	<i>studyGroup(0.6)</i>
Nexus 4	<i>GCC(0.3)</i>	<i>studyGroup(0.7)</i>
Galaxy S3	<i>GCC(0.6)</i>	<i>studyGroup(0.7)</i>
Nexus 5(start)	-	-
Nexus 5(end)	<i>GCC(1)</i>	<i>studyGroup(1)</i>

Device	Test2	
	Location	Activity
Nexus 5	<i>GCC(0.8)</i>	<i>studyGroup(0.8)</i>
Nexus 4	<i>Greendale(0.7)</i>	<i>studyGroup(0.7)</i>
Galaxy S3	<i>StudyRoomF(0.7)</i>	<i>meeting(0.4)</i>
Nexus 5(start)	<i>GCC(0.5)</i>	<i>meeting(0.6)</i>
Nexus 5(end)	<i>Greendale(1), GCC(0.74), StudyRoomF(0.26)</i>	<i>meeting(1) studyGroup(0.6)</i>

Device	Test3	
	Location	Activity
Nexus 5	<i>GCC(0.8)</i>	<i>meeting(0.9)</i>
Nexus 4	<i>Greendale(0.7)</i>	<i>studyGroup(0.7)</i>
Galaxy S3	<i>StudyRoomF(0.7)</i>	<i>meeting(0.7)</i>
Nexus 5(start)	<i>Home(0.9)</i>	<i>reading(0.4)</i>
Nexus 5(end)	<i>Greendale(1), GCC(0.48)</i>	<i>meeting(0.85), studyGroup(0.26)</i>

Table B.2: Results for the tests performed: context creation with consistent information (Test1), context enrichment with consistent information (Test2), context correction with inconsistent information (Test3).

We show in Figure B.18 the results obtained for the computation time when increasing the number of devices (notice that the time axis follows a logarithmic scale). The graph shows the average of the computation time of the four smartphones. The standard deviation was situated under 10% up to 16 devices (less than 0.1 seconds), and under 20% for the rest (with the increase of the simulated devices the Nexus 5 over performed the other smartphones). We send small strings of information from one device to another using Bluetooth and did not observe any relevant communication delays. Therefore we have not included communication time in the graph as we believe that the delays in short range Bluetooth communication (around 10 meters) would be insignificant. In addition, while our system continuously enriches the context of a device, the

set up of the Bluetooth network will be usually performed just once for most of the scenarios. We wish to highlight that the computation time remained around 1.5 seconds when 32 devices were used. Given the restrictions of Bluetooth communications (a device can communicate up to seven active devices), 32 devices is probably enough for most scenarios. These results also show that the system could enrich the context of users continuously every second in highly dynamic situations. Regarding the memory usage (see Figure B.18), it is maintained in the range of 10MB to 20MB. This includes the memory allocated for the ontology and the semantic reasoner. Taking into account that current mobile devices have at least 1GB of RAM (current Android versions usually provide apps with a maximum heap size of 256MB), our approach does not overload the device even in extreme situations. To summarize, Triveni maintains computation time and memory usage on a mobile device within a practically acceptable rate even when the ontology and the reasoner runs on the device.

B.4 MultiCAMBA: Multi-CAMera Broadcasting Assistant

In this section we present the prototype developed to evaluate the SHERLOCK module in charge of managing camera views. The prototype, called *Multi-CAMBA* (*Multi-CAMera Broadcasting Assistant*), enables the user to define the camera view she is interested in and it checks the views of the different cameras available to obtain those that can fulfill her requirements.

B.4.1 Architecture of MultiCAMBA

Figure B.19 shows the high-level architecture of the MultiCAMBA prototype. The prototype is based on the following steps:

1. *3D model management.* The 3D model of the scenario is an essential part of our system as it stores the information about the different objects and cameras in the scenario. The system maintains this 3D model updated in parallel to the processing of user queries.
2. *Obtaining the user requirements.* The requirements of the user are captured through an easy-to-use interface. The prototype provides three mechanisms for that: a) the TD uses the 3D interface to define the scene she is interested in, b) the TD selects from lists the target object/s that

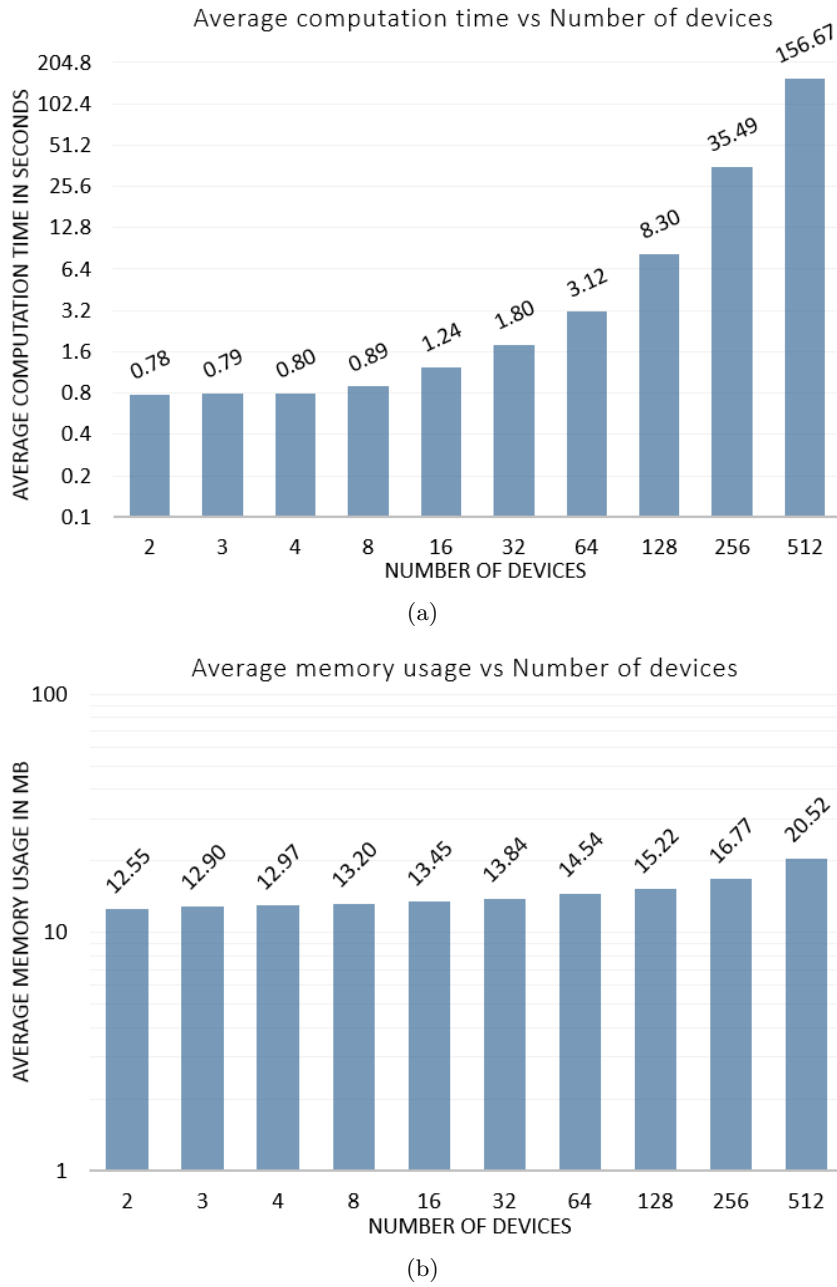


Figure B.18: Computation time and memory usage on a Triveni node with increasing number of devices.

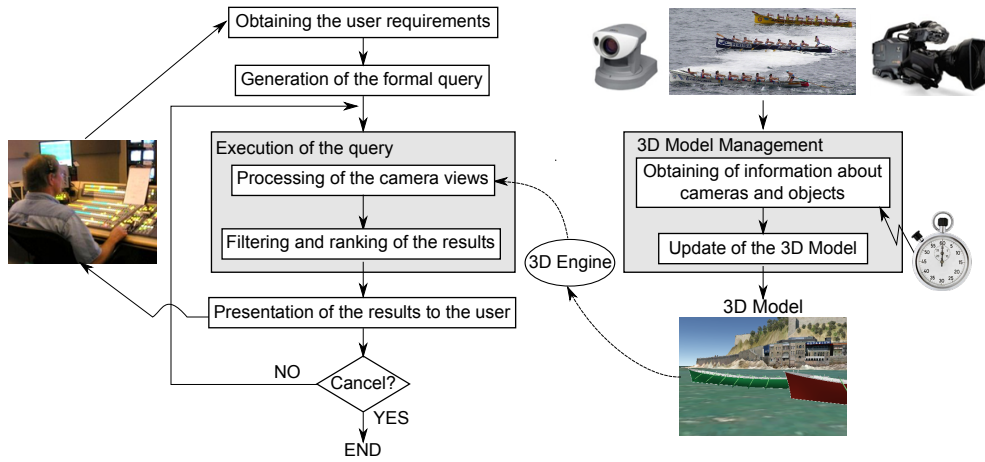


Figure B.19: Main steps followed by the MultiCAMBA prototype.

must be part of the view and the constraints that the camera view must fulfill, or c) the TD clicks on predefined queries.

3. *Generation of the formal query.* By analyzing the information provided by the user, the system generates a formal query capturing her requirements.
4. *Execution of the query.* The system obtains high-level features of the camera views (objects viewed, amount of them covered, kind of view, etc.) and the cameras are filtered to obtain those whose view fulfills the user requirements. Then, the answer set is ranked according to the user preferences.
5. *Presentation of the results to the user.* The results obtained by the system are presented to the user in the GUI, both in a tabular form and in a 3D reconstruction of the scenario.

B.4.2 Prototype of MultiCAMBA

Regarding the details of the implementation (see Figure B.20), the MultiCAMBA prototype has been developed as a Web application using HTML5 because the latest Google Earth API (which we use to recreate the scenario) is based on JavaScript and meant to be used in web pages. Then, the core of our prototype has been developed as a Java Applet that has been integrated into the Web application. We selected the Java programming language because for the processing of the camera views we are using a free and powerful Java 3D

engine: JMonkeyEngine (as we explained in Section 8.2). Both JMonkeyEngine and the Google Earth plugins extract the 3D meshes of objects-of-interest from OBJ (a geometry definition format) files. Also, we are using a MySQL database to store the 3D model of the scenario and other interesting information for the tests.

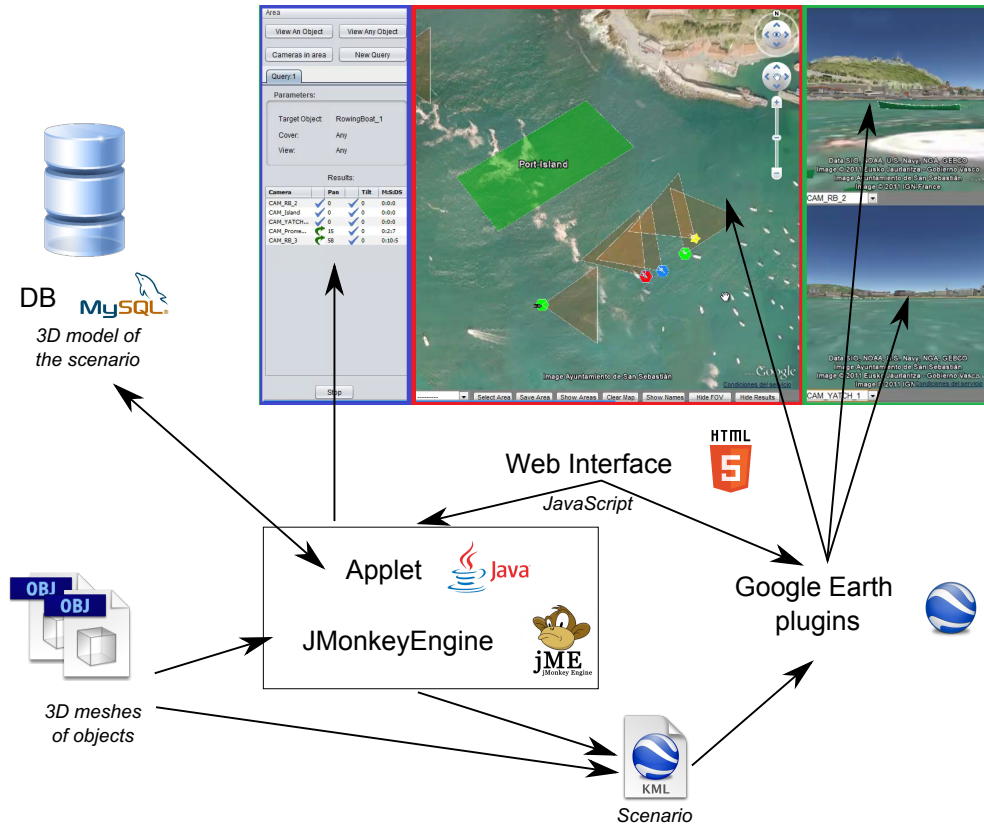


Figure B.20: Technical architecture diagram of the MultiCAMBA prototype.

We need to enable the communication between the different technologies and the transfer of information among them. For example, for the communication among the applet and the Google Earth plugins our prototype uses the Keyhole Markup Language (KML)¹². For this task, the applet creates and maintains updated a KML file with the current location, direction, and other information of cameras and objects in the scenario, that all the Google Earth plugins use to update the scene they show.

¹²<https://developers.google.com/kml>

Inspired by mobile production units in our third motivating use case (see Section 3.1.3), we have developed a friendly GUI that models a Technical Director (TD) work environment (see Figure B.21 and <http://sid.cps.unizar.es/MultiCAMBA>) where the TD can express her requirements and the results are displayed. The GUI is mainly composed of three modules:

1. The *query interface*, where the TD defines (using HTML forms or the 3D interface) the requirements that the cameras have to fulfill and stores / loads / submits her queries.
2. The *overview map*, which is a 3D representation of the scenario, with the moving objects and cameras involved, where the results of the queries are shown.
3. The *camera inputs*, which are several windows where the TD can preview the camera video streams before broadcasting them.

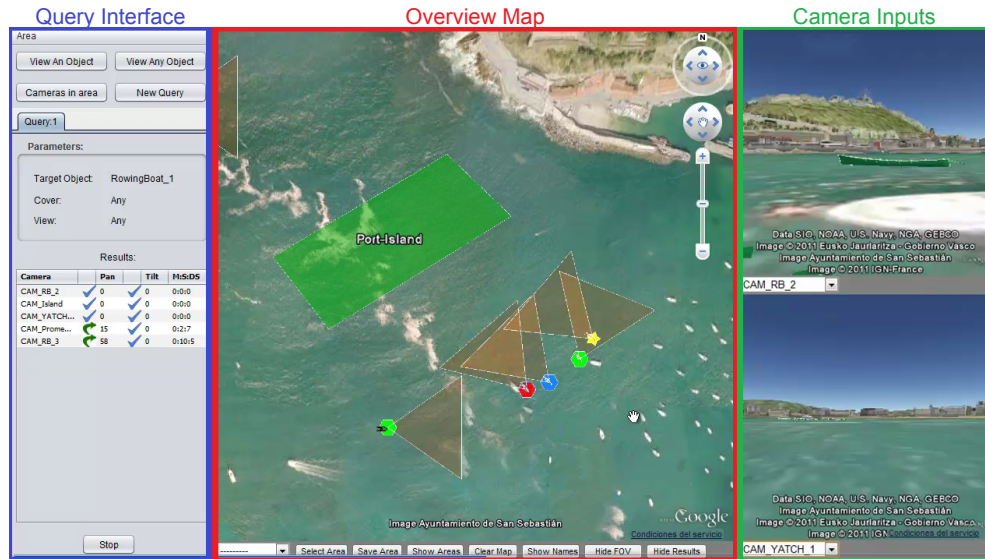


Figure B.21: MultiCAMBA app: Graphical User Interface (GUI) for the Technical Director.

Query Interface

The query interface of the system allows the TD to define the constraints that an interesting shot has to fulfill. This can be done using HTML forms, which

enable a quick definition of a query out the expense of less precision in the definition, or using a 3D interface, which enables the TD to define very precise camera views but requires more time.

In the example of Figure B.22 we show the usage of the HTML forms by a TD that has shown interest in obtaining “Any” camera that could provide a shot covering a certain rowing boat (“Kaiku”). At least “50%” of the boat has to appear in the shot, and the camera has to cover at least “75%” of the front view of the boat (a shot covering 100% of the front view is obtained by a camera located in front of the object pointing to it).

Figure B.22: MultiCAMBA app: Example of low-level input form.

We also provide the TD with a Query-by-Example 3D interface for the definition of 3D scenes (see Figure B.23) that supports a precise definition of the query shot required. So, instead of defining the constraints that the shot has to fulfill, the TD shows exactly the kind of shot she wants [ABP02] and the prototype extracts the required constraints. The interface has been developed using the Java 3D engine, *JMonkey Engine*, that we have used also for the analysis of the camera views.

Overview Map

Delivering the information easily and effectively is essential to quickly select the camera to broadcast in live. To achieve this goal we use a powerful and free software tool, the *Google Earth API*, to display the results in a friendly interface. Google Earth is a geographic information system that offers a vast

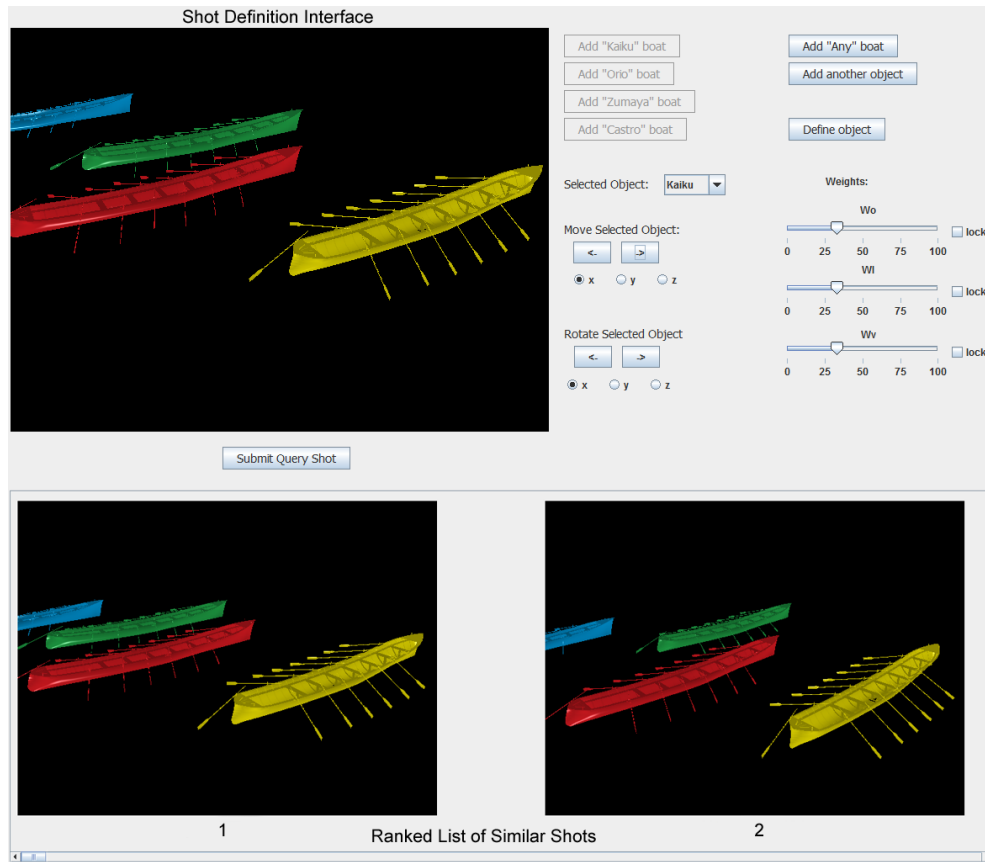


Figure B.23: MultiCAMBA app: Snapshot of the Query-by-Example 3D interface.

amount of geospatial data (satellite images, 3D buildings, 3D terrains, etc.), that helps to develop virtual scenes similar to those in the real world. This way, the overview map recreates and keeps up-to-date the scene in a Google Earth plugin allowing the TD to navigate through the scenario. In the center of Figure B.21 the overview map shows an example of the moving objects and cameras (a brown triangle indicates its current FOV) in a sport scenario. Besides, it shows the results to a query submitted by the TD to retrieve the cameras that can view a certain object (a yellow star is used to represent the target object, a green hexagon for the cameras fulfilling the requirements, a blue hexagon for the cameras that will fulfill them if rotated, and a red hexagon for the cameras unable to fulfill them).

Camera Inputs

The system also uses Google Earth to recreate the view of a camera. This is very useful mainly in two situations: when the real camera video stream is not available (as in the camera inputs of Figure B.21) and when a camera is not currently viewing the target and the system estimates the scene it will capture if it is rotated. The possibility to estimate future camera views with the combined use of Google Earth technology allows the system to show a realistic recreation of what a camera will view if rotated. This is interesting because sometimes the best shot is not the one that can be obtained the fastest. For example, a camera that is not currently viewing a target, but will be in a matter of seconds, could then provide a better background scene than a camera that is viewing the target currently.

B.4.3 Evaluating the Processing of Multimedia Information

In this section, we show the experimental evaluation performed to validate our approach to obtain images that fulfill the requirements of a user, focusing on the multimedia aspect¹³.

Experimental Setup

Testing a live broadcasting of a rowing race in a real-life environment is difficult because there are many real objects, devices, and wide-area scenarios involved. Besides, testing the system several times in similar situations in a real-life environment is challenging. Therefore, we have developed a simulator that enables us to manage the different cameras and objects in the scenario. To simulate the rowing race we have used a file containing the *real* GPS location data of each rowing boat captured every second during the race celebrated in San Sebastian in September 2010; this rowing race covers a total distance of 3 miles logically divided in two parts by a turning point. Therefore, *real* trajectories are used to move objects in the simulations. Moreover, the simulator allows us to dynamically change other parameters, such as the current pan and tilt of each camera, in order to rotate them as the TD would request in the real scenario.

The parameters used in the tests are the following ones:

1. There are four rowing boats equipped with a camera; as commented

¹³Some videos and interesting moments of the tests are available at <http://sid.cps.unizar.es/MultiCAMBA/Experiments>.

before, these boats move according to the real GPS location data captured during the race celebrated in San Sebastian in September 2010.

2. There are three other cameras: one on top of the island, one on the promenade, and one on a sailing boat near the rowing boats. The cameras are set with horizontal focus $\beta_h = 70^\circ$, vertical focus $\beta_v = 45^\circ$, pan range $\pm 130^\circ$, tilt range $\pm 90^\circ$, and pan and tilt speed 5.5 degrees/second.
3. The tests were performed in an Intel Core i5-480M with graphics card NVIDIA Geforce GT 540M.

For the experimental evaluation we will consider this simulated scenario and one of the most interesting queries for the TD. In our sample scenario, the TD may want to know, during the whole event, which cameras are viewing each of the rowing boats, as she could need a shot of a certain boat at anytime. In fact, during the broadcasting of any event, the TD would be interested on the cameras that are viewing the main agents (e.g., for a soccer match it is interesting to know the cameras that are viewing each of the star players). So, as a representative query in the context of our experiment we will continuously process the following one:

“Cameras that can view at least 70% of the Kaiku boat”

Evaluating the Quality of the Result Set

In this first experiment, we want to evaluate whether the cameras provided by the system as an answer are good candidates to provide the views required by the TD (see Figure B.24). We represent the number of cameras in the answer (vertical axis) along the event duration (horizontal axis). The blue line shows the number of cameras provided by the system (some of them currently fulfilling the requirements of the user and the others estimated by the system to be able to do it in the near future) and the red dashed line shows the number of wrongly chosen cameras. We consider a camera as wrongly chosen when the system makes an estimation error greater than 25 seconds in the estimated time to wait until the camera could provided the view required. For example, at the beginning the system estimates that the camera on board the farthest boat (that has two boats between it and the target and does not currently view the target) will cover 77% of “Kaiku” in 10 seconds, but when those 10 seconds have passed the camera is only able to view 21% because then the occlusion of “Kaiku” is greater than estimated. This occlusion remains for 30 seconds, and so the system makes a mistake considering this camera as part of the answer. These kinds of errors only happen when the target is occluded due to

variations in the speed and direction of the objects that make the system fail in the estimation of the future scene. Again, around time 17:30, where the boats are in the final sprint, the system estimates the time needed to view the target and shows the cameras in the answer, but some of them overtake “Kaiku” and thus they are not able to view it for the rest of the race (due to their rotation limits). However, thanks to the continuous query processing, estimation errors due to unexpected changes in the trajectories are quickly corrected.

For the test, the cameras are rotated automatically in order to track “Kaiku” according to the results provided by the system. The maximum number of cameras that can be part of the answer is six (because the camera of “Kaiku” cannot view itself due to the physical rotation limits), and the system shows in the answer at least three cameras during the most critical time intervals, that is, when the boats are at the turning point (which is a key moment during the race) around time instant 10:30 and when the other boats are overtaking “Kaiku” around time instant 17:00-20:00.

Even though the system must perform its calculations every second, the results are quite satisfactory for the total period of twenty minutes, and the errors are corrected quickly enough to avoid a negative impact on the decisions of the TD. Notice that, in our scenario, for the most part of the race a good number of cameras fulfill the TD requirements, as they are rotated automatically following the system commands. In this case, the system succeeds in discarding the irrelevant cameras at each moment and in ranking the most interesting cameras first.

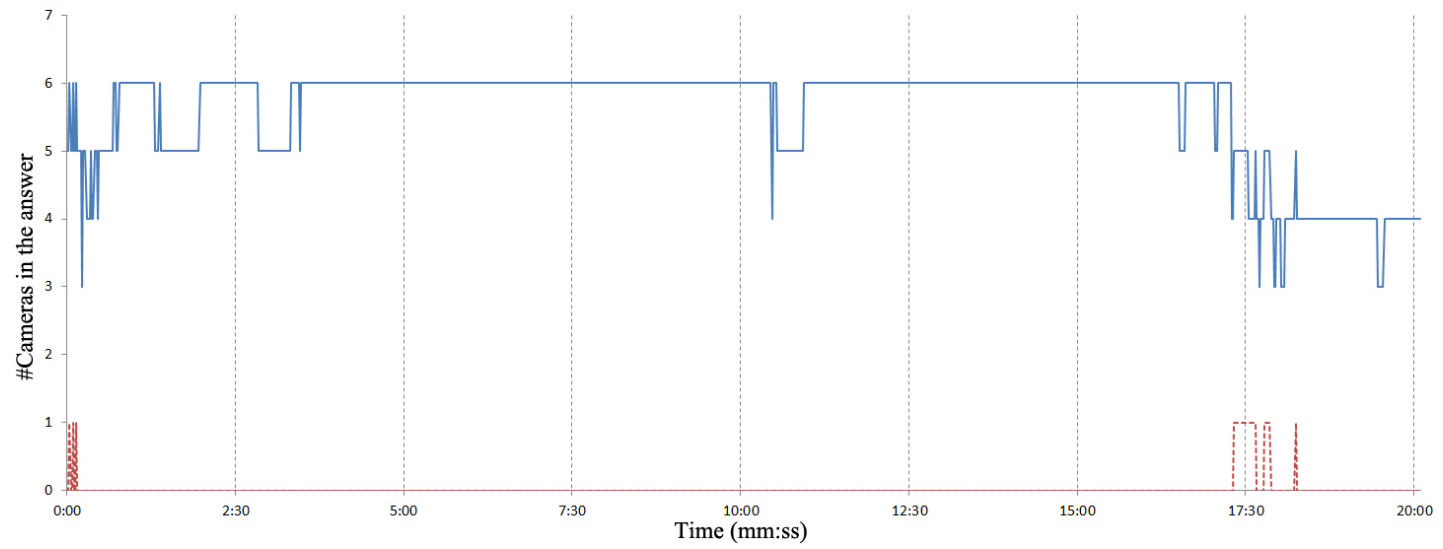


Figure B.24: Quality of the result set (i.e., cameras fulfilling the user requirements): number of cameras in the answer set (blue line) and number of wrongly chosen cameras (red dashed line).

Testing the Precision of the Estimated Time to View

The goal of this test is to evaluate the error in the time estimation, measured in seconds between the estimated time and the actual time when the camera views the target.

In Figure B.25, we show the sum of the time errors for all the cameras at every time instant. Notice that there are positive and negative values, that represent when the time estimated by the system is greater than the actual value (positive) or when it is smaller (negative). We have decided to make this distinction because a negative error could make the TD to keep an eye on a camera that actually will need more time than expected to view the target, which may be an important problem. On the other hand, a positive error (if it is not too big) means that a camera viewed the target earlier than expected, and so the negative impact of selecting that camera is minimized. Anyway, notice that the sum of all the errors ranges only between -5 and $+3$ seconds, which is very small for the sum of the errors of all the cameras.

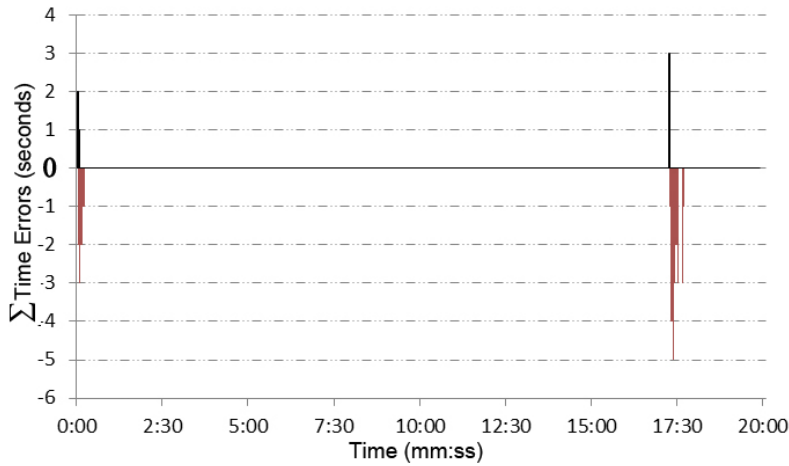


Figure B.25: Error in the estimated time needed for the cameras to view the target object: the error is localized at two specific time intervals (the start and the end of the race).

The errors are localized within two specific time intervals. The first errors occur at the start of the race, when all the cameras are pointing at the same direction as the front vector of their rowing boats. In this scenario there are not big changes on the objects' altitude (only slight changes caused by waves), so in order to test the pan and tilt estimation we have set initially all the

cameras pointing upwards at the start (maximum tilt). Thus, as the test starts, the cameras have to pan and tilt to view “Kaiku” and the system has to estimate the time needed to do it. As at the starting point the system has not enough information to precisely estimate the speeds of the boats, it makes some little mistakes that, overall, do not exceed 3 seconds. Thus, this error is small enough to provide the TD with accurate information. Around time 17:30 the end of the race is near and there are big variations in the speed and distance between the boats (as the rowers are making their last efforts) and some boat is even overtaken. The system estimates here that the time needed to view “Kaiku” is smaller than the actual time needed, as the boats increase their speed in a final attempt to win the race.

This test shows that the errors concerning the estimated time to view the target are small. Besides, those errors are localized in two narrow time intervals and they are quickly fixed (in the test, in ten seconds at most), since the corresponding location-dependent query is continuously processed.

Testing the Precision of the Estimated Percentage Viewed

We also tested the precision of the estimated percentage of the target viewed by the cameras. Due to the specific features of the scenario, a camera views a percentage below 100% when the target is partially occluded by another boat (the target usually remains inside the FOV of the cameras). As explained before, the system is able to compute the percentage of “Kaiku” that a camera will cover when it is able to view it. For the camera closest to “Kaiku” (that has no other boat between them), the errors in the estimation of the percentage only occur when there is an error in the estimation of the time needed to view it. We have considered these errors in Figure B.25. We show in Figure B.26 an example for a camera that has two boats between it and the target (and thus occlusions are possible) and focusing on a short time interval (the first 8 seconds) where some errors occur. The system starts estimating that the camera will view a smaller percentage than what it will really view, and as the time goes by the error in this estimation decreases.

Testing the System Against Real Camera Footage

We have also tested the system against real camera footage. Specifically, we have used the video produced by the Spanish broadcaster *EiTB* for the rowing race celebrated in September 2010. The goal of this test was to compare the results offered by our system with the real event to check the behavior of our proposal. Figure B.27 shows an extract of two seconds of a camera

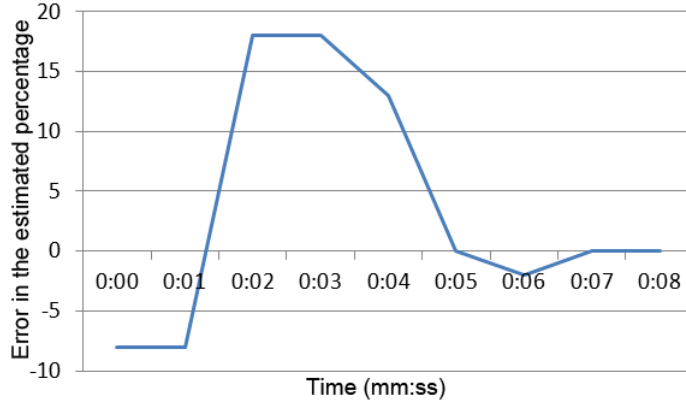


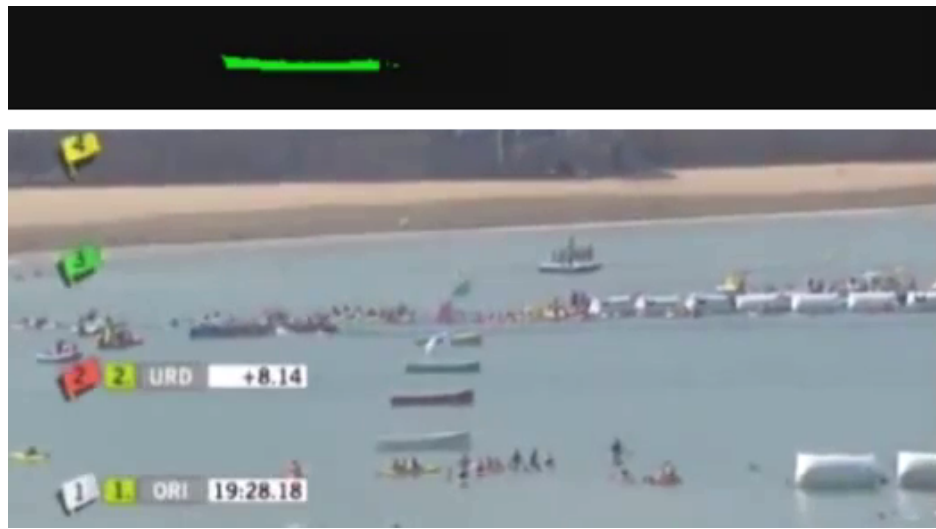
Figure B.26: Error in the estimated percentage of the target that a camera will view in the first eight seconds.

covering the end of the race (the real images are shown on the bottom of each frame) compared with the information generated and exploited by the query processing in our system (shown at the top of each frame). Notice that the extents of the rowing boats are represented in green by the system, for an easy comparison with the real footage. At the beginning of this live footage, the “Urdaibai” boat has just crossed the finish line (it is the only boat we see in the first frame in Figure B.27(a)); it is followed by “Kaiku”, that enters the FOV of the camera 1 second later, being completely captured by the camera exactly at time instant 2.08 seconds (Figure B.27(b)).

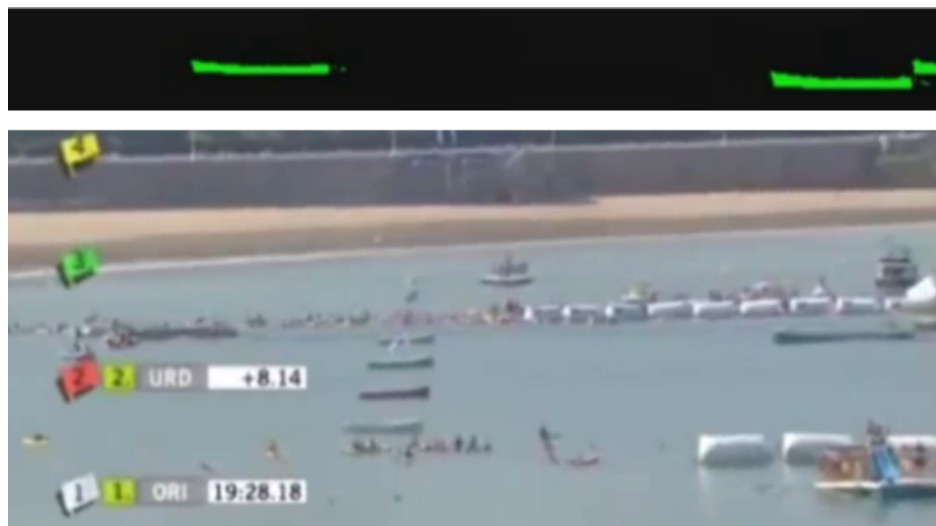
At the time instant when the live footage begins, our system estimates that the camera will obtain a full view of the “Kaiku” boat in 1.85 seconds. Notice that the error committed is not very significant: the TD will be alerted just 0.23 seconds before the desired situation is captured by the real camera. This small error is caused by the slight inaccuracy of the GPS data transmitted by the boats and the fact that the location of the real camera was estimated (the TV broadcaster did not provide us with this information). However, the results obtained by the system would have been good enough to help the TD to select this camera to view the “Kaiku” boat.

Evaluation of the Ranking Presented to the User

The purpose of this experiment is to evaluate the ranked answer provided by the system for a given query image instead of the description of a scene like in the previous tests. The ranking criteria are very important because, due to



(a)



(b)

Figure B.27: Testing the system against real camera footage (the information generated by our system is on the top): in two consecutive seconds.

the need of selecting the next camera to broadcast as quickly as possible, the TD will consider only the first positions in the ranking.

In [SB05] the authors emphasized that “image retrieval is only meaningful

in its service to people, performance characterization must be grounded in human evaluation”. However, evaluating an image retrieval system is a difficult task [Tho10] and testing it with real users is time-consuming (so, in many approaches tests are performed with a limited number of participants [ABP02; BP97]). Moreover, in our case finding real TDs with experience in the live broadcasting of sport events able to take part in our experiments was not possible, although it would have been very interesting. So, for our test, 10 users familiarized with the use of 3D interfaces were recruited and we used four query images that could be interesting for a TD (see Figure B.28). The users were presented with 45 images given in arbitrary order (see Figure B.29) and the four queries, and their goal was to select the images that they considered similar to each query and then to rank the selected images according to their similarity. To accomplish this task, the users were allowed to choose the order in which they wanted to answer the queries and modify their previous answers whenever necessary. We have selected these 45 images as they show different numbers of objects in different configurations. The number is high enough to obtain at least five similar images per query image and at the same time is low enough for the users to be able to check all the images correctly (the higher the number of images available, the higher the difficulty to keep the concentration of the user to verify them all carefully, as the amount of information that can be kept in the working memory of humans is considered to be limited [Mil56]).

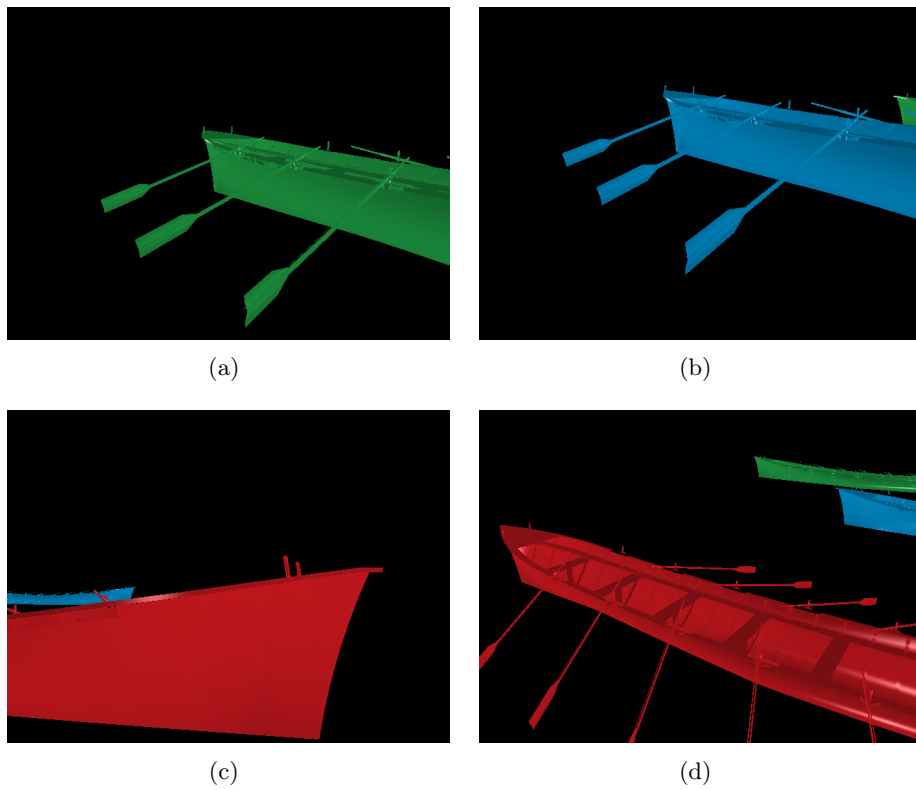


Figure B.28: Query images used in the tests.

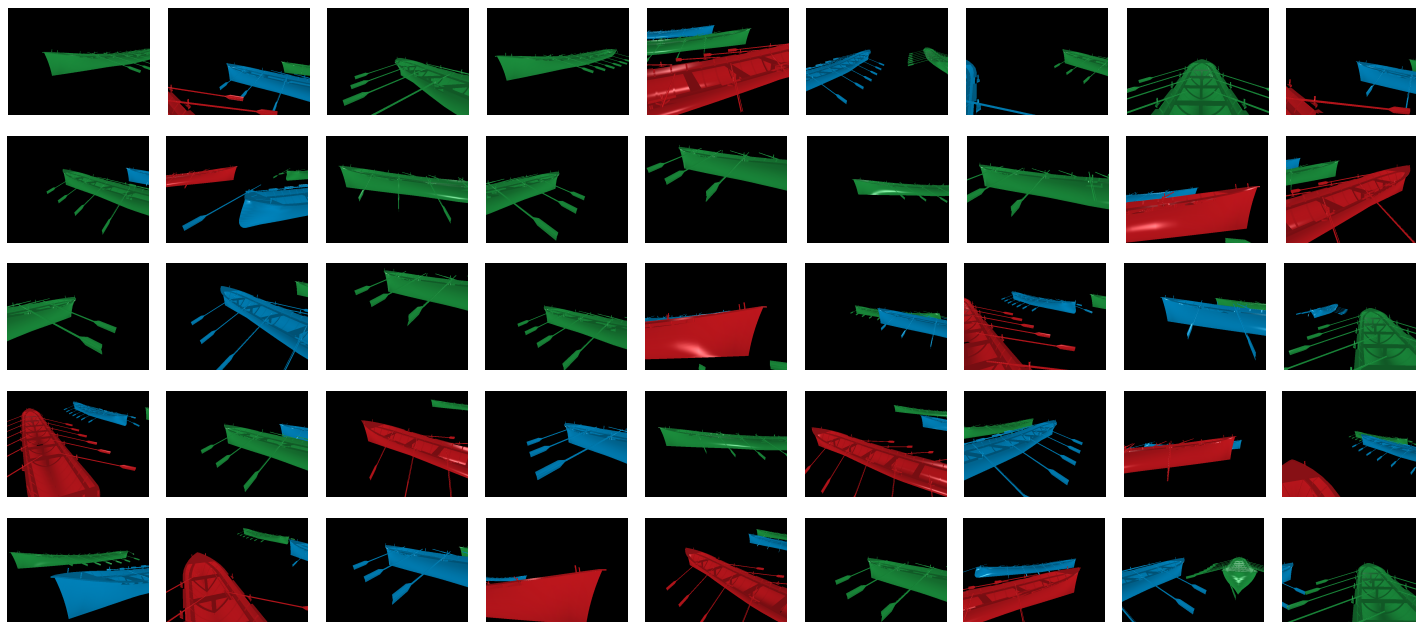


Figure B.29: Set of 45 images used in the tests.

Considering an image as similar to another one is a very subjective matter. For some users an image is similar to a query image if it shows the same objects, for others if it shows the same percentage of the object, and for others if the viewpoint of the most prominent object is the same. So, as explained in Section 8.3, we used weights equally distributed when computing the similarity in our tests. According to the results obtained from the users concerning the number of similar images for each query (see Figure B.30), we can observe three kinds of users: some users have a “demanding profile”, as they select only a few pictures as similar (e.g., the ninth user); other users have a “lax profile”, as they select more images than the average (e.g., second and eighth users); finally, the rest of users have an “average profile”. Thus, we think that the system has to rank all the images without discarding any image: if we considered instead a *top-k ranking*, the appropriate k would depend on the subjectivity of the specific user and the specific query and set of images available. However, ranking all the images also emphasizes the importance of providing a good ordering, such that images that are very different from the query image have to be placed at the end of the result list.

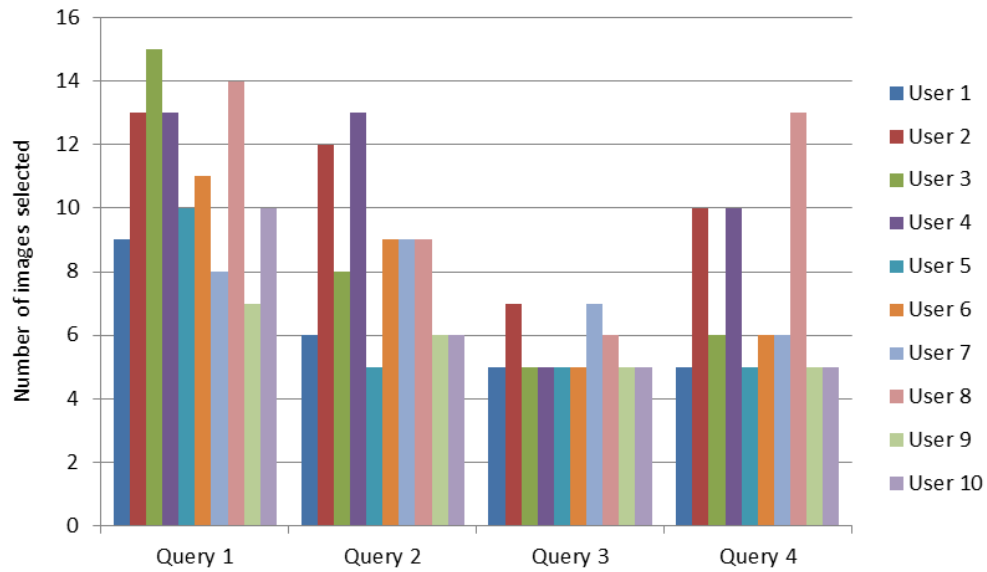


Figure B.30: Number of images selected by the users as similar to each query image.

The next step is to compare the users’ rankings with the ranking provided by our system. As the users’ rankings for a query could include different images

and a different number of results, we have to compare *partial lists of varying length*. It is interesting to consider that it is quite common for a user to make a precise definition of the firsts positions of a ranking and the precision decreases as the element is located in the last positions. This has been confirmed by some users, that have explained that the images they placed at the end of the list had all approximately the same similarity to the query image (according to their opinion). So, we will focus on the first five images of each user's ranking (five is the maximum number of images selected for all the users for all the queries) to compare it with the system's ranking. *Kendall's tau* [Ken38] is a measure widely used to compare rankings. However, it has the limitation that the rankings to compare have to contain the same elements (i.e., they have to be *full rankings*). In [FKS03] a *generalized Kendall's tau* that overcomes this limitation is presented, that can be applied to compare top-k lists:

$$K^{(p)}(\tau_1, \tau_2) = \sum_{i,j \in P(\tau_1, \tau_2)} \bar{K}_{i,j}^{(p)}(\tau_1, \tau_2) \quad (\text{B.1})$$

where τ_1 and τ_2 are the lists to compare and $P(\tau_1, \tau_2)$ is the union set of the elements in both lists. In addition, $\bar{K}_{i,j}^{(p)}(\tau_1, \tau_2) = 1$ if any of the following conditions hold: (a) i and j appear in both lists but in reverse order (i.e., i is ranked higher than j in one list but lower in the other); (b) i and j both appear in one list (and j is ahead i) and exactly one of i or j appears in the other list; (c) i , but not j , appears in one list, and j , but not i , appears in the other top-k list. Otherwise, $\bar{K}_{i,j}^{(p)}(\tau_1, \tau_2) = 0$, as we are considering the “optimistic approach” of Kendall's tau with $p = 0$ (i.e., $K^{(0)}$), which is a frequent instantiation of Kendall's tau in the literature. In order to normalize $K^{(0)}$ in such a way that two identical lists have a value of 1 and two lists that share no element have a value of 0, we use the normalized K [MN07]:

$$K = 1 - \frac{K^{(0)}(\tau_1, \tau_2)}{k^2} \quad (\text{B.2})$$

We have computed the normalized distance, K , between our system's ranking and the users' ranking (the results are shown in Figure B.31, where $K(u_i, s)$ represents the K value between the ranking provided by the user $user_i$ and the ranking of the system s). Values below 0.5 usually indicate that the system does not select some images that appear in the user's ranking, and values above 0.5 indicate that the system selects all the images selected by the user but the order is exactly the same only if the value of K is 1. So, for *Query 3* and *Query 4* the similarity between the rankings provided by the users

and the one provided by the system is particularly high, as with the dataset used in the experiments the users found it quite easy to select and rank images similar to those query images. However, for *Query 1* the users found more difficulties to rank the images, as there is a higher number of images that could be considered similar to the query image; in the ranking provided, for example, some users considered as more similar the images where a similar percentage of the green rowing boat was shown, whereas others considered more similar the images that showed a similar perspective.

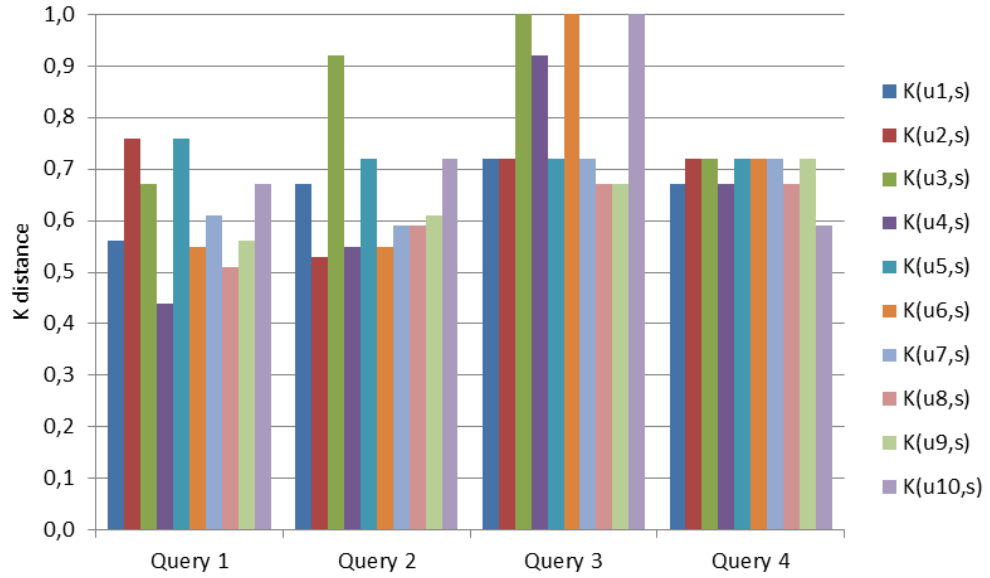


Figure B.31: Comparing the ranking of images obtained by our system (s) and the users' rankings (u_i) for each query (normalized Kendall tau distance $K(u_i, s)$).

However, it has to be noticed that different users usually propose different rankings for the same set of images (i.e., there is some disagreement between the users about the best ranking, due to the subjectivity of the process). Therefore, comparing only the system's ranking with each of the users' rankings would be unfair. To take the subjectivity into account, we apply a similar approach to the one used in [TPIBM11] to obtain the level of disagreement between the system and the users. First, we define for each query a global level of disagreement between all the users and the system, called *System Disagreement (SD)*:

$$SD = 1 - \frac{\sum_{i=1}^M K_{i,s}}{M} \quad (\text{B.3})$$

where $K_{i,s}$ is the normalized distance K for each of the users' rankings compared with the ranking provided by our system, and M is the number of users (10 in our case). The value of SD can be interpreted intuitively as follows. Considering the extreme cases, $SD = 0$ would mean that the system obtains exactly the same results (the same images in the same order) than all the users (this would be possible only if all the users provided exactly the same answer), and $SD = 1$ would mean that the system results are completely different from the results provided by any of the users. For the intermediate cases, the rankings provided by the users and the system are more similar when the value of SD is low.

Then, we define for each query a global level of disagreement among all the users called *Tester's Disagreement (TD)*, as the users play the role of testers for our system:

$$TD = 1 - \frac{\sum_{i=1}^M \frac{\sum_{j=1, j \neq i}^{M-1} K_{i,j}}{M-1}}{M} \quad (\text{B.4})$$

where $K_{i,j}$ is the normalized distance K for two users' rankings. The value of TD can be interpreted similarly to what was explained before for the value of SD , but in this case TD measures the difference among the rankings provided by the users.

In Figure B.32 we show the resulting SD and TD for each of the four queries. As $K = 1$ indicates that the two selected rankings are exactly the same, we consider that the ranking of similar pictures for a query image is correct as long as $SD \leq TD$ (i.e., when the disagreement between the users and the system is not higher than the disagreement between the users themselves). According to this, the system always behaves well except for the last query (*Query 4*), where $SD = 0.31$ and $TD = 0.27$. So, we analyze this query in the following to explain this behavior.

Figure B.33(a) shows the ranking of images that the system obtains for *Query 4*. We noticed that the system locates in the fourth position an image that was not present in any of the users' top-5 rankings. This was the cause that led to obtaining a SD slightly greater than TD for this query. The reason for this behavior is related to the weights assigned to each term of

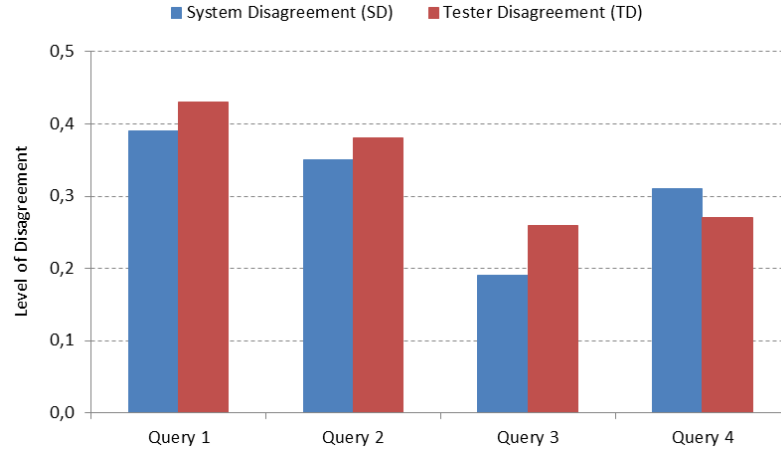


Figure B.32: Tester Disagreement and System Disagreement.

the similarity function used by our system (Formula 8.12 in Section 8.3.4). Specifically, we consider by default $w_l = w_o = w_v = 1/3$, as there is no objective criterion to assign different weights. However, analyzing the users' rankings for the last query, we have noticed that for most of them the percentage of the objects viewed and the percentage of the shot occupied by the objects were more important than other factors. Taking this into account, we have also set the weights $w_o = 0.7$, $w_v = 0.25$, and $w_l = 0.05$, and reevaluated the query, obtaining a new ranking (see Figure B.33(b)) and a new $SD = 0.27$, which makes the disagreement between the system and the users equal to the disagreement between the users. So, by adapting the weights used in the similarity function (which can be performed easily by using the sliders available in the GUI, as shown in Figure B.23), we can customize the system according to the preferences of a specific user. Another example of the potential interest of adjusting the weights is presented in the following section.

Evaluation of the User Satisfaction when Entering an Arbitrary Query

We have performed other tests where “expert” users (persons who are not only familiar with the use of 3D interfaces but also fond of photography) have used the prototype to formulate their own queries. In this section, for illustration purposes, we explain some results obtained with one of these expert users. Similar conclusions can be drawn from the tests performed with other expert users.

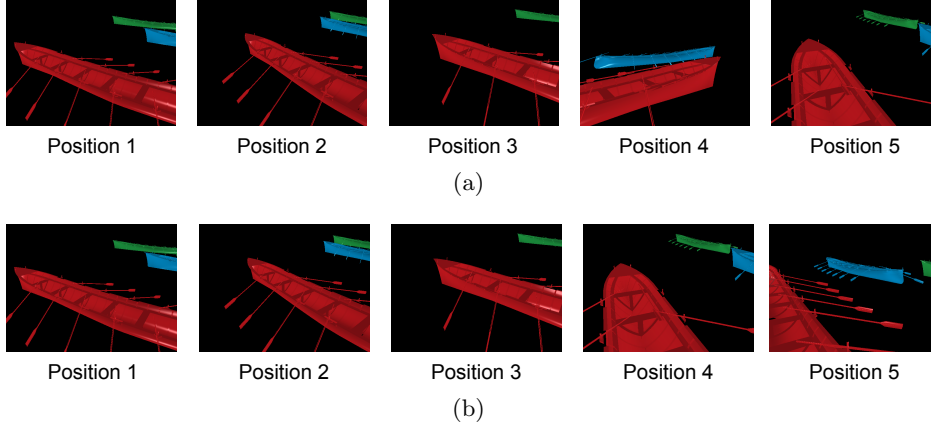


Figure B.33: Ranking obtained by the system for *Query 4*: before (a) and after (b) modifying the weights of the similarity formula.

First, we asked the user to define a query scene; the user found no difficulties in performing this task to compose the wanted query image. Afterwards, to expand the dataset used in the previous tests, we generated some images by moving the rowing boats in the scene and by moving/rotating the camera randomly. Then, based on the new dataset of images that we generated, the system presented to the user a ranked list of images similar to his query image (see Figure B.34(a)) and we asked him to make some comments about the results. He pointed out that, for him, the second image was more similar to the query than the first one, due to the viewpoint of the camera. He also noted that the rest of the images were somewhat similar to the query but he would rule out them compared to the first and second ones (the user showed a “demanding profile”).

We analyzed the user answer and decided to modify the weights of the similarity function (Formula 8.12) to match his preferences. By increasing the weight of the views and decreasing the weight of the location (we used $w_o = 0.3$, $w_v = 0.6$, and $w_l = 0.1$) we obtained a new ranking (see Figure B.34(b)) where the positions of the first two images are inverted, as the user would have expected. Moreover, by adapting the weights to the preferences of that specific user, the differences between the computed similarity of the image in the second position and the third, fourth, and fifth increased from 0.04, 0.04, and 0.06 (in Figure B.34(a)), to 0.14, 0.15, and 0.21 (in Figure B.34(b)), respectively; this means that with the new weights the last three images were considered by the system much less similar to the query image than before. So, it is

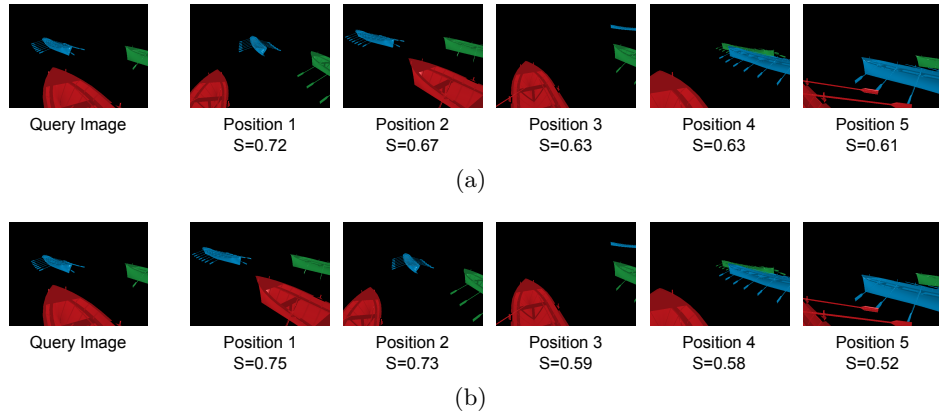


Figure B.34: Ranking of images obtained for a user query: before (a) and after (b) modifying the weights of the similarity formula.

possible to fine-tune the weights according to the preferences of the user. Some complementary works have proposed to automatically infer suitable preference weights based on past user's interactions with the system [RHOM98].

We shall overcome.

Bibliography

- [AAA13] Samina Raza Abidi, Syed Sibte Raza Abidi, and A Abusharek. “A Semantic Web based mobile framework for designing personalized patient self-management interventions”. In: *1st Conference on Mobile and Information Technologies in Medicine*. 2013.
- [ABP02] Jürgen Assfalg, Alberto Del Bimbo, and Pietro Pala. “Three-Dimensional Interfaces for Querying by Example in Content-Based Image Retrieval”. In: *IEEE Transactions on Visualization and Computer Graphics* 8.4 (2002), pp. 305–318.
- [ADBDSS99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. “Towards a Better Understanding of Context and Context-Awareness”. In: *1st International Symposium on Handheld and Ubiquitous Computing (HUC 1999)*. 1999, pp. 304–307.
- [ADV07] Tarik Filali Ansary, Mohamed Daoudi, and Jean-Phillipe Vandeborre. “A Bayesian 3-D Search Engine Using Adaptive Views Clustering”. In: *IEEE Transactions on Multimedia* 9.1 (2007), pp. 78–88.
- [AEJJIM13] Raeda F AbuAlRub, Fadi El-Jardali, Diana Jamal, Abdulkareem S Iblasi, and Susan F Murray. “The challenges of working in underserved areas: a qualitative exploratory study of views of policy makers and professionals”. In: *International Journal of Nursing Studies* 50.1 (2013), pp. 73–82.

- [AEMPW07] Arnon Amir, Alon Efrat, Jussi Myllymaki, Lingeshwaran Palaniappan, and Kevin Wampler. “Buddy tracking – efficient proximity detection among mobile friends”. In: *Pervasive and Mobile Computing* 3.5 (2007), pp. 489–511.
- [AKK06] Yasuo Ariki, Shintaro Kubota, and Masahito Kumano. “Automatic Production System of Soccer Sports Video by Digital Camera Work Based on Situation Recognition”. In: *8th IEEE International Symposium on Multimedia (ISM 2006)*. IEEE Computer Society, 2006, pp. 851–860.
- [ARCGHJR13] Ana Armas-Romero, Bernardo Cuenca-Grau, Ian Horrocks, and Ernesto Jiménez-Ruiz. “MORE: a Modular OWL Reasoner for Ontology Classification”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. Ulm, Germany: CEUR-WS, 2013, pp. 61–67.
- [AWH10] Chris J. van Aart, Bob J. Wielinga, and Willem Robert van Hage. “Mobile Cultural Heritage Guide: Location-Aware Semantic Search”. In: *17th International Conference on Knowledge Engineering and Management by the Masses (EKAW 2010)*. 2010, pp. 257–271.
- [BB09] Christian Becker and Christian Bizer. “Exploring the Geospatial Semantic Web with DBpedia Mobile”. In: *Journal of Web Semantics* 7.4 (2009), pp. 278–286.
- [BBCCGMMV00] Domenico Beneventano, Sonia Bergamaschi, Silvana Castano, Alberto Corni, R Guidetti, G Malvezzi, Michele Melchiori, and Maurizio Vincini. “Information integration: the MOMIS project demonstration”. In: *26th International Conference on Very Large Data Bases (VLDB 2000)*. 2000, pp. 611–614.
- [BBHINRR10] Claudio Bettini, Oliver Brdiczka, Karen Henriksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. “A Survey of Context Modelling and Reasoning Techniques”. In: *Pervasive and Mobile Computing* 6.2 (2010), pp. 161–180.

- [BBIM14] Carlos Bobed, Fernando Bobillo, Sergio Ilarri, and Eduardo Mena. “Answering Continuous Description Logic Queries: Managing Static and Volatile Knowledge in Ontologies”. In: *International Journal on Semantic Web and Information Systems* 10.3 (2014), pp. 1–44.
- [BBMI13] Jorge Bernad, Carlos Bobed, Eduardo Mena, and Sergio Ilarri. “A Formalization for Semantic Location Granules”. In: *International Journal of Geographical Information Science* 27.6 (2013), pp. 1090–1108.
- [BBYEM14] Carlos Bobed, Fernando Bobillo, Roberto Yus, Guillermo Esteban, and Eduardo Mena. “Android Went Semantic: Time for Evaluation”. In: *3rd International Workshop on OWL Reasoner Evaluation (ORE 2014)*. Vol. 1207. CEUR-WS, 2014, pp. 23–29.
- [BCMNP03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook. Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [BDR07] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. “A Survey on Context-Aware Systems”. In: *International Journal of Ad Hoc and Ubiquitous Computing* 2.4 (2007), pp. 263–277.
- [BG04] Dan Brickley and Ramanathan V Guha. *RDF vocabulary description language 1.0: RDF schema*. 2004.
- [BGJRMPS13] Samantha Bail, Birte Glimm, Ernesto Jiménez-Ruiz, Nicolas Matentzoglou, Bijan Parsia, and Andreas Steigmiller. “Summary ORE 2014 Competition”. In: *3rd International Workshop on OWL Reasoner Evaluation (ORE 2014)*. Vol. 1207. Vienna, Austria: CEUR-WS, 2013, pp. IV–VII.
- [BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. “Linked Data-the story so far”. In: *International Journal on Semantic Web and Information Systems* 5.3 (2009), pp. 1–22.

- [BHJV08] Jrgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. “Benchmarking OWL Reasoners”. In: *International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008)*. 2008.
- [Bia98] Michael H. Bianchi. “AutoAuditorium: A Fully Automatic, Multi-Camera System to Televisе Auditorium Presentations”. In: *1998 Joint DARPA/NIST Smart Spaces Technology Workshop*. 1998.
- [BJKS06] Rimantas Benetis, Christian S. Jensen, Gytis Karčiauskas, and Simonas Šaltenis. “Nearest and reverse nearest neighbor queries for moving objects”. In: *The VLDB Journal* 15.3 (2006), pp. 229–249.
- [BK11] Robert Battle and Dave Kolas. “GeoSPARQL: enabling a geospatial Semantic Web”. In: *Semantic Web Journal* 3.4 (2011), pp. 355–370.
- [BKOK04] Noboru Babaguchi, Yoshihiko Kawai, Takehiro Ogura, and Tadahiro Kitahashi. “Personalized abstraction of broadcasted American football video by highlight selection”. In: *IEEE Transactions on Multimedia* 6.4 (2004), pp. 575–586.
- [BKOTV11] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Zdravko Tashev, and Ruslan Velkov. “OWLIM: A family of scalable semantic repositories”. In: *Semantic Web Journal* 2 (1 2011), pp. 33–42.
- [BLHL+01] Tim Berners-Lee, James Hendler, Ora Lassila, et al. “The Semantic Web”. In: *Scientific American* 284.5 (2001), pp. 28–37.
- [BLKABCH09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sren Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. “DBpedia - A crystallization point for the Web of Data”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 7.3 (2009), pp. 154–165.
- [BLS06] F. Baader, C. Lutz, and B. Suntisrivaraporn. “CEL - A Polynomial-time Reasoner for Life Science Ontologies”. In: *3rd International Joint Conference on Automated*

- Reasoning (IJCAR 2006)*. Vol. 4130. Springer, 2006, pp. 287–291.
- [BMJ07] Ronald Beaubrun, Bernard Moulin, and Nafaa Jabeur. “An architecture for delivering location-based services”. In: *International Journal of Computer Science and Network Security* 7.7 (2007), pp. 160–166.
- [BP97] Alberto Del Bimbo and Pietro Pala. “Visual Image Retrieval by Elastic Matching of User Sketches”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19.2 (1997), pp. 121–132.
- [BS16] Fernando Bobillo and Umberto Straccia. “The fuzzy ontology reasoner fuzzyDL”. In: *Knowledge-Based Systems* 95 (2016), pp. 12–34.
- [BSS10] Max Braun, Ansgar Scherp, and Steffen Staab. “Collaborative Semantic Points of Interests”. In: *7th Extended Semantic Web Conference (ESWC 2010)*. 2010, pp. 365–369.
- [BTAABT11] Azzedine Boukerche, Begumhan Turgut, Nevin Aydin, Mohammad Z. Ahmad, Ladislau Bölöni, and Damla Turgut. “Routing protocols in ad hoc networks: A survey”. In: *Computer Networks* 55.13 (2011), pp. 3032–3080.
- [Bur10] Ed Burnette. *Hello, Android: Introducing Google’s Mobile Development Platform*. The Pragmatic Programmers, LLC., 2010.
- [BYBIBMTLG15] Carlos Bobed, Roberto Yus, Fernando Bobillo, Sergio Ilarri, Jorge Bernad, Eduardo Mena, Raquel Trillo-Lado, and Angel Luis Garrido. “Chapter 4: Emerging Semantic-Based Applications”. In: *Semantic Web: Implications for Technologies and Business Practices*. Ed. by Michael Workman. Springer International Publishing, 2015, pp. 39–83.
- [BYBM15] Carlos Bobed, Roberto Yus, Fernando Bobillo, and Eduardo Mena. “Semantic Reasoning on Mobile Devices: Do Androids Dream of Efficient Reasoners?” In: *Journal of Web Semantics* 35 (2015), pp. 167–183.

- [CBLZW11] Muhammad Aamir Cheema, Ljiljana Brankovic, Xuemin Lin, Wenjie Zhang, and Wei Wang. “Continuous Monitoring of Distance-Based Range Queries”. In: *IEEE Transactions on Knowledge and Data Engineering* 23.8 (2011), pp. 1182–1199.
- [CCCCDVF12] Irene Celino, Dario Cerizza, Simone Contessa, Marta Corubolo, Daniele Dell’Agllo, Emanuele Della Valle, and Stefano Fumeo. “Urbanopoly - A Social and Location-Based Game with a Purpose to Crowdsources Your Urban Data”. In: *2012 International Conference on Social Computing (SocialCom 2012)*. 2012, pp. 910–913.
- [CCL03] Imrich Chlamtac, Marco Conti, and Jeffifer J. N. Liu. “Mobile Ad Hoc Networking: Imperatives and Challenges”. In: *Ad Hoc Networks* 1.1 (2003), pp. 13–64.
- [CDH11] Amparo Elizabeth Cano, Aba-Sah Dadzie, and Melanie Hartmann. “Who’s Who - A Linked Data Visualisation Tool for Mobile Environments”. In: *8th Extended Semantic Web Conference (ESWC 2011)*. 2011, pp. 451–455.
- [CDV10] Fan Chen and Christophe De Vleeschouwer. “Personalized Production of Basketball Videos from Multi-sensored Data Under Limited Display Resolution”. In: *Computer Vision and Image Understanding* 114.6 (2010), pp. 667–680.
- [CDV11] Fan Chen, Damien Delannay, and Christophe De Vleeschouwer. “An autonomous framework to produce and distribute personalized team-sport video summaries: a basket-ball case study”. In: *IEEE Transactions on Multimedia* 13.6 (2011), pp. 1381–1394.
- [CEBMPN13] David Corsar, Peter Edwards, Chris Colin Baillie, Milan Markovic, Konstantinos Papangelis, and John D. Nelson. “GetThere: A Rural Passenger Information System Utilising Linked Data & Citizen Sensing”. In: *12th International Semantic Web Conference (ISWC 2013)*. 2013, pp. 85–88.

- [CH06] Chih-Chieh Cheng and Chiou-Ting Hsu. “Fusion of audio and motion information on HMM-based highlight extraction for baseball games”. In: *IEEE Transactions on Multimedia* 8.3 (2006), pp. 585–599.
- [CHK97] David M. Chess, Colin G. Harrison, and Aaron Kershbaum. “Mobile Agents: Are They a Good Idea?”. In: *2nd International Workshop on Mobile Object Systems - Towards the Programmable Internet (MOS 1996)*. London, UK, UK: Springer-Verlag, 1997, pp. 25–45.
- [CK00] Guanling Chen and David Kotz. *A Survey of Context-Aware Mobile Computing Research*. Tech. rep. Hanover, NH, USA, 2000.
- [CLY09] K. Choi, S.W. Lee, and Seo Y. “Automatic Broadcast Video Generation for Ball Sports From Multiple Views”. In: *International Workshop on Advanced Image Technology (IWAIT 2009)*. 2009.
- [Cro12] Adam Crowe. *Disasters 2.0: The application of social media systems for modern emergency management*. CRC press, 2012.
- [CSLC12] Yuxin Chen, Hariprasad Sampathkumar, Bo Luo, and Xue-wen Chen. “iLike: Bridging the Semantic Gap in Vertical Image Search by Integrating Text and Visual Features”. In: *IEEE Transactions on Knowledge and Data Engineering* PP.99 (2012), p. 1.
- [CTHH13] Mario Henrique Cruz Torres, Robrecht Haesevoets, and Tom Holvoet. “CooS: Coordination Support for Mobile Collaborative Applications”. In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQ 2013)*. 2013, pp. 152–163.
- [CVNMMNORU14] Andrea Calì, Roberto De Virgilio, Tommaso Di Noia, Luca Menichetti, Roberto Mirizzi, Luca Nardini, Vito Claudio Ostuni, Fabrizio Rebecca, and Marco Ungania. “Semantic Search in RealFoodTrade”. In: *8th Alberto Mendelzon Workshop on Foundations of Data Management*. 2014.

- [DA11] Melwyn D'Souza and V.S. Ananthanarayana. "Decentralized registry based architecture for location-based services". In: *6th IEEE International Conference on Industrial and Information Systems (ICIIS 2011)*. 2011, pp. 136–139.
- [DCTK11] Kathrin Dentler, Ronald Cornet, Annette ten Teije, and Nicolette de Keizer. "Comparison of reasoners for large ontologies in the OWL 2 EL profile". In: *Semantic Web Journal 2.2* (2011), pp. 71–87.
- [DE10] Jérôme David and Jérôme Euzenat. "Linked data from your pocket: The Android RDFContentProvider". In: *9th International Semantic Web Conference (ISWC 2010)*. 2010, pp. 129–132.
- [DFJPCPRDS04] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs. "Swoogle: a search and metadata engine for the semantic web". In: *13th ACM International Conference on Information and Knowledge Management (CIKM 2004)*. 2004, pp. 652–659.
- [DFKSS09] Julian Dolby, Achille Fokoue, Aditya Kalyanpur, Edith Schonberg, and Kavitha Srinivas. "Scalable highly expressive reasoner (SHER)". In: *Journal of Web Semantics 7.4* (2009), pp. 357–361.
- [DGGHJMPSS15] Michel Dumontier, Birte Glimm, Rafael S. Gonçalves, Matthew Horridge, Ernesto Jiménez-Ruiz, Nicolas Marentzoglou, Bijan Parsia, Giorgos B. Stamou, and Giorgos Stoilos, eds. *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description Logics (DL 2015), Athens, Greece, June 6, 2015*. Vol. 1387. CEUR-WS, 2015.
- [DK09] Vincent Delaitre and Yevgeny Kazakov. "Classifying \mathcal{ELH} ontologies in SQL databases". In: *6th International Workshop on OWL: Experiences and Directions (OWLED 2009)*. Vol. 529. CEUR-WS, 2009.

- [DL10] Tiziana D’Orazio and Marco Leo. “A review of vision-based systems for soccer video analysis”. In: *Pattern Recognition* 43 (8 2010), pp. 2911–2926.
- [DL14] Preeti R Dodwad and LMRJ Lobo. “A Context-Aware Recommender System Using Ontology Based Approach for Travel Applications”. In: *International Journal of Advanced Engineering and Nano Technology* 1 (2014).
- [dM11] Mathieu d’Aquin and Enrico Motta. “Watson, more than a Semantic Web search engine”. In: *Semantic Web* 2.1 (2011), pp. 55–63.
- [dNM11] Mathieu d’Aquin, Andriy Nikolov, and Enrico Motta. “Building SPARQL-enabled applications with android devices”. In: *10th International Semantic Web Conference (ISWC 2011)*. 2011, pp. 23–27.
- [DWE13] Adrian Dabrowski, Edgar R. Weippl, and Isao Echizen. “Framework Based on Privacy Policy Hiding for Preventing Unauthorized Face Image Processing”. In: *2013 IEEE International Conference on Systems, Man, and Cybernetics (SMC 2013)*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 455–461.
- [EFHIJMMNPS+09] Jérôme Euzenat, Alfio Ferrara, Laura Hollink, Antoine Isaac, Cliff Joslyn, Véronique Malaisé, Christian Meilicke, Andriy Nikolov, Juan Pane, Marta Sabou, et al. “Results of the ontology alignment evaluation initiative 2009”. In: *4th International Workshop on Ontology Matching (OM 2009)*. 2009.
- [EHTA11] Timofey Ermilov, Norman Heino, Sebastian Tramp, and Sören Auer. “OntoWiki Mobile Knowledge Management in Your Pocket”. In: *8th Extended Semantic Web Conference (ESWC 2011)*. Vol. 6643. Lecture Notes in Computer Science. Springer, 2011, pp. 185–199.
- [EKA14] Timofey Ermilov, Ali Khalili, and Sören Auer. “Ubiquitous Semantic Applications: A Systematic Literature Review”. In: *International Journal on Semantic Web Information Systems* 10.1 (2014), pp. 66–99.

- [ELLL12] Mikel Emaldi, Jon Lázaro, Xabier Laiseca, and Diego López-de-Ipiña. “LinkedQR: Improving Tourism Experience through Linked Data and QR Codes”. In: *6th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2012)*. 2012, pp. 371–378.
- [ES+07] Jérôme Euzenat, Pavel Shvaiko, et al. *Ontology matching*. Vol. 18. Springer, 2007.
- [ETM03] A. Ekin, A. M. Tekalp, and R. Mehrotra. “Automatic Soccer Video Analysis and Summarization”. In: *IEEE Transactions on Image Processing* 12.7 (2003), pp. 796–807.
- [Fer99] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.
- [FG97] Stan Franklin and Art Graesser. “Is It an Agent, or Just a Program?: A Taxonomy for Autonomous Agents”. In: *Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages (ECAI 1996)*. London, UK, UK: Springer-Verlag, 1997, pp. 21–35.
- [FG99] Y. Fan and S. Gauch. “Adaptive Agents for Information Gathering from Multiple, Distributed Information Sources”. In: *n AAAI Symposium on Intelligent Agents in Cyberspace*. 1999.
- [FGPSV14] Riccardo Falco, Aldo Gangemi, Silvio Peroni, David Shotton, and Fabio Vitali. “Modelling OWL Ontologies with Graffoo”. In: *11th Extended Semantic Web Conference (ESWC 2014)*. 2014, pp. 320–325.
- [FKS03] Ronald Fagin, Ravi Kumar, and D. Sivakumar. “Comparing top k lists”. In: *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*. Society for Industrial and Applied Mathematics, 2003, pp. 28–36.
- [FLR13] Nirosinie Fernando, Seng W. Loke, and Wenny Rahayu. “Mobile cloud computing: A survey”. In: *Future Generation Computer Systems* 29.1 (2013), pp. 84–106.

- [GA13] Jorge Gracia and Kartik Asooja. “Monolingual and cross-lingual ontology matching with CIDER-CL: evaluation report for OAEI 2013.” In: *8th International Workshop on Ontology Matching (OM 2013)*. 2013.
- [GBJRMPGK13] Rafael S. Gonçalves, Samantha Bail, Ernesto Jiménez-Ruiz, Nicolas Matentzoglou, Bijan Parsia, Birte Glimm, and Yevgeny Kazakov. “OWL Reasoner Evaluation (ORE) Workshop 2013 Results: Short Report”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. Ulm, Germany: CEUR-WS, 2013, pp. 1–18.
- [GDFLP02] Nicola Muscettola Gregory, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. “IDEA: Planning at the Core of Autonomous Reactive Agents”. In: *3rd International NASA Workshop on Planning and Scheduling for Space*. 2002.
- [GHMSW14] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. “HermiT: An OWL 2 Reasoner”. In: *Journal of Automated Reasoning* 53.3 (2014), pp. 245–269.
- [GI13] Andrey V. Grigorev and Alexander G. Ivashko. “TReasoner: System Description”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. Ulm, Germany: CEUR-WS, 2013, pp. 26–31.
- [GL06] Bugra Gedik and Ling Liu. “MobiEyes: A Distributed Location Monitoring Service Using Moving Location Queries”. In: *Transactions on Mobile Computing* 5.10 (2006), pp. 1384–1402.
- [GLdSMM07] Jorge Gracia, Vanessa Lopez, Mathieu d’Aquin, Marta Sabou, Enrico Motta, and Eduardo Mena. “Solving Semantic Ambiguity to Improve Semantic Web based Ontology Matching”. In: *2nd International Workshop on Ontology Matching (OM 2007)*. 2007.
- [GMPS13] Rafael S. Gonçalves, Nicolas Matentzoglou, Bijan Parsia, and Uli Sattler. “The Empirical Robustness of Description Logic Classification”. In: *26th International*

- Workshop on Description Logics (DL 2013)*. Vol. 1014. CEUR-WS, 2013, pp. 197–208.
- [GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *Journal of Web Semantics* 3.2–3 (2005), pp. 158–182.
- [GPZ04] Tao Gu, H.K. Pung, and Da Qing Zhang. “A middleware for building context-aware mobile services”. In: *IEEE 59th Vehicular Technology Conference (VTC 2004)*. Vol. 5. 2004, pp. 2656–2660.
- [Gru93] Thomas R. Gruber. “A translation approach to portable ontology specifications”. In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220.
- [Gru95] Thomas R. Gruber. “Toward Principles for the Design of Ontologies Used for Knowledge Sharing”. In: *International Journal of Human-Computer Studies* 43.5-6 (1995), pp. 907–928.
- [GTH06] Tom Gardiner, Dmitry Tsarkov, and Ian Horrocks. “Framework for an automated comparison of description logic reasoners”. In: *5th International Semantic Web Conference (ISWC 2006)*. Vol. 4273. Springer, 2006, pp. 654–667.
- [GWPZ04] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. “An ontology-based context model in intelligent environments”. In: *Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS 2004)*. 2004.
- [GWZTDZ11] Yue Gao, Meng Wang, Zheng-Jun Zha, Qi Tian, Qionghai Dai, and Naiyao Zhang. “Less is More: Efficient 3-D Object Retrieval With Query View Selection”. In: *IEEE Transactions on Multimedia* 13.5 (2011), pp. 1007–1018.
- [HB11] Matthew Horridge and Sean Bechhofer. “The OWL API: A Java API for OWL Ontologies”. In: *Semantic Web Journal* 2.1 (2011), pp. 11–21.
- [HHMW12] Volker Haarslev, Kay Hidde, Ralf Möller, and Michael Wessel. “The RacerPro Knowledge Representation and Reasoning System”. In: *Semantic Web Journal* 3 (3 2012), 267–277.

- [HKPPSR09] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. “OWL 2 web ontology language primer”. In: *W3C recommendation* 27.1 (2009), p. 123.
- [HM12] Abdeltawab M. Hendawi and Mohamed F. Mokbel. “Predictive spatio-temporal queries: a comprehensive survey and future directions”. In: *1st ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems (MobiGIS 2012)*. 2012, pp. 97–104.
- [HMFC14] Elena Hervalejo, Miguel A. Martínez-Prieto, Javier D. Fernández, and Óscar Corcho. “HDTourist: Exploring Urban Data on Android”. In: *13th International Semantic Web Conference (ISWC 2014)*. 2014, pp. 65–68.
- [Hor98] Ian Horrocks. “Using an Expressive Description Logic: FaCT or Fiction?”. In: *6th International Conference on the Principles of Knowledge Representation and Reasoning (KR 1998)*. Trento, Italy: Morgan Kaufmann Publishers, 1998, pp. 636–647.
- [HPSBTGD04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosz, and Mike Dean. *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. W3C Member Submission. World Wide Web Consortium, 2004.
- [HSS13] Benjamin Henne, Christian Szongott, and Matthew Smith. “SnapMe if You Can: Privacy Threats of Other Peoples’ Geo-tagged Media and What We Can Do About It”. In: *6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2013)*. Budapest, Hungary: ACM, 2013, pp. 95–106.
- [HWG07] Rachel Heck, Michael Wallick, and Michael Gleicher. “Virtual Videography”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications* 3.1 (2007).

- [HXL05] Haibo Hu, Jianliang Xu, and Dik Lun Lee. “A generic framework for monitoring continuous spatial queries over moving objects”. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*. ACM, 2005, pp. 479–490.
- [IBM11] Sergio Ilarri, Carlos Bobed, and Eduardo Mena. “An approach to process continuous location-dependent queries on moving objects with support for location granules”. In: *Journal of Systems and Software* 84.8 (2011), pp. 1327–1350.
- [Ila06] Sergio Ilarri. “LOQOMOTION: Processing of Continuous Location-Dependent Queries in Mobile Environments”. PhD thesis. University of Zaragoza, 2006, p. 247.
- [IIMS11] Sergio Ilarri, Arantza Illarramendi, Eduardo Mena, and Amit Sheth. “Semantics in Location-Based Services – Guest Editors’ Introduction for Special Issue”. In: *IEEE Internet Computing* 15.6 (2011), pp. 10–14.
- [IMI06] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. “Location-dependent queries in mobile contexts: distributed processing using mobile agents”. In: *IEEE Transactions on Mobile Computing* 5.8 (2006), pp. 1029–1043.
- [IMI08] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. “Using Cooperative Mobile Agents to Monitor Distributed and Dynamic Environments”. In: *Information Sciences: an International Journal* 178.9 (2008), pp. 2105–2127.
- [IMI10] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. “Location-dependent Query Processing: Where We Are and Where We Are Heading”. In: *ACM Computing Surveys* 42.3 (2010), 12:1–12:73.
- [IMIYLM12] Sergio Ilarri, Eduardo Mena, Arantza Illarramendi, Roberto Yus, Maider Laka, and Gorka Marcos. “A Friendly Location-Aware System to Facilitate the Work of Technical Directors When Broadcasting Sport Events”. In: *Mobile Information Systems* 8.1 (2012), pp. 17–43.

- [Ind] Ministry of Health & Family Welfare of the Government of India. *Accredited Social Health Activist (ASHA) Manual*. URL: <http://nrhm.gov.in/communitisation/asha/asha-manual.html>.
- [ITM06] Sergio Ilarri, Raquel Trillo, and Eduardo Mena. “SPRINGS: A Scalable Platform for Highly Mobile Agents in Distributed Computing Environments”. In: *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2012)*. IEEE Computer Society, 2006, pp. 633–637.
- [JCHWTL04] Xiaodong Jiang, Nicholas Y. Chen, Jason I. Hong, Kevin Wang, Leila Takayama, and James A. Landay. “Siren: Context-aware Computing for Firefighting”. In: *2nd International Conference on Pervasive Computing (PERVASIVE 2004)*. Springer, 2004, pp. 87–105.
- [JMSK10] Yves R Jean-Mary, E Patrick Shironoshita, and Mansur R Kabuka. “ASMOV: Results for OAEI 2010”. In: *5th International Workshop on Ontology Matching (OM 2010)*. 2010.
- [JT05] James Jayaputera and David Taniar. “Data retrieval for location-dependent queries in a multi-cell wireless environment”. In: *Mobile Information Systems 1.2* (2005), pp. 91–108.
- [Kaz09] Yevgeny Kazakov. “Consequence-driven reasoning for Horn *SHIQ* ontologies”. In: *21st International Joint Conference on Artificial intelligence (IJCAI 2009)*. 2009, pp. 2040–2045.
- [KD12] Jacek Kopecký and John Domingue. “ParkJam: Crowdsourcing Parking Availability Information with Linked Data”. In: *9th Extended Semantic Web Conference (ESWC 2012)*. 2012, pp. 381–386.
- [KDNCB13] Vikas Kumar, Vishal Dave, Rohan Nagrani, Sanjay Chaudhary, and Minal Bhise. “Crop cultivation information system on mobile devices”. In: *IEEE Global Humanitarian Technology Conference: South Asia Satellite (GHTC-SAS 2013)*. 2013, pp. 196–202.

- [KDSW15] Katharina Krombholz, Adrian Dabrowski, Matthew Smith, and Edgar Weippl. “Financial Cryptography and Data Security: FC 2015 International Workshops, BITCOIN, WAHC, and Wearable”. In: ed. by Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. Chap. Ok Glass, Leave Me Alone: Towards a Systematization of Privacy Enhancing Technologies for Wearable Computing, pp. 274–280.
- [Ken38] M. G. Kendall. “A New Measure of Rank Correlation”. In: *Biometrika* 30.1/2 (1938), pp. 81–93.
- [KFJ03] Lalana Kagal, Tim Finin, and Anupam Joshi. “A Policy Based Approach to Security for the Semantic Web”. In: *2nd International Semantic Web Conference (ISWC 2003)*. 2003, pp. 402–418.
- [KK13] Yevgeny Kazakov and Pavel Klinov. “Experimenting with ELK Reasoner on Android”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. CEUR-WS. 2013, pp. 68–74.
- [KKS14] Yevgeny Kazakov, Markus Krötzsch, and František Simančík. “The Incredible ELK”. In: *Journal of Automated Reasoning* 53 (1 2014), pp. 1–61.
- [Kle06] Thomas Kleemann. “Towards Mobile Reasoning”. In: *2006 International Workshop on Description Logics (DL 2006)*. 2006.
- [CLK12] Yong-Bin Kang, Yuan-Fang Li, and Shonali Krishnaswamy. “A Rigorous Characterization of Classification Performance - A Tale of Four Reasoners”. In: *1st International Workshop on OWL Reasoner Evaluation (ORE 2012)*. Vol. 858. CEUR-WS. 2012.
- [KLXWL05] Dazhou Kang, Jianjiang Lu, Baowen Xu, Peng Wang, and Yanhui Li. “A Framework of Checking Subsumption Relations Between Composite Concepts in Different Ontologies”. In: *Knowledge-Based Intelligent Information and Engineering Systems*. Vol. 3681. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005.

- [Kol11] Maheshkumar H. Kolekar. “Bayesian belief network based broadcast sports video indexing”. In: *Multimedia Tools and Applications* 54.1 (2011), pp. 27–54.
- [KWM11] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. “Activity Recognition Using Cell Phone Accelerometers”. In: *ACM SIGKDD Explorations Newsletter* 12.2 (2011), pp. 74–82.
- [KZ08] Wei-Shinn Ku and Roger Zimmermann. “Nearest neighbor queries with peer-to-peer data sharing in mobile environments”. In: *Pervasive and Mobile Computing* 4.5 (2008), pp. 775–788.
- [LB10] M. Lawley and C. Bousquet. “Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner”. In: *Australasian Ontology Workshop 2010 (AOW 2010)*. 2010, pp. 45–50.
- [Les99] Victor R. Lesser. “Cooperative Multiagent Systems: A Personal View of the State of the Art”. In: *IEEE Transactions on Knowledge and Data Engineering* 11.1 (1999), pp. 133–142.
- [LIS01] B. Li and M. Ibrahim Sezan. “Event detection and summarization in sports video”. In: *IEEE Workshop on Content-Based Access of Image and Video Libraries (CBAIVL 2001)*. 2001, pp. 132–138.
- [LJMKHS12] Youngki Lee, Younghyun Ju, Chulhong Min, Seungwoo Kang, Inseok Hwang, and Junehwa Song. “CoMon: cooperative ambience monitoring platform with continuity and benefit awareness”. In: *10th Int. Conf. on Mobile systems, applications, and services (MobiSys 2012)*. 2012.
- [LKFWB02] Qiong Liu, Don Kimber, Jonathan Foote, Lynn Wilcox, and John Boreczky. “FlySPEC: a multi-user video camera system with hybrid human and automatic control”. In: *10th ACM International Conference on Multimedia (MULTIMEDIA 2002)*. ACM Press, 2002, pp. 484–492.

- [LMLPCC10] N.D. Lane, E. Miluzzo, Hong Lu, D. Peebles, T. Choudhury, and A.T. Campbell. “A survey of mobile phone sensing”. In: *IEEE Communications Magazine* 48.9 (2010), pp. 140–150.
- [LMZBWPY07] Jing Lu, Li Ma, Lei Zhang, Jean-Sébastien Brunner, Chen Wang, Yue Pan, and Yong Yu. “SOR: A Practical System for Ontology Storage, Reasoning and Search”. In: *33rd International Conference on Very Large Data Bases (VLDB 2007)*. ACM, 2007, pp. 1402–1405.
- [LO99] Danny B. Lange and Mitsuru Oshima. “Seven good reasons for mobile agents”. In: *Communications of the ACM* 42 (3 1999), pp. 88–89.
- [LPQPH11] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth. “The linked sensor middleware—connecting the real world and the semantic web”. In: *10th International Semantic Web Conference (ISWC 2011)*. 2011.
- [LS97] Louisa Lam and Ching Y. Suen. “Application of Majority Voting to Pattern Recognition: An Analysis of Its Behavior and Performance”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans* 27.5 (1997), pp. 553–568.
- [Mac67] James MacQueen. “Some methods for classification and analysis of multivariate observations”. In: *15th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297. 1967.
- [MBK06] Christopher J. Matheus, Kenneth Baclawski, and Mieczysław M. Kokar. “BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules”. In: *2nd International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML 2006)*. 2006, pp. 67–74.
- [McB02] Brian McBride. “Jena: A Semantic Web Toolkit”. In: *IEEE Internet Computing* 6.6 (2002), pp. 55–59.
- [Men12] Julian Mendez. “jcel: A Modular Rule-based Reasoner”. In: *1st International Workshop on OWL Reasoner Evaluation (ORE 2012)*. Vol. 858. CEUR-WS. 2012.

- [MGK14] Tamás Matuszka, Gergo Gombos, and Attila Kiss. “mSWB: Towards a Mobile Semantic Web Browser”. In: *11th International Conference on Mobile Web Information Systems (MobiWIS 2014)*. 2014, pp. 165–175.
- [MHK12] Boris Motik, Ian Horrocks, and Su Myeon Kim. “Delta-reasoner: A Semantic Web reasoner for an intelligent mobile platform”. In: *21st World Wide Web Conference (WWW 2012)*. 2012, pp. 63–72.
- [MHLN06] Felix Müller, Michael Hanselmann, Thorsten Liebig, and Olaf Noppens. “A Tableaux-based Mobile DL Reasoner - An Experience Report”. In: *19th International Workshop on Description Logics (DL 2006)*. Vol. 189. CEUR-WS. 2006.
- [Mil56] George A. Miller. “The magical number seven, plus or minus two: Some limits on our capacity for processing information”. In: *Psychological Review* 63.2 (1956), 81–97.
- [Mil95] George A. Miller. “WordNet: A Lexical Database for English”. In: *Communications ACM* 38.11 (1995), pp. 39–41.
- [MK14] Tamás Matuszka and Attila Kiss. “Alive Cemeteries with Augmented Reality and Semantic Web Technologies”. In: *International Journal of Computer, Information Science and Engineering* 8 (2014), pp. 32–36.
- [MMM+04] Frank Manola, Eric Miller, Brian McBride, et al. “RDF primer”. In: *W3C recommendation* 10.1-107 (2004), p. 6.
- [MN07] Frank McCown and Michael L. Nelson. “Agreeing to disagree: Search engines and their public interfaces”. In: *7th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL 2007)*. ACM, 2007, pp. 309–318.
- [MRM04] Nikola Mitrovic, Jose-Alberto Royo, and Eduardo Mena. “ADUS: Indirect Generation of User interfaces on Wireless Devices”. In: *7th International Workshop Mobility on Databases and Distributed Systems (MDDS 2004)*. IEEE Computer Society, 2004, pp. 662–666.

- [MS02] Alexander Maedche and Steffen Staab. “Measuring Similarity Between Ontologies”. In: *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2002)*. 2002.
- [MS05] B. Motik and R. Studer. “KAON2—A Scalable Reasoning Tool for the Semantic Web”. In: *2nd European Semantic Web Conference (ESWC 2005)*. 2005.
- [NBWMPW14] Hai H. Nguyen, David E. Beel, Gemma Webster, Chris Mellish, Jeff Z. Pan, and Claire Wallace. “CURIOS Mobile: Linked Data Exploitation for Tourist Mobile Apps in Rural Areas”. In: *4th Joint International Conference on Semantic Technology (JIST 2014)*. 2014, pp. 129–145.
- [NTB09] Naoko Nitta, Yoshimasa Takahashi, and Noboru Babaguchi. “Automatic personalized video abstraction for sports videos using metadata”. In: *Multimedia Tools and Applications* 41.1 (2009), pp. 1–25.
- [Nwa96] Hyacinth S Nwana. “Software agents: An overview”. In: *The knowledge engineering review* 11.03 (1996), pp. 205–244.
- [OKCM12] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. “Evaluation of Android Dalvik virtual machine”. In: *10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2012)*. 2012.
- [ONMRS12] Vito Claudio Ostuni, Tommaso Di Noia, Roberto Mirizzi, Davide Romito, and Eugenio Di Sciascio. “Cinemappy: a Context-aware Mobile App for Movie Recommendations boosted by DBpedia”. In: *International Workshop on Semantic Technologies meet Recommender Systems & Big Data*. 2012, pp. 37–48.
- [Pan05] Zhengxiang Pan. “Benchmarking DL Reasoners Using Realistic Ontologies”. In: *Workshop on OWL: Experiences and Directions (OWLED 2005)*. Vol. 188. CEUR-WS, 2005.

- [PLMC09] Han-Saem Park, Soojung Lim, Jun-Ki Min, and Sung-Bae Cho. “Optimal View Selection and Event Retrieval in Multi-Camera Office Environment”. In: *Multisensor Fusion and Integration for Intelligent Systems*. Lecture Notes in Electrical Engineering 35 (2009). Ed. by Hern-soo Hahn, Hanseok Ko, and Sukhan Lee, pp. 45–53.
- [PM09] Evan W Patton and Deborah L McGuinness. “The mobile wine agent: Pairing wine with the social Semantic Web”. In: *2nd Workshop on Social Data on the Web (SDoW 2009)*. 2009.
- [PM14] Evan M. Patton and Deborah L. McGuinness. “A Power Consumption Benchmark for Reasoners on Mobile Devices”. In: *13th International Semantic Web Conference (ISWC 2014)*. Vol. 8796. Lecture Notes in Computer Science. Springer, 2014, pp. 409–424.
- [PPRH10] Danh Le Phuoc, Josiane Xavier Parreira, Vinny Reynolds, and Manfred Hauswirth. “RDF On the Go: RDF Storage and Query Processor for Mobile Devices”. In: *9th International Semantic Web Conference (ISWC 2010)*. 2010.
- [PS+08] Eric Prud’hommeaux, Andy Seaborne, et al. “SPARQL query language for RDF”. In: *W3C recommendation* 15 (2008).
- [PWLZB07] Amir Padovitz, Seng Wai Loke, Arkady Zaslavsky, and Bernard Burg. “Verification of uncertain context based on a theory of context spaces”. In: *International Journal of Pervasive Computing and Communications* 3.1 (2007), pp. 30–56.
- [PYDFMJ14] Primal Pappachan, Roberto Yus, Prajit Kumar Das, Tim Finin, Eduardo Mena, and Anupam Joshi. “A Semantic Context-Aware Privacy Model for FaceBlock”. In: *2nd International Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014), co-located with the 13th International Semantic Web Conference (ISWC 2014)*. Vol. 1316. CEUR-WS, 2014.

- [PYJF14] Primal Pappachan, Roberto Yus, Anupam Joshi, and Tim Finin. “Rafiki: A Semantic and Collaborative Approach to Community Health-care in Underserved Areas”. In: *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2014)*. 2014, pp. 322–331.
- [RAMC04] Anand Ranganathan, Jalal Al-Muhtadi, and Roy H. Campbell. “Reasoning About Uncertain Contexts in Pervasive Computing Environments”. In: *IEEE Pervasive Computing* 3.2 (2004), pp. 62–70.
- [RGKR07] Jonathan Raper, Georg Gartner, Hassan Karimi, and Chris Rizos. “Applications of Location-based Services: A Selected Review”. In: *Journal of Location Based Services* 1.2 (2007), pp. 89–111.
- [RHGL01] Yong Rui, Liwei He, Anoop Gupta, and Qiong Liu. “Building an intelligent camera management system”. In: *9th ACM International Conference on Multimedia (MULTIMEDIA 2001)*. ACM Press, 2001, pp. 2–11.
- [RHOM98] Yong Rui, Thomas S. Huang, Michael Ortega, and Sharad Mehrotra. “Relevance feedback: A power tool for interactive content-based image retrieval”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 8.5 (1998), pp. 644–655.
- [RM00] Bradley J. Rhodes and Pattie Maes. “Just-in-time Information Retrieval Agents”. In: *IBM Systems Journal* 39.3-4 (2000), pp. 685–704.
- [RNCME96] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.
- [RS07] Chantal Reynaud and Brigitte Safar. “Exploiting WordNet as Background Knowledge”. In: *2nd International Workshop on Ontology Matching (OM 2007)*. 2007.

- [RSFIBS14] Michele Ruta, Floriano Scioscia, Danilo De Filippis, Saverio Ieva, Mario Binetti, and Eugenio Di Sciascio. “A Semantic-enhanced Augmented Reality Tool for OpenStreetMap POI Discovery”. In: *Transportation Research Procedia* 3 (2014), pp. 479–488.
- [RSILS12] Michele Ruta, Floriano Scioscia, Saverio Ieva, Giuseppe Loseto, and Eugenio Di Sciascio. “Semantic Annotation of OpenStreetMap Points of Interest for Mobile Discovery and Navigation”. In: *1st International Conference on Mobile Services (MS 2012)*. 2012, pp. 33–39.
- [RSLDF12] Kurt Rothermel, Stephan Schnitzer, Ralph Lange, Frank Dürr, and Tobias Farrell. “Context-aware and quality-aware algorithms for efficient mobile object management”. In: *Pervasive and Mobile Computing* 8.1 (2012), pp. 131–146.
- [RSS11] Michele Ruta, Eugenio Di Sciascio, and Floriano Scioscia. “Concept abduction and contraction in semantic-based P2P environments”. In: *Web Intelligence and Agent Systems* 9 (3 2011), pp. 179–207.
- [RSSGL12] Michele Ruta, Floriano Scioscia, Eugenio Di Sciascio, Filippo Gramegna, and Giuseppe Loseto. “Mini-ME: the Mini Matchmaking Engine”. In: *1st OWL Reasoner Evaluation Workshop (ORE 2012)*. 2012.
- [RT98] European Institute for Research and Strategic Studies in Telecommunications. *Agent based computing – a booklet for executives*. 1998.
- [SAW94] B. Schilit, N. Adams, and R. Want. “Context-Aware Computing Applications”. In: *1st Workshop on Mobile Computing Systems and Applications (WMCSA 1994)*. 1994, pp. 85–90.
- [SB05] Nikhil V. Shirahatti and Kobus Barnard. “Evaluating image retrieval”. In: *Conference on Computer Vision and Pattern Recognition (CVPR 2005)*. Vol. 1. IEEE Computer Society, 2005, pp. 955–961.

- [SCC10] Jang-Ping Sheu, Guey-Yun Chang, and Chiung-Hung Chen. “A Distributed Taxi Hailing Protocol in Vehicular Ad-Hoc Networks”. In: *71st IEEE Vehicular Technology Conf. (VTC 2010)*. IEEE Computer Society, 2010, pp. 1–5.
- [SdM08] Marta Sabou, Mathieu d’Aquin, and Enrico Motta. “SCARLET: semantic relation discovery by harvesting online ontologies”. In: *5th European Semantic Web Conference (ESWC 2008)*. 2008, pp. 1–5.
- [SE13] Pavel Shvaiko and Jérôme Euzenat. “Ontology Matching: State of the Art and Future Challenges”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.1 (2013), pp. 158–176.
- [Ser13] Barış Sertkaya. “The ELepHant Reasoner System Description”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. Ulm, Germany: CEUR-WS, 2013, pp. 87–93.
- [SK05] Alex Sinner and Thomas Kleemann. “KRHyper In Your Pocket”. In: *20th International Conference on Automated Deduction (CADE-20)*. Vol. 3632. Lecture Notes in Computer Science. Springer, 2005, 452–458.
- [SK08] Luke Steller and Shonali Krishnaswamy. “Pervasive Service Discovery: mTableaux Mobile Reasoning”. In: *International Conference on Semantic Systems (I-SEMANTICS 2008)*. Springer, 2008, pp. 620–625.
- [SK87] Lawrence Sirovich and M. Kirby. “Low-dimensional procedure for the characterization of human faces”. In: *Journal of the Optical Society of America A* 4.3 (1987), pp. 519–524.
- [SKG09] Luke Steller, Shonali Krishnaswamy, and Mohamed Medhat Gaber. “Enabling Scalable Semantic Reasoning for Mobile Services”. In: *International Journal on Semantic Web and Information Systems* 5.2 (2009), pp. 91–116.

- [SKH11] Frantisek Simanck, Yevgeny Kazakov, and Ian Horrocks. “Consequence-Based Reasoning beyond Horn Ontologies”. In: *22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*. 2011, pp. 1093–1098.
- [SLG14] Andreas Steigmiller, Thorsten Liebig, and Birte Glimm. “Konclude: System Description”. In: *Journal of Web Semantics* 27–28 (2014), pp. 78–85.
- [SM06] Marta Sabou and Enrico Motta. “Using the Semantic Web as background knowledge for ontology mapping”. In: *1st International Workshop on Ontology Matching (OM 2006)*. 2006.
- [Smi77] Reid G. Smith. “The Contract Net: A Formalism for the Control of Distributed Problem Solving”. In: *5th International Joint Conference on Artificial Intelligence*. Cambridge, MA. 1977, p. 472.
- [SMMSG07] Jeremy Schiff, Marci Meingast, Deirdre K. Mulligan, Shankar Sastry, and Ken Goldberg. “Respectful cameras: detecting visual markers in real-time to address privacy concerns”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007)*. 2007, pp. 971–978.
- [Sow99] John F Sowa. *Knowledge representation: logical, philosophical, and computational foundations*. Course Technology, 1999.
- [SP07] Evren Sirin and Bijan Parsia. “SPARQL-DL: SPARQL Query for OWL-DL”. In: *OWLED*. Vol. 258. 2007.
- [SPCGKK07] Evren Sirin, Bijan Parsia, Bernardo Cuenca-Grau, Aditya Kalyanpur, and Yarden Katz. “Pellet: A practical OWL-DL reasoner”. In: *Journal of Web Semantics* 5.2 (2007), pp. 51–53.
- [SPS09] Heiner Stuckenschmidt, Christine Parent, and Stefano Spaccapietra. *Modular ontologies: concepts, theories and techniques for knowledge modularization*. Vol. 5445. Springer, 2009.

- [SS02] Srividhya Sathyanath and Ferat Sahin. “AISIMAM—An Artificial immune system based intelligent multi agent model and its application to a mine detection problem”. In: *1st International Conference on Artificial Immune Systems (ICARIS 2002)*. 2002, pp. 22–31.
- [SS12] SR Shrivastava and PS Shrivastava. “Evaluation of trained Accredited Social Health Activist (ASHA) workers regarding their knowledge, attitude and practices about child health.” In: *Rural & Remote Health* 12.4 (2012).
- [SSD12] Weihong Song, Bruce Spencer, and Weichang Du. “WS-Reasoner: a Prototype Hybrid Reasoner for \mathcal{ALCHOI} Ontology Classification using a Weakening and Strengthening Approach”. In: *1st International Workshop on OWL Reasoner Evaluation (ORE 2012)*. Vol. 858. CEUR-WS. 2012.
- [SSD13] Weihong Song, Bruce Spencer, and Weichang Du. “A Transformation Approach for Classifying $\mathcal{ALCHI}(\mathbf{D})$ Ontologies with a Consequence-based \mathcal{ALCH} Reasoner”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. Ulm, Germany: CEUR-WS, 2013, pp. 39–45.
- [SSK05] Giorgos Stoilos, Giorgos Stamou, and Stefanos Kollias. “A string metric for ontology alignment”. In: *4th International Semantic Web Conference (ISWC 2005)*. 2005, pp. 624–637.
- [SSLPMC13] Fuming Shih, Oshani Seneviratne, Ilaria Liccardi, Evan Patton, Patrick Meier, and Carlos Castillo. “Democratizing Mobile App Development for Disaster Management”. In: *Joint Workshop on AI Problems and Approaches for Intelligent Environments and Workshop on Semantic Cities (AIIP 2013)*. 2013, pp. 39–42.
- [SU11] Kimiaki Shirahama and Kuniaki Uehara. “Query by Virtual Example: Video Retrieval Using Example Shots Created by Virtual Reality Techniques”. In: *6th International Conference on Image and Graphics (ICIG 2011)*. IEEE Computer Society, 2011, pp. 829–834.

- [SV04] Jochen Schiller and Agnes Voisard. *Location-Based Services*. Morgan Kaufmann, 2004.
- [SVK10] Vassilis Spiliopoulos, George A Vouros, and Vangelis Karkaletsis. “On the discovery of subsumption relations for the alignment of ontologies”. In: *Web Semantics: Science, Services and Agents on the World Wide Web* 8.1 (2010), pp. 69–88.
- [SVV08] Vassilis Spiliopoulos, Alexandros G Valarakos, and George A Vouros. “CSR: discovering subsumption relations for the alignment of ontologies”. In: *5th European Semantic Web Conference (ESWC 2008)*. 2008, pp. 418–431.
- [SWW06] Kindra Serr, Thomas Windholz, and Keith Weber. “Comparing GPS receivers: a field study”. In: *URISA Journal* 18.2 (2006).
- [SXJMTL11] Alexander Savelyev, Sen Xu, Krzysztof Janowicz, Christoph Mülligann, Jim Thatcher, and Wei Luo. “Volunteered geographic services: developing a linked data driven location-based service”. In: *2011 International Workshop on Spatial Semantics and Ontologies (SSO 2011)*. 2011, pp. 25–31.
- [TFAEA11] Sebastian Tramp, Philipp Frischmuth, Natanael Arndt, Timofey Ermilov, and Sören Auer. “Weaving a Distributed, Semantic Social Network for Mobile Users”. In: *8th Extended Semantic Web Conference (ESWC 2011)*. Vol. 6643. Lecture Notes in Computer Science. Springer, 2011, pp. 200–214.
- [TFPL04] Yufei Tao, Christos Faloutsos, Dimitris Papadias, and Bin Liu. “Prediction and Indexing of Moving Objects with Unknown Motion Patterns”. In: *ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*. ACM Press, 2004, pp. 611–622.
- [TH06] Dmitry Tsarkov and Ian Horrocks. “FaCT++ description logic reasoner: system description”. In: *3rd International Joint Conference on Automated Reasoning (IJCAR 2006)*. 2006.

- [Tho10] Bart Thomee. “A picture is worth a thousand words – Content-based image retrieval techniques”. PhD thesis. Leiden University (Germany), 2010.
- [TIM07] Raquel Trillo, Sergio Ilarri, and Eduardo Mena. “Comparison and Performance Evaluation of Mobile Agent Platforms”. In: *3rd International Conference on Autonomic and Autonomous Systems (ICAS 2007)*. IEEE Computer Society, 2007.
- [TJV13] Bjørnar Tessem, Bjarte Johansen, and Csaba Veres. “Mobile Location-Driven Associative Search in DBpedia with Tag Clouds”. In: *9th International Conference on Semantic Systems (I-SEMANTICS 2013)*. 2013, pp. 6–10.
- [TKS12] Gerald Töpper, Magnus Knuth, and Harald Sack. “DBpedia Ontology Enrichment for Inconsistency Detection”. In: *8th International Conference on Semantic Systems (I-SEMANTICS 2012)*. 2012, pp. 33–40.
- [TP12] Dmitry Tsarkov and Ignazio Palmisano. “Chainsaw: a Metareasoner for Large Ontologies”. In: *1st International Workshop on OWL Reasoner Evaluation (ORE 2012)*. Vol. 858. CEUR-WS. 2012.
- [TPIBM11] Raquel Trillo, Laura Po, Sergio Ilarri, Sonia Bergamaschi, and Eduardo Mena. “Using Semantic Techniques to Access Web Data”. In: *Information Systems. Special Issue on Semantic Integration of Data, Multimedia, and Services* 36.2 (2011), pp. 117–133.
- [TPR10] Edward Thomas, Jeff Z. Pan, and Yuan Ren. “TrOWL: Tractable OWL 2 Reasoning Infrastructure”. In: *7th Extended Semantic Web Conference (ESWC 2010)*. 2010, pp. 431–435.
- [TPRT11] Rémi Tournaire, Jean-Marc Petit, Marie-Christine Rousset, and Alexandre Termier. “Discovery of Probabilistic Mappings between Taxonomies: Principles and Experiments”. In: *Journal of Data Semantics* 15 (2011), pp. 66–101.

- [TPS02] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. “Continuous Nearest Neighbor Search”. In: *28th International Conference on Very Large Data Bases*. VLDB 2002. Hong Kong, China: VLDB Endowment, 2002, pp. 287–298.
- [TS03] Goce Trajcevski and Peter Scheuermann. “Triggers and continuous queries in moving objects databases”. In: *14th International Workshop on Database and Expert Systems Applications*. 2003, pp. 905–910.
- [TWHC04] Goce Trajcevski, Ouri Wolfson, Klaus Hinrichs, and Sam Chamberlain. “Managing Uncertainty in Moving Objects Databases”. In: *ACM Transactions on Database Systems* 29.3 (2004), pp. 463–507.
- [VDV14] Silviu Vert, Bogdan Dragulescu, and Radu Vasiu. “LOD4AR: Exploring Linked Open Data with a Mobile Augmented Reality Web Application”. In: *13th International Semantic Web Conference (ISWC 2014)*. 2014, pp. 185–188.
- [VNP15] Edgaras Valincius, Hai H. Nguyen, and Jeff Z. Pan. “A Power Consumption Benchmark Framework for Ontology Reasoning on Android Devices”. In: *4th International Workshop on OWL Reasoner Evaluation (ORE)*. 2015, pp. 80–86.
- [WCT10] William Van Woensel, Sven Casteleyn, and Olga De Troyer. “Applying Semantic Web Technology in a Mobile Setting: The Person Matcher”. In: *10th International Conference on Web Engineering (ICWE 2010)*. 2010, pp. 506–509.
- [Wer03] Christph Wernhard. *System Description: KRHyper*. Tech. rep. Fachberichte Informatik 14-2003, Universität Koblenz-Landau, 2003.
- [WHAA14] William Van Woensel, Newres Al Haider, Ahmad Ahmad, and Syed S. R. Abidi. “A Cross-Platform Benchmark Framework for Mobile Semantic Web Reasoning Engines”. In: *13th International Semantic Web Conference (ISWC 2014)*. Vol. 8796. Lecture Notes in Computer Science. Springer, 2014, pp. 389–408.

- [Whi04] Stephen A White. “Introduction to BPMN”. In: *IBM Cooperation 2* (2004).
- [WRSOS05] Max L. Wilson, Alistair Russell, Daniel A. Smith, Alistair Owens, and Monica M. C. Schraefel. “mSpace Mobile: A mobile application for the Semantic Web”. In: *2nd International Workshop on Interaction Design and the Semantic Web*. 2005.
- [WXCLT08] Jinjun Wang, Changsheng Xu, Engsiong Chng, Hanqing Lu, and Qi Tian. “Automatic composition of broadcast sports video”. In: *Multimedia Systems* 14.4 (2008), pp. 179–193.
- [WYZ11] Anna Wu, Xin Yan, and Xiaolong (Luke) Zhang. “Geo-tagged Mobile Photo Sharing in Collaborative Emergency Management”. In: *2011 Visual Information Communication - International Symposium (VINCI 2011)*. 2011, 7:1–7:8.
- [WZL13] Waskitho Wibisono, Arkady Zaslavsky, and Sea Ling. “Situation-awareness and reasoning using uncertain context in mobile peer-to-peer environments”. In: *International Journal of Pervasive Computing and Communications* 9.1 (2013), pp. 52–71.
- [XCDS02] Lexing Xie, Shih-Fu Chang, A. Divakaran, and Huifang Sun. “Structure analysis of soccer video with hidden Markov models”. In: *2002 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2002)*. Vol. 4. 2002, pp. IV–4096–IV–4099.
- [XWLZ08] Changsheng Xu, Jinjun Wang, Hanqing Lu, and Yifan Zhang. “A Novel Framework for Semantic Annotation and Personalized Retrieval of Sports Video”. In: *IEEE Transactions on Multimedia* 10.3 (2008), pp. 421–436.
- [XZZRLH08] Changsheng Xu, Yi-Fan Zhang, Guangyu Zhu, Yong Rui, Hanqing Lu, and Qingming Huang. “Using Web-cast Text for Semantic Event Detection in Broadcast Sports Video”. In: *IEEE Transactions on Multimedia* 10.7 (2008), pp. 1342–1355.

- [YAMIII11] Roberto Yus, David Anton, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. “MultiCAMBA: A System to Assist in the Broadcasting of Sport Events”. In: *8th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQ-uitous 2011)*. Vol. 104. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (LNICST). Springer, 2011, pp. 238–242.
- [YBBM15] Roberto Yus, Fernando Bobillo, Carlos Bobed, and Eduardo Mena. “The OWL Reasoner Evaluation Goes Mobile”. In: *4th International Workshop on OWL Reasoner Evaluation (ORE 2015)*. Vol. 1387. CEUR-WS, 2015, pp. 38–45.
- [YBEBM13] Roberto Yus, Carlos Bobed, Guillermo Esteban, Fernando Bobillo, and Eduardo Mena. “Android goes Semantic: DL Reasoners on Smartphones”. In: *2nd International Workshop on OWL Reasoner Evaluation (ORE 2013)*. Vol. 1015. CEUR-WS, 2013, pp. 46–52.
- [YCPSA11] Zhixian Yan, Dipanjan Chakraborty, Christine Parent, Stefano Spaccapietra, and Karl Aberer. “SeMiTri: a framework for semantic annotation of heterogeneous trajectories”. In: *14th International Conference on Extending Database Technology (EDBT 2011)*. ACM, 2011, pp. 259–270.
- [YIM15] Roberto Yus, Sergio Ilarri, and Eduardo Mena. “Real-time Selection of Video Streams for Live TV Broadcasting Based on Query-by-Example Using a 3D Model”. In: *Multimedia Tools and Applications* 74.8 (2015), pp. 2659–2685.
- [YM15a] Roberto Yus and Eduardo Mena. “Continuous Processing of Real-Time Multimedia Requests Using Semantic Techniques”. In: *13th International Conference on Advances in Mobile Computing and Multimedia (MoMM 2015)*. ACM, 2015, pp. 216–220.
- [YM15b] Roberto Yus and Eduardo Mena. “Cooperative Network of Mobile Agents to Remotely Process User Information

- Requests". In: *2015 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2015)*. IEEE, 2015, pp. 227–228.
- [YM15c] Roberto Yus and Eduardo Mena. "Emergency Management Using SHERLOCK". In: *13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2015)*. ACM, 2015, pp. 495–495.
- [YM15d] Roberto Yus and Eduardo Mena. "Mobile Endpoints: Accessing Dynamic Information from Mobile Devices". In: *14th International Semantic Web Conference (ISWC 2015)*. Vol. 1486. CEUR-WS, 2015, p. 4.
- [YMBII11] Roberto Yus, Eduardo Mena, Jorge Bernad, Sergio Ilarri, and Arantza Illarramendi. "Location-Aware System Based on a Dynamic 3D Model to Help in Live Broadcasting of Sport Events". In: *19th ACM International Conference on Multimedia (ACMMM 2011)*. ACM, 2011, pp. 1005–1008.
- [YMI13] Roberto Yus, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. "SHERLOCK: A System for Location-Based Services in Wireless Environments Using Semantics". In: *22nd International World Wide Web Conference (WWW 2013)*. ACM, 2013, pp. 301–304.
- [YMI14] Roberto Yus, Eduardo Mena, Sergio Ilarri, and Arantza Illarramendi. "SHERLOCK: Semantic Management of Location-Based Services in Wireless Environments". In: *Pervasive and Mobile Computing* 15 (2014), pp. 87–99.
- [YMIIB15] Roberto Yus, Eduardo Mena, Sergio Ilarri, Arantza Illarramendi, and Jorge Bernad. "MultiCAMBA: A System for Selecting Camera Views in Live Broadcasting of Sport Events Using a Dynamic 3D Model". In: *Multimedia Tools and Applications* 74.11 (2015), pp. 4059–4090.
- [YMSB15] Roberto Yus, Eduardo Mena, and Enrique Solano-Bes. "Generic Rules for the Discovery of Subsumption Relationships based on Ontological Contexts". In: *2015 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2015)*. IEEE, 2015, pp. 309–312.

- [YP15] Roberto Yus and Primal Pappachan. “Are Apps Going Semantic? A Systematic Review of Semantic Mobile Applications”. In: *1st International Workshop on Mobile Deployment of Semantic Technologies (MoDeST 2015)*, co-located with the *14th International Semantic Web Conference (ISWC 2015)*, Bethlehem, PA (USA). Vol. 1506. CEUR-WS, 2015, pp. 2–13.
- [YPDFJM14] Roberto Yus, Primal Pappachan, Prajit Kumar Das, Tim Finin, Anupam Joshi, and Eduardo Mena. “Semantics for Privacy and Shared Context”. In: *2nd International Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2014)*, co-located with the *13th International Semantic Web Conference (ISWC 2014)*. Vol. 1316. CEUR-WS, 2014.
- [YPDMJF14] Roberto Yus, Primal Pappachan, Prajit Kumar Das, Eduardo Mena, Anupam Joshi, and Tim Finin. “Face-Block: Privacy-Aware Pictures for Google Glass”. In: *12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys 2014)*. 2014, pp. 366–366.
- [YYY13] Yakup Yildirim, Adnan Yazici, and Turgay Yilmaz. “Automatic Semantic Content Extraction in Videos Using a Fuzzy Ontology and Rule-Based Model”. In: *IEEE Transactions on Knowledge and Data Engineering* 25.1 (2013), pp. 47–61.
- [ZHXXGY07] Guangyu Zhu, Qingming Huang, Changsheng Xu, Liyuan Xing, Wen Gao, and Hongxun Yao. “Human Behavior Analysis for Highlight Ranking in Broadcast Racket Sports Video”. In: *IEEE Transactions on Multimedia* 9.6 (2007), pp. 1167–1182.
- [ZL01] Baihua Zheng and Dik Lun Lee. “Semantic Caching in Location-Dependent Query Processing”. In: *7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*. Springer, 2001, pp. 97–116.
- [ZRCH08] Cha Zhang, Yong Rui, Jim Crawford, and Li-Wei He. “An Automated End-to-end Lecture Capture and Broadcasting System”. In: *ACM Transactions on Multime-*

dia Computing, Communications, and Applications 4.1 (2008), 6:1–6:23.

[And] Android goes semantic! <http://sid.cps.unizar.es/AndroidSemantic>.

[Dat] Data Never Sleeps. <https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/>.