‘If you wait to do everything until you’re sure it’s right, you’ll probably never do much of anything.”

- Win Borden on spontaneity
Abstract

This project explores mobile interactions with pervasive services (e.g. provided by printers, public displays, multimedia stations, workstations, etc.) supported by spatial information. It intends to ease mobile interactions by proposing the End-to-End Service Discovery model which bootstraps spontaneity. In this context we address spontaneity by helping users to identify and consistently interact with services their mobile device did not have prior knowledge of.

We start by helping the user to identify services in the environment. The Spatial Discovery layer uses relative positioning information to provide a novel and more natural representation of available co-located services. This spatial context is provided by an ad-hoc network of sensors. Once a service is identified the two remaining layers take care of the service installation and configuration in order to enable the interaction. The Network and Service Discovery layer ensures an automated connection of the mobile device with the service providers. The Interaction and Interoperability layer offers to consume the previously unknown services in a uniform and dynamic manner.

Furthermore, we present a concrete implementation of the model in the RelateGateways architecture. The Spatial Discovery is implemented in the gateways user interface which supports end-user discovery of services dynamically encountered in their immediate environment. The interface indicates available services through small widgets arranged at the periphery of the screen based on a compass metaphor. To gather spatial context, we propose an extension of Relate, an ad-hoc sensor network providing relative positioning information. The two other layers are supported by a Service Oriented Architecture based on dynamic discovery and a novel type of mobile code that enables the seamless installation and configuration of newly discovered services.

Additionally, the layers’ implementation is integrated into a single tool supporting the rapid prototyping of mobile spatial interactions. Finally, the implementation was used to support an evaluation of the model and interaction techniques in a user study presented in the last part of this report.

Keywords: Mobile interactions, spatial context, service and network discovery, invocation, interoperability, pervasive services, service oriented architecture, mobile code, visualization of services.
Preamble

Foreword

Like any other Master Thesis this document reports on the work both in terms of new contributions and description of the developed system. However, it reports on a Master Thesis for the Master in Computer Science at the University of Fribourg in Switzerland achieved at Lancaster University in the northern United Kingdom. As a consequence, it aims to describe a bit further the tasks achieved during the stay abroad in order to sustain the work’s evaluation in Switzerland. Furthermore as formulated by the research team at Lancaster University, it describes the system and documents the software architecture in greater details in order to support future work and extensions.

Acknowledgments

I would like to use these first few lines to express my gratitude to a number of colleagues, supervisors and friends who supported me on this project.
I shall start with my supervisor at Lancaster University: Hans Gellersen who warmly welcomed me in his team. Hans was really what I would call a great supervisor. Giving his time to define and refine the projects, sharing his ideas and listening to mine. He treated me as a regular member of his research group from the beginning on which was really touching.
I’ll continue with my partner on the first part of this project: Sara Streng from LMU in Munich (Germany). Together we imagined and built the gateways user interface and ran two user studies. Her skills in the HCI (Human Computer Interactions) domain helped me to learn a handful of things in a topic relatively new to me.
Next are my colleagues at Lancaster University who helped us on this project: Carl Fischer for being a genuine Swiss-army knife: always open to help and skilled in so many subjects that I shall not list here for the sake of space. It is worth noting that Carl will take the lead of the future work on RelateGateways at Lancaster University. Roswitha Gostner for her skills in spatial visualization and user studies. Matthew Oppenheim for his amazing hardware skills, Vasughii Sundraamoorthy for her expertise in service discovery system (and conference organizing) as well as Rene Mayrhofer for his help on the integration of Relate.
Next are all the other members of the Ubicomp Group: Dave, Yukang, Momo, Nick, Flo, Urs, Henoc, Mike, Enrico, Jamie. I shall not forget Bine, Andre and Andrew. When I
arrived it was a hard job to get to know so many new colleagues, now that I am about to leave it is quite hard to leave so many friends.

On the Swiss side I'd like to thank the Diva Group for supporting my “weird” idea of going under rain-showers for eight month. In particular thanks to Denis Lalanne and Rolf Ingold for supervising my project on the Swiss side as well as Karin Altorfer for sending me chocolate once in a while.

A special thank you to Olivier Liechti who, once again, took me under his wing and introduced me to Hans. It going to be hard to forget how much you already did for me Olivier! Thanks a well to Patrik for his wise advices in software engineering.

Finally, I want to express my gratitude to the Hasler Foundation in Switzerland. Their financial support was of great help in a country where paying your rent makes you feel like asking whether it does include a castle in Spain for the same price.

My deepest thanks go for my loving family whose support was a helping hand wherever I was and wherever I shall be tomorrow...

Workload and Repartition of the Different Parts

This Master Thesis presents a conceptual model as well as its concrete implementation. I imagined the model and led the concrete implementation of the three layers it is composed of. However, I would not have been able to achieve it without constructing my work on others work as well as collaborating with other researchers. The table below clarifies this by stating, as transparently as possible, my role and tasks for each deliverable described in this work. The percentage column (%) quantifies my own work for each output as a fraction of the overall work for this task. The last column contains the name of the other participants for each part (i.e. other than myself): S.S stands for Sara Streng, C.F for Carl Fischer and M.O for Matthew Oppenheim.

It is worth noting that the table does not mention the work of my supervisor, Prof. Hans Gellersen, who supervised almost every step of the project.

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Task(s) Description</th>
<th>Roles</th>
<th>%</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-to-End Discovery Model</td>
<td>Establishing the conceptual model based on the study of many scientific articles.</td>
<td>Concept</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>Gateways User Interface</td>
<td>This part was a common project between Sara Streng and myself. Both of us participated to the conceptual and implementation tasks.</td>
<td>Concept, impl.</td>
<td>50 S.S</td>
<td></td>
</tr>
<tr>
<td>Spatial Condition Toolkit</td>
<td>Refactoring the toolkit (concept and implementation) to make it support dynamic environments.</td>
<td>Software refactoring</td>
<td>10 S.S</td>
<td></td>
</tr>
<tr>
<td>Deliverable</td>
<td>Task(s) Description</td>
<td>Roles</td>
<td>%</td>
<td>Other</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>User study</td>
<td>The organization of this user study was a common work with Sara Streng. Both of us took part to all the tasks involved in organizing, running and evaluating a user study. My focus was on the technical aspects (system, settings) of the study.</td>
<td>Concept, technical settings, Execution</td>
<td>50</td>
<td>S.S</td>
</tr>
<tr>
<td>Extension of Relate</td>
<td>Application concept, first integration and adaptation of the Relate toolkit, participation to the DOT taskforce meetings.</td>
<td>Application</td>
<td>20</td>
<td>C.F,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>M.O</td>
</tr>
<tr>
<td>Network and Service</td>
<td>Concept and implementation of a network and service discovery layer for the RelateGateways architecture. Using Relate to implement a first version of discovery system. Implementation of the software part of the system.</td>
<td>Concept,</td>
<td>90</td>
<td>C.F</td>
</tr>
<tr>
<td>Discovery Layer</td>
<td></td>
<td>impl.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service Framework</td>
<td>Concept and implementation of a service framework supporting the rapid prototyping of mobile spatial services.</td>
<td>Concept,</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>impl.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invocation and Inter-</td>
<td>Concept and implementation of the mobile code system to support the “plug-and-play” discovery and invocation of previously unknown services.</td>
<td>Concept,</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>operability Layer</td>
<td></td>
<td>impl.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final Report</td>
<td>Redaction of the Master thesis, trying to detail my work as much as possible in both to allow an evaluation of the work in Switzerland and support future work on the architecture.</td>
<td>Redaction</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>(Master Thesis)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notations and Conventions**

- **Bold** and *italic* are used for emphasis and to signify the first use of a term.

- **SansSerif** is used for web addresses.
• **Code** is used for all the source codes and generally for anything that would be typed literally when programming, including keywords, constants, method names, and variables, class names, and interface names.

• The present report is divided in chapters. Chapters are broken down into sections. Where necessary, sections are further broken down into subsections, and subsections may contain some paragraphs.

• Figures, Tables and Listings are numbered inside a chapter. For example, a reference to Figure \( j \) of chapter \( i \) will be noted *Figure \( i,j \).*

• As a respect to both genders, he and she are used interchangeably in this document.

• Source code is displayed as follows:

```java
Matrix3f rotMat = new Matrix3f();
rotMat.fromAngleNormalAxis(FastMath.DEG_TO_RAD * 45.0f, new Vector3f(1.0f, 0.0f, 0.0f));
box.setLocalRotation(rotMat);
```

### About This Documentation

This documentation is based on a \LaTeX\ template created by Patrik Fuhrer and Dominique Guinard (Software Engineering Group, University of Fribourg, Switzerland). The template is open source and freely available from: [http://diuf.unifr.ch/softeng/guidelines.html](http://diuf.unifr.ch/softeng/guidelines.html).
# Table of Contents

1. Introduction ................................................................. 2
   1.1. Motivation ............................................................. 3
      1.1.1. Scenario 1: Spontaneous Mobile Interaction with Pervasive Services 3
      1.1.2. Scenario 2: Interactions Across Devices .......................... 4
   1.2. Research Questions .................................................. 4
   1.3. Approach ............................................................... 4
   1.4. Outputs ...................................................................... 5
      1.4.1. Softwares and Prototypes .......................................... 5
      1.4.2. Publications ......................................................... 5

2. A Model for Spontaneous Mobile Spatial Interaction with Pervasive Services ....... 7
   2.1. Spatial Discovery ....................................................... 8
   2.2. Network and Service Discovery ....................................... 9
   2.3. Invocation and Interoperability ....................................... 9

3. Related Work ..................................................................... 11
   3.1. Network and Service Discovery ...................................... 11
      3.1.1. Commercial Systems ............................................... 11
      3.1.2. Discovery by Scanning ............................................. 12
   3.2. Identifying Services .................................................... 12
      3.2.1. Physical Markers ................................................... 13
      3.2.2. Spatial User Interfaces ............................................ 13
   3.3. Invocation and Interoperability ....................................... 14
      3.3.1. Universal Remote Control ....................................... 14
      3.3.2. Recombinant Computing ......................................... 14
      3.3.3. Cross Device Interactions ....................................... 15
   3.4. Pervasive System Software ............................................ 15
      3.4.1. Prototyping Pervasive Environments ............................ 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Identifying Pervasive Services</td>
<td>17</td>
</tr>
<tr>
<td>4. Spatial Discovery</td>
<td>18</td>
</tr>
<tr>
<td>4.1. The Gateways User Interface Concept</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1. The Compass Metaphor</td>
<td>21</td>
</tr>
<tr>
<td>4.1.2. Scanning and Conditional Modes</td>
<td>21</td>
</tr>
<tr>
<td>4.2. Implementing the Gateways User Interface</td>
<td>24</td>
</tr>
<tr>
<td>4.2.1. Gateways</td>
<td>25</td>
</tr>
<tr>
<td>4.2.2. Main Toolbar</td>
<td>28</td>
</tr>
<tr>
<td>4.2.3. Spatial Simulation Tool: Wizard</td>
<td>28</td>
</tr>
<tr>
<td>4.3. Spatial Context Toolkit</td>
<td>29</td>
</tr>
<tr>
<td>4.3.1. Making the SCT Dynamic</td>
<td>30</td>
</tr>
<tr>
<td>5. Realtime Spatial Provisioning</td>
<td>36</td>
</tr>
<tr>
<td>5.1. About the Relate Project</td>
<td>36</td>
</tr>
<tr>
<td>5.1.1. Introducing the Relate Sensor Nodes</td>
<td>37</td>
</tr>
<tr>
<td>5.2. Extending Relate</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1. Relate Dots</td>
<td>39</td>
</tr>
<tr>
<td>5.2.2. Software Integration</td>
<td>41</td>
</tr>
<tr>
<td>II. Towards a Software Architecture to Support Spontaneity.</td>
<td>43</td>
</tr>
<tr>
<td>6. Service Oriented Architecture</td>
<td>45</td>
</tr>
<tr>
<td>6.1. Concepts of SOA</td>
<td>46</td>
</tr>
<tr>
<td>6.1.1. Service</td>
<td>47</td>
</tr>
<tr>
<td>6.1.2. Client</td>
<td>51</td>
</tr>
<tr>
<td>6.1.3. Directory</td>
<td>52</td>
</tr>
<tr>
<td>6.1.4. Why SOA?</td>
<td>52</td>
</tr>
<tr>
<td>7. Network and Service Discovery</td>
<td>53</td>
</tr>
<tr>
<td>7.1. Service Discovery in a Mobile Context</td>
<td>54</td>
</tr>
<tr>
<td>7.1.1. Services and Devices</td>
<td>54</td>
</tr>
<tr>
<td>7.1.2. Directories and Registration</td>
<td>55</td>
</tr>
<tr>
<td>7.1.3. Clients and Queries</td>
<td>57</td>
</tr>
<tr>
<td>7.1.4. Naming VS Discovery</td>
<td>58</td>
</tr>
<tr>
<td>7.1.5. Summarizing the Service Discovery Process</td>
<td>59</td>
</tr>
<tr>
<td>8. Invocation and Interoperability for Pervasive Services</td>
<td>61</td>
</tr>
<tr>
<td>8.1. Consuming Services</td>
<td>61</td>
</tr>
<tr>
<td>8.2. Addressing Interoperability in Pervasive Environments</td>
<td>62</td>
</tr>
<tr>
<td>8.3. Classifying Discovery and Interoperability Systems</td>
<td>63</td>
</tr>
</tbody>
</table>
9. **The RelateGateways Architecture** 66
   9.1. RelateGateways’ SOA 67
   9.1.1. Physical Servers, Devices and Services 68
   9.1.2. Descriptive Service Architecture 68
   9.1.3. Service Contract 70
   9.1.4. Implemented Services 77
   9.1.5. Service Management Tool 78
   9.2. Network and Service Discovery 78
   9.2.1. Introducing Relate as a Network Discovery System 79
   9.2.2. Service Registration 80
   9.3. Invocation and Interoperability 82
   9.3.1. Common Properties Through OOP 82
   9.3.2. Extracting a Mobile Code Using RMI Serialization 83
   9.3.3. About Dynamic Class Loading and Downloading 86
   9.4. RelateGateways’ Interactions in a Nutshell 88
   9.5. Service Discovery Simulation Tool 89

10. **Enabling Cross Device Interactions** 90
   10.1. Use Cases 90
   10.2. Enhanced Binding Layer (EBL) 91
   10.3. EBL meets RGW 93
      10.3.1. Migration Service Requester 93
      10.3.2. Migration Service Provider 94
   10.4. Implementing the Shared Notepad Application 94

11. **Developer’s Manual** 98
   11.1. Extending the RelateGateways Architecture 98
   11.2. Creating a New Service 99
      11.2.1. The Service class 99
      11.2.2. Creating a ServiceProvider 100
      11.2.3. Creating a ServiceRequester 101
      11.2.4. Creating a Device 102
      11.2.5. Creating a Condition 103
      11.2.6. Adding the Device to the Application 104
   11.3. Running the Application 104
      11.3.1. Network Configuration 104
      11.3.2. Using the WoZ Interface 105
      11.3.3. RMI Registry Browser 105
   11.4. Versions’ History 106
# Table of Contents

11.5. EBL and RGW ........................................ 107

## III. Evaluation, Conclusion and Future Work

12. Evaluation ........................................... 109
   12.1. Setup ........................................... 110
      12.1.1. Spatial Conditions ...................... 110
      12.1.2. WOZ ........................................ 111
   12.2. Test Run ........................................ 111
      12.2.1. Scenario .................................. 112
      12.2.2. Questionnaire ......................... 113
   12.3. Execution ....................................... 114
   12.4. Qualitative Results ......................... 114
      12.4.1. General Impression .................... 114
      12.4.2. Assessing the Disturbing Factor .... 116
      12.4.3. Comparing the Interaction Techniques 118
      12.4.4. Comparing the Modes of Operation ... 118
      12.4.5. Comments .................................. 120
   12.5. Quantitative Results ......................... 120
      12.5.1. Critical Evaluation ................... 121

13. Conclusion ........................................... 123
   13.1. Contribution .................................... 123
   13.2. Future Work .................................... 124
   13.3. Outlook ........................................ 125

## IV. Appendix .......................................... 127

A. EBL/RGW Notepad ...................................... 128
B. Questionnaire for User Study ....................... 131
C. Common Acronyms .................................... 134
D. Licenses ............................................... 137
   D.1. Documentation .................................. 137
   D.2. Software ........................................ 137
E. Website of the Project ................................ 138
F. CD-ROM ............................................... 140
Table of Contents

References 142

Referenced Web Resources 148

Index 150
List of Figures

1.1. Software components of the RelateGateways architecture. .......... 5

2.1. The End-to-End Service Discovery Model for spontaneous mobile spatial interactions with pervasive services. ....................... 7
2.2. Using Relate as a medium for network discovery. .................. 9

4.1. Discovering pervasive services on a Bluetooth device, visualization as a list. 19
4.2. Discovering pervasive services on a Bluetooth device, visualization as a list of icons. ..................................................... 20
4.3. The gateways user interface as a compass. ............................ 21
4.4. “User-in-the-middle” mapping of the environment. .................. 22
4.5. Facing the keyboard gateway in scanning mode. ...................... 23
4.6. Facing the keyboard gateway in conditional mode. ................... 23
4.7. A smartphone manufactured by Blackberry. ............................ 24
4.8. Alpha version of the gateways running on the OQO hand-held PC. 25
4.9. Gateways appearing on the OQO hand-held PC. ...................... 25
4.10. The three interaction zones of two concrete gateways. ............... 26
4.11. Evolution of the gateways’ design. .................................... 33
4.12. The MVC pattern applied to the gateways UI. ....................... 34
4.13. The administration (left) and main (right) toolbars. ................ 34
4.14. Wizard of Oz interface used to simulate the user’s spatial context. 34
4.15. UML class diagram of the SCT/Gateways UI integration. ........... 35
4.16. Concept of the SCT gateways UI integration. ....................... 35

5.1. Sample of the Relate sensor nodes. .................................... 38
5.2. Initial model of Relate. .................................................. 38
5.3. Display, mobile device, dot and dongle. ............................... 39
5.4. Extending the Relate model for RelateGateways. .................... 40
5.5. Current Dot (left) and expected size of the final Dot (right). ......... 40
5.6. Board of the Dot 1.0 Sensor Node. .................................... 41
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Structure of the Relate Toolkit.</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Historical background of Service Oriented Architectures.</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Basic pattern of actors and interactions for a Service Oriented Architecture.</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Registry (right) and non-registry topologies (left).</td>
<td>55</td>
</tr>
<tr>
<td>7.2</td>
<td>Naming and Discovery Systems</td>
<td>58</td>
</tr>
<tr>
<td>7.3</td>
<td>A generic model of discovery</td>
<td>59</td>
</tr>
<tr>
<td>8.1</td>
<td>Representing the User-Oriented Taxonomy as a BCG Matrix.</td>
<td>65</td>
</tr>
<tr>
<td>9.1</td>
<td>A Venn-diagram of the relationships between physical servers, devices and</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>services.</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>UML view of the Descriptive Service Architecture.</td>
<td>69</td>
</tr>
<tr>
<td>9.3</td>
<td>A universal requester consuming a music-player service</td>
<td>70</td>
</tr>
<tr>
<td>9.4</td>
<td>UML diagram of the ServiceProvider abstract class.</td>
<td>71</td>
</tr>
<tr>
<td>9.5</td>
<td>UML diagram of the ServiceRequester abstract class.</td>
<td>72</td>
</tr>
<tr>
<td>9.6</td>
<td>UML representation of a ServiceRequest</td>
<td>73</td>
</tr>
<tr>
<td>9.7</td>
<td>Schematic view of service communication</td>
<td>75</td>
</tr>
<tr>
<td>9.8</td>
<td>Building a printing service with a RMI communication contract</td>
<td>76</td>
</tr>
<tr>
<td>9.9</td>
<td>Mechanism underlying the abstract RMI Service.</td>
<td>77</td>
</tr>
<tr>
<td>9.10</td>
<td>The Service Management Tool</td>
<td>79</td>
</tr>
<tr>
<td>9.11</td>
<td>Using relate as a service discovery system</td>
<td>80</td>
</tr>
<tr>
<td>9.12</td>
<td>Service Registration in the RelateGateways architecture.</td>
<td>81</td>
</tr>
<tr>
<td>9.13</td>
<td>Software model for interoperability.</td>
<td>82</td>
</tr>
<tr>
<td>9.14</td>
<td>UML representation of the DeviceMobileCode class.</td>
<td>85</td>
</tr>
<tr>
<td>9.15</td>
<td>Creating and registering mobile code.</td>
<td>85</td>
</tr>
<tr>
<td>9.16</td>
<td>Plug and Play Invocation using mobile code.</td>
<td>87</td>
</tr>
<tr>
<td>9.17</td>
<td>Communicating the address of the display’s service registry.</td>
<td>88</td>
</tr>
<tr>
<td>9.18</td>
<td>Invoking the presentation service.</td>
<td>88</td>
</tr>
<tr>
<td>9.19</td>
<td>The discovery simulation tool.</td>
<td>89</td>
</tr>
<tr>
<td>10.1</td>
<td>A simple presentation tool.</td>
<td>91</td>
</tr>
<tr>
<td>10.2</td>
<td>Group collaboration using the EBL Migration service.</td>
<td>92</td>
</tr>
<tr>
<td>10.3</td>
<td>The EBL widget importer.</td>
<td>92</td>
</tr>
<tr>
<td>10.4</td>
<td>Concept of the requester part of the migration service.</td>
<td>94</td>
</tr>
<tr>
<td>10.5</td>
<td>Concept of the provider part of the migration service.</td>
<td>94</td>
</tr>
<tr>
<td>10.6</td>
<td>Notepad application implemented in EBL.</td>
<td>97</td>
</tr>
<tr>
<td>11.1</td>
<td>State of the Service Registry after a new Service was Started.</td>
<td>106</td>
</tr>
<tr>
<td>12.1</td>
<td>Location of services in the meeting room.</td>
<td>110</td>
</tr>
<tr>
<td>12.2</td>
<td>Spatial Conditions modeled for each device.</td>
<td>111</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>12.3</td>
<td>Alpha Version of the User interface and Wizard of Oz interface.</td>
<td>112</td>
</tr>
<tr>
<td>12.4</td>
<td>Circumstances under which the applications would be useful.</td>
<td>115</td>
</tr>
<tr>
<td>12.5</td>
<td>Perceived benefits of the application.</td>
<td>115</td>
</tr>
<tr>
<td>12.6</td>
<td>Detailed results for the ratings part of the general assessment.</td>
<td>116</td>
</tr>
<tr>
<td>12.7</td>
<td>Asserting how much the gateway's appearance was interrupting.</td>
<td>117</td>
</tr>
<tr>
<td>12.8</td>
<td>Global results for always displaying the gateways or rather only when needed</td>
<td>119</td>
</tr>
<tr>
<td>12.9</td>
<td>Percentage of predicted use between the scanning and the conditional mode.</td>
<td>120</td>
</tr>
<tr>
<td>12.10</td>
<td>Mean time per service using the two different mobile devices.</td>
<td>121</td>
</tr>
<tr>
<td>E.1</td>
<td>Screenshot of the project's official website</td>
<td>139</td>
</tr>
<tr>
<td>F.1</td>
<td>The CD-ROM of this project</td>
<td>141</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. The OnlyTrackableWithinZone class before the refactoring process.</td>
<td>31</td>
</tr>
<tr>
<td>4.2. The OnlyTrackableWithinZone class after the refactoring process.</td>
<td>32</td>
</tr>
<tr>
<td>5.1. Method triggered on updates of the Relate model.</td>
<td>42</td>
</tr>
<tr>
<td>6.1. Implementing the MathWebService in Java EE.</td>
<td>48</td>
</tr>
<tr>
<td>6.2. Implementing the MathSessionBean in EJB 3.0.</td>
<td>48</td>
</tr>
<tr>
<td>6.3. Extract of MathWebService’s WSDL file.</td>
<td>49</td>
</tr>
<tr>
<td>6.4. The description interface for local calls to the MathSessionBean service.</td>
<td>50</td>
</tr>
<tr>
<td>6.5. The description interface for remote calls to the MathSessionBean service.</td>
<td>51</td>
</tr>
<tr>
<td>7.1. Service registration in DNS SD, using JmDNS.</td>
<td>55</td>
</tr>
<tr>
<td>7.2. Service registration in jSLP.</td>
<td>56</td>
</tr>
<tr>
<td>7.3. Service query in jSLP.</td>
<td>57</td>
</tr>
<tr>
<td>9.1. The common interface of all the providers and requesters.</td>
<td>71</td>
</tr>
<tr>
<td>9.2. Sample of service request loop for a user interface programmed on top of the service framework.</td>
<td>73</td>
</tr>
<tr>
<td>9.3. Pseudo-code of a request(Object o) method.</td>
<td>73</td>
</tr>
<tr>
<td>9.4. Pseudo-code of a request(Object o) method.</td>
<td>75</td>
</tr>
<tr>
<td>9.5. startProvider(...) method of the ProviderStarter GUI.</td>
<td>81</td>
</tr>
<tr>
<td>9.6. Core method of the DevicesMobileCodeFactory class.</td>
<td>84</td>
</tr>
<tr>
<td>9.7. Using a home-made class loader.</td>
<td>86</td>
</tr>
<tr>
<td>10.1. Request method of the EBLMigrationServiceRequester.</td>
<td>95</td>
</tr>
<tr>
<td>10.2. Request method of the EBLMigrationServiceProvider.</td>
<td>96</td>
</tr>
</tbody>
</table>
1

Introduction

1.1. Motivation ........................................... 3

1.1.1. Scenario 1: Spontaneous Mobile Interaction with Pervasive Services ..................... 3

1.1.2. Scenario 2: Interactions Across Devices .................................................. 4

1.2. Research Questions .................................... 4

1.3. Approach .................................................. 4

1.4. Outputs .................................................. 5

1.4.1. Softwares and Prototypes .................................................. 5

1.4.2. Publications ............................................... 5

Back in the late eighties, a handful of scientists at the Xerox PARC (Palo Alto Research Center) had the vision of another type of computing: ubiquitous computing (aka Ubicomp or pervasive computing). Mark Weiser and his team envisioned a world where computers would merge with their environments [Wei91]. Far from virtual reality, which transports human beings from their environment to the virtual world, Weiser wanted to bring computers to the everyday life of common people: “[...] Our preliminary approach: Activate the world. Provide hundreds of wireless computing devices per person per office, of all scales [...]”

Various requirements were formulated in order to enable such a world. One of them is particularly relevant for the presented work: “A less-traveled path I call the "invisible"; its highest ideal is to make a computer so embedded, so fitting, so natural, that we use it without even thinking about it.” In essence, there exist two ways of addressing this requirement. One can focus on the “invisible computer” issue by trying to make the computer working in the background, inferring from users’ behavior to trigger appropriate actions and mechanisms. This first approach groups the domains that we shall call technical. These range from sensor networks technologies, distributed systems (network discovery, communication protocols), not to forget context aware computing.

On the other hand, one can focus on the natural aspect of the requirement. This latter approach is the matter of the HCI (or Human Computer Interactions). Pioneered by figures such as Alphonse Chapanis, Ben Shneiderman, Bill Curtis and Douglas Engelbart [Shn07] in the early eighties, HCIIs bring the human back in the center of attention by
1.1. Motivation

We share our living and working spaces with smart appliances providing various ranges of services. We are surrounded by desktop computers, printers, displays, servers, access points, etc. We carry mobile phones, PDAs, laptops or multimedia players with us. Having these observations in mind it is not hard to imagine the power and the convenience of using our mobile devices to interact with the pervasive services surrounding us. Yet, while a number of research focus on service discovery and interoperability few address the underlying HCI issues, placing the user in the center of the service discovery process. As a result, the user experience is still quite frustrating \cite{Nie02}. It is precisely this frustration we want to address in order to offer to the mobile user more spontaneous interactions with pervasive services. In order to have a better understanding of this issue, two scenarios are provided below.

1.1.1. Scenario 1: Spontaneous Mobile Interaction with Pervasive Services

Imagine Bob visiting a new research laboratory for a week. As a modern IT user, Bob bring his laptop, cellphone and PDA along. Having finished writing the presentation for the seminar he will be giving on the next day, Bob is now wanting to print it. While this task may be easy for the local researcher, Bob will have to go through various steps:

1. Physically locate the printer.

2. Find out about the printer’s properties (brand, queue name, does it print four slides on a page, ...).

3. Install and configure the printer on his personal laptop.

4. Print the document.
1.2. Research Questions

Addressing spontaneity in ubiquitous environment is still a rather vague concept and thus, we refine the challenges we really want to solve in terms of a general model as well a concrete answers (i.e. implementations) to the following questions:

1. In a world with a plethora of potentially accessible pervasive services how does the user actually identifies the services that are available.

2. Once a service is identified, how can the user interact with it in a uniform and natural manner, without the need for him to install, configure or learn using yet another system.

1.3. Approach

As a general approach for addressing these questions we propose the End-to-End Service Discovery model (E2ESD) (see 2). The main contribution in this model is an answer to the first question. In the first layer of our model we propose the use real-time spatial information to provide a more natural representation of services surrounding the user.

The other parts of the model are inspired from previous works in the field service discovery and answer the second question by specifying two layers supporting the plug-and-play installation and configuration of newly discovered services.

Beyond a purely theoretical approach, we provide a concrete application implementing the E2ESD model. The RelateGateways architecture implements the three layers composing the model. As such it served as a validation support for our model. Furthermore, the implementation is consolidated into a single “run-and-use” application making use of a
novel type of mobile code that enables the seamless installation of previously unknown services. These facts make the RelateGateways a good candidate for the prototyping of spontaneous spatial interactions with pervasive services.

1.4. Outputs

1.4.1. Softwares and Prototypes

Figure 1.1.: Software components of the RelateGateways architecture.

The output in terms of software components is summarized on Figure 1.1 and further described in Part II of this Master Thesis. In total the software framework is composed of 162 classes\(^1\) for about 17,000 lines of Java code.

1.4.2. Publications

Additionally to this Master Thesis (freely available from the official website: [27]), a number of articles were produced as listed below.

Published

1. [GSG07b] Dominique Guinard, Sara Streng, Hans Gellersen. "Extending Mobile Devices with Spatially Arranged Gateways to Pervasive Services". Presented at the International Workshop on Pervasive Mobile Interaction Devices (PERMID 2007), in frame of the International Conference on Pervasive Computing, (PERVASIVE 2007) in Toronto (Canada). This paper focuses on the spontaneous interaction aspects. After a brief introduction to the user interface, it provides a (brief) insight of the spontaneous interaction issues both form a conceptual and an implementation point of view. Additionally, it discusses the early results of a user study conducted in Munich (Germany), see Chapter 12.

\(^1\)About 130 of the 162 classes were written by the author of this thesis.
   This paper focuses on the user interface of the project, especially on the aspect of spatial and visual discovery.

   Abstract, Poster and demonstration of the RelateGateways architecture and the underlying Relate ad-hoc network.

**Accepted**

   This short paper is a demonstration proposal of the whole RelateGateways architecture including the first integration of the sensor network.

**Pending**

1. The RelateGateways architecture was selected for an article in a special issue of the ACM “Personal and Ubiquitous Computing” on Mobile Spatial Interactions to be released in 2008.
A Model for Spontaneous Mobile Spatial Interaction with Pervasive Services

2.1. Spatial Discovery

2.2. Network and Service Discovery

2.3. Invocation and Interoperability

Figure 2.1.: The End-to-End Service Discovery Model for spontaneous mobile spatial interactions with pervasive services.

Public areas such as airports, exhibition halls or universities are highly dense places in terms of pervasive services provided by co-located devices: access points, information booths, vending machines, picture printers, copiers, interactive and information displays. However, as formulated before, this power remains under-exploited both because of the difficulties for users to identify the services available in the immediate environment as well as consume the identified services in a natural and straightforward manner.

Our approach addresses these two problems using a simple conceptual model named “End-to-End Service Discovery” (E2ESD). This model is summarized on Figure 2.1. It basically addresses every step in the life-cycle of a spontaneous mobile interaction with pervasive services. The metric used for the X-axis of the figure are the steps in the interaction life-cycle. The starting point (far-left) is the user carrying his mobile device. His intention is
identify and interact with co-located pervasive services. On the end-point (far-right) are the devices offering various pervasive services. In between the user and the service, three building blocks represent the mandatory layers for spontaneous interactions. These layers will be introduced below and further described in the following section of this thesis.

This model was used as point of focus for the implementation of the framework and prototypes. While the upper part (Model) of Figure 2.1 represents the conceptual bricks of the E2ESD model, the lowest part (Implementation) exposes the concrete implementation of the various layers in the RelateGateways architecture. It starts with the gateways user interface helping users to identify services. It is followed by extending the Relate sensor network to enable network and service discovery. Eventually, a novel SOA using mobile code is responsible for Invocation and Interoperability. The next subsections provide a brief overview of these components, the next chapters discuss them in more details.

2.1. Spatial Discovery

According to many studies and articles such as [KTKG07] the biggest barrier for users to interact with pervasive services is the identification of the service themselves. Interestingly enough this fact was confirmed, not to say emphasized, by our user study (see Chapter 12). Users are literally surrounded by electronic appliances. While each one of these device might potentially deliver a service accessible through a mobile interface, it is very unlikely for every single electronic device to do so. As a consequence users must have a way to identify these services or devices. This is the matter of the first layer of our model: the Spatial Discovery Layer.

A number of solutions have began to appear. Some of them are summarized in Section 3. Our implementation proposes to address the problem using a novel kind of user interface (UI): the RelateGateways UI. This concrete implementation uses spatial information to pop up small widgets on the periphery of mobile screen. These widgets, called “gateways” represent the nearby services and thus, help the user identifying services in his immediate environment.

Once a service of interest is identified it can be consumed by drag-and-dropping an object on the gateway to directly invoke the service or by clicking on this latter to obtain more options before actually consuming the service. The concept and concrete implementation of the user interface is discussed in Chapter 4.

Yet alone helping the user identifying service is not enough for a user interface to enable “spatial discovery”. We add a spatial dimension to the user interface in two ways: Firstly by moving the gateways at the screen periphery according to the current user’s location. This way the gateways always “point” towards the co-located services, acting like a compass user interface. Secondly the services can advertise contextual rules, or conditions that are used to decide whether a gateways should appear. An example of condition could be to show a gateway for interacting with a public display if and only if the user is currently facing the display.

In order to model and manage these latter rules the gateways UI is built on top of a Spatial Context Toolkit (SCT) described in 4.3. Furthermore, the former location information is provided by the Relate ad-hoc sensor network as described in Chapter 5.
2.2. Network and Service Discovery

As exposed in [GSG07a], the user-study conducted to evaluate the user interface and interaction techniques of this work (see Chapter 12) emphasized the importance of the auto-configuration. Indeed, for our users the fact that the system was taking care of all the technical details was the most significant step towards spontaneity. Thus, once the user identified a service she would like to use (Spatial Discovery layer) the system needs to address network and service discovery in order to connect the mobile device to the machine providing the selected service.

We propose to use an extends of the Relate sensor network (see Chapter 5) to address this problem. This way the system initiates the network connections without the need for the user to configure anything. Chapter 7 further explore our solution as well as the general concepts of Network and Service Discovery.

2.3. Invocation and Interoperability

Once the mobile user identified a service (Spatial Discovery layer), her mobile device is successfully networked with the service provider (Network and Service Discovery layer) the interaction is ready to occur. During the study users pointed at the importance of auto-installation when trying to interact spontaneously we believe this layer must be addressed as well. Thus, we propose a system based on packages of MobileCode containing everything a client needs to know in order to invoke a previously unknown service. These packages are dynamically downloaded and loaded by the mobile client at run time. This way, the client discovers the services “on the fly” and can use them in a “plug-and-play” fashion. Furthermore, we designed a service framework taking care of dynamically wrapping local
2.3. Invocation and Interoperability

services into packages of MobileCode. The framework enables the rapid prototyping of all kinds of pervasive services through a set of interfaces and call-back methods. Invocation and Interoperability mechanisms are further discussed in Chapter 8.
Work related to the presented architecture originate from several domains such as mobile interactions, network and service discovery, spatial interactions, visualization of pervasive services, invocation and interoperability systems, recombinant computing, prototyping architectures for pervasive computing and cross device interactions.

As a starting point let us consider to the motto of our research: enhancing the user experience when interacting with services they discover whilst on the move. There are many ways in which mobile devices can facilitate or support their interactions with pervasive services. Generally these fall into two big classes, the support for discovery of available services, and the support for access to and interaction with identified services.

### 3.1. Network and Service Discovery

#### 3.1.1. Commercial Systems

Examples of systems supporting discovery include a range of research works and commercial products. When considering a large spectrum of mobile devices (and not just mobile phones) on-the-shelves technologies like Jini [15], UPnP (Universal Plug and Play) [32],
Bonjour [5], DNS SD (DNS Service Discovery) [7] have to be considered. These systems all provide powerful service discovery mechanisms but they are network-centric and focus on technological issues. As a consequence they do not include users in the discovery loop. Furthermore, many of these technologies are heavy-weighted and although they were used in an number of pervasive computing projects such as in [HK05], using them on mobile devices with limited resources is not optimal1.

When considering small, less-powerful mobile devices, Bluetooth [4] and its SDP (Service Discovery Protocol) is an interesting option. Available in many types of mobile device, this system can be used to scan the environment for Bluetooth-enabled services. Again, this approach is network-based and has the major drawback that there is no direct way for a user to associate names of discovered services with actual devices in their environment.

### 3.1.2. Discovery by Scanning

Other approaches to support interaction with pervasive services are based on some form of physical scanning, using mobile devices like a probe that is moved through the environment in order to find interaction opportunities. Examples are Fitzmaurice’s pioneering work in which the mobile device is used like a lens [Fit93] or Rekimoto’s Navicam which uses augmented reality techniques [RN95]. In this domain services are often tagged to enable computer identification. RFID (Radio Frequency Identification) enabled devices, camera or barcodes readers are used to scan the tagged services. This interaction technique is commonly called NFC (Near Field Communication) due to the fact that the probing device needs to be close (usually at most 10 centimeters) to the tagged services in order for the interaction to occur. In [RSH04] mobile interactions with tagged posters offering interactive services are considered. Such techniques were also deployed in fields trials such as in [OTSA07]. Similarly, [WTM+07] focuses on the use of standards (EPC, Electronic Product Codes) for massive NFC deployments.

A number of user studies such as [RLC+06] and [BSR+07] tended to show that NFC techniques were the most appreciated by users for interactions between a mobile device and everyday object such as posters. More importantly these studies emphasized the fact that the most automated the interactions were, the most likely the users were to use the provided services. Indeed, most users perceived NFC interactions as a good experience both because they were easy and their learning process was really quick.

The probing method is a way of involving the mobile user in the discovery loop. However, these approaches support physical identification of a service in the environment, but they do not help the user obtain an overview of services that are available in their proximity. We try to address this issue by combining a user-oriented Spatial Discovery layer (Chapter 4) with a Network and Service Discovery layer (Chapter 9).

### 3.2. Identifying Services

A fundamental question driving this work could be formulated as: “which device provide a service I can access with my mobile device, which one does not?”. While this might not be a problem in familiar environments, it was identified as a real problem for the mobile

---

1 An interesting solution to counter the impact of limited resources is the ‘surrogate’ architecture: [30].
3.2. Identifying Services

user [KTKG07, Gre06]. As the number of pervasive services available in the everyday environments increases, this problem will intensify.

3.2.1. Physical Markers

There are basically two types of approaches for helping users to identify services. The first one is based on the development of a set of icons that can be physically stuck (or displayed) on the services as in [Gre06]. Similarly, in [KTKG07] Koltenger et al. suggest the use of a novel kind of visual stickers, inspired from the well-known "I" of the tourist information centers. These approaches are straightforward but do have some drawbacks such as the fact that this type of identification needs for the user to move quite close to the service to actually identify it. Additionally, the information bandwidth of such physical tags is quite low.

3.2.2. Spatial User Interfaces

Another approach is to provide user interfaces to visualize pervasive service on the mobile screen. Research works like [KTH00, QKRS05] or commercial applications such as [14] try to enhance a commercial discovery system (namely Jini) with a better visualization than the simple but unnatural [HB00] lists of services.

Similarly, the user interfaces of bigger projects like ICrafter [PLF+01], Cooltown [KB01] or Speakeasy [NSN+02, NIE+02] map the spatial arrangements of the environment on the mobile screen. By visualizing services using spatial metaphors these applications offer a better user experience of the discovery process. Our implementation of the Spatial Discovery layer goes along this path. However, unlike these projects we would like the spatial arrangements on the mobile screen to be dynamic, i.e. to be updated in real-time as the user moves in the environment.

Several works like explore this area. A number of them such as [LPRT07, LK07] or [19] are based on visualizations of geo-tagged services or points of interests. Most of them are built on top of commercial applications like Google Maps [13] or Microsoft MapPoint [18] and use tagged maps to represent available services. While this type of representation is fine for services at a macro level (country, city, street) our application needs to visualize services at a micro level (building, airport/conference hall, floor, office, etc.). Other geo-tagging based system use alternative representations. As an example [JSP07, JGJ07] use a radar-representation of the services in the environments.

While all the pre-cited examples require the use of dedicated applications to represent spatial information, some systems explore ways of using this information to the extend the desktop metaphor. In [KKG05] Kortuem et al. introduce the spatial widgets to be able to build spatial graphical user interfaces. The gateways UI extends this approach. The concept is to integrate the gateways in an unobtrusive manner at the perimeter of the screen but closely integrated such that direct interactions with the desktop or with other applications are facilitated. Furthermore, the gateways move as the user moves around the screen, thus transforming the extended desktop into a compass pointing towards co-located services.
3.3. Invocation and Interoperability

To fully address the interaction with pervasive services we need to consider the support for access to and interaction with identified services. This part of pervasive service architectures was a lot less explored than the discovery of services. Yet, two areas of research influenced our system: universal remote controls and recombinant computing.

3.3.1. Universal Remote Control

Research on “universal remote controls” are also related to this work since they tend to make use of mobile devices such as mobile phone to control common devices (coffee machines, TVs, multimedia appliances, etc.).

uPhoto provides a “camera” to capture images that have embedded links to services in the photographed scene [KIS+04]. In the same trend [TKKN05] proposes to use the physicality of mobile device to control real-world devices: rotating your mobile phone can be use to control the sound volume of your TV.

In [RLFS07] the user experience when using a mobile phone as a universal remote control was carefully evaluated. Qualitative results revealed the fact that for more complex interactions such as configuration tasks (e.g. re-configure the coffee machine) users perceived a universal remote control as valuable. On the other hand it appeared they would rather use the device controls for everyday tasks (e.g. getting an expresso). Quantitative results confirmed this fact by showing that executing configuration tasks was faster using the mobile device whereas simple tasks were significantly faster using the machines’ controls.

These systems use the mobile device as remote controls. We would like to take a different approach and consider the mobile devices as real actors of the interaction: sending content to and gathering content from the pervasive services.

3.3.2. Recombinant Computing

Work in the area of recombinant computing push the interaction metaphor beyond the “universal remote control” model. In [KB01] interoperability relies on web standards. Since every sufficiently modern mobile device can read webpages this approach is quite powerful in terms of interoperability. Nevertheless, for more complex and low-level interaction, relying on web-standards can be a limiting factor. Moreover, web interfaces are not necessarily well adapted to mobile devices.

Other systems rely on a set of specific interfaces. Jini is perhaps the first significant implementation of this scheme. It uses mobile codes extending specific interfaces for each type of service (e.g. Printer interface). Since clients need to have a prior knowledge of interfaces the system is not entirely open to interactions with completely unknown services.

In Speakeasy2 [NSN+02, NIE+02] Edwards et al. propose an architecture for recombinant computing based on packages of mobile code loaded by clients during run-time. A slightly different version of this system is used in [PLF+01, JFW02]. Our implementation

---

2The results of this work were incorporated into the Obje interoperability platform marketed by PARC.
of the Invocation and Interoperability layer was largely influenced by the recombinant computing model.

Just as in Speakeasy we use mobile code but since our focus is on the rapid prototyping of pervasive service, we couple the interoperability system with a service oriented framework. As a result the mobile code can be automatically generated from the service, freeing the programmer from this task not directly related to his core concerns.

3.3.3. Cross Device Interactions

We further extend the gateways interaction technique modeled by our user interface to cross device interactions in order to support the relocation or migration of applications. In [SER03] an interactive table is used as the migration end-point for various mobile devices. [BB04] and [BB06] focus on user interfaces to support this interaction scheme. They propose various types of 2D iconic (static) maps of the workspace allowing relocation per drag-and-drop. These works explore the interaction space when working with relocation but do not offer a general platform to support it.

ICrafter, on the other hand, provides a system to relocate an application by migrating the service that runs it to another location. In [aVR05] Grolaux et al. go towards this direction but offers to migrate widgets rather than whole applications. The next version this research work is called the Extended Binding Layer (EBL). It consists of a toolkit for building user interfaces offering migration capabilities without the need for the UI designer to address these capabilities.

In the RelateGateways architecture we merge EBL with our project in order for the Extended Binding Layer to be used with a dynamic representation of the environment.

3.4. Pervasive System Software

As mentioned in the introduction, this work is an architecture implementing the E2ESD model. Rather than addressing a single issue it focuses on enabling a particular type of interaction, namely mobile spatial interaction with pervasive services. It aims to address every step in the interaction life-cycle to make the interaction as spontaneous as possible. Thus, the closest works are the ones addressing the interaction as a whole as well. In this area impressive projects like ICrafter and [PLF+01] the IRoom project [JFW02], Cooltown [KB01], Elope [PBW05] or Speakeasy [NSN+02] inspired the RelateGateways architecture. However, as mentioned before these projects rely on a discretisation of the spatial visualization. The user gets a mapping of the environment when polling the system to obtain it (e.g. using an RFID reader to scan tags on services or scan a room).

In the RelateGateways architecture we want to explore discovery using a continuous flow of spatial information.

3.4.1. Prototyping Pervasive Environments

To close this related work section we shall also have a closer look at prototyping tools for pervasive environments. As a starting point, [KF02] Kindberg et al. discuss the
requirements of system software in ubiquitous computing. In [TMA+07] propose a high-level language for the rapid prototyping of pervasive services. IStuff [BMRB07] addresses the rapid prototyping of mobile phone interfaces for ubiquitous computing. This work is centered on developing user interfaces for mobile phones in order for these devices to control and interact with pervasive environments.

Closer to our work, Topiary [LHL04] is a prototyping tool for location-enhanced applications. Using this tool designers can create active maps, which model the location of people, objects and places. On top of these maps interactions storyboards can be created. Eventually, the generated prototypes can be exported to mobile devices and controlled using a Wizard-of-Oz type interface to simulate location and context.

Tools like Topiary have a macro-view on location-enhanced applications using maps as a support for modeling location. With the RelateGateways architecture we want to explore mobile spatial applications in order to model pervasive services at a micro level (offices, airports, etc.).

The strength of our toolkit strength resides in the fact that we integrate all the layers required to address mobile spatial interaction in these environments into a single tool that does not require any particular infrastructure or complicated setup. Creating a service with the RelateGateways architecture makes it immediately available to be tested in the dynamic conditions required by pervasive environments.
Part I.

Identifying Pervasive Services
To begin our exploration of spontaneous mobile interactions with pervasive services we shall start by addressing the first part of the E2ESD model: the Spatial Discovery Layer.

In the title of this master thesis, the adjective *spontaneous* is particularly important. A quick look for “spontaneous” in the Oxford Dictionary of English gives us the following: “Adjective 1 performed or occurring as a result of an unpremeditated impulse and without external stimulus. 2 open, natural, and uninhibited. 3 (of a process or event) occurring without apparent external cause. 4 Biology (of movement or activity) instinctive or involuntary.”
Both the first and second definitions are relevant for us. In our context, we define a spontaneous interaction between two devices as one unpremeditated or unplanned. Ubiquitous context are highly dynamic, devices and services appear and disappear on a regular basis. Thus we want to enable devices to interact with each other without prior knowledge of one another. We want the newly discovered services to be invokable from any mobile device in a “plug-and-play” fashion.

This is, however, a technology centric view-point on spontaneity, in this first layer, we would like to extends these definitions to include the user. In the context of this work, we would like to make human-computer interactions as natural as possible from a human point of view, offering comprehensive interaction schemes with the devices and services.

Two main motivations served as a starting point for addressing this part of the work. The first one is a simple findings: one of the biggest challenges for users when trying to interact with pervasive services is the actual identification of available services. Indeed, according to a number of studies and articles such as [KTKG07], users are literally surrounded by pervasive services but do not interact with them primarily because they are unable to identify them.

![Figure 4.1. Discovering pervasive services on a Bluetooth device, visualization as a list.](image)

Thus, there is a need for helping users to identify services. Many solutions already exist in this field. Mobile devices can scan for available devices using technologies like Bluetooth, Bonjour or Jini (see Chapter 7). The visualization of the retrieved services is traditionally offered as a list as on Figure 4.1 (Source: [25]) or at best as a list of icons as on Figure 4.2 (Source: [31]).

This representation is adapted for devices with (very) small displays. However, human beings structure their environment primarily spatially [HB00]. Indeed, their representation of the environment is based on the position of objects and people relatively to them: the user is in the center of his own representation of the surrounding world. As a consequence visualizing pervasive devices as lists (iconic or not) is not a natural way of representing the environment. This statement was supported by a number of user studies such as [MG07] showing that spatial representation of services (or systems) were more straightforward for most users. As a consequence several studies aiming to create new visualization techniques for co-located objects have been undertaken such as [KKG05], exploring the use of spatial widgets instead of using simple list-views.
Not only this kind of view is opposed to our spatial representation of the environment, it is also unpractical. Indeed, consider the printer on Figure 4.2. It only informs the user that there is a Bluetooth printer somewhere, but does not help the user to actually identify which one and where it is located. This is even more of a problem when considering systems supporting the discovery of physically distant services, such as Jini or UPnP.

These issues form the basis of the Spatial Discovery Layer. In general terms it aims to introduce spatial information in the user interface to help user identify available services. This chapter describes the various elements of the RelateGateways user interface (or gateways interface), a concrete implementation of the spatial discovery layer first presented in [GSG07b]. We start by describing the user interface which uses positioning information to dynamically and contextually display the services users can interact with in their immediate environment (see Chapter 4). The devices providing these services appear as gateways in the interface, taking the form of small widgets at the periphery of the users’ screens as they move in the environment. The gateways are spatially arranged around the screen based on a compass metaphor. They serve as “access point” for association and interaction with services, and are integrated with a mobile desktop interface to support direct manipulations such as the “drag-and-drop” of documents to a service.

Section 4.3 introduces the Spatial Context Toolkit [Str07]. This software system enables the gateways to be shown to the user according to certain rules and conditions (e.g. the user is 30 cm away from the device, user is facing the device, etc.). We explain how it was integrated with the user interface.

The RelateGateways user interface concept needs a sensing technology providing real-time relative positioning information such that the spatial relationships between a mobile user’s device and devices encountered in their environment can be tracked real-time. In Chapter 5 we introduce the Relate ad-hoc sensor network in the architecture and discuss its ongoing extension to fit the needs of the RelateGateways UI.

In Chapter 10 we push the limits of spontaneous interactions in our system a bit further by introducing cross-device interactions based on the EBL framework [aVR05] and the RelateGateways service framework. Using an integration of both systems users can drag-and-drop parts of users interfaces from one computer (or mobile device) to another.
4.1. The Gateways User Interface Concept

4.1.1. The Compass Metaphor

As introduced above, the user interface is based on so-called “gateways”. The gateways are interactive areas, effectively widgets, on the edges of the screen, indicating where a device (providing services) is located. The gateway’s position depends on the direction of the device. For example if the user is standing in front of a printer, he will see a printer gateway on the top of his screen. If the printer is on the left, the gateway will appear on the left. Thereby, the RelateGateways user interface has the functionality of a compass. This is illustrated in more detail in Figure 4.3 (source: [Str07]). In the left picture a user with his PDA is indicated by a black spot. The arrow represents the user’s orientation. The directions to the devices are mapped onto the PDA’s screen as shown on Figure 4.3. As the user moves around the gateways’ positions are updated. Thus, the interface maps the user’s view of her environment onto the mobile screen. For the concept to work the user has to imagine himself as being in the center of the desktop. This way, the environment in front of him is mapped at the top of the screen, the rear at the bottom and the left and right parts of the environment on the left and right side of the screen as shown on Figure 4.4.

Since the immersion of computer users in their virtual desktop is usually fairly high, the mapping is quite natural. This fact was confirmed by many users during the evaluation of the user interface (see Chapter 12).

4.1.2. Scanning and Conditional Modes

Scanning Mode

The user interface offers two different modes of operation. In scanning mode the system discovers all the devices and their corresponding services within visibility range, i.e. within line-of-sight from a sensing perspective. The goal of this first mode is to replicate, in the user interface, what the user actually sees.

Figure 4.5 shows the user interface (running on a Linux laptop) when the scanning mode is active. The left part of the screenshot is a view of the current spatial context. The user is represented by a blue dot. His orientation and direction are represented using an oriented blue line starting from the dot. In this example the user is going towards the
4.1. The Gateways User Interface Concept

Figure 4.4: “User-in-the-middle” mapping of the environment.

keyboard. Since the scanning mode is active all the available services are shown on the mobile screen and the gateways are disposed at the edge of the screen according to the compass metaphor exposed before.

Conditional Mode

Alternatively, in the conditional mode the gateways appear if and only if certain spatial conditions are met. The purpose of this latter mode is to indicate that the service exists and might be useful in the current spatial context. Imagine a user sitting in front of the PC writing text on his PDA. A small keyboard gateway will appear on the screen, advertising use of a keyboard instead of the more limited keypad provided by the PDA itself.

In the conditional mode, the spatial conditions which cause the gateways to appear differ depending on the service. For example, in our initial prototype the keyboard gateway is only shown if the PDA is within 50 cm of the typing device since the user needs the keyboard and the PDA to be within reach.

On Figure 4.6 the user is still going towards the keyboard. However, this time the conditional mode is activated. As a result the gateways will be shown if and only if their spatial conditions are satisfied. In this simple example the conditions are depicted using green circles. These represent the range the user has to be within for the corresponding gateways to be shown. Since our user is only within range of the keyboard, the keyboard gateways is the only one to appear on the mobile screen.

Desktop Integration and Interaction

In terms of interaction with the gateways one requirement was core to our approach: the gateways should be fully integrated to the desktop so that interactions can be perceived
4.1. The Gateways User Interface Concept

as an extension of the desktop paradigm, not as “using yet another program”. Thus, the gateways user interface is designed to be a seamless extension of any existing desktop-style interface. The concept is to display the gateways in an unobtrusive manner at the perimeter of the screen but closely integrated such that direct interactions with the desktop or with other applications are facilitated.

Each gateway can be used in two different ways. First, it is a target area for drag-and-drop. For example a file can be printed by dragging it onto the printer gateway and a slideshow can be started by dragging a presentation onto the display gateway. Secondly, it can be used as a button. If the user clicks on it several (or a least one) options dialogs will be opened before actually using the service. This is an important functionality since drag-and-drop always triggers a default action without allowing for any specific settings. By dragging a file on the printer gateway the whole file will be printed with the default options. However, if the user wants to print only some pages or change the properties,
this is possible by invoking the service menu by “clicking” on the gateway.

4.2. Implementing the Gateways User Interface

Since the gateways user interface focuses on mobile interactions, one of the most important requirements is for the implementation to run on relatively small and heterogeneous devices. The application is primarily meant to be used on advanced mobile devices such as hand-held (OQO, Palm, Pocket PC, etc.), laptops (Lenovo ThinkPads, Macbooks, etc.), tablet-PCs (Paceblade, Motion Tablets, Fujitsu Siemens Tablet PC, etc.). It is worth noting that such an application is not really adapted to a plain cellular-phone. The very small screen-size of these devices is not appropriate to the gateways UI. However, the latest generation of cellular-phones, often called “smartphones” would be more adapted to our UI. As an example the smartphones produced by Blackberry [2], as shown on Figure 4.7, have a bigger screen which makes them good candidates to run the gateways UI on. To meet the requirement for heterogeneity, the choice of a portable implementation is obvious. Hence, the user interface is implemented as a Java application using Swing GUI components. It is basically composed of four distinguishable parts:

- The Gateways are the most important components since they stand as access-points to the pervasive services.
- The Main Toolbar offers quick access to various functionalities such as swapping the current mode (scanning/conditional).
- The Service Management tool used to start and stop pervasive services.
- The Simulation tools used to simulate parts of the system such as the spatial information.

The rest of this section describes the architecture and concrete implementation of these parts. Even if the main goal of the following sections is to provide a conceptual view of user interface’s implementation, it still requires some knowledge of Object Oriented Programming (OOP), since it refers to important concepts of this programming paradigm.

![A smartphone manufactured by Blackberry.](image)
4.2. Implementing the Gateways User Interface

The reference language is the Java programming language but the exposed concepts are similar in any other OO language.

4.2.1. Gateways

Implementing the Design

The implementation of the gateways user interface was an iterative process driven by the results of the test-run, user study and the various demonstration sessions. We collected feedbacks after each of these events and tried to implement the most relevant suggestions. Figure 4.11 shows an overview of the evolution of the gateway’s design and Figure 4.8 is
4.2. Implementing the Gateways User Interface

![Image of gateways](image)

**Figure 4.10.** The three interaction zones of two concrete gateways.

a picture of the alpha version of the gateways running on a real mobile device. The logic of the iterative design process is described into further details in the evaluation (Chapter 12).

The visual part of the gateways is implemented as small windows in order to be able to move them easily at the screen periphery. However the borders, decorations and window icons were removed in order to make the gateways look like an extension of the desktop rather than windows of an independent application. Furthermore, the user can not move these windows. This is to ensure that the compass metaphor is always consistent.

As shown on Figure 4.10 each gateway is composed of three distinct zones. The first one (1) is also the most important since it is the invocation zone. It is to be used whenever the user wants to drag-and-drop an object on the gateway or get more options by clicking on it. This part also contains an iconic representation of the device the gateway stands for. This icon has two main purposes:

1. It emphasizes the visual mapping. If two devices are situated quite close to one another, the icon will help identifying which gateway stands for which device.

2. It attracts the user attention directly to this central zone. As a consequence most test users intuitively tried clicking and drag-and-dropping on this zone.

The next zone (2) is used for additional control. The cross (x) offers the user to hide (close) a gateway. It is especially useful when a gateway is hiding some other elements or when the user needs more space on the mobile desktop. The second icon (i) can be used to gather more information about the services offered by the represented device. This information can be given to the user using two different output mode.

The first mode is purely textual, and thus more adapted to bigger screen. It provides an information dialog containing service description. Alternatively the second mode provides a vocal description of the offered services. This mode is especially well-suited for devices with a smaller form factor. The vocal descriptions of services are not pre-recorded. Instead the system provides the `VoiceSynthesizerManager` which reads out the textual services’
4.2. Implementing the Gateways User Interface

description. This component is based on the publicly available FreeTTS system, a speech synthesizer written in the Java programming language [12].

The last zone (3) is meant to enhance the spatial user experience. It is worth noting that it is a mock up for the time being. However, it will provide an estimation of the distance to the device the gateway represents when the integration with the underlying sensing layer will be fully completed. To have a better feeling of the gateways’ look-and-feel Figure 4.9 is an enhanced picture of the application running on a mobile device.

MVC Architecture

Technically speaking, the user interface is implemented as a Java Swing GUI using the Swing Layout Extensions library. In terms of software components, a gateway is composed of:

1. Two concrete views, subclasses of VerticalView and HorizontalView. These are the visual representation of the gateways. The former is used when the gateway has to be mapped on the left or right side of the mobile display (see Figure 4.10, left part), whereas the latter appears when the gateway has to be displayed on the top or bottom (Figure 4.10, upper part).

2. A GatewayController in charge of abstracting the positioning information in order to display and move the gateways’ views at the periphery of the mobile screen.

Thus, the gateways’ architecture implements the well-known MVC (Model View Controller) pattern. This pattern is meant to help achieving a clean design of graphical user interfaces by decoupling data access and business logic. As exposed in [Eck07], it suggests to decomposing the application into three main conceptual parts:

1. The Model encapsulates the application state and the data to be represented by the GUI. It also responds to state queries and exposes the application functionality. Finally, it is in charge of notifying the views about changes in the data model and application state.

2. The Controller is the interface between users and the model, it is the core logic of the GUI and thus defines the application behavior. Furthermore, it turns user actions into changes in the model and selects the appropriate views.

3. The View are visually (or using other output channels if the user interface is multimodal) rendering the model’s content. They also request updates form the model and communicate user actions to the controller.

As shown on Figure 4.12, the gateways UI structure is inspired from the MVC framework. The VerticalView and HorizontalView of a gateway represent the views, the concrete GatewayController is the controller and the model is the positioning information acquired from a sensing layer as well as the service description obtained using the second and third layers of the E2ESD model.
4.2. Implementing the Gateways User Interface

4.2.2. Main Toolbar

The GUI provides an additional toolbar for operations that are not directly related with the identification and consumption of pervasive service. This toolbar comes in two different versions: the MainToolbar (Figure 4.13, right part) is meant for mobile users. It offers all the functionalities required to interact with the services. The AdminToolbar (Figure 4.13, left part) extends the MainToolbar with additional functionality. Both toolbars have different menus (zone 1, Figure 4.13) but their layout and other elements are common.

Zone 2 offers the three most important functionalities. It order to make them easily accessible and identifiable on various types of devices the functionalities are represented by three big buttons. The first one (from the left) is used to start the application (i.e. start showing the gateways). The next button is used to swap between the scanning and conditional modes. The last button is used to cleanly shut down the application.

Zone 3 is the status bar. A message is displayed there whenever the application wants to communicate something about its state to the user. For instance when swapping between modes this zone will inform the user about the mode currently in operation.

Zone 1 is the menu bar. As mentioned before the menu is different for both toolbars. The MainToolbar provides two basic menus that can be used to get more information (About) or exit the application (File). In the AdminToolbar two additional menus are available. The first one (Admin) offers access to two operations. “Provide Services” can be used to start and stop services offered by the machine it runs on. “Start Relate” is used to initialize the underlying ad-hoc sensor network feeding the user interface with relative positioning information using Relate (see Chapter 5).

4.2.3. Spatial Simulation Tool: Wizard

size. Accessible through the “Simu” menu of the AdminToolBar, the spatial simulation GUI permits to “fake” the spatial context by simulating the user’s position within the covered area. Creating such a tool was particularly important for two reasons. First of all it permits to test the spatial discovery layer (and thus the gateways user interface) without actually requiring a spatial information system. This eases prototyping and user studies a lot. Secondly, the RelateGateways architecture was developed in tandem with a specific sensing technology for direct sensing of the spatial arrangement between colocated devices (described in Chapter 5). In order for one project not to be influenced by the limitation of the other too much they were intentionally developed independently. As a consequence, we needed a way to simulate the sensing technology for the design of the gateways user interface.

From a user centric point of view the simulation GUI is what is commonly called a “Wizard of Oz interface”. Besides being the title of one of the most well-known children book ever printed, the WOZ was first introduced as a concept for user studies in [Kel83]. Its aim to simulate parts of a system in order to run a user study on a not yet fully-implemented application. In such a study the person simulating a feature is called the “Wizard”. It is particularly useful in situations where it is known to the researchers that the results of a user study would be biased by using a not fully-working technology.
4.3. Spatial Context Toolkit

As shown on Figure 4.14, the RelateGateways WoZ interface is a simple swing frame providing a map view of the environment. The whole window represents the interaction space, the X-axis and Y-axis display the position within the room in centimeters. Devices providing pervasive services are symbolized by red rectangles. Their size is relative to the device.

The user’s position is represented by a blue dot. To update the user’s position the wizard simply needs to click on the grid. If the wizard wishes not only to simulate the position but also the orientation he can click and move the mouse towards the user’s direction. When the mouse button is released the interface will draw a blue line starting from the dot and oriented towards the simulated direction.

Technically speaking, when the simulation interface is started it connects to the mobile device using Java RMI (Remote Method Invocation, see [29]) and gets a lightweight version of the devices the client already discovered. Whenever the wizard simulates the user’s position or orientation the simulation interface sends an object containing the new position to the application’s instance running on the client. On the mobile device side (client) this information is consumed by a toolkit on top which the gateways user interface runs. This latter component is called the Spatial Context Toolkit and is the matter of the next section.

4.3. Spatial Context Toolkit

Whilst operating in scanning mode (see Subsection 4.1.2), positioning information emanating from a sensing technology can be directly mapped on the user interface, determining the gateway’s position. In conditional mode, however, the spatial information needs to be further processed to determine which gateways shall appear and which shall not.

To formalize the differences between both modes let us propose two pseudo-codes defining what to do with the gathered positioning information in each case. We begin with the scanning mode:

```java
1) positions = getRelativePositionsOfDevices();
2) foreach (device in positions) {
3)   showGatewayForAt(device, position);
}
```

and follow with the conditional mode:

```java
1) positions = getRelativePositionsOfDevices();
2) foreach (device in positions) {
3)   foreach (condition in device) {
4)     result = evaluateCondition(condition, position);
5)     if (result is true) {
6)       showGatewayForAt(device, position);
7)     }
8)   }
9) }
10)
```

From the pseudo code we can identify the fact that the conditional mode needs to check all the conditions attached to a device prior actually showing a gateway. Recall that a condition is a rule using spatial properties. A typical example of spatial condition would be: “the user is closer than 30 cm from the device”, “the user is facing the device” or “the user is within a zone which comprises at most two devices”. 

Modeling and evaluating spatial conditions is the aim of the Spatial Context Toolkit [Str07]. Presented by Sara Streng as a Master Thesis at Lancaster University, this toolkit is intended for applications that use spatial information to trigger actions. It is composed of a number of Java classes that can be used to model and evaluate spatial conditions. Since the SCT and the gateways interface were developed in tandem it was natural to base the gateways UI on this toolkit. The Spatial Context Toolkit is well documented in [Str07] and therefore will not be extensively described in this thesis. Instead, we will focus on the integration of both software.

The concept of this integration is quite straightforward as shown on Figure 4.16. The toolkit defines an abstract object called Condition. When this object is evaluated to true the system triggers the corresponding abstract interface: Action. To integrate the gateways UI we define concrete spatial conditions extending the abstract Condition object and a concrete Action which shows a gateway.

Let us detail a bit more the integration using Figure 4.15 (source: [Str07]). The core of the SCT is designed around a handful of objects as shown on the yellow part of Figure 4.15. The Trackable class represent objects that the toolkit can monitor. In the RelateGateways architecture we want to trace devices providing pervasive services, thus RGW Devices naturally extend the Trackable class. Furthermore, we’ve seen that the visual representation of a TrackableDevice in the gateways UI is controlled by a GatewayController. As a consequence we assign a concrete Condition to each concrete GatewayController. A GatewayController has to implement the Action interface which enables it to show/hide the gateways whenever the SCT evaluates the condition to true. Finally, the SCT is centered around a Monitor class. This class is the integration point between the toolkit and a positioning system. Thus, it is this class that we feed with positioning information, whether it comes from the Wizard of Oz interface (spatial simulator) or from the Relate sensor network.

### 4.3.1. Making the SCT Dynamic

The concept of the SCT is really clean and well documented. However, it has one weakness: it does not support dynamic environments. The primary usage of the SCT was to support user studies requiring spatial conditions. As a consequence it was designed for application in which every actor in the environment is known at compile time. While this assumption is fine for a lab study where everything can be planned in advance, it is not for the RelateGateways application which needs, by essence, to be highly dynamic. The motto of this thesis is to address spontaneity in mobile interactions. Spontaneity is correlated with dynamism. Spontaneous systems are able to adapt “on the fly”, during run-time.

At the early stage of the integration we did not identify this problem. However, when implementing both layers 2 and 3 of the End-to-End Service Discovery model we realized that a minor refactoring of the SCT to make it dynamic was required.

To better understand this issue and the required modifications, let us expose the OnlyTrackableWithinZone condition provided by the SCT before and after the refactoring process. Listing 4.1 is the source of this class before the modifications, Listing 4.2 is the new version. The major difference is the defineInvolvedTrackables() method appearing in the dynamic version. This method is defined as abstract in the refactored Condition class.
4.3. Spatial Context Toolkit

```java
package uk.ac.lancs.relate.trigger.conditions;

public class OnlyTrackableWithinZone extends ZoneCondition {
    public OnlyTrackableWithinZone(Application app, String id1, Zone zone) {
        super(app, id1);
        super.setZone(zone);
        // since we need to check if any of the other trackables
        // is in the zone, all trackables are involved!
        ArrayList Trackable involvedTrackables = app.getTrackables();
        super.setInvolvedTrackables(involvedTrackables);
    }

    /**
     * Return true if the condition is fulfilled, else false.
     * @return true if the condition is fulfilled, else false
     */
    public boolean evaluate() {
        Trackable trackable = super.getTrackable();
        if (inZone(trackable)) {
            // get number of trackables in this zone
            if (super.getNrTrackablesInZone() == 1)
                return true;
            else
                return false;
        } else
            return false;
    }
}
```

Listing 4.1: The OnlyTrackableWithinZone class before the refactoring process.

As a consequence, every concrete class extending a Condition needs to implement it. The initial version defined the involved trackables once forever, at initialization time, i.e. in the constructor of concrete Conditions. The new version ensures that the Trackables (i.e. TrackableDevices in the case of the gateways UI) involved when evaluating a condition are re-defined every time the condition needs to be evaluated during run-time. This way, when the user discovers a new device the OnlyTrackableWithinZone condition takes it into account as well.

Most importantly, when exporting the Conditions from the service provider to the mobile device, the system does not reference local TrackableDevices anymore since these are available on the service provider only. The export of Conditions to the mobile devices is further detailed and discussed in Chapter 9.

The refactored version of the SCT is available in the source code of the RelateGateways architecture, in the packages uk.ac.lancs.relate.trigger.conditions and uk.ac.lancs.relate.trigger.zone. Thanks to their modular architecture the classes in the other core packages of the Spatial Context Toolkit did not have to be modified.
package uk.ac.lancs.relate.trigger.conditions;

public class OnlyTrackableWithinZone extends ZoneCondition {
    public OnlyTrackableWithinZone(String id1, Zone zone, String deviceId) {
        super(id1, deviceId);
        super.setZone(zone);
    }

    /* This method defines the trackables to be considered as involved when evaluating the condition.
    * Since we need to check if any of the other trackables is in the zone, all trackables are involved. */
    public void defineInvolvedTrackables() {
        super.setInvolvedTrackables(super.getApp().getTrackables());
    }

    /**
    * Return true if the condition is fulfilled, else false.
    * @return true if the condition is fulfilled, else false
    */
    public boolean evaluate() {
        Trackable trackable = super.getActiveTrackable();
        if (inZone(trackable)) {
            // get number of trackables in this zone
            if (super.getNrTrackablesInZone() == 1)
                return true;
            else
                return false;
        } else
            return false;
    }
}

Listing 4.2: The OnlyTrackableWithinZone class after the refactoring process.
Figure 4.11.: Evolution of the gateways’ design.
4.3. Spatial Context Toolkit

Figure 4.12.: The MVC pattern applied to the gateways UI.

Figure 4.13.: The administration (left) and main (right) toolbars.

Figure 4.14.: Wizard of Oz interface used to simulate the user’s spatial context.
### 4.3. Spatial Context Toolkit

**Figure 4.15.** UML class diagram of the SCT/Gateways UI integration.

**Figure 4.16.** Concept of the SCT gateways UI integration.
5

Realtime Spatial Provisioning

5.1. About the Relate Project

The subject of this thesis is part of the RELATE project: Relative Positioning of Mobile Objects in Ad hoc Networks [26]. Lancaster University leads this European project involving six universities and engineering schools. Its motto is to study the means and benefits of relative positioning for mobile devices and object.

Into more details, Relate focuses on the spatial arrangement of the devices and objects, free of surrounding infrastructure. In global terms the research program of the project addresses the following aspects:

- Measurement of the spatial relationship between mobile objects with embedded sensors.

Positioning information is core to the Spatial Discovery layer of the End-to-End Service Discovery model. In order to provide a spatial mapping of the user’s view of his environment on the mobile screen we need to gather spatial context from some system.

Many systems provide this information. However, as mentioned before the Relate ad-hoc spatial sensing system is quite adapted. It provides real-time information, it is ad-hoc and can thus be deployed within minutes. It also provide relative positioning. Spatial (or positioning) information is said to be relative if it is expressed by taking the information’s requester as the origin of the Cartesian system. Thus, if a user polls a system for relative position, the system will return the location of all the objects relatively to the user’s position.

Relate fits most of our requirements and hence was a fair choice. This chapter introduces the Relate project and discusses the extension we have been working on to even better fit the needs of the RelateGateways project.

5.1. About the Relate Project

5.1.1. Introducing the Relate Sensor Nodes

5.2. Extending Relate

5.2.1. Relate Dots

5.2.2. Software Integration

36
5.1. About the Relate Project

- Scalable communication and organization for ad-hoc networks of relative positioning devices
- Algorithms for distributing localization within the sensor network.
- Architecture and software platforms for relative positioning system.

The initial stages of the project have been concerned with creating sensing technologies and systems appropriate for collaborative positioning on surfaces. Such systems are particularly relevant in the fields of mobile computing, computer-supported cooperative work, and user interfaces, where the locations and orientations of objects on surfaces are often required. To address these issues newly created, small, wireless devices perform peer-to-peer sensing and produce relative location and orientation estimates without relying on pre-existing infrastructure. In the current project's stage, teams of researchers work on improving the sensing technology. Other researchers focus on concrete use cases of this ad-hoc sensor network.

The RelateGateways project is part of this latter focus. Nevertheless, the idea was to develop it as decoupled as possible from the spatial sensing technology. This strategy presents two main advantages. First of all it permits to explore the concrete application without depending on the limitations of the sensing technology. Secondly, since the sensing technology and RelateGateways are decoupled it enables the application to be tested and adapted to many positioning technologies, not just Relate.

An extension of the Relate sensing technology was designed and developed in tandem with the RelateGateways application. The aim of this new version of the sensor network was to fit the needs of spontaneous mobile interaction with pervasive services.

Rather than exposing Relate's complete history this chapter focuses on the extension showing what was available and which adaptations were required. Additional information about the Relate hardware and overall project can be found in about 30 publications listed and available on the project's website [28]. However, to have a better understanding of the achieved extension we shall start with a brief discussion of the general concepts on which the Relate sensor nodes are based.

5.1.1. Introducing the Relate Sensor Nodes

Each hardware sensor of the Relate system is a wireless sensing node built from custom hardware and using dedicated protocols in order to create an ad-hoc sensor network for relative positioning [KKG05]. All the nodes are composed (at least) of three hardware components as shown on Figure 5.1: a ultrasonic transducer (which can both emit and receive ultrasound), a Smart-Its Particle board [23] containing a PIC microcontroller and a Radio Frequency (RF) module [HKG+05].

Each node performs three basic tasks.

1. It emits ultrasonic signals using three to four hardware transducers simultaneously.
2. It listens to incoming ultrasound signals. Using the peak signal and time-of-flight of these signals a node can estimate the distance and angle to another node.
3. It transmits the observations processed in 2 to other nodes using a Radio Frequency network channel.
5.2. Extending Relate

Using slightly adapted versions of these three tasks all the sensors of the Relate system form an spontaneous network able to sense the relative position of peers.

Before the RelateGateways project started, the Relate hardware was composed of "Dongles" (see Figure 5.1). The particularity of these sensors nodes is that even if the network they form is completely had-hoc and does not require any pre-existing infrastructure, the nodes are connected to a computer and hence are not stand-alone: periodically each dongle uploads the observations it made over its USB link to its host. The data is then further processed on the computer using the Relate toolkit [KKG05] and eventually made available to higher-level applications. This operation scheme is summarized on Figure 5.2.

While this might seem just fine for the gateways user interface the approach has two major problems when considering the RGW use cases. Recall that the gateways are access-points to services provided by devices surrounding the user. While some devices actually provide (as in compute) the service they stand for, most of them do not. Take
the case of a public display or a loudspeaker. The former provides a presentation service, the latter a music player service. Neither the display nor the loudspeaker execute the service they stand for. Instead, a computer (or server) connected to them by some mean and potentially located on a completely different site does. As a consequence the device standing for the service and the system actually executing the service are spatially and physically decoupled as shown on Figure 5.4. This is an issue since it means we can not connect a dongle to the server providing the service, that we shall name as “physical server”. From a user perspective we are not interested in locating the physical server but we want to identify the display, or loudspeakers. Moreover we can not expect to a connect a USB device to loudspeaker or to a display or any pervasive service. This assumption would seriously limit the services one can offer. Hence the idea of designing a new type of Relate sensor that is fully stand-alone: the Relate Dots.

5.2.1. Relate Dots

Application Concept

The mere idea is to “stick” one of these nodes onto every device. Additionally, we connect a Dongle to the mobile device. This setting is shown on Figure 5.3. A dot is stuck on the screen “providing” the display service, a Dongle is connected to the OQO mobile device.
5.2. Extending Relate

Conceptually, the system does not change a lot. The Dots are now stand-alone and communicate the observations amongst themselves as well as to the Dongle which will further communicate this information to the mobile device through its USB interface. In the current state of the integration the mobile device will then process this information using the Relate toolkit.

![Diagram](image1)

**Figure 5.4.:** Extending the Relate model for RelateGateways.

**Hardware Implementation**

The Dots were developed in tandem with another class of Relate sensors called Bricks (see Figure 5.1). These devices are hybrid since they can be both stand-alone or connected to a USB interface. Dots are basically stand-alone only Bricks with a reduced form factor in order to make them easier to stick onto a broad range of pervasive devices.

![Image of current Dot and expected size of final Dot](image2)

**Figure 5.5.:** Current Dot (left) and expected size of the final Dot (right).
5.2. Extending Relate

The left part of Figure 5.5 shows the current size of a Dot whereas the right part is the expected form factor of the next version of the Dots. In terms of hardware the Dots are similar to the other relate sensors and dispose of three ultrasound transducers, a PIC microcontroller as well as an RF module. For the sake of documentation, the electronic schema of the Dots’ boards is shown on Figure 5.6.

![Figure 5.6: Board of the Dot 1.0 Sensor Node.](image)

5.2.2. Software Integration

At the time of writing a first integration was undertaken. It is worth noting that it is more a “proof of concept” integration since more filtering work needs to be done in order for the user experience to be acceptable.

![Figure 5.7: Structure of the Relate Toolkit.](image)
5.2. Extending Relate

```java
public void updatedModel(Model model) {
    // iterate over the model to see what happened */
    for (Service currentService : model.getServices()) {
        String deviceIdStr = Integer.toString(currentService.getDeviceId());
        // this device was already discovered, just inform the monitor
        // about the position update... */
        if (deviceIdStr.equals("91")) {
            // do noting, it is the user's number
        } else {
            log.fine(currentService.toString());
            if (gwApp.isRegisteredDevice(deviceIdStr)) {
                sendUpdateToMonitor(currentService);
            } else {
                // this device is yet to discover ! Inform the
                // ServiceDiscovery manager. */
                manager.discover(currentService.getHostname());
            }
        }
    }
}
```

Listing 5.1: Method triggered on updates of the Relate model.

Basically, the RelateGateways application uses the Relate toolkit. Since this latter was programmed in Java as well, the integration was seamless in terms of programming language. The toolkit converts sensing observation into a higher level of abstraction and builds a model containing all the positioning information it could gather, as shown on Figure 5.7 (source: [GG07]). The information enclosed in the model is relative to the user. The model will contain tuples like: node 76, x=400, y=300, angle=75deg Which basically means that the node 76 is on coordinate 400,300 (the origin of the Cartesian system being the user’s position) with a relative angle of 75 degrees.

To use this model in order to feed the gateways user interface and make the gateways move, a listener is triggered everytime the Relate model gets updated. The core method of this listener is shown on Listing 5.1. When getting a new model we start by iterating through it. For every entry (i.e. Relate Service) we check whether the corresponding device was already discovered. If not we ask the discovery manager to discover it (this part is the matter of Chapter 9). If the device is already discovered but its position was updated we ask the monitor of the RelateGateway’s user interface to update the gateway’s position.
Part II.

Towards a Software Architecture to Support Spontaneity.
In technical terms spontaneity is all about dynamism. The more dynamic an application is, the fewer hardcoded bits it contains, the more it is likely to support spontaneity. Hence, designing a software architecture enabling spontaneity is about designing a highly dynamic system. Towards this aim machine discovery, interoperability and service oriented architectures are the keywords.

In the first part of this work we addressed user discovery (i.e. localization) of services (see Chapter 4). The user was in the center of what we define as the spatial discovery process. This is part of the E2ESD but it is not yet sufficient. To enable spontaneity in the interaction with the visually discovered service one needs to address machine discovery as well. Once a service was visually discovered, we ought to find a way for the mobile device to establish a software and hardware link with the device running the service. This mechanism called network discovery or service discovery\(^1\) is the second layer we shall discuss.

While many discovery systems stop at this point, establishing a link between the client (mobile device) and the devices running services it is still not enough to enable spontaneous interactions. Once a service is visually discovered, once a link between the device providing it and the mobile client is established there still need to be an agreement or contract on how to use or consume the service. This is especially important since the client does not know in advance the services it will encounter and hence, needs a way to dynamically consume them without prior knowledge of their semantics. This is the concerns of the third layer of the E2ESD: the invocation and interoperability layer.

The present part follows this logical structure. The general concepts of network and service discovery are first discussed (Chapter 7). Invocation and interoperability mechanisms are then introduced (Chapter 8). In order to get a better feeling of the existing industrial solutions and research projects and a basic taxonomy of discovery systems tailored to the project is provided (Section 8.3). To finish with we propose a concept and concrete implementation to address the exposed concerns and combine it with the user interface for spatial discovery exposed in Chapter 4.

Put together these sections constitute the basic of what is comonly called SOA or Service Oriented Architectures. As a consequence it does make sense to begin with a small insight of this set of best-pratice patterns and show how it influenced the RelateGateways architecture.

\(^1\)For the sake of readability, when talking about “discovery” through the rest of Section, we implicitly refer to network or service discovery, not to spatial discovery which is the matter of Chapter 4.
Amongst all the buzz-words of the software engineering field, SOA is probably number one. Behind the hype of this concept lies a real value that has been proved many times already. SOA is slowly, but definitely shifting the programming paradigm of application softwares, as stated by the Gartner group in [Nat03]:

“Through 2007, growing enterprise experience with SOA process and SOA-based applications will eliminate the myths and instill appreciation for the real benefits of SOA in most enterprises. [...] This change will shift the massive software industry mainstream into the new software-engineering reality: By 2008, SOA will be a prevailing software-engineering practice, ending the 40-year domination of monolithic software architecture (0.7 probability). [...]”

SOA is not new, not really innovating, nor a technology. It is simply a concept reflecting the evolution of computing. Computer software used to be designed in a monolithic manner. The machine was in the center and was designed to solve or compute all the problems
on its own. Two major evolutions questioned these practices. Distributed Computing permitted to connect computers with one another, enabling machine communication. With languages like SmallTalk-80, Eiffel or Java, Object Oriented Programming tended to show that softwares could be thought of in modular terms, decomposing big tasks into smaller units according to a certain hierarchy. This is practice is known as Component Based Architecture.

The combination of both component based architecture and distributed computing is commonly called a SOA or Service Oriented Architecture: a set of best practices towards a distributed component based architecture. The evolution that lead to Service Oriented Architectures is illustrated on Figure 6.1.

The primary goal of all SOAs is to ensure reusability. While this goal is shared with components architectures SOAs' distributed dimension pushes the limit of reusability by enabling re-purposing of specific components. A component is not programmed as part of a system anymore, it is instead meant as a single, atomic unit that one may combine with others to form a complex system.

While this Section does not pretend to be a cookbook on SOAs, it offers a summary of the basic concepts in order to support our implementation choices (see Chapter 9).

### 6.1. Concepts of SOA

![Figure 6.2: Basic pattern of actors and interactions for a Service Oriented Architecture.](image)

Nowadays SOAs are mainly applied to pure business architectures (e.g. banking systems, e-commerce, etc.) mainly because of the dawn of webservices. But webservices are not
6.1. Concepts of SOA

SOAs. A SOA is a set of best practices whereas webservices are a set of technologies. We can imagine the concept of SOA as being a cookbook and webservices as being some ingredients that one can use to “cook a SOA”. In these terms [Ort05] provides a good insight of a savory SOA-Webservice mix. It is important to realize that other ingredients such as EJB (Enterprise Java Beans) [10], ebXML [9]1 can be used to “cook a good SOA” as well. Moreover, the concept is not constrained at all to be applied to the field of business software. When exploring the world of pervasive services we consider a service oriented architecture as well. Since we want the mobile user to be able to interact with a number of services offered by devices surrounding him, we need to address a distributed component based architecture.

Now that we know what a SOA is not, let us expose what it is by exposing the pattern’s concepts. According to [Nic05] an architecture can be called SOA if it addresses the following concepts:

- Services.
- Service description.
- Advertising and discovery.
- Specification of an associated data model.
- Service contract.

We will take a slightly different approach by organizing our description according to the main components or actors of a SOA rather than the concepts themselves:

- Service.
- Client.
- Directory.

The interactions between these actors, often called resources are pictured on Figure 6.2. While the pattern on this figure represents the minimal architecture that one may call SOA, it omits a number of elements making the SOAs’ best practices an interesting guideline for modern pervasive applications. Thus, we enhance the model with a thinner descriptions of its actors and interactions.

6.1.1. Service

In general computing terms a service is defined as:

“Functionality derived from a particular software program. For example, network services may refer to programs that transmit data or provide conversion of data in a network. Database services provide for the storage and retrieval of data in a database. Web services are applications that interact with each other on the Internet (see Web services)[1]”

1EbXML is an initiative of OASIS (Organization for the Advancement of Structured Information Standards) providing a number of standards and specifications for the electronic business.
6.1. Concepts of SOA

In the context of SOAs we can refine this definition. Services are the functional endpoints of modern architectures. A service is thus a self-contained functionality or behavior implemented and exposed by a component and consumed by another.

A service on its own is not really interesting since it is only an abstraction of a particular functionality. What makes the value of services in the context of SOAs is their global accessibility. Services offer interfaces making them available to all sorts local and distant clients. While there is nothing fundamentally new in this approach core to the art of Distributed Computing, the important point is that SOAs propose to make this availability as loosely-coupled as possible. Loosely-coupled functionalities are the motto of SOAs. In essence it means that a client does not know anything about the service implementation, symmetrically a service does not need to know anything about the client consuming it. Nevertheless, the way clients and services communicate together must be standardized. More precisely all the actors need to be aware of clear protocols for service retrieval, service publication and service invocation.

It is worth noting at this point that a service is not necessarily atomic. According to its definition a service is a behavior, which can offer various operations. As a parallel the service could be a Java class (not atomic) and the operation level a method (atomic). Thus, a service could be but does not need to be atomic.

SOAs do not give technical details about the implementation of a service or any other component, instead they provide a conceptual, high-level view of what these components should do. A service can be a Java class running on a server (as it is in our case), as much as a microcode on a sensor-node, as long as it complies the the SOA conceptual view of a service.

As an example Listing 6.1 is the source code of a plain web service (W3C standard) implemented in Java EE 1.5 using the Netbeans IDE [21]. The behavior it implements is arithmetic (class level), the operations it offers are the addition and the substraction (operation level).

```java
@WebMethod
public int add(@WebParam(name = "a") int a, @WebParam(name = "b") int b) {
    return a + b;
}

/**
 * This operation can be used to subtract up two integers.
 */

@WebMethod
public int sub(@WebParam(name = "a") int a, @WebParam(name = "b") int b) {
    return a - b;
}
```

Listing 6.1: Implementing the MathWebService in Java EE.

For the sake of consistency we also provide the same service modeled using another service oriented system. Listing 6.2 is the Math service implemented as a Session Bean using the EJB 3.0 framework.

```java
public int add(int a, int b) {
    return a + b;
}

public int sub(int a, int b) {
    return a - b;
}
```
Listing 6.2: Implementing the MathSessionBean in EJB 3.0.

Service Provider

The service provider is the resource offering a service on the network. The term network here is to be taken in its large sense of connecting one resource to another, not constrained to any particular technology. Conceptually a provider is the opposite end-point of a client. Implementing a service is not the only tasks of a provider. Indeed, towards the loosely-coupling goal exposed before, it also needs to describe the service and advertise it. This is the role of the service description and the data model that we describe below.

Service Description

The service description represents a contract governing the mechanics of interactions with a service. Additionally, it should contain a number of fields describing the service to allow its (machine or human) retrieval from a directory. Several standardized languages propose to describe services. Again, building a SOA does not advocate for any of these. The pattern only states the fact that every resource in the system should be able to understand the provided descriptions, whatever form they take. The Descriptive Service Architecture (see Subsection 9.1.2) of the RelateGateways application is a valid service description in SOA terms, since every actor of our system understands it and can use it to consume a service.

Amongst the standardized service description languages, WSDL (Web Services Description Language) is a key player. Created by the W3C (World Wide Web Consortium) as part of the Web Service standards, this language is mostly cited as an example when talking about service description. The difference between such a global (business-oriented) standard and the descriptive architecture of the RelateGateways system is great in terms of implementation. Yet, both ideas pursue the same simple goal, describe services in a way such that all the actors of the system can understand it. Whereas Webservices target a global acceptance (i.e. world-wide business) we work at an another level (i.e. rapid prototyping of spontaneous mobile interactions with pervasive services) and thus need a different approach.

Listing 6.3 provides an example of service description. It basically defines the operations (or methods) offered by the service and their parameters. Since all the actors of a web-service infrastructure know how to read and interpret the WSDL, this file will permit clients to know how to interact with this the MathWebService.

```xml
</xsd:schema>
</types>
<message name="add">
  <part name="parameters" element="tns:add"/>
</message>
<message name="addResponse">
  <part name="parameters" element="tns:addResponse"/>
```
6.1. Concepts of SOA

Listing 6.3: Extract of MathWebService’s WSDL file.

Once again, Listing 6.4 and Listing 6.5 provide a description of the same service when implemented using the EJB framework. The service description takes the form of two Java interfaces called MathSessionBeanLocal and MathSessionBeanRemote.

Listing 6.4: The description interface for local calls to the MathSessionBean service.
6.1. Concepts of SOA

```java
package sessionbeans;

import javax.ejb.Remote;

// This is the business interface for MathSession enterprise bean.
@Remote
public interface MathSessionRemote {
    int add(int a, int b);
    int sub(int a, int b);
}
```

Listing 6.5: The description interface for remote calls to the MathSessionBean service.

Data Model

All in all, consuming a service is similar to invoking a method. As such there is a need to specify the parameters of this call. In a SOA world specifying the format of parameters is part of what we call the common data model. A specification meant to ensure that both parties of an interaction do understand the data and know what to expect in terms of parameters.

In the Web service world the data model is formalized using the WSDL language. As an example in the WSDL extract of Listing 6.3, lines 33-35 express a part of the data model as they define the type of parameters (literal) required to consume the add operation of the MathWebService.

In the EJB world the data model is expressed through the methods definitions in the local and remote interfaces. In Listing 6.5 line 12 specifies the types of parameters (int) required to invoke the add method of the MathSessionBean.

Service Contract

The service contract groups all the metadata that describe how the different parties will interact. It comprises the service descriptions and the data models. Additionally, it defines topologies, interaction models and communication mechanisms. A service contract is in essence a higher, and often not fully formalized level of abstraction regrouping everything that defines the way the interaction will occur. From technical details such as the protocols to be used to transmit data, to higher-level descriptions such as the service usage or security policies.

6.1.2. Client

Obviously the client in a SOA is the entity wanting to consume a service. It is often called service requester or service consumer. A client can be just about any kind of software component able to consume a service, ranging from a C command-line program or a dynamic PHP webpage to a Java SE program. A user interface could be the client of a service but a service can be the client of another service as well.

In the case of the webservice described above, the client is a simple JSP (Java Server Page) which consumes the MathWebService. Similarly, in the EJB project, the MathSessionBean can be invoked using a dynamic webpage (such as a JSP or PHP page).
6.1.3. Directory

Directories is to a service what yellow pages are to businesses: a way to advertise to the world. It offers a way clients to find services and for service to publish part of their contracts. A directory can be implemented in plenty of different ways. However, every directory should offer a registration system in order for services to advertise and a polling or querying interface in order for clients to search for services. Furthermore, it should be able to return a handle on the services it references in order for clients to be able to contact the listed services.

In the case of webservices the UDDI (Universal Description, Discovery and Integration) server is a concrete instance of directory. It is worth noting that a directory is not necessarily a centralized server. As an example, Bluetooth services advertise themselves by broadcasting information to other Bluetooth services within range.

6.1.4. Why SOA?

Recall that components in SOAs are not embedded as part of the same unit anymore, as a consequence SOAs enable highly dynamic architectures, where the implementation of an application’s core business is picked at runtime. SOAs best practices help to achieve reusability, interoperability, scalability and flexibility.

Since layers two and three of the E2ESD model form the bricks of a SOA for spontaneous mobile interactions, this chapter helps to understand our design choices. The RelateGateways architecture was deeply influenced by the blueprints exposed in this chapter. They provided a conceptual basis on which we built the RelateGateways service framework.

The following chapters describes the building blocks of a SOA into more details, focusing on discovery, invocation and interoperability. Finally, Chapter 9 relates how the RelateGateways SOA was concretely implemented.
In the vision of Ubicomp, the use of pervasive computers has to become so natural as to disappear. Users should be free to focus on the core of their tasks rather than on the tools’ usage: in order to limit the loss of fingers, one wants to be able to focus on the nail rather than on the hammer’s usage when hammering. Similarly, when using a pencil we focus on the drawing rather than on the pencil’s usage. Facing these simplistic examples with the reality and underlying complexity of computers is questioning. Pervasive environments are full of computers providing functionalities. Interconnected with one another these functionalities form an immense computing power. In a perfect world mobile users should be able to use this power just as easily as they manipulate hardware tools.

Discovery mechanisms do not fully achieve this (utopic?) goal, yet they represent an important step towards networked systems diminishing the needs for human administration. Indeed, these systems enable services and devices to become aware of each other without
requiring human configuration or activation. The only assumption being for the peer to be networked with one another.

Services and devices can advertise themselves using the discovery systems and, hence become available to all the actors of the system. The clients can then query the discovery system in order to retrieve services. As resources (i.e. services or devices) join and leave the network, the discovery system is updated. This makes architectures based on discovery truly dynamic, an essential feature to achieve spontaneous interactions with pervasive services. It is these mechanisms that we describe in the next subsections.

### 7.1. Service Discovery in a Mobile Context

Since discovery mechanisms are a core component of ubiquitous computing systems, plenty of them were and still are developed both at research and industrial levels. However, all the systems are based on the same generic model pursuing the simple goal of linking computers with one another in a dynamic fashion and without the need for human administration and configuration.

We focus our exploration of discovery systems on mobile interactions with pervasive services. Additionally, the goal here is to provide a generic model of discovery in this context. Thus, concepts and mechanisms are included in this model if and only if they are common to most of the discovery systems. It is worth noting that base our study on a number of surveys of systems for pervasive and ubiquitous computing such as [ZMN05, Sun06] and [Edw06]. We can start by extracting four main components and actors:

1. **Clients** are mobile devices (e.g. PDAs, tablet PCs, hand-held, smart phones, laptops, etc.) discovering services.

2. **Registries**, also called registers, directories or lookup services are devices containing information about the available pervasive services.

3. **Devices** are electronic devices running $1..n$ services, more generally called service providers (e.g. desktop, laptop, workstation, server, mobile devices, etc.).

4. **Services** are atomic functionalities provided by devices.

Let us explore the interactions between these components taking a bottom-up approach, from services up to clients.

#### 7.1.1. Services and Devices

Services are the end-points of discovery mechanisms. They represent atomic functionalities offered to the clients. “Playing a song” is a service provided by a multimedia station, “printing a file” is a service offered by a printer. Thus, a service is nothing without the machine running it. As a consequence services are said to be offered by devices.

In this context a device can be just about any entity fitting two criteria: (i) it encloses a certain amount of computing power (ii) it has the ability to be networked. While the former criterion is probably clear, the latter needs a bit more attention. As mentioned before the main assumption for actors of a discovery system is for them to be somehow
7.1. Service Discovery in a Mobile Context

import javax.jmdns.*;

JmDNS jmdns = new JmDNS();
jmdns.registerService(new ServiceInfo("_http._tcp.local.", "_Printer Lexmark 41 C Floor._http._tcp.local.", 1234, 0, 0, "path=www.guinard.org/guinard")
);

Listing 7.1: Service registration in DNS SD, using JmDNS

able to communicate with the actors they need to interact with. Discovery is a general model that does not enforce any particular protocol, hardware feature or technology, only implementations of discovery do. Pairs in a discovery system can be networked with one another over an ethernet network, an RF link, an ultrasound channel, an optical recognition system, etc. Similarly, a discovery system is not bound to use a single communication technique, as long as bridges exist between the technologies everything is fine.

For instance, the discovery system proposed in this thesis (see Chapter 9) uses ultrasound, RF (Radio Frequency) together with wireless and wired ethernet. Jini’s Surrogate Architecture is a good example on how to deal with heterogeneity amongst the communication protocols and technologies [30].

Additionally, it is worth noting the connection between clients and registries as shown on Figure 7.3. In a number of discovery systems clients are (or become) service providers as well. Thus, devices are sometimes clients as well and clients sometimes become devices as well. An architecture is called symmetric if all the clients are services providers as well. The discovery protocol of the system we present (see Subsection 9.2.1) is symmetric.

7.1.2. Directories and Registration

Devices willing to offer services need to inform clients about them. This mechanism often called service advertisement or simple service registration is done using a registry or directory. In general terms the device contacts the registry and publishes information about the services it provides.

![Diagram of registry and non-registry topologies](image)

Figure 7.1.: Registry (right) and non-registry topologies (left).

As shown on Figure 7.3 a particular instance of a discovery system may comprise several registries. Some architectures take the extreme approach and turn each service provider
7.1. Service Discovery in a Mobile Context

Listing 7.2: Service registration in jSLP

```java
/* get Advertiser instance */
Advertiser serviceRegistry = ServiceLocationManager.getAdvertiser(new Locale("en"));

/* create service, set type to printer and time to live to 60 seconds */
ServiceURL printingService = new ServiceURL("service:printer:printingService://eis.lancs.ac.uk", 60);

/* attributes of the printing service */
Hashtable attributes = new Hashtable();
attributes.put("persistent", Boolean.TRUE);
attributes.put("name", "Lexmark40LX");
attributes.put("max-connections", new Integer(5));
attributes.put("color-supported", Boolean.FALSE);
attributes.put("document-format-supported", "application/postscript");

/* register service */
serviceRegistry.register(printingService, attributes);
```

into a registry. This explains the connection between registries and devices on Figure 7.3: in some architectures the devices are the registries. This method was chosen for the Bluetooth SDP (Service Discovery Protocol) [4] or for the system described in this work. In the specialized literature ([Sun06] for instance), we call the latter topology a *non-registry* architecture since no server is specialized in service registration. On the other side, the former topology is called a *registry* architecture, meaning that 1..n resources are dedicated to service registration. This concept is summarized on Figure 7.1.

The syntax and semantics of the information submitted to a registry is often standardized (as in UPnP with the WSDL, Web Service Description Language [32]) or partially standardized (as in Jini [15] or in DNS SD [7]). Typical elements of these standardized descriptions are, resource identifier (canonical name), service description, service attributes, access protocols, URLs, icons, service type, etc.

Listing 7.1 provides an example of service registration using the Java implementation (JmDNS [16]) of DNS SD. Listing 7.2 is a code snippet needed to register a printing service in the Java implementation of SLP (Simple Location Protocol) [17].

The most important (and only required) registration field is the *resource identifier*. We define a resource as any device or service available within the system’s scope. In turn, the scope is defined in terms of the set of resources that are discoverable by all the clients of a particular instance of a discovery system. Although the name “resource identifier” is not common to every discovery mechanisms, its semantics is shared by all the systems: it provides a *unique* fingerprint that clients use to retrieve or access a resource or an aggregate of resources.

The notion of uniqueness is not absolute. While some identifiers are universally unique, an ID needs to be unique at least within the scope of the discovery system. To have a better feeling of these identifiers let us provide some examples. Good examples of universally unique identifiers are the UUIDs (Universally Unique IDenfiers). These are used in a number of discovery system such as Bluetooth SDP or Jini. This number essentially a unique 128-bit codeword used to identify a service:

550e8400-e29b-41d4-a716-446655440000

Or the EPC (Electronic Product Code) GID (General IDenfier) [ONS06] number used in the EPCNetwork. This world-wide unique identifier is a 96-bits codeword pointing to
7.1. Service Discovery in a Mobile Context

```java
/* get the Locator instance for queries */
Locator locator = ServiceLocationManager.getLocator(new Locale("en"));

/* find all the printing services with attribute color-printing set as true */
ServiceLocationEnumeration sle = locator.findServices(new ServiceType("service:printer"), null, "(color-supported=true)");;

/* display results */
while (sle.hasMoreElements()) {
    ServiceURL foundService = (ServiceURL) sle.nextElement();
    System.out.println(foundService);
}
```

Listing 7.3: Service query in jSLP

An aggregate of services:

```
urn:epc:id:sgtin:0614141.100734.1245
```

A resource identifier does not required to be a number. Some take the form of URLs or strings. The service identifier in the DNS SD system is worth citing since it takes a slightly different approach. This system-wide ID is designed to be human-readable. The string:

```
Printer Lexmark 41 C Floor
```

on Listing 7.1 is an example of a DNS SD service identifier. The identifiers used in the initial architecture described in the thesis are similar to the ones used in DNS SD.

7.1.3. Clients and Queries

Clients need service discovery in pervasive environments because they want to use the immense computing power running around them without having to spend hours interconnecting the sources of this power. The functional end-point of this power is represented by services. Thus, in general terms a client is defined as a resource wanting to use a number of services. When applied to our context, a client is represented by a mobile device such as a PDA, a cellular phone, a tablet PC, a laptop, etc.

As mentioned before services have to announce their existence in a process called service registration. Once registered, a service can be queried for by a client. That is, a client can search for the services it wants to use by formulating requests to a registry. Although queries can be formulated using various strategies their semantics is always the same: expressing the needs of the client in terms of pervasive services. Systems typically offer “web search engines like” querying interfaces. However, the query language usually contains a number of fields that are not available when searching for web-pages on the Internet, such as: service types, service attributes, hashtables, key-value pairs, objects, UUIDs, XML description of the service, URLs, etc. As an example, in systems like UPnP, DNS SD or SLP queries take the form of URLs possibly containing wildcards. Traditional discovery systems address the search for services in a programmatic way, focusing on the way systems can find services. For instance, Listing 7.3 is a request for all the services of type Printer supporting color-printing.

While most of the discovery systems constrain their concerns for the search of services to a textual query language, other systems add an extra layer to the search for services. These (traditionally more research-oriented) system focus on the way users can find and
query for services. Most of these architectures use the physicality of pervasive services: if the user wants to use a printer he can physically approach the printer. In systems like [KB01, ENSI02, Fis06] but also in some uses of the EPCNetwork (or more generally RFID) and Bluetooth, the physical proximity is the query. Instead of formulating her query for a service with words, the user formulates it with physical proximity. Some projects such as the Kinetic User Interface [PBG+06] use both proximity and displacement to formulate queries.

In our system the queries are implicit as well. Users formulate them by entering new environments and fullfilling spatial conditions.

### 7.1.4. Naming VS Discovery

![Naming and Discovery Systems](image)

In order to fully understand what a discovery system is and is not, an important distinction between naming (or name) and discovery systems is to be addressed. In [Edw06] three main distinctive aspects are proposed. We slightly adapted them here to better fit our needs:

1. Naming systems require directory’s (registry) location to be a preconfigured parameter. Discovery systems do not have (or minimize) such requirements, in these latter systems the centralized directory is sometimes even omitted (this is the case of the discovery mechanism we propose).

2. In the case of naming systems, a directory update requires explicit human intervention. The discovery systems are highly dynamic architectures with automatic state updates.

3. Naming systems act as servers with clients polling the directories. Discovery systems usually provide asynchronous notifications as resources enter or leave the system.

To sum up these points one can state that discovery systems are real-time, dynamically adaptive systems whereas naming systems are more static and human managed.

To clarify the difference let us take two example. We start with the Domain Name System. As its name suggests, DNS is a naming system. Basically, one can consider it
as a phonebook. The client knows in advance\textsuperscript{1} the address of the DNS server and polls it with a key (URL) and it sends you the corresponding IP address (or a NAPTR record to be more precise). The clients poll the directory since they certainly do not want to be informed whenever a new address is added to the Internet as much as you probably would not like to receive a mail everytime someone adds a new phone number. In these terms, DNS is truly a naming system.

Let us now consider Jini. Sun Microsystem’s system is probably one of the most well known discovery systems. Unlike DNS, Jini is highly dynamic and notifies the users as resources enter or leave the system. To summarize we will consider the approach taken in [Sun06] were discovery systems are considered as “third generation” naming systems emphasizing the fact that the latter systems provide an added value to the pure naming. Hence, as shown on the Venn diagram of Figure 7.2 discovery systems include and enhance naming systems.

7.1.5. Summarizing the Service Discovery Process

\textsuperscript{1}DHCP (Dynamic Host Configuration Protocol), an add-on to DNS, solves this issue by broadcasting the address of the DNS server.
7.1. Service Discovery in a Mobile Context

*Devices* are computing entities. The computation capabilities they provide as a whole can be decomposed into smaller functional parts called *services*. When a device enters the environment it can *describe* its *services* using a *description formalism*. Devices then send the service description to 1..n *registries* or directories in order to advertise their services to all the potential consumers or *clients*. In turn, clients entering the environment can formulate their needs in terms of pervasive services using a *query language*. In discovery systems particularly designed for mobile interactions with pervasive services queries can also be formulated based on physical proximity or physical displacement. Once a query is formulated it is sent to 1..n *registries*.

The answer of a registry to a query is a subject on its own. This issue is the matter of the next chapter.
8.1. Consuming Services

Using network and service discovery, mobile clients can search for and get connected to pervasive services. Once these issues are solved one question remains open: “how does the client actually consumes the service”.

In a world where everything is known at compile-time the answer to such a question is very straightforward: pre-program the interaction. If the mobile device knows in advance it is going to be able to interact with a vending machine then it just needs to have a class modeling this interaction, there is now need for an invocation and interoperability model. Note than in such a perfect world there is not even a need for a discovery system at all, every feature of the targeted service is known in advance and can be hardcoded in the mobile device.
But pervasive environments are not at all similar to this situation. They are highly dynamic. A mobile device evolving in such an environment does not know at compile-time neither at run-time what kind of service it will be offered to interact with. It discovers them “on the fly” while running and hence, can only make little assumptions about their very nature.

The traditional solution to this problem is... not to address it! Most of the discovery systems (such as DNS SD, Bonjour, SLP, BT SLP, etc.) do not address this issue. They return a handle on the service and consider the client being responsible of knowing what it might interact with and how. This is fair enough for discovery systems. After all the aim of a discovery system is to help discovering services, not interacting with them.

However, in a bigger picture addressing interactions with services, there is a need for an invocation and interoperability system. SOA’s blueprints address this problem: recall the fact that a Service Contract aims for pairs to agree on generic models to describe the interaction and hence enable it.

Before digging into the concept of interoperability let us think about the common definition of this term:

"Interoperability: The capability of two or more hardware devices or two or more software routines to work harmoniously together. [...]"

In the context of pervasive services we need to extend the definition with both a notion of “service” and “dynamism”:

"The capability of a client to work harmoniously with services it did not have prior knowledge of."

8.2. Addressing Interoperability in Pervasive Environments

There are a number of different ways to approach this problem but for pervasive environment a general model has been extracted. Known as “recombinant computing” it was introduced for the Speakeasy project at the Palo Alto Research Center. The result of this research was then incorporated into the Obje interoperability platform marketed by PARC. In [ENSI02] Edwards and al. exposes the conceptual basis of the model. They propose three premises to address interoperability in ubiquitous environments, we shall summarize them here:

1. Agree on a fixed set of universally-known interfaces without requiring each party to have prior domain-specific knowledge of the type of entity it may encounter.

2. Use Mobile Code to allow parties to agree on a small and fixed set of generic interfaces allowing to acquire new behaviors. This approach is more flexible than requiring prior knowledge of specific communication protocols, data types, etc.

3. Users will be the ultimate arbiters. They decide whether an interaction amongst compatible entities does make sense. The arbitration occurs at run time rather than at development time.
As for SOAs, these rules define a general approach towards interoperability, not a concrete system. Since they are quite relevant to our application domain, these premises influenced our implementation of the interoperability layer exposed in Section 9.3.

8.3. Classifying Discovery and Interoperability Systems

Before proposing our own service discovery architecture, we shall conclude with a brief “state-of-the-art” of the existing discovery systems and architectures for spontaneity in terms of interactions with pervasive services. We will then continue by introducing Relate and the RelateGateways project as a novel discovery system.

Two excellent articles about existing discovery technologies can be used as a starting point. “Discovery System in Ubiquitous Computing” from W.Keith Edwards in Georgia Tech [Edw06] provides an interesting summary of the systems. It also proposes a classification focusing on five parameters:

1. Topology: network structure of the discovery system (e.g. Peer-to-Peer, Directory, Beacon-and-reader, etc).
2. Transport: method used to transport discovery messages (e.g. Unicast, multicast, Infrared, Short-range RF, etc.).
3. Scope: action space for the discovery mechanism (e.g. Subnet, Proximity, etc.).
4. Search: degree of granularity when searching for services (e.g. type, ID, XML-Based description, etc.).
5. Security: provided security mechanisms (e.g. Java Security, Capability-based access control, Physical Proximity, etc.).

While this classification is really valuable to study the systems from an technical point of view, it is not fully adapted to our use. “A Taxonomy of Service Discovery Systems” from Vasughi Sundramoorthy in Twente [Sun06] focuses on a functional and (formal) classification towards a better understanding of the main differences between the systems. A similar approach is investigated in [FML05].

Yet, a functional classification, taking the role of users into consideration, is required for the RelateGateway project. This will permit to distinguish End-to-End Service Discovery systems from other discovery systems. This simple classification is the matter of the next Subsection.

8.3.1. User-Oriented Taxonomy of Discovery Systems

The proposed classification is in fact a simple functional taxonomy of the service discovery and invocation mechanisms enhanced with a user activity parameter. Into more details, it roots back to the definition and the model of introduced in this work, where three layers are distinguished:
1. Spatial Discovery.

2. Service and Network Discovery.

3. Invocation and Interoperability.

In order to take these three layers into consideration three parameters are derived: (i) Discovery (ii) Invocation (and interoperability) (iii) User activity. The first parameter is also the most obvious. It determines the type of discovery that is addressed by the system. This is used to distinguish systems addressing network discovery only from systems working both on the service and the network level. The question here is basically whether a particular implementation helps computers to find each other or clients to find services. Note that it is very likely for systems working on the service level to address the network discovery as well, the contrary is not necessarily true.

Furthermore, invocation and interoperability mechanisms are considered. As mentioned before, some discovery systems provide a handle on a service (e.g. URL, IP, ID, etc.) and leave the client alone for actually consuming the service. Some, on the other hand, address the invocation part as well by providing invocation and/or interoperability mechanisms.

Finally, we take into account the user activity. Taking this parameter into consideration makes the proposed taxonomy a novel classification. As mentioned in the previous chapter, we envision a model of discovery where the user has an active role. She is the first strongest link in the end-to-end discovery process. She initiates the network and service discovery by getting physically close to the devices providing services. Introducing this parameter will also permit to focus the taxonomy on discovery and interoperability systems taking the user into the loop.

It is worth noting that we quantify the user activity in terms of ability to control the discovery process, not in terms of the work it involves for the end user. Indeed, since we want to achieve spontaneous interactions the user should have as little to do as possible. Thus, this parameter measures how much control the user has on the discovery process as well as how much the system is oriented towards users' comprehension and needs, not how much administration and configuration work is left to the user.

A graphical representation of this taxonomy is provided on Figure 8.1 as a BCG (Boston Consulting Group) Matrix. The X-Axis represents the type of service discovery (first parameter). Ranging from hardware and network discovery to service discovery only. Thus, a discovery system in the middle of this axis will offer both network and service discovery.

The Y-Axis quantifies the degree of involvement of the system in terms of service execution. Systems with low Y values do not address invocation, neither interoperability. Systems with high Y values address both invocation and interoperability.

The size of the circle depicts the user activity in the discovery process. The most active the user is (again, in terms of control not in terms of configuration and administration), the bigger the circle. Squares are used to emphasize the fact that the user is completely absent of the mechanism. The systems mentioned in this matrix are referenced in the related works Chapter of this document (Chapter 3).

It is worth noting that a BCG matrix is a visualization tool rather than a formal, scientific method of classifying systems. It does not pretend to provide a formal overview of
8.3. Classifying Discovery and Interoperability Systems

systems’ scope but offers a way to position we wanted from our system with regards to other works.
9

The RelateGateways Architecture

9.1. RelateGateways’ SOA ................................................. 67
  9.1.1. Physical Servers, Devices and Services ...................... 68
  9.1.2. Descriptive Service Architecture ............................ 68
  9.1.3. Service Contract .................................................. 70
  9.1.4. Implemented Services ............................................ 77
  9.1.5. Service Management Tool ...................................... 78
9.2. Network and Service Discovery ................................. 78
  9.2.1. Introducing Relate as a Network Discovery System ........ 79
  9.2.2. Service Registration ............................................. 80
9.3. Invocation and Interoperability ................................. 82
  9.3.1. Common Properties Through OOP ............................ 82
  9.3.2. Extracting a Mobile Code Using RMI Serialization ....... 83
  9.3.3. About Dynamic Class Loading and Downloading .......... 86
9.4. RelateGateways’ Interactions in a Nutshell .................. 88
9.5. Service Discovery Simulation Tool ............................... 89

As mentioned before the End-to-End Service Discovery Model is a rough conceptual model. Rather than providing definitive rules on how to implement spontaneity, it is meant to be used as a guideline for creating the bricks of a prototype implementation.

Our implementation of the model is called the RelateGateways Architecture. It enables spontaneous mobile spatial interaction with pervasive services and address every layer of the model. In terms of software implementation this architecture is basically a patchwork of different toolkits aggregated as a unique software framework.

The first brick of this architecture is the gateways user interface described in Chapter 4 and fulfilling the requirements of the Spatial Discovery layer. While this part was centered on Human Computer Interaction (HCI), especially focusing on spatial mobile interactions, the present part focuses on technical aspects enabling spontaneity. In the next sections we expose our concrete implementation of both the Network and Service Discovery and the Invocation and Interoperability layers.
Nevertheless, since our overall system is a concrete instance of a SOA (see Chapter 6) we shall start by discussing the method we use to model the pervasive services as well as the premises of our own Service Oriented Architecture. This chapter is meant to provide a conceptual view of the RelateGateways architecture. Thus, rather than focusing on implementations details we discuss the way the system works and its main components. A more "down-to-earth" view of the RelateGateways architecture is provided in Chapter 11. Yet, it is worth noting that this chapter still requires some knowledge of Object Oriented Programming (OOP) since it refers to important concepts of this programming paradigm. The reference language is the Java programming language but the exposed concepts are similar in any other object oriented language. Similarly, it is important to note that the RelateGateways service framework comprises about 150 classes for a total of about 15'000 lines of code. Thus, this section does not pretend to present it extensively but rather provides an overview of the system in a concern for re-usability.

We describe the architecture from both a SOA and a Network and Service Discovery point of view. Since the former concept partially encloses the latter some redundancy occurs. To solve it we decide to treat the subject where it appears to be the most suitable. As an example the Service Registration process is not exposed in the SOA section but rather in the Network and Service Discovery section.

9.1. RelateGateways’ SOA

As mentioned in Chapter 6 there are plenty of techniques that can be used to concretely implement a SOA. This section presents our own technique based on SOA principles. The choice to implement our own service architecture was conscious. While the use of standards systems such as Webservice or EJBs\(^1\) would have been a better choice in terms of global scalability and interoperability with other systems, we chose not to go for such an approach.

There are many reasons for this choice. The first being a requirement: our system was meant to deal with real-life pervasive devices (e.g. printers, keyboards, displays, vending machines) rather than abstract computational services (e.g. computing available flights). While Webservice and EJBs are well suited for highly computational tasks (such as business oriented tasks) they are not yet well adapted for interacting with concrete, "everyday life" devices.\(^2\) Furthermore we needed was an architecture enabling the rapid prototyping of pervasive services for our user-studies and explorations of the interaction space. Again, business implementations of SOAs are great in terms of productive environments since they address many transversal concerns such as transaction control, security management, load-balancing, etc. But this approach is quite heavy weighted and thus, not really adapted to prototyping tasks. Finally, creating our own service architecture enabled us to make it fit as well as possible with our technology (e.g. the Relate hardware, see Chapter 5) and existing software (e.g. the Spatial Context Toolkit, see Section 4.3). Let us start this exploration of our service architecture with a couple of definitions.

---

\(^1\)Or both Webservices and EJBs since the latter components can be turned into standard Webservices as well.

\(^2\)Current projects such as the European SOCRADES consortium are focused on adapting Webservices to common devices and machines so that these devices can offer standard interfaces.
9.1.1. Physical Servers, Devices and Services

As dictated by SOAs best practices a service in our architecture is self-contained functionality. It exposes to mobile clients an atomic functionality provided by a pervasive device. Thus, a service (e.g. uploading a file) is provided by a device (e.g. file server). However, the hierarchy is sometimes slightly more complicated. Indeed, in many cases a computer (in the wide sense) is controlling various devices offering many services. Consider a workstation for instance. Such a machine could manage various devices each offering a couple of services such as a mouse, a keyboard and a display for instance. In this case the workstation is considered as what we call a physical server, managing two devices each offering n services. This hierarchy is summarized on Figure 9.1. Physical

![Figure 9.1: A Venn-diagram of the relationships between physical servers, devices and services.](image)

Servers are the master entities, they manage devices which in turn offer services.

9.1.2. Descriptive Service Architecture

As seen in Chapter 6. One of the most important guideline in a SOA is to have a uniform way of describing services (see 6.1.1). As seen before, traditional service description use standardized XML-based languages such as WSDL. As global interoperability is not our main concern we choose a more straightforward manner to define a service. We define services in a programmatic way. A service has to respect a certain number of programmatic constraints to useable by the RelateGateways service framework.

These constraints form what we call a Descriptive Service Architecture. That is: the architecture itself is a description of the service. More concretely, we use interfaces and abstract classes to describe services in an abstract manner. Concrete services then have
9.1. RelateGateways’ SOA

Figure 9.2: UML view of the Descriptive Service Architecture.

Every service implements two main components: a ServiceRequester and a ServiceProvider. The latter part encloses the logic, or semantics of the service and is situated on the machine providing it. On the other hand, the former part contains everything a program needs to know in order to invoke a service and is situated on the machine requesting the service. This construct allows a clean separation of the concerns. Consider the case of a content-publishing service on a public display. All a client needs to know in order to publish content is how to send it to the display. There is no need for them to know how to format or show the content, this is the display’s concern.

Furthermore, a service needs to be either of type Push or Pull-and-Push. Push services represent simple asynchronous stateless components. The client of such a service only needs to push an object (e.g. file, image, document, etc.) to the provider. Pull-and-push services allow to represent more complex functionalities such as statefull services requiring particular processing on the client side as well. These services need to pull information from the user before the request can be sent.

Consider the case of a printer. This device could offer a service for printing documents using the default settings. Such an operation is the ideal candidate to be modeled as a Push service, since no other input than the object on which to apply the service is
required. On the contrary, if the printer offers a service enabling the user to specify his own settings (e.g. paper orientation, size, etc.) it will need to be implemented as a Pull-and-Push service, prompting the user for more information through various dialogs before actually executing the service logic.

This architecture was directly inspired from the design of the gateway interaction technique implementing the Spatial Discovery Layer (see Chapter 4). Indeed, drag-and-drop actions can easily be mapped on Push services, whereas a *click* action on a gateway correspond to the invocation of a Pull-and-Push service.

### About a Universal Requester

![Figure 9.3: A universal requester consuming a music-player service](image)

As mentioned before, a *PushService* represents stateless asynchronous service whereas a *PullAndPushService* is used to model “statefull” services requiring more user input than the object to apply the service on, also known as the service subject (e.g. the file to print, the presentation to start, etc.). Besides the fine mapping on the gateways user interface, this distinction has another interesting property: it permits to create a “universal requester” for *PushServices*. Indeed, since such a service simply pushes the service subject, it does not need to know anything about the service logic and semantics. Thus, while each concrete *PullAndPushService* needs to offer a proper *ServiceRequester* to the mobile device, all the *PushServices* could be addressed using a single generic requester, a universal *PushServiceRequester*. This could be especially useful to simulate use-cases where security policies are such that the mobile system is not allowed or does not accept to download and dynamically load code provided by newly discovered pervasive devices. More precisely, the use of universal requesters basically represents an alternative to the interoperability system proposed in Section 9.3.

An typical use of a universal requester is shown on Figure 9.3. Here the generic universal requester is used to consume a music-player service. Since the only thing we need to provide to this service is a service subject (i.e. an object to apply the service on) we can address it just by pushing the drag-and-dropped file, without a need for the mobile client to have any further knowledge of the service it sends the file to.

### 9.1.3. Service Contract

Next to the service description a typical SOA should define a service contract. A contract is nothing than a description of how parties will interact when trying to consume a service. In our architecture the contracts are not expressed in a globally standardized way but rather in a way fitting our special needs (i.e. the prototyping of pervasive
9.1. *RelateGateways’* SOA

This interface exposes the properties shared amongst the ServiceProviders and ServiceRequesters.

```java
public interface ProvidersAndRequestersCommonInterface {
    /* Method used to start a ServiceRequester or a ServiceProvider */
    public void startup();
    /* Method used to stop a ServiceRequester or a ServiceProvider */
    public void shutdown();
}
```

Listing 9.1: The common interface of all the providers and requesters.

For both the providers and requesters of services we define a number of interfaces and a class hierarchy to be respected. These definitions form the basis of the interface contract that is detailed below.

**Interface Contract**

![UML diagram of the ServiceProvider abstract class.](image)

For both the providers and requesters of services we define a number of interfaces and a class hierarchy to be respected. These definitions form the basis of the interface contract that is detailed below.

**Providers and Requesters**. We shall start by exploring the methods both ServiceProviders and ServiceRequesters have to implement. These are defined in an interface common to both providers and requesters called: ProvidersAndRequestersCommonInterface which prototype is shown in Listing 9.1. These two methods define the entry and exit point of a service life cycle. They start and stop the providers, respectively requesters.
The actual implementation of these methods is part of the contract of a particular service and thus cannot be generically defined. As shown on Listing 9.2 (line 6-8) these methods are called by the framework whenever a provider or requester needs to be used.

**Requesters**. Recall that a requester represents the client part of a service. It is the "stub" used by a mobile client to access a remote service. As shown on Figure 9.5, a concrete ServiceRequester has to implement the following method:

```java
/** Method request a service from a service provider */
public abstract void request(Object o);
```

This way the user interface (e.g., the gateway UI) executes all the service requests in a uniform manner using the `request(Object o)` method every concrete ServiceRequester offers. Keeping this method very generic enables to move concerns related to a particular service away from the user interface. Consider the case of the gateway UI: whenever the user drops an object on a gateway the UI only needs to send this object, without any concerns of its very nature. It is the concrete ServiceRequester's business to know what kind of object it accepts and what type it will reject. As a consequence the service request loop of any user interface plugged on top of the service framework is quite straightforward to write. An sample loop in Java is provided on listing Listing 9.2 and further discussed in Chapter 4.

Even if the concrete implementation of the `request(Object o)` method is the service programmer's business, the framework expects such a method to do at least one operation, which is part of the interfacing contract: the body of a requester's `request(Object o)` should at least issue a call to its counterpart on the provider side. From these rules we can derive the pseudo-code of a `request(Object o)` method as shown on Listing 9.3.

**Providers**. On the provider side the concept is similar, only the semantics is inverted. As shown on Figure 9.4 each concrete ServiceProvider has to implement a `provide()` method:

```java
public abstract ServiceResponse provide(ServiceRequest serviceRequest);
```
for (Service currentService : getDevice().getServices()) {
    if (currentService instanceof PushService) {
        foundService = true;
        log.fine("Found a Push service, let’s invoke it: "+
            currentService.getName());
        currentService.getRequester().startup();
        currentService.getRequester().request(fileName);
        currentService.getRequester().shutdown();
        break;
    }
}

Listing 9.2: Sample of service request loop for a user interface programmed on top of the service framework.

Listing 9.3: Pseudo-code of a request(Object o) method.

This method represents the other end-point of the interaction. It is going to be triggered by the framework whenever the counter-part of a concrete provider (i.e. a concrete requester) executed its request(Object o) method.

provide(ServiceRequest serviceRequest) is the core method of a service. It contains the service logic. Thus, if a service offers to play audio files this method will actually consume the audio file and any play it.

<table>
<thead>
<tr>
<th>ServiceRequest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
</tr>
<tr>
<td>private Object serviceSubject</td>
</tr>
<tr>
<td>private boolean isSubjectAFile</td>
</tr>
<tr>
<td>private String canonicalClassNameOfSender</td>
</tr>
<tr>
<td>Operations</td>
</tr>
<tr>
<td>public ServiceRequest(String canonicalClassNameOfRequester)</td>
</tr>
<tr>
<td>public ServiceRequest(Object serviceSubject, String canonicalClassNameOfRequester)</td>
</tr>
<tr>
<td>public ServiceRequest(file serviceSubjectFile, String canonicalClassNameOfRequester)</td>
</tr>
<tr>
<td>public Object getSubject()</td>
</tr>
<tr>
<td>public String getCanonicalClassNameOfSender()</td>
</tr>
<tr>
<td>public file createTempFileForSubject()</td>
</tr>
<tr>
<td>protected String getFileName()</td>
</tr>
</tbody>
</table>

Figure 9.6: UML representation of a ServiceRequest.

This method has two requirements, part of the contract. At first it accepts a ServiceRequest as argument. Such an component is based on a well-known enterprise design pattern called: TO (Transfer Object) [ACM03]3, as shown on Figure 9.6. It is basically an object

3Note that a ServiceRequest is a degenerated version of Transfer Object since it implements some service methods. See [ACM03] for the pure definition of a TO.
containing some parts of another object that needs to be transferred over the network. Thus, a ServiceRequest contains the data a provider needs to send to a requester. In particular it contains the ‘service subject’, defined as the object one wants to apply the service on. In the case of a public display service it would be the object (e.g. presentation, picture, etc.) we want to display.

The `provide(…)` method returns a ServiceResponse object. This extension of the ServiceRequest class is also its counter-part in the interaction protocol. As a consequence when a ServiceProvider receives a request it:

1. Unpacks the ServiceRequest to extract the service subject.
2. Applies its core logic on the service subject.
3. Packs and returns a ServiceResponse.

Note that the returned ServiceResponse might be an empty object, simply acting as an acknowledgment of the service delivery. In fact, most services we implemented do not need to return any kind of object to the mobile client. However, more complicated services need to return something. Consider for instance the case of a TV service. It will certainly want to return a ‘remote control’ to the caller in order for him to be able to browse the available channels.

Providing a service subject is not the only goal of ServiceRequest and ServiceResponse objects. These can also be used for authentication purposes. Both objects can be extended to contain security related data fields in order to authenticate and log the service usage. As an example one such field is already implemented in the proposed version of the service framework. Indeed, the method:

```java
public String getCanonicalClassNameOfSender()
```

returns the full classname of the requester, respectively provider. This way, a provider can check whether the service request originated from a valid provider. However, even if the RelateGateways service framework supports authentication methods it does not implement any: for the timebeing it is the programmer’s responsibility to implement an authentication mechanism using the ServiceResponse and ServiceResponse enhanced transfer objects. An extension of the framework towards this direction was suggested by a number of users and specialists, which makes it a good candidate for future improvements.

Connectors . In order to have a more precise idea of what a concrete requester needs to implement in its `provide(…)` method, we shall take the DirectPrintingServiceProvider of Listing 9.4 as an example. In this example we start by unpacking the ServiceRequest, transforming it into a PrintRequest. That is: we turn the request from a generic object into an object specific to the current service. We then log the request and instantiate what is called a Connector. Recall that the provider is responsible for executing the service’s logic. In order to reach this goal and make the services’ logic as reusable as possible, a provider might instantiate and use a Connector.

Such a software component is an abstraction of a service’s logic. Note the use of connectors is not enforced in the current framework’s state. Providers are free to implement the service logic as they wish. However, connectors help to extract common properties
9.1. RelateGateways’ SOA

9.1.1. Direct Printing Service

The direct printing service is provided by RelateGateways as a service to requesters wanting to use it. The service is implemented as a method in a ServiceRequest object, as shown in Listing 9.4:

```java
public ServiceResponse provide(ServiceRequest request) {
    try {
        /* 1) Unpacking the ServiceRequest into a PrintRequest. */
        PrintRequestTO printRequest = (PrintRequestTO) request.getSubject();

        /* 2) Writing request to log. */
        log.fine("Flavor: " + printRequest.flavor.toString());
        this.getPrinterConnector().printAttributeSet(printRequest.printingAttrs);

        /* 3) Sending the request to the printer connector for default printing */
        getPrinterConnector().defaultPrint(printRequest);
    } catch (ServiceRequestException ex) {
        ex.printStackTrace();
        log.severe("Empty request received from requester.");
    }

    /* 4) Nothing to tell the requester, create ServiceResponse to acknowledge but leave its service subject empty. */
    return new ServiceResponse(this.getClass().getCanonicalName());
}
```

Listing 9.4: Pseudo-code of a request(Object o) method.

amongst implementation of the services’ logic. According to our experience while implementing service, the part of the logic that you can actually re-use is not insignificant. As a consequence the use of Connectors might empower re-usability when implementing the core logic of a service provider. The framework already provide a number of ready-to-use connectors to interface various software components such as OpenOffice [22] or the Java Robot API as well as hardware devices such as printers. The use of these connectors is described in further details in Chapter 11.

In the case of Listing 9.4 we call the PrinterConnector which will take care of communicating print jobs to the printer. Finally, we create a ServiceResponse to acknowledge the fact that the service logic was executed.

Communication Contract

![Schematic view of service communication](image)

Figure 9.7.: Schematic view of service communication

The communication contract of the RelateGateways services is quite straightforward and simple: the concrete providers and requesters form the networked communication end-
points of the architecture. A ServiceRequester will always communicate with its corresponding ServiceProvider as shown on Figure 9.7.

While the actual communication mechanism is not globally defined, it has to be defined at the service level in order for providers and requesters to be able to understand each other. That is: a service must include a communication mechanism in its contract but services can use different communication systems. As a consequence the Service Architecture does not enforce the use of any particular protocol or system. However, the developer is not let alone since a number of abstract services implement communication protocols. This way the programmer can focus on the core logic of the service rather than on transversal concerns. While the feature of the framework is described into further details, from a programmer’s point of view in Chapter 11, we shall have a conceptual overview of the subclassing mechanism.

Consider one wants to implement a printing service as shown on Figure 9.8. The programmer begins with designing a PrintingService class subclassing the Service class. As specified in the Descriptive Service Architecture she now needs to implement a PrintingServiceProvider and its corresponding PrintingServiceRequester. Once this is done she has to decide on a communication protocol to adopt for the service. As the provider and requester were both implemented in the Java programming language she chooses to implement them as RMI (Remote Method Invocation) [29] services. Thus, she need to make:

1. The PrintingService class subclass RMIService
2. The PrintingServiceRequester class subclass RMIServiceRequester
3. The PrintingServiceProvider class subclass RMIServiceProvider

This will hide the RMI concerns from the service’s core logic. As shown on Figure 9.9, the system will:

1. Instantiate generic stub (i.e. remote interface) for the PrintingServiceProvider.
2. Bind it to a remotely accessible RMI Registry.

\footnote{Note that depending on the exact nature of the service one may also want to use a UniversalRequester instead.}
3. Relay PrintingServiceRequester calls to provide(ServiceRequestTO) on the provider side.

9.1.4. Implemented Services

To assert and evaluate the service framework, various services were implemented. The table below summarizes these services.

<table>
<thead>
<tr>
<th>Device</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printer</td>
<td>Direct</td>
<td>Push</td>
<td>Prints document with the printers’ default options.</td>
<td>Java Print</td>
</tr>
<tr>
<td></td>
<td>Printing</td>
<td></td>
<td></td>
<td>Service API</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>Pull- and-</td>
<td>Prompts the user for file selection and open a printing option dialog box. Prints document with the selected options.</td>
<td>Java Print</td>
</tr>
<tr>
<td></td>
<td>Printing</td>
<td>Push</td>
<td></td>
<td>Service API</td>
</tr>
<tr>
<td>Display</td>
<td>Presentation</td>
<td>Push</td>
<td>Starts a presentation.</td>
<td>UNO API</td>
</tr>
<tr>
<td></td>
<td>Show</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Presentation</td>
<td>Pull- and-</td>
<td>Pops up a file selection dialog and loads several types of presentations (.ppt, .odp, etc.) in OpenOffice Impress.</td>
<td>UNO API</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>Push</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.9.: Mechanism underlying the abstract RMI Service.
### 9.2. Network and Service Discovery

According to the E2ESD model the next part to explore is the Network and Service Discovery layer. We already have a service framework helping us to prototype pervasive services. In order to have a real Service Oriented Architecture we now need to make these services available to the mobile user on a dynamic network.

Recall the basic problem that needs to be solved by this layer: once the mobile user identified a service she would like to use with the Spatial Discovery layer, the system needs to connect the mobile device to the machine providing the selected service. Basically we want to automate the connection process so that it appears transparent for the end-user.

We propose a solution based on an extend of the Relate Ad-Hoc Sensor Network. This sensor network introduced in Chapter 5 was already used as a mean to feed the gateways

#### 9.1.5. Service Management Tool

In order to start and stop services offered by a physical server a graphical user interface is provided. The ServiceManagementTool is accessible through the AdminToolbar described in the first part of this work. As shown on Figure 9.10, this tool automatically retrieves all the services for which the current host has a provider. It enables the service administrator to start and stop the providers for each service. Note that starting a provider with this tool will result in actually providing the service to any interested client (unless authentication or restriction methods have been implemented).

#### 9.2. Network and Service Discovery

<table>
<thead>
<tr>
<th>Device</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Conn.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>Typing</td>
<td>Pull-and-Push</td>
<td>Can be used to “borrow” a keyboard from another computer. It basically multicasts the typed keys on the network, to the mobile device(s).</td>
<td>Java Robot API</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OOWriter</td>
<td>Typing</td>
<td>Pull-and-Push</td>
<td>Can be used to “borrow” a keyboard from another computer in order to create and edit OpenOffice Writer documents.</td>
<td>UNO API and Java AWT Event API</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computer</td>
<td>EBL Migration</td>
<td>Pull-and-Push</td>
<td>Offers to migrate a part of a user interface to another computer.</td>
<td>EBL Framework</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mouse</td>
<td>Pointing Service</td>
<td>Pull-and-Push</td>
<td>Can be used to “borrow” the mouse of another computer.</td>
<td>Java Robot API</td>
</tr>
</tbody>
</table>

**Note:**

- **Pull-and-Push** refers to the method of connecting and disconnecting devices over a network.
- **Java Robot API** and **UNO API** are used for interfacing with the services.
- **EBL Framework** is used for migration services.
- **OOWriter** and **UNO API** are used for typing services.
- **Computer** and **Mouse** are examples of devices that can be connected through service discovery.
user interface with real-time spatial positioning information, we now extend it to use it as a straightforward, user directed, network discovery system.

9.2.1. Introducing Relate as a Network Discovery System

The concrete providers and requester form the networked communication end-points between the actors. Thus, an important assumption for our architecture is the fact that all the devices providing services as well as the mobile device have able to access one another over a TCP/IP network. Since nowadays a lot of mobile devices tend to have wireless connectivity we believe that this assumption is not excessively restrictive. Bearing this assumption in mind all we need for the mobile device to be able to dynamically discover the machine running as service, is to be able to send the provider’s IP address to the mobile client.

In Chapter 5 we explained how our extend of Relate created an ad-hoc network providing spatial information suitable for the gateways user interface. In this model each service was given a Relate Dot. This Dot is virtually mapped to the service since it represents it within the spatial context. Thus, if we include the IP address of the sender in the packet sent by the Dots to the Relate Brick attached to the mobile device, we enable network discovery.

An illustration of this concept is provided on Figure 9.11. In this situation the user want to discover the public display using his mobile device:

- Because a Relate Dot standing for the display was deposed next to this later and the mobile device embed a Relate Brick an RF communication is initiated.
9.2. Network and Service Discovery

A packet is sent from the Dot to the Brick using the communication channel of the newly created ad-hoc network.

- The packet contains positioning information as well as the IP address of the sender (or of the physical server managing the device that runs the service to be more accurate).

- Using this network address the mobile device is able to connect to the display service provider using the TCP/IP wireless network.

9.2.2. Service Registration

The model exposed above is a straightforward way of interconnecting machines using Relate, yet this does not make the system a so-called network and service discovery system. Recall Subsection 7.1.4 proposing four characteristics of discovery systems. In essence what we still miss is a way to expose services to clients rather than simple machines. To achieve this we create a directory or registry system on which services advertise themselves to the mobile clients. Since it is based on an ad-hoc network, the RelateGateways architecture does not offer a central directory but rather a registries where \( n \) is the number of physical servers in the environment.
9.2. Network and Service Discovery

The registration process and components are shown on Figure 9.12. Recall that a Device is the master entity. It encloses and manages a number of Services that it needs to advertise on the local physical server’s registry. Now, since the RelateGateways architecture offers a service framework it is the responsibility of this latter to publish a description of the service on the registry. Thus, this process is transparent for the programmer. Nevertheless, for the sake of clarity, let us expose the underlying system.

As shown on Listing 9.5, when a service is started using the ProvidersStarter GUI the system uses the callback method startup() to initialize and start the service on the current physical server. The framework then request the ServiceRegistrar class to extracts a description of the running service and publishes it to the local service registry. This directory is basically a RMI registry running on the mobile device to which service description objects are bound.

When the client gets the IP of the physical server managing 1..n devices running 0..n services it connects to this IP and retrieves the services descriptions. Note that since the registry is updated everytime a service is started/stopped the mobile device will only get the description of services started at the time of discovery.

After a new service was started the registry contains a RGWMobileCode object. This latter encloses all the descriptions of all the services currently running on all the devices held by the physical server. To be more accurate this package of MobileCode contains everything a client needs to know in order to invoke a previously unknown services. The details of this interoperability system is the matter of the next section.

Listing 9.5: startProvider(...) method of the ProviderStarter GUI.
9.3. Invocation and Interoperability

At this stage of our system’s implementation a mobile user can already spatially discover a service using the gateways user interface, an implementation of the Spatial Discovery layer. His mobile device can then be automatically connected to the physical server providing the identified service using the Relate discovery mechanism. Furthermore the mobile device is able to retrieve a description of the discovered service. These two parts form a concrete implementation of the Network and Service Discovery layer.

The last point to address in order to achieve an end-to-end spontaneous interaction is invocation and interoperability: once a service is identified and discovered how does the user actually manages to consume it.

As seen before there are several solutions to address this problem. The solution proposed in the RelateGateways architecture is based on three main bricks as shown on Figure 9.13.

9.3.1. Common Properties Through OOP

The first brick is OOP: using the concept of object oriented programming, especially the inheritance mechanisms, we can develop a model of what a service will look like in abstract terms. The concrete implementation of this part was described in the Service Contract of our SOA Figure 9.1.3. Using interfaces and abstract classes the service consumption is modeled in a generic manner.

As a consequence, the system knows it needs a concrete ServiceProvider to consume a service. Some services (the PushServices) can be addressed in a generic way by using a UniversalServiceProvider and simply pushing any object selected by the user to the service, as detailed in 9.1.2. However, this construct presents two important drawbacks. First of all, while it is fine for the mobile device not to know anything about the service it interacts...
with, it is certainly not suitable for the final user. Users need a strong confidence in the service they interact with. Simply displaying a generic gateway in the user interface is not satisfactory. For a better experience gateways need to describe (textual description, visual representation, etc.) the service they stand for. Thus, the mobile device needs to download this information from the physical server’s service registry.

This solution is fine for simple stateless and asynchronous services which do not require particular processing on the client side (i.e. PushServices, see Subsection 9.1.2). Nevertheless, downloading a service textual or visual description is not enough for more complex services requiring processing on the client side as well (i.e. PullAndPushServices, see Subsection 9.1.2). For such services we need to “install” additional code on the client side in order for the mobile device to get a knowledge of how to handle and interact with such services.

According to the results of our user study in Chapter 12, the absence of installation tasks is a very strong incentive towards user acceptance of any ubiquitous system. Thus, we propose to achieve these “installation” tasks automatically, using the two upper bricks of Figure 9.13 described below.

### 9.3.2. Extracting a Mobile Code Using RMI Serialization

The second brick we use towards interoperability and plug-and-play invocation is the object serialization mechanism. In this part of the Invocation and Interoperability layer we want to export part of a service from the provider to the client. More precisely, for each service we need to export:

1. A description of the service containing a textual description and an iconic representation.

2. The concrete ServiceRequester of the current service that will be used by the mobile device to consume the service.

3. The spatial condition assigned to the device offering the service, as exposed in Section 4.3. This way the service will be made available on the mobile device if and only if the context fulfills the condition.

Furthermore, since the RelateGateways service framework is meant to be used for the rapid prototyping of pervasive services we would like to automate this process and take this concern away from the programmer of pervasive services.

To achieve this goal packages of mobile code are made available to the mobile device through the service registration process introduced in Subsection 9.2.2. These packages are modeled using the DeviceMobileCode class as shown on Figure 9.14. An instance of this class encloses the descriptions, ServiceRequesters and conditions required to interact with all the services a particular device offers. To make it transparent for the programmer the framework automatically creates and updates the packages of mobile code made available in the service registry. This task is launched by the DevicesMobileCodeFactory each time the state of a service changes (i.e. started/stopped).

Listing 9.6 presents the core method of this class. Each time a service is started or stopped the framework calls this method. It starts by iterating through the devices looking for
This method is the core of the factory. It looks for available
* (started) services and packs them into a DevicesMobileCode
* ready to be sent to the client. */

```java
public MobileCodeList packDevicesMobileCode() {
    MobileCodeList mobileCodes = new MobileCodeList();

    /* iterate on all the devices */
    for (TrackableDevice currentDevice : deviceMap) {
        ArrayList<Service> runningServices = new ArrayList();
        for (Service currentService : currentDevice.getServices()) {
            try {
                ServiceProvider provider = currentService.getProvider();
                if (provider.isRunning()) {
                    /* provider runs, add service to the list */
                    Service castedService = (Service) currentService;
                    runningServices.add(castedService);
                }
            } catch ( ServiceException ex ) {
                /* not so bad, in the current
                * architecture a service does not
                * necessarily have a provides, thus do nothing... */
            }
        }
        /* we iterated through the services, now if at least one was
        * available then create a DeviceMobileCode to pack it... */
        if (!runningServices.isEmpty()) {
            /* create an ImageIcon in order to be able to serialize the
            * image */
            ImageIcon iconicRep = currentDevice.getIconicRepresentation();
            /* create device mobile code */
            DeviceMobileCode mobileCodeForCurrentDevice = new
            DeviceMobileCode(currentDevice.getName(),
            iconicRep,
            currentDevice.getHostname(),
            runningServices,
            currentDevice.getController().getCondition(),
            currentDevice.getLocation());

            /* add it to the list to be sent to clients */
            mobileCodes.addDeviceMobileCode(mobileCodeForCurrentDevice);
        }
    }
    return mobileCodes;
}
```

9.3. Invocation and Interoperability

Running services. If a service is running for any given device a new `DeviceMobileCode` is created and added to the list that will be published on the service registry. The novelty and practicality of our approach is that we generate the Mobile Code “on-the-fly”, extracting it from a running system. When the client contacts the registry it automatically retrieves the latest mobile code. The process is schematized on Figure 9.15.

![Figure 9.15: Creating and registering mobile code.](image)

Again, the whole process is transparent for both the end-user and the programmer using the RelateGateways service framework. However, in order to fully understand the concept and perceive its incentives let us expose what happens when the mobile code is sent over the network. A package of mobile contains a series of objects. When the mobile device downloads the mobile code it basically gets a number of serialized objects and deserializes them.

This well-known process of OOP allows what is technically called the flattening of objects [Gre00]. When an object is created using an OOP language it exists in the RAM (Random Access Memory), or volatile memory only. To have a persisting version of this object we need to flatten it, that is to extracts its fields and write them to a file on a permanent storage, or a network socket: this is the serialization process.
9.3. Invocation and Interoperability

When the object is needed again we load the fields from the file and feed them to the object constructor, creating a new object with the same state as the original object: this is the deserialization process. In our system the services are automatically serialized when published on the registry, deserialized when loaded by the mobile device at the other end-point of the network.

9.3.3. About Dynamic Class Loading and Downloading

There is one remaining issue to solve. In Java (as well as in most other OO languages), when an object from a class ADumbService is deserialized the system will look for the class ADumbService in the class pool of the current JVM (Java Virtual Machine). This is fine in a static environment but not in a dynamic pervasive environment. While the structure of the yet to discover services is known in advance thanks to the service contract, concrete instances of service classes are not and must not be known at compile-time in order to ensure spontaneous interactions.

Conceptually to solve this issue the mobile code should enclose not only a serialization of the service related objects but also their classes. We started by using this approach and programed our own class loader component to achieve what is commonly called “dynamic class loading” [Chr05, Tay03]. Listing 9.7 is an example of class loader. It basically loads a class (ADumbService) retrieved from a .jar (Java AChive) file and tries to deserialize an instance from this dynamically loaded class. While this code is conceptually correct it will raise a ClassNotFoundException. The reason behind this exception is that when loading the class using our own class loader we do not make it available on the same level of the JVM as java core classes for deserialization and thus, the class ADumbService is still not recognized.
Luckily enough, the “dynamic code downloading” system of RMI [War06] can be used to solve this problem. This feature offers to download missing classes “on-the-fly”, at runtime, from an HTTP or FTP repository. Since our service registry is based on RMI we easily integrate this feature using a JVM parameter called “remote code base”:

\[ -Djava.rmi.server.codebase=http://192.168.1.101:8080/RelateGateways.jar \]

This parameter, set on the service provider side, is sent to the clients [8]. As mentioned before, when the mobile client discovers a service it deserializes the classes contained in the package of mobile code retrieved from the physical server’s service registry. Now, if a class is missing during this process, before throwing a ClassNotFoundException exception, the system will use an URLClassLoader to retrieve classes from the specified (remote) RMI codebase. Conceptually, this system is similar to our first trial described above. However, since the system takes care of the class loading in this case, the loaded classes will be made available in the entire JVM space and thus will be available for the system deserialization process.

**Class Server Component**

![Figure 9.16: Plug and Play Invocation using mobile code.](image)

This solution is quite elegant and achieves interoperability by ensuring that previously unknown services can be dynamically discovered during run-time, both in terms of existence and invocation semantics. On the other hand it has a major conceptual drawback: it breaks the transparency and, as a consequence, complicates the design process for programmers. One of our requirements was for the programmer to be able to create the semantics of services while the framework takes care of discovery, invocation and interoperability concerns. Introducing the RMI dynamic code downloading system forces the programmer to manually make new services available on an HTTP or FTP server.

To solve this problem we introduce a new component in the framework: the ClassServer. In a nutshell, this class is responsible for providing the latest version of the RelateGateways compiled classes to the mobile clients through a publicly accessible Webservice running on each physical server. It basically extends and manages a simple (but still multithreaded) HTTP server and copies the latest version of the RelateGateways jar file containing the service classes to be retrieved by the mobile clients. In order to make the operation transparent to the final programmer, the framework will start this server if and only if the physical server starts offering services.

Using OO interfaces and inheritance mechanisms, objects’ serialization in RMI, dynamic class loading and an automated class server we offer dynamic invocation and interoperability. This makes the RelateGateways framework especially useful to prototype pervasive
services since it considerably reduces the configuration and installation work. New services can be created and started on one machine without the need to explicitly inform the other about it. Indeed, each running instance of the RelateGateways architecture will take care, through its framework, of keeping the physical server it runs on up to date in terms of service classes.

9.4. RelateGateways’ Interactions in a Nutshell

To conclude this overview of the RelateGateways’ architecture we shall provide a summary of the most important steps in the interaction life-cycle.

A mobile user (running the RelateGateways software on his device) wants to display a presentation on a public screen. As shown on Figure 9.17, the mobile device obtains a network address using its USB brick to access the Relate ad-hoc network. Thanks to this address it can now connect to the service registry of the physical server offering the display service.

From the registry the user can get a package of mobile code containing all the ServiceRequesters and service description classes it needs to consume services offered by the devices the physical server manages. This package is automatically downloaded by the mobile client and the code it contains dynamically loaded into the running virtual machine. This way, the client discovers the services “on the fly” as shown on Figure 9.16.

Additionally, a package of MobileCode contains the spatial condition allocated to the device providing services. The shipped condition is evaluated on the mobile client as this later moves in the environment.

As mentioned before, an update of the spatial context will force the re-evaluation of the spatial conditions. If one of these is found to be “true” the corresponding gateway window
will be shown. The steps coming next in the interaction life-cycle are shown on Figure Figure 9.18. In this case the user chooses to drag-and-drop a presentation on the display gateway. This forces the execution of the freshly downloaded Push PresentationRequester. The requester then sends the presentation to the other end-point, on the provider side (PresentationProvider). This latter component contains the services’ semantics and thus, knows what to do with the retrieved presentation.

It starts by retrieving the ServiceRequest object sent by the requesters and extracting the presentation it contains. It then instantiates an OpenOfficeConnector and use it to start the presentation on the display using OpenOffice Impress.

9.5. Service Discovery Simulation Tool

![Discovery Simulator GUI](image)

Figure 9.19.: The discovery simulation tool.

In the best case, testing both the Service Discovery and Interoperability system requires to have a working physical service discovery system like Relate running underneath the application. While more realistic and suitable for demonstrations it complicates the full testing of newly implemented services. To be able to test the Service Discovery system as well as the Interoperability layer without the need of specialized hardware service discovery system as tool is provided.

As shown on Figure 9.19, the DiscoverySimulator GUI offers a graphical access to this tool. It exposes a simple interface where the programmer can enter the IP address of the service repository to poll for mobile code. This way the programmer replaces a service discovery hardware like Relate and can test newly designed services in a straightforward manner.
10.1. Use Cases

In order to better understand this concept let us consider two simple examples. Figure 10.1 shows the mockup of a simple presentation application. Imagine you need to present your work. You connect your laptop to the projector and start the presentation. Because the presentation has to be full screen on your system you will not see the notes...
zone” anymore. Since your PDA is situated next to your laptop a PDA gateway appear on your screen. You can now drag-and-drop the “notes zones” on the gateway to export this part of the user interface to your PDA. Similarly, you could drag-and-drop the “previous” and “next” buttons to your mobile phone in order to control the slides from this latter device.

The interesting part is that the application stays consistent. Whenever you click on the “next” button it will change the slide on your laptop as well as the notes on your PDA. This way you use the computing power “service” of all the devices surrounding you to spontaneously distribute your task.

Similarly, one could think of people working on a common task, such as discussing and correcting a scientific paper. Imagine each of the actor has his own portable device running the RelateGateways application. As a consequence each device identifies the other devices as providing the migration service and displays the corresponding gateways at the correct position, as shown on Figure 10.2. Now, whenever a scientist wants her colleague to edit the text, she can drag-and-drop the textbox onto the corresponding gateway to send the widget to her peer.

### 10.2. Enhanced Binding Layer (EBL)

EBL (aka DistriXML) [AVR05] is a toolkit developed in frame of the UsiXML project [24] at the University of Louvain-La-Neuve in Belgium (HCI Lab). Developed by Donatien Grolaux (supervised by Jean Vanderdonckt) this system aims to support a novel form of distributed user interfaces. Indeed, the UIs created with this toolkit can be decomposed into pieces in order to migrate parts of it (i.e. widgets) across a number of networked devices.
10.2. Enhanced Binding Layer (EBL)

Figure 10.2.: Group collaboration using the EBL Migration service.

computers. Furthermore, using EBL the migration of UI elements can be achieved without breaking the application’s consistency.

The basic concept of EBL is to provide a toolkit for building distributed user interfaces. It offers a set of widgets that are "migratable". The UI programmer can manipulate these widgets as he would for normal (i.e. not distributable) GUI components.

Figure 10.3.: The EBL widget importer.

EBL provides the necessary architecture to easily and seamlessly migrate widgets. However, since the focus of EBL is on providing a clean layer to base user interfaces on, it legitimately does not address the spatial and spontaneous aspects of the interaction. Using the current demonstrator of the EBL project the user has to go through the following steps to migrate a widget:
1. Start the widget importer (UniversalReceiver) on the destination machine.

2. Enter the IP address of the origin machine.

3. Choose the widget to import from a list Figure 10.3.

This procedure is not really intuitive. Thus, merging the power of EBL with the spontaneity of RGW enables quite interesting use cases. The next section describes the merging concept into further details.

## 10.3. EBL meets RGW

The idea is to integrate EBL as a new "migration" service using the RelateGateways architecture. Using the gateways user interface permits the exportation of widgets to be more natural and to have a spatial dimension. Furthermore, the Network and Service Discovery Layer of RGW enables the destination machine(s) and the origin machine(s) to be connected without any user involvement. In a nutshell, the provider part of this service called EBLMigrationService is responsible for informing the requester part about the widget to import. In turn, the requester part has to import the widget.

The EBL system is implemented on top of the Mozart Programming System, an advanced development platform for intelligent, distributed applications [20]. Unfortunately, there is currently no bridges between Java and Mozart. As a consequence the creation of the "migration" service was somewhat more challenging than creating a plain Java service. Luckily enough, this part of the project is a collaboration between both the Ubicomp group at Lancaster University and the HCI Lab of the University of Louvain-La-Neuve in Belgium. Thus, we could always count on the Belgian researchers’ support when facing integration problems. This helped to come up with the solution proposed below, described both in conceptual and concrete implementation terms.

### 10.3.1. Migration Service Requester

The requester’s concept is summarized on Figure 10.4. Basically whenever the user drags and then drops a widget form an EBL user interface a ticket (aka capability) is created. A capability is a value containing just enough information for the destination machine to import a widget. The creation of capability is core to EBL and thus was not implemented for this use-case. However, in our case the capability is written to a file instead of directly sent to the destination machine over the network. Furthermore, the capability needs to be generated whenever a drop action occurs, which was not the case in the initial EBL model and requires to implement a drag-and-drop method for EBL.

When the user drops the dragged widgets on a gateway offering an access to the migration service, the RGW framework starts the EBLMigrationServiceRequester. This component reads the capability from a path common to both EBL and RGW. From this path it gathers the capability and sends it further to the requester wrapped in a ServiceRequest just as any other service subject.

The concrete implementation of the requester’s request(...) method is shown on Listing 10.1.
10.4. Implementing the Shared Notepad Application

In order to test the EBL/SGW merging we programmed a small EBL application supporting the collaborative scenario exposed in Section 10.1.

As shown on Figure 10.6 the application is a basic notepad but since it is implemented using EBL, every widget it is composed of can potentially be migrated to another com-
10.4. Implementing the Shared Notepad Application

This method requests a widget of a user interface to be migrated to the device on which the EBLMigrationServiceProvider runs.

@param o The gateway view (or any other subclass of JFrame) attached to the calling controller.

```java
public void request(Object o) {
    try {
        GatewayView gatewayView = (GatewayView) o;
        File capFile = new File(getPathToCapability());

        // confirm the fact that we are sending the request
        super.displayMessage(
            "The user interface part is being migrated."
            + 
            "\nPlease allow a few seconds for " +
            "it to be opened on the workstation.", "Confirmation");

        // create the request
        ServiceRequest request = new ServiceRequest(capFile,
            this.getClass().getCanonicalName());

        super.request(request);
    } catch (ServiceRequestException ex) {
        ex.printStackTrace();
    }
}
```

Listing 10.1: Request method of the EBLMigrationServiceRequester.

We begin by declaring the widgets that our user interface will use, using the EBL language (Mozart syntax).

```plaintext
Window={ETk.etk.build_window(name:top
td(glue:nsw
    Toolbar
    lr (glue:nsw
        text(name:textbox
             relief:raised
             glue:nsw
            handle:TextHandle
            bg:ETk.etk.color.white)
        scrollbar(name:scrollbar
                     orient:vertical
                     glue:ns
            handle:Scroll)))}
```

Notice that the widget modeling the text zone of our application is named `textbox`.

We then create the migration method which will write the capability to a file called `migration.cap` whenever it is called with a reference to a particular widget.

```plaintext
Window=ETk.etk.build_window(name:top
td(glue:nsw
    Toolbar
    lr (glue:nsw
        text(name:textbox
             relief:raised
             glue:nsw
            handle:TextHandle
            bg:ETk.etk.color.white)
        scrollbar(name:scrollbar
                     orient:vertical
                     glue:ns
            handle:Scroll)))
```

Eventually, we add an event listener on the `textbox` widget.

```plaintext
{ForAll [textbox]
    proc($WidgetName)
    in
        Widget=Window.WidgetName
```
public ServiceResponse provide(ServiceRequest request) {
    try {
        /* Write the capability file to a temporary file. */
        File capFile = request.createTempFileForSubject();

        /* Initialize the capability reader to extract the correct field. */
        EBLCapabilityReader reader = new EBLCapabilityReader(capFile);

        /* Read the next entry */
        String valueOfCap = reader.valueOfNextCapabilityEntry();

        /* Execute the EBL loader */
        String commandAndArgs[] = new String[2];
        commandAndArgs[0] = capabilityLoaderName;
        commandAndArgs[1] = valueOfCap;
        Runtime.getRuntime().exec(commandAndArgs);

    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    } catch (IOException ex) {
        ex.printStackTrace();
    } catch (EBLCapabilityReaderException ex) {
        ex.printStackTrace();
    }

    /* no response required so create an empty service response object, containing authentication fields only. */
    return new ServiceResponse(this.getClass().getCanonicalName());
}

Listing 10.2: Request method of the EBLMigrationServiceProvider.
Everytime a "mouse button released" events occurs the listener is triggered and invokes the Migrate method with a reference to the widget where the event occurred as parameter.

```plaintext
[Widget bind (event: 'B1-ButtonRelease')
action: proc($)
try {Thread.terminate @ThId} catch _ then skip end
{Widget set(bg:@OldCol cursor:arrow)}
if @Side=none then
%Call migrate for current widget:
{Migrate {Widget getRef($)}@Side}
end
end]
```
11.1. Extending the RelateGateways Architecture .......................... 98
11.2. Creating a New Service .................................................. 99
  11.2.1. The Service class .................................................... 99
  11.2.2. Creating a ServiceProvider .......................................... 100
  11.2.3. Creating a ServiceRequester ........................................ 101
  11.2.4. Creating a Device .................................................. 102
  11.2.5. Creating a Condition ................................................ 103
  11.2.6. Adding the Device to the Application ............................. 104
11.3. Running the Application ................................................ 104
  11.3.1. Network Configuration .............................................. 104
  11.3.2. Using the WoZ Interface ........................................... 105
  11.3.3. RMI Registry Browser ............................................. 105
11.4. Versions’ History ....................................................... 106
11.5. EBL and RGW ............................................................. 107

The Developers’s manual contains some pointers and tutorials on how to use the RelateGateways framework for the prototyping of mobile interaction with pervasive services. It complements Chapter 9 by providing some concrete howto’s.

It is worth noting that this manual is not extensive and more information can be found on the project’s website [27] on the troubleshooting blog [3] as well as in the RGW Java API documentation located in the doc folder of the project.

11.1. Extending the RelateGateways Architecture

Since this is an Open Source project everyone is warmly welcomed to contribute and enhance the system. The project’s sources are available in the src folder. Since it was developed in the Netbeans 5.5 IDE [2], it can be opened as a project and easily recompiled in this environment. However, it is based on an Ant script which makes it easy to be integrated to other IDEs such as Eclipse or to be recompiled using the Ant commands.

98
11.2. Creating a New Service

This tutorial demonstrates the required steps to create a new service. As an example we show how the first prototype of the EBLMigrationService was created. This service basically offers to migrate parts of a user interface from a device (e.g., the mobile device) to another (see Chapter 10 for a more detailed description of this service).

11.2.1. The Service class

In general terms, the first step is to subclass the Service class and decide whether the new service will be of type PullAndPushService or PushService. Next, the default constructor containing a service name and description should be implemented. As these fields are used by the final user interface and help the user find services, the description should be as complete as possible and oriented towards human-readability.

```java
package uk.ac.lancs.relate.gateways.services.impl.migration;

import java.util.concurrent.ExecutionException;

public class EBLMigrationService extends RMIService implements PullAndPushService {
    private String EBLWorkingDirectory;

    // Creates a new instance of <code>EBLMigrationService</code>
    public EBLMigrationService(TrackableDevice device,
            String serviceHost,
            int servicePort,
            String providerStubName,
            String EBLWorkingDirectory) {
        super("MigrationService", "This service offers "+
                "to migrate part of a user interface to another computer."+
                device, serviceHost, servicePort, providerStubName);
        this.setEBLWorkingDirectory(EBLWorkingDirectory);
    }

    // Returns the working directory of the EBL libraries and programs.
    public String getEBLWorkingDirectory() {
        return EBLWorkingDirectory;
    }

    // Sets the working directory of the EBL libraries and programs.
    private void setEBLWorkingDirectory(String EBLWorkingDirectory) {
        this.EBLWorkingDirectory = EBLWorkingDirectory;
    }
}
```

While extending the Service class directly is perfectly fine, but one may also want to extend a more specialized service class. As mentioned before the framework does not require the services to use a particular communication API, such as the Java Sockets API or RMI. However, it provides a number of specialized Service that can be subclassed in order to use a particular communication API. These classes are located in uk.ac.lancs.relate.gateways.services.architecture.communication.* and offer a very efficient way of designing a new service. Indeed, these components fully implement various communication protocols. As a consequence, the programmer can simply subclass them and focus on the business core of the service she designs instead of having to deal with communication issues as well. For this service we choose to use the RMI connectivity tools and extend the RMIService class.
11.2. Creating a New Service

Notice the EBLWorkingDirectory field declared in this class and accessible through the getEBLWorkingDirectory(). Properties that need to be shared at the service level (i.e., used by both requesters and providers) are to be defined here. In this case, since the working directory of the EBL system is required by both the service requester and the provider, we define it at the service level.

### 11.2.2. Creating a ServiceProvider

The next step consists in creating a ServiceProvider. As explained in Subsection 9.1, a provider is the component that implements the business logic of a service. In our case, the EBLMigrationServiceProvider loads a ticket corresponding to the part of the interface to import on the provider side. Since the EBLMigrationService uses the RMI communication, the provider extends the RMIServiceProvider class. Note, that this is not required. A service could well make use of two different communication APIs, one for the provider and another one for the requester. Nevertheless, a provider needs to extend the ServiceProvider class either directly or indirectly by extending a specialized provider such as the RMIServiceProvider.

```java
package uk.ac.lancs.relate.gateways.services.impl.migration;

/**
 * This class provides UI migration service using the EBL framework. It
 * extends the RMIServiceProvider and thus uses RMI connectivity.
 */
public class EBLMigrationServiceProvider extends RMIServiceProvider {
    private static Logger log = LoggingManager.createLogger("EBLMigrationServiceProvider");
    private String capabilityLoaderName;

    /** Creates a new instance of <code>EBLMigrationServiceProvider</code>
     * @param capabilityLoaderName Name of the program used to load capabilities
     * @param on the local host.
     */
    public EBLMigrationServiceProvider(EBLMigrationService service, String capabilityLoaderName) {
        super(service);
        this.capabilityLoaderName = service.getEBLWorkingDirectory() + File.separator + capabilityLoaderName;
    }

    public ServiceResponse provide(ServiceRequest request) {
        try {
            // Write the capability file to a temporary file. */
            File capFile = request.createTempFileForSubject();

            // Initialize the capability reader. */
            EBLCapabilityReader reader =
                new EBLCapabilityReader(capFile);

            // Read the next entry */
            String valueOfCap = reader.valueOfNextCapabilityEntry();

            // Execute the EBL loader */
            String commandAndArgs[] = new String[2];
            commandAndArgs[0] = capabilityLoaderName;
            commandAndArgs[1] = valueOfCap;

            Runtime.getRuntime().exec(commandAndArgs);
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        } catch (EBLCapabilityReaderException ex) {
        }
    }
}
```
11.2. Creating a New Service

The core of this class is the `public ServiceResponse provide(ServiceRequest request)` method. Each provider has to implement such a method. It consumes a `ServiceRequest` on which it applies the provider’s logic. When this is done it returns a `ServiceResponse` object.

Recall that this object can contain anything the provider would like to communicate to the requester. In the case of a TV service this object would typically contain a remote control to further navigate through the channels. In our case it does not contain anything in particular (other than authentication fields). Still, the provider must return a `ServiceResponse` as an acknowledgment that the `ServiceRequest` was executed.

### 11.2.3. Creating a ServiceRequester

Similarly to the provider we now create a requester. A requester must at least extend the `ServiceRequester` class or one of its specialization classes implementing a communication API. In our case we extends the `RMIServiceRequester`. Just as for providers, requesters must implement a method containing their core logic. The `request(Object o)` method is responsible for packing the service subject and sending it to the corresponding provider in a `ServiceRequest` object. Recall that the requester will be executed on the client side and thus should contain everything the service needs to gather from the client in order to fulfill a valid request.
11.2. Creating a New Service

```java
private String pathToCapability;

/** Creates a new instance of <code>EBLMigrationServiceRequester</code> */
public EBLMigrationServiceRequester(EBLMigrationService service,
        String capabilityFileName) {
    super(service);
    this.pathToCapability = service.getEBLWorkingDirectory() +
            File.separator + capabilityFileName;
}

/** This method requests a widget of a user interface to be migrated
 * to the device on which the EBLMigrationServiceProvider runs.
 * @param o The gateway view (or any other subclass of JFrame) attached
 *         to the calling controller. */
public void request(Object o) {
    try {
        GatewayView gatewayView = (GatewayView) o;
        File capFile =
                new File(getPathToCapability());
        /* confirm the fact that we are sending the request */
        super.displayMessage(
                "The user interface part is being migrated."
                        +
                "\nP lease allow a few seconds for " +
                "it to be opened on the workstation.\", " Confirmation");
        /* create the request */
        ServiceRequest request = new ServiceRequest(capFile,
                this.getClass().getCanonicalName());
        super.request(request);
    } catch (ServiceRequestException ex) {
        ex.printStackTrace();
    }
}

/** Returns the path (filename included)
 * to the capability (i.e. ticket) to be
 * used to import a widget (i.e. a part of the user interface. */
public String getPathToCapability () {
    return pathToCapability;
}

/** Sets the the path (filename included) capability (i.e. ticket) to be
 * used to import a widget (i.e. a part of the user interface. */
private void setPathToCapability(String pathToCapability) {
    this.pathToCapability = pathToCapability;
}
```

11.2.4. Creating a Device

In our model (recall Figure 9.1) services are offered by Devices. In the case of the EBLMigrationService the device offering it is a workstation. Thus, we create a WorkstationDevice extending the TrackableDevice class. A device is in charge of instantiating its own services. This task are implemented in the createServices() method. We start by creating the services, their respective providers and requesters and then we add each service to the service list. The framework will take care of registering the services and make them available to clients at run-time, whenever these are started using the Service Management Tool (see 9.1.5).

```java
package uk.ac.lancs.relate.gateways.devices;
/**
```
11.2. Creating a New Service

This class models a workstation.

```java
public class WorkstationDevice extends TrackableDevice {

    private Configuration conf;

    public WorkstationDevice(String name, int width, int height, Location p,
                               String host, GatewayApplication app,
                               ImageIcon iconRepresentation) {
        super(name, width, height, p, host, app, iconRepresentation);
        conf = new Configuration(ConfigurationTools.CONF_FILE_NAME);
        this.createServices();
        super.setToolTipTextMessage(
            "drag&drop an object here to invoke a service on it " +
            "or click for more options.");
        super.setController(new GatewayController(this));
    }

    protected void createServices() {
        /* Setting up the services offered by this workstation: */
        /* Creating an EBL Migration service... */
        EBLMigrationService migrationService =
            new EBLMigrationService(this, super.getHostname(),
                                    conf.getInt("RMI_REGISTRY_PORT"),
                                    conf.getString("EBL_MIGRATION_STUBNAME"),
                                    conf.getString("EBL_WORKING_DIRECTORY"));

        /* Detect the OS as the cap loader is system dependant */
        String os = System.getProperty("os.name").toLowerCase();
        String loaderName = null;
        if(os.contains("linux")) {
            loaderName = "receiver_import_unix";
        } else {
            loaderName = "receiver_import.exe";
        }

        /* ...and creating its provider and requester */
        EBLMigrationServiceProvider provider =
            new EBLMigrationServiceProvider(migrationService, loaderName);
        EBLMigrationServiceRequester requester =
            new EBLMigrationServiceRequester(migrationService,
                                             conf.getString("EBL_MIGRATION_CAPFILE"));

        /* eventually we add it to the list of provided services */
        super.addService(migrationService);
    }
}
```

11.2.5. Creating a Condition

The next step is to create a Condition for the WorkstationDevice. As explained in Section 4.3 a condition represents the rules to be evaluated (in conditional mode only) on the client side before in order to decide whether or not to show a gateway on the mobile screen.

The WorkstationCondition extends the WithinZone class provided by the Spatial Context Toolkit. This type of condition enables to define a zone within which the user should be located in order to see the gateway appear. In our case the gateway will appear if the user is standing at most 80 centimeters away from
11.2.6. Adding the Device to the Application

The last step is to instantiate the device in the application in order for it to be taken into account by the framework. This is done in the GatewayApplication class of the uk.ac.lancs.relate.gateways package. The setupDevices() method is the place where we want to instantiate our WorkstationDevice, since it is called automatically by the framework when the running instance wants to provide services.

```java
/**
 * Create all the devices.
 */
public void setupDevices() {
    WorkstationDevice workstation = new WorkstationDevice("Workstation", 10, 30, new Location(0, 0, 0),
        conf.getString("EBL_MIGRATION_HOST");
    new javax.swing.ImageIcon(getClass().getResource("/images/" +
        Settings.WORKSTATION_ICON));
    workstation.getController().setCondition(
        new WorkstationCondition(this, "user", workstation));
    deviceMap.add(workstation);
}
```

Note that the last line of this method deviceMap.add(workstation) adds the device to the map of devices currently available. Since the framework uses this list whenever an operation is required on the devices you should not omit to add your new device to the list.

11.3. Running the Application

11.3.1. Network Configuration

The RelateGateways framework is programmed in such a way that most of the network parameters are guessed by the system at run-time. However, in the current version two groups of parameters need to be set up before starting the application.

The first group of parameters to set are the JVM arguments:

```
-Djava.library.path=.../native/lib/linux/
-Djava.rmi.server.hostname=192.168.1.101
-Djava.security.policy=src/policies/universal.policy
```
The first parameter is only required if you use the application with the Relate sensor nodes. It specifies the path to the operating system native libraries required by Relate.

The second should be set to the IP address (or hostname) of the computer you are running the system on. It is not strictly required but since Java tends to take the address of the localhost interface as the hostname (especially on UNIX/Linux hosts) this will lead to several problems when providing services. [8] further discusses this RMI-related issues.

The third parameter is similar. It should be set to the IP address (or hostname) of the host as well. This parameter is the address of the ClassServer automatically started by the framework whenever services are provided by the current host.

The last one contains the path to the security policies and does not need to be changed unless you need to. Note that the universal.policy is wide open and thus should never be used in production environments.

Since the RelateGateways application is symmetric every host (including the mobile device) should use these JVM arguments. However, if the mobile host does not provide any service you may wish to use a slightly simpler version, that does not require any modification when changing the network configuration:

```
-Djava.library.path=nativelib/win32/
-Djava.security.policy=src/policies/universal.policy
```

### 11.3.2. Using the WoZ Interface

As presented before, the WoZ interface permits to simulate the spatial context. This tool is Accessible through the “Simu → WoZ” menu of the AdminToolBar.

The address to which the wizard should connect in order to send the simulated context has to be specified in the ipsettings.conf file located in the config folder of the project. In this file, the line:

```
# Wizard of Oz
MOBILE_DEVICE_WIZARD_ADDRESS = 192.168.1.102
```

sets the address of the mobile device to which to connect.

This file contains a number of other settings used by the framework. Normally these settings do not need to be further modified.

### 11.3.3. RMI Registry Browser

As shown on Figure 11.1 a tool is provided to browse a RMI registry. This application is especially useful to debug the dynamic discovery mechanism and monitor the various RelateGateways Service registries.

To start the tool, launch the RMIRegistryBrowserGUI main class located in the uk.ac.lancs.relate.gateways.monitoring.rmi.gui package.
11.4. Versions’ History

The RelateGateways application was developed using a top-down approach. While this allows for early study of the prototype it also implies a fair amount of refactoring to create a clean and extensible framework from the prototype.

This refactoring process as well as the software evolution through all the project’s stages are summarized on the table below.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes Summary Description</th>
<th>Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>First prototype. First version of the gateways user interface for the test run in Lancaster. Implemented three basic (and hard-coded) services.</td>
<td>D. Guinard, S. Streng.</td>
</tr>
<tr>
<td>Version</td>
<td>Changes Summary</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>0.1</td>
<td>Transforming the prototype to an application.</td>
<td>UI improvements according to study in Munich, refactoring of services in order to create a full distributed application. Partial refactoring of the Spatial Context Toolkit. Implementation of the serialization/deserialization layer.</td>
</tr>
<tr>
<td>0.2</td>
<td>Service Discovery and Interoperability.</td>
<td>Implementation of the Dynamic Class Loading components using RMI. Consolidation of the Service architecture by extracting the connectivity API from the architecture. Turned the Descriptive Service Architecture to a Service framework for the prototyping of pervasive Services. First integration of the EBL framework as a RelateGateways service. Improved UI with feedbacks from Cakes Talk and demonstrations. New services using the Java Robot API.</td>
</tr>
<tr>
<td>1</td>
<td>Transforming the application into a prototyping framework.</td>
<td>Refactoring the SCT to make it dynamic. Voice synthesis tool for reading the service descriptions. Consolidation of the whole system into a single prototyping application. Configuration management classes. Implementation of an HTTP server to serve as a Class Server in order for clients to retrieve missing classes. First integration with relate. Bug fixes.</td>
</tr>
</tbody>
</table>

### 11.5. EBL and RGW

In order to be able to use the migration service, the Oz language has to be installed. The installation is documented on [20]. Currently, the EBL system supports the version 1.3.2 of Oz only, thus it is currently the version to download to avoid incompatibilities. An installation bundle of this version, for both Windows and Linux, can be found in the project’s CD-Rom (see F).

The EBL toolkit is located in the migration folder of the project and does not require any further installation. To run the “notepad” example, open the NoteAppDND.oz source code in the Oz IDE (which requires Emacs to be installed) and select: “Oz → Feed Buffer”. The application should start after a few seconds.

For more information about the EBL toolkit please refer to the official documentation on [6] or on the troubleshootings blog of the RelateGateways project [3].
Part III.

Evaluation, Conclusion and Future Work
This project was defined and implemented with a top-down approach in order to be able to run a user study on the interface prototype as early as possible. The first part of this chapter describes the different phases of this user study. Besides being an evaluation tool, the study was used a mean for improvements. So were the various public demonstrations of the the application which influenced the final design. In order to test the usability and relevance of the application from an end-user point of view, the user study was conducted on an early prototype (beta version, see Chapter 11 for a description of the features included in each version.).

This prototype did not include all the features of the final application. More precisely it was a first implementation of the spatial discovery layer and a partial implementation of the service framework. However, since the goal of the user study was to evaluate the user interface and the underlying interaction technique this had no real influence.

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1. Setup</td>
<td>110</td>
</tr>
<tr>
<td>12.1.1. Spatial Conditions</td>
<td>110</td>
</tr>
<tr>
<td>12.1.2. WOZ</td>
<td>111</td>
</tr>
<tr>
<td>12.2. Test Run</td>
<td>111</td>
</tr>
<tr>
<td>12.2.1. Scenario</td>
<td>112</td>
</tr>
<tr>
<td>12.2.2. Questionnaire</td>
<td>113</td>
</tr>
<tr>
<td>12.3. Execution</td>
<td>114</td>
</tr>
<tr>
<td>12.4. Qualitative Results</td>
<td>114</td>
</tr>
<tr>
<td>12.4.1. General Impression</td>
<td>114</td>
</tr>
<tr>
<td>12.4.2. Assessing the Disturbing Factor</td>
<td>116</td>
</tr>
<tr>
<td>12.4.3. Comparing the Interaction Techniques</td>
<td>118</td>
</tr>
<tr>
<td>12.4.4. Comparing the Modes of Operation</td>
<td>118</td>
</tr>
<tr>
<td>12.4.5. Comments</td>
<td>120</td>
</tr>
<tr>
<td>12.5. Quantitative Results</td>
<td>120</td>
</tr>
<tr>
<td>12.5.1. Critical Evaluation</td>
<td>121</td>
</tr>
</tbody>
</table>
12.1. Setup

In order to evaluate the application we designed a small scenario. As shown on Figure 12.1, three devices are installed in a large room: a printer, a display, and a keyboard. Each device is attached to a computer. Each computer is then connected to a router coupled with a wireless access point in order to create both a wired and wireless local network.

Through their respective gateways the three devices offered one to two services each. The display provided two services. When drag-and-dropping a presentation it would display it in full screen mode, when clicking on the corresponding gateway the display would open the OpenOffice document (presentation, text) in edit mode. Similarly, the printer directly printed the drag-and-dropped documents whereas a click on the printer gateway would prompt the user for selecting more printing options (such as orientation, number of pages, format, etc.). Finally, the keyboard could be "borrowed" to type text into a new OpenOffice writer document.

Additionally, the RelateGateways prototype was installed on two types of mobile devices: a OQO handheld PC (see Figure 4.9) and a Paceblade tablet PC both running Windows XP.

12.1.1. Spatial Conditions

Recall that in the conditional mode of operation the gateways appear if and only if all the spatial conditions for this device are evaluated to true. In order to test the conditional mode of the application each device was assigned a condition.

The keyboard was assigned a condition on the user's distance to the device. Indeed, it does make sense for a user to use the keyboard if and only if this latter is situated close enough. If the user was located within 0.5 meters of the keyboard the system would display the corresponding gateway:

```java
if (dist(user, device) <= 0.5)) then true;
else false;
```
The display’s condition was slightly more complicated. Here we wanted to enforce the fact that the user needs to face the display to use it. Furthermore, since the range of use of a display is much bigger than for a keyboard, we set the distance 3 meters:

\[
\begin{align*}
\text{if} & \quad (\text{dist(user, device)} \leq 3 \text{ AND isFacing(user, device)}) \text{ then true} ; \\
\text{else} & \quad \text{false};
\end{align*}
\]

Finally, the printer gateway is shown if the user is within a meter of the device or if the user is moving towards the printer:

\[
\begin{align*}
\text{if} & \quad ((\text{dist(user, device)} \leq 1 \text{ OR isMovingTowards(user, device)}) \text{ then true} ; \\
\text{else} & \quad \text{false};
\end{align*}
\]

Note that when the user switches to the scanning mode all these conditions are ignored and the system shows all the available gateways, all the time.

### 12.1.2. WOZ

In order for the user study not to be influenced by limitation of the early releases of the positioning technology the spatial context is simulated using the Wizard of Oz interface (see Figure 12.3). This makes the application independent of the current sensing constrains and lets us focus on the user interaction and spatial discovery concerns. Since the Wizard interface was already discussed in Subsection 4.2.3 we will not discuss its functionalities any further here.

### 12.2. Test Run

Before the actual user study a test run was organized. The idea of this first test with real users was, to identify the prototype (alpha version) flaws and design issues to improve before the actual study.

This section describes both the test run and the study. The test run was setup at Lancaster University in a working office. Five users (4 computer scientist, one management
12.2. Test Run

student) were guided through three different tasks arranged as explained above. We asked the users to provide direct feedbacks about what they were doing and how they felt about it when trying to accomplish the tasks. This information was video recorded as well as using a microphone. Additionally, during each test run the log file of the application was recorded in order to be able to post-analyze it for detecting software bugs.

Besides allowing to identify bugs, observations during this phase permitted to improve some parts of the user interface. As an example, most user expressed a feeling of having to disconnect from the keyboard gateway once the service was consumed. This feature was not provided nor really required by the application but it was included in the next release.

The interaction zones of the gateways were also redesigned after the test run. Indeed, the initial version comprised two zones corresponding to the two interaction techniques: a drag-and-drop zone and a button. Since our five first users mentioned this fact as quite confusing we decided to redesign the gateways with a single zone for both interaction schemes. Figure 4.11 shows this evolution of the gateway’s design.

Finally, during this first phase we identified the fact that clearer feedbacks were required from the application on the mobile device when consuming a service. Since the invocations may take a few seconds before having an effect on the service provider, confirmation feedback were added to the system.

12.2.1. Scenario

Each user was expected to give us about 20 to 30 minutes. This time was decomposed into three distinct parts: an introduction, a testing phase and a concluding interview.

Introduction

During this part of the tests, the users were given an overview of the project’s aim. The setup, service and tasks were explained. Additionally, a short introduction to the device
was given. This was especially important for users using the OQO mobile PC since the clicking and pointing system of this device is not completely straightforward.

It is worth noting that no indication on the use of the gateways interaction technique was given. Since the study aimed to address the usability, intuitiveness and relevance of interaction technique it was a key point not to teach them how to use the user interface.

**Testing Phase**

The testing phase was decomposed into two distinct parts. During the first part the user was given three tasks:

1. The first task consisted in finding a keyboard and using it order to type some text in OpenOffice Writer.
2. The second task was to use a display in the room to display one of the presentations available on the mobile desktop.
3. The third task consisted in printing a document available on the desktop. After printing a first document, we asked the user to try printing a second one but in a landscape format.

Since the application operated on conditional mode during the tasks solving phase (see 4.1.2 for a conceptual description of these modes), we informed the users about the scanning mode and let them try it for a while.

**12.2.2. Questionnaire**

At the end of the testing phase, all the participants were interviewed during about 10 minutes. A questionnaire was prepared but the questions were asked orally in order to prevent misunderstandings and allow discussions going out of the questionnaire’s scope.

The questionnaire’s content is available in Appendix B. It basically consisted of four distinct parts. In the first part we aimed to capture the user’s general feeling about the interface and the interaction technique. Furthermore, we tried to assert the relevance of the system as a whole, asking whether they could perceive any benefit in using such an application and whether they had concerns (security, privacy) about it. In this part they were also asked to suggest services that would fit well with the system and that they would like to see implemented. Finally, a couple of questions captured usability issues.

The next part was really focused on the user interface. It analyzed the disturbing factor induced by the gateways appearing and moving at the edge of the screen. Furthermore we tried to assess the degree of attention in order to know if the gateways should be more/less discrete.

Part three was a comparison of both interaction techniques (click/drag-and-drop), trying to capture both their intuitiveness and enjoyment factor.

Part four was comparing the two modes of operation (scanning and conditional mode). The questions were focused on understanding the perceived use and usage of each mode. A question of this part aimed to spot the right degree of intelligence the user interface should have in order to fit the users’ needs.
12.3. Execution

Using the data of the test run, a new release (beta version) of the gateways user interface was developed to be tested during the user study. This study was organized in a large meeting room at LMU (Ludwig-Maximilians-Universitaet) in Munich, Germany, with the setup exposed above.

In total 15 users took part to the study which was executed during two consecutive days. All of them were students in Multimedia or Computer Science. 13 of them were men, 2 women. Twelve participants were between 21 and 25 years old, two between 26 and 30 and one between 31 and 40.

The 20 to 30 minutes each user gave us were distributed as follow. The introduction/device training time lasted for about five minutes. The task solving part took about five minutes. Followed by the scanning mode testing which lasted about two minutes. The remaining time (15 minutes) was spent on the interview.

During the two days the study lasted no major bug prevented the users to execute the tasks. A few minor flaws had an influence on the timing but these were taken into account and are clearly marked in the final results.

Finally, it is worth noting that the participants where given two different devices. Nine people were given an OQO mobile PC to test the application whereas six tested the system with a Paceblade Tablet PC. In terms of form factors and functionalities we could easily assume the former as being a smart phone and the latter as being a laptop. This fact makes them two quite different, but still “mobile”, devices to test the application on.

12.4. Qualitative Results

The qualitative results of the study are primarily based on the interviews. Thus, we enumerate the four sections of the questionnaire and comment the global results.

12.4.1. General Impression

The general feedbacks were quite positive. We asked users to rate their answers from 1 (not at all) to 5 (very much). When asking them whether they liked the application they rated it as 4.47 in the mean with a standard deviation of 0.499. That is between “rather yes” and “very much”.

More interesting, when asked whether they would use such an application they rated this as 4.47 again with a standard deviation of 0.718. Which means that answers ranged from “neutral” to “rather yes” and “very much”. When asking them more details about where they would use it the majority of users (7) said they would find it useful in places they are not familiar with. This result was well welcomed since it matches our vision of the application. Next to unfamiliar places, they cited public places such as university buildings or public offices as shown on Figure 12.4.
12.4. Qualitative Results

The fact that no installation or configuration was required to interact with the service was the most widely perceived benefit of the application. This shows, once more, the relevance of Weiser’s vision [Wei91] of the disappearing computer, in which the users’ administrative tasks (e.g. device installation or configuration) are reduced to as few as possible. Our users expressed this fact as a requirement when it comes to visiting a new environment, which tends to show the pertinence of this assumption when addressing interactions with pervasive services.

Perceived as well by 7 users, the fact that the system enabled to use devices on site, without having to carry them along, is also a result that confirms the relevance of our project. Indeed, it emphasize the fact that users would really like to be able to use the potential power of pervasive services.

They also enjoyed being able to interact with all the available services using the same intuitive techniques. All users mentioned the drag-and-drop interactions across devices as being particularly appealing. A summary of the perceived benefits is shown on Figure 12.5. When discussing security issues five users had concerns about it. Three users said they might have problems depending on the service they would use. When summing up these two groups we have more than one user out of two expressing concerns about security. This is an interesting result showing that spontaneity does raise security concerns for both users and system designers. This will be further discussed in Section 13.2.

In order to get a better insight of what kind of service users would use we asked them to rank the three provided services. The most popular service was the printer, followed by the display and the keyboard. We further asked the users what kind of services they would like to see implemented. A number of interesting suggestions were formulated.
They mentioned file sharing services to share files between mobile phones, notebooks or hard disks. They suggested a couple of multimedia services to play mp3 songs or music play-lists. Furthermore, they proposed shared applications such as blackboards, calendars or public displays where people could publish information in a quick an easy manner.

Using the system as a token for vending machines such as DVD rental systems was also suggested. Someone imagined an access control system in which you could gain access by virtually sending a key file to the door gateway. In the same trend, a virtual vallet system was hinted: users could have virtual representation of cards (insurance, credit, etc.) that they could drag and drop on the corresponding gateways.

![Figure 12.6: Detailed results for the ratings part of the general assessment.](image)

The remaining of this section was composed of directed questions rated from one to five. These questions aimed to capture the general user experience when interacting with the user interface. The results were globally really positive with rather low standard deviations. As a consequence we can say that generally all the users had a good experience with the user interface. The global results for all the ratings of the general assessment part are summarized on Table 12.1. A visualization of the full results is provided on Figure 12.6. In this figure, bars (or horizontal lines) represent the distribution of ratings, vertical lines represent the standard deviations intervals, dots and points show isolated answers.

### 12.4.2. Assessing the Disturbing Factor

During the second phase of the interview the questions were focused on measuring whether extending the desktop metaphor with the gateways was disturbing for our users. For these
12.4. Qualitative Results

<table>
<thead>
<tr>
<th>Question</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I like the application.</td>
<td>4.47</td>
<td>0.516</td>
</tr>
<tr>
<td>I would use the application.</td>
<td>4.47</td>
<td>0.718</td>
</tr>
<tr>
<td>The application is easy to use.</td>
<td>4.40</td>
<td>0.737</td>
</tr>
<tr>
<td>The application is useful.</td>
<td>4.13</td>
<td>0.640</td>
</tr>
<tr>
<td>The application attractive.</td>
<td>4.33</td>
<td>0.617</td>
</tr>
<tr>
<td>The application is intuitive.</td>
<td>4.20</td>
<td>0.941</td>
</tr>
<tr>
<td>The application is clumsy to use (frustrating).</td>
<td>1.27</td>
<td>0.594</td>
</tr>
</tbody>
</table>

Table 12.1.: Ratings for the directed questions of the general assessment.

questions we asked the users to imagine themselves using the application in the everyday life (while being on the move for instance), rather than in the test laboratory. The aim of this remark was for users to nuance their answers when thinking about the use of such an application in real-life.

![Figure 12.7: Asserting how much the gateway’s appearance was interrupting.](image)

In the first question we asked the users to rate how interrupting they would find the automatic appearance of gateways on the screen. The mean answer to this question was “rather not”, or 2.07 in numerical terms. The standard deviation was slightly higher (1.100) than for the previous one, as a consequence the answers ranged from 1 (not at all) to 4 (rather yes). Figure 12.7 provides an overview of the answers’ repartition on the ranking scale.

The two next questions were related since we asked the users whether the gateways should attract more, respectively less attention. None of the user said it should attract less attention. Three users said it should attract slightly more attention. The gateways’ design was generally well received but some users suggested design improvements as listed below:

- Use “flashier” icons.
• Make the gateways more colorful.
• Make the gateways bigger when they appear.
• Use transparent windows in order not to completely hide the background.

Furthermore, a number of users discussed the size of the gateways. We designed them in order to be as unobtrusive as possible on mobile devices, i.e. quite small. While this was fine with the users as long as they did not interact with the gateways, some suggested the system to somehow detect their will to use a particular gateway, e.g. when approaching the mouse, and make it bigger in order to ease the interaction.

When asking whether the gateways were hiding other important elements of the desktop 14 users said no but one had troubles closing OpenOffice Writer since the gateway was hiding the X button on the top right of the window.

12.4.3. Comparing the Interaction Techniques

One of the gateways user interface motto was to provide a set of uniform and simple ways of accessing all the services. As mentioned before two techniques were proposed: the drag-and-drop for direct invocations and the click for more options.

The goal of this part of the interview was to compare and contrast the two techniques. At first we asked how easy it was to find out about these two possible interactions. The mean answer was 3.40 with a standard deviation of 1.183.

Eleven users out of fifteen first identified the click interaction. While this result may appear surprising it is in fact strongly influenced by the study scenario. Indeed, by simply asking a few user to start by using the display gateway instead of the keyboard gateway (on which you could only use the click interaction) the results were completely inverted: users would identify the drag-and-drop interaction as more intuitive. As a consequence it is likely for the most intuitive technique to be the one suggested by the first device users interact with.

This result and the previous one suggests that the interactions techniques where well perceived once experienced but not completely obvious at first. This emphasize the need for teaching the users about the possible interaction techniques. Various systems can help towards this goal such as providing tooltip messages, message bubbles, tutorial animations or audio commentaries which are more adapted to devices with small screens.

When considering which technique was the most appealing, all the users picked the drag-and-drop. This confirms the fact that drag-and-dropping objects is a method of choice when considering interaction with pervasive services. This result is not surprising since the drag-and-drop technique mimics the real-life tangible interactions.

12.4.4. Comparing the Modes of Operation

As mentioned before, the gateways user interface offers two modes of operation. While the focus of the user study was on the conditional mode we briefly introduced the users to the scanning mode in order to compare them.

In the first questions we asked the users whether they would prefer see all the gateways of service in the room all the time or if they would rather have the system guessing for
12.4. Qualitative Results

Figure 12.8.: Global results for always displaying the gateways or rather only when needed.

them when it did make sense to show the gateways. The answers to these questions were rather contrasted as shown on Figure 12.8. The mean answer for seeing all the gateways all the time was 2.27 with a standard deviation of 1.486, meaning that people were quite neutral. The same is true when asking whether the system should rather guess when to show a gateway, the mean was 3.00 with a standard deviation of 1.558.

When asking how often they would use the scanning mode and the conditional mode the same kind of result appear. They would use the scanning mode 57% of the time and the conditional mode 43%. The distribution of these results on a per user basis is shown on Figure 12.9.

Although the trend here was rather hard to extract, the conditional mode seemed to be slightly preferred for everyday use whereas the scanning mode was the favorite in unknown environments. We further asked our users what part of the current context the system should take into account in order to show the right gateway at the right moment (in the conditional mode). The most common answer was to take into account the context in the current application. As an example the keyboard gateway should appear whenever a word processor is opened. When saving a text document the printer gateway could appear similarly, when saving a presentation the display gateway should appear.

Another quite interesting suggestion was to monitor the current use of the file system to infer the gateways to pop up. This way, a user opening a folder containing presentations would see the display gateways appear. A user browsing for mp3 files could see the HiFi gateway pop up.

Yet another user proposed to record the user’s activity when entering a new room in order to infer the gateway he was most likely to use. This way when entering a room for the tenth time to print documents, the printer gateway should automatically appear. Similarly, a user often looking for keyboards should see the keyboard gateways as soon as a corresponding device is available in the environment.
12.5. Quantitative Results

Figure 12.9.: Percentage of predicted use between the scanning and the conditional mode.

Even if their implementation raises non-trivial problems that are close to artificial intelligence and machine learning, these suggestions give a good sense of the fact that users expect applications on mobile devices to be smart enough to limit the manipulations they have to make to discover and interact with pervasive services.

12.4.5. Comments

The last section of the interview was an open discussion during which users were invited to comment on the system and provide suggestions.

Some users proposed to enhance the application with query methods in order to tell the application you were looking for a HiFi player and for the application to display the gateways to services matching the query. Another user suggested to display some information about the distance to a device, this feature was included in the next version of the user interface.

In terms of services many user hinted the fact that a service should allow more complex interactions. When displaying a slideshow for instance many users suggested that the service should return a remote control that allows to move to the next or previous slide. The support for such interactions was implemented in the new release of the service framework. They also proposed sharing software rather than only hardware functionalities. The gateways user interface could be used to identify applications available on other co-located computers. A slightly different version of this idea was implemented using the EBL framework to migrate user interfaces.

12.5. Quantitative Results

The study was mostly aiming at obtaining qualitative results on the user interface. However, we obtained two quantitative sets of results as well.
12.5. Quantitative Results

The first one is the time required by each user to perform all the tasks. On the average a user spent 4.57 minutes solving all the three tasks, with a standard deviation of 1.720 minutes. This is very encouraging and was perceived as such by our users. Imagine achieving the same three tasks with your mobile device in a previously unknown environment. Identifying, installing and configuring each of the service to work with your device would take a lot more time than five minutes.

From this simple quantitative result another quite interesting fact was extracted. It reveals the importance of the mobile device one considers when running a user study. Indeed, on the average users using the Paceblade tablet PC performed the tasks 2.3 minutes faster than those using the OQO. This was mainly due to the fact that the clicking on a gateway using the OQO required to be quite used to the device. Not so much because of the gateways themselves but rather because of the OQO (1.0) touch pad which is quite hard to use at first. The mean time per service and mobile device is shown on Figure 12.10.

![Figure 12.10: Mean time per service using the two different mobile devices.](image)

12.5.1. Critical Evaluation

This study was formative, as opposed to a comparative study. The focus was on addressing the usability of the user interface. As such it hints but does not formally proves that the use of the gateways interface and the RelateArchitecture improves the user experience with pervasive services. To complement such a user study we would need a comparative study, assessing the performances and overall appreciation of the gateways user interface when comparing it to other service visualization techniques such as service lists provided by Bluetooth or Jini.

Still, the results were quite satisfying. Almost all the users liked the user interface very much. People really enjoyed the novelty of the UI as well as the idea of being able to spatially discover services. Moreover, the fact that the system was installing service “on-the-fly” without the need for installation or configuration was well perceived. The click
and especially the drag-and-drop techniques were received as natural and intuitive. Most of the people said they would use a system such as the RelateGateways architecture if the critical mass of provided service was reached. Most of our aims when creating this project met the users’ perceived benefits, which is alone quite encouraging. The study also shown that addressing spontaneity in interactions with pervasive services meets the users’ future expectations in terms of mobile interactions.

However, our sample of users is not fully representative of the end users. Indeed, since out of the 20 users (test run and user study) only two were not studying multimedia engineering or computer science. While one may state that such students are usually more critical towards computer science projects they are also known as the “early adopters” of technologies. This fact definitely does influences results and should be taken as a safeguard, somewhat lowering the very positive results we obtained.

It is worth noting only a few users mentioned the fact that the system could help them search (i.e. identify) available service as a benefit. Although two users mentioned it, we were expecting this to be perceived by more people as a benefit. While this result is somewhat disappointing, two points need to be taken into account when evaluating it. First of all the room was not filled with enough other appliances to create possible confusions and thus make the identification of co-located services definitely useful.

More importantly study was focused on the conditional mode which, according to our user experience and the users’ feedback, not the best mode to spatially discover services. This leads to the general remark that another study would be required to really assess the usefulness of the spatial discovery. Such a study would be less focused on the user interface issues but more on a study of how do user manage to identify services with and without the application.
13.1. Contribution

As a whole the project addresses the steps of mobile spatial interactions with pervasive services. In the End-to-End discovery model we extracted three simple conceptual layers that need to be considered in order to enable spontaneous interactions in such environments. These layers offer general guidelines for providing an answer to the two research questions exposed in the introduction of this work, namely:

- In a world with a plethora of potentially accessible pervasive services how does the user actually identify the services that are available.
- Once a service is identified, how can the user interact with it in a uniform and natural manner, without the need for him to install, configure or learn using yet another system.

We further developed the RelateGateways architecture which is a concrete implementation of our model. In this architecture, the Spatial Discovery layer is based on a gateways user interface. This novel type of UI offers both a simple and uniform interaction technique as well as a way of representing pervasive services based on spatial information. This information can be gathered from any positioning system but we chose to both simulate it using a Wizard-of-Oz technique and implement it using an extension (Dots) of the Relate ad-hoc sensor network. The interface was also evaluated in a formative user study which results were reported in this thesis.

The Network and Service Discovery Layer is implemented in the RelateGateways service framework, a SOA for pervasive services. This systems supports the Relate Service Discovery system that we imagined.

The Invocation and Interoperability Layer is a novel implementation of the Mobile Code concept of the recombinant computing area. Supported by the service framework, this
system ensures that the application is fully dynamic. Indeed, it allows a mobile device to “install” and consume services it had no semantic knowledge of prior to discovery.

We consolidated and documented the RelateGateways architecture in order to integrate all the layers within a single, “run-and-use”, Java application does not require any particular infrastructure. This makes the RelateGateways architecture an ideal framework for the prototyping of spontaneous mobile spatial interactions with pervasive service.

13.2. Future Work

The RelateGateways project does not end with the end of this Master Thesis. A team of researchers at Lancaster University jointly with the author of this thesis, takes the project over. The focus is first set on achieving a clean integration of the Relate hardware to create a usable demonstrator of the system when coupled with a real sensor network. Furthermore, a number of interesting path to explore for this project were identified. We shall give an overview of some of them below.

Enabling Security  As hinted in the evaluation Chapter 12 spontaneity is a need when talking about mobile interaction with the services surrounding us. However, spontaneity has a cost directly derived from its benefit. In a spontaneous pervasive environment things are made easier for the end-user. The system takes care of interconnecting the discovered services and makes them directly available to the end user. While this might be the perfect solution in a perfect world it leads to security and privacy issues in the real-world. One probably does want Bob to be able to publish, some interesting news on a public display, but do we want Alice, detractor of our institution to be able to display disgracefull information with the same ease. Adding authentication and more generally security mechanisms seems to be really important. While the proposed version of our system does not address these mechanisms, Mayrhofer et al. in [MGH07] shape how the RelateGateways system could be used as the support for authentication methods based on proximity.

Building a Symmetric System  The model we have presented so far is asymmetric: mobile clients needs to connect to services and physical servers need to accept the clients requests. Although presented as such for the sake of simplicity, the RelateGateways application is not really asymmetric. Mobile clients and physical servers use the same instance of the application. Thus, we can imagine mobile clients becoming physical servers. Furthermore, instead of packaging everything required to invoke a service in the MobileCode we could bundle all that is required to provide a service. This way the architecture would enable the ad-hoc distribution of services amongst the physical server and mobile devices.

As a example consider a mobile device $d$, a physical server $a$ providing a file-storage service and a physical server $b$ providing a keyboard service. If symmetry was enabled the $d$ could discover the file-storage service on $a$. But since $d$ has a hard-drive as well it discover the whole service rather than just the requester part and thus start providing the file-storage service as well. Furthermore, $a$ could discover the keyboard service of $b$ and provide it as well. Indeed, in a symmetric model nothing prevents physical servers
to discover other physical servers and their services. In such a model servers are clients
and clients are servers.

The RelateGateways architecture supports symmetry and with minor changes these use-
cases could be enabled. Since it is highly dynamic, the impact of such a model certainly
raises a number of interesting challenges and novel uses.

Creating Macro and Micro Visualization of the Services The gateways user interface
provides a micro view of the environment. Using this interface the user can identify co-
located services. This system is adapted to find services in a relatively small area (such
as a floor, an office, a hall, etc.) but it does not really help to find service in a city or
a complex of buildings such as a whole campus. To help the user identify services in
the large we suggest extending the spatial discovery layer with a macro visualization of
services. Tagged maps could be used to represent available service on a campus, street
or city scale.

From an interface point of view maps such as the ones provided by GoogleMaps [13] or
[18] could be tagged with available services. The user could then “zoom in” to obtain
the obtain the micro visualization offered by the gateways user interface. From a sensing
point of view the proposed Relate sensor-network could be used for the micro visualization
whereas the GPS (Global Positioning System) could be user for the macro visualization.

Evaluating the Prototyping Framework As mentioned before, the integration all the
E2ESD layers into a single, runnable application enhanced with a service framework
makes the RelateGateways architecture a nice prototyping tool for mobile spatial appli-
cations.

While our preliminary tests hinted that a new service could be created and integrated in
an hour. Furthermore, using the application to build a study lab for testing spontaneous
mobile spatial interactions with pervasive services could be all done within an hour as
well.

However, these tests were informal and internal. A user study evaluating the prototyping
framework would be required to confirm these results and improve the system. In such a
study we would ask a couple of programmers to create new services (e.g. a multimedia-
player service, a vending machine service, a file-sharing service, etc.) using the RGW
service framework. The ideal setting would be to get students from another university to
work on small projects using the RelateGateways architecture.

13.3. Outlook

Long gone is the time when the computer had its own room in our home. Today’s
computing power is carried along and comes into many portable form factors. Our mobile
devices meet other form of computing power, everywhere, all the time. Yet, we still rarely
exploit the great power of coupling our mobile devices with co-located services.

The user study of this project proves it: users are really eager to create such ad-hoc
interactions. The biggest remaining barrier towards this goal is the lack of spontaneity.
Whilst on the move users have even less time to spend on configuration, installation and
identification processes. Thus, projects like the RelateGateways architecture, exploring new ways of addressing spontaneity, reflect an existing need.

To solve this need and address spontaneity in the bigger picture, industrial leaders have to join their efforts. Common standards for discovery and global interoperability are the only way to reduce the configuration and installation tasks, thus enabling seamless and spontaneous interactions between mobile devices and pervasive services. A number of industrial projects tackle this goal.

But enabling smooth mobile interactions is definitely not only about technical concerns and standards. It also goes through a complete re-thinking of the supporting user interfaces, creating entirely novel systems, without any fear for introducing drastically novel interaction techniques, questioning accepted ergonomics. Because enabling spontaneity in computing environments starts with understanding human centered design...
Part IV.

Appendix
The following source is the entire code of the EBL notepad used to demonstrate the EBL/RGW integration.

```plaintext
declare
ETk={Module.link ["ETk.ozf"]}

Window Scroll
Toolbar=lr (glue:we
    button (text:"Restore" glue:w relief:flat action:Restore)
    button (text:"Quit" glue:w relief:flat action:Exit))

TextHandle
Window={ETk.etk.build window (name:top
    td (glue:nsw)
    Toolbar
    lr (glue:nsw
        text (name:textbox
            relief:raised
            glue:nsw
            handle:TextHandle
            bg:ETk.etk.color.white)
        scrollbar (name:scrollbar
            orient:vertical
            glue:nsw
            handle:Scroll)))}

class TextFile from Open.text Open.file end

% Converts the capability to a numerical value
fun {BytesToString L}
    case L of X | Xs then
        Low High
        in
            Low=X div 16 % AAAABBBB => AAAA
            High=(X-(Low*16)) % AAAABBBB => BBBB
            (Low+64)|(High+64)||{BytesToString Xs}
        else nil end
end

% This procedure is called in order to create a capability
% when the user requested a migration.
proc {Migrate Ref Side}
    File={New TextFile init name:"migration.cap" flags:[create truncate write text]}
    in
        {File putS (Side#="#{BytesToString {VirtualString.toString Ref}}")}
    {File close}
end
```
% This procedure migrates the UI parts back to the initial state
proc { Restore }
  (Window restoreInitialGeometry)
end

% Exit the programm
proc { Exit }
  (Window destroyAll)
end

local
  OldCol=(NewCell unit)
  ThId=(NewCell unit)
  Side=(NewCell none)

  [ForAll [textbox scrollbar]
    proc {$ WidgetName}
      Widget=Window.$WidgetName
      in
        [Widget bind event: '1'
          action: proc {$}
            try {Thread.terminate @ThId} catch _ then skip end
            {Widget set(bg: ETk.etk.color.'LightBlue')} end]
      [Widget bind event: 'B1-Motion'
        action: proc {$ X Y}
          WX={Widget winfo(screenwidth: $)}
          WY={Widget winfo(screenheight: $)}
          in
            if X<10 then left
            elseif X=(WX-10) then right
            elseif Y<10 then top
            elseif Y=(WY-10) then bottom
            else none end
            {Widget set(cursor: r(none:shuttle
              left:left_side
              right:right_side
              top:top_side
              bottom:bottom_side).@Side}
            if @Side=none andthen @ThId=unit then
              try {Thread.terminate @ThId} catch _ then skip end
              ThId:= unit
            elseif @Side=none andthen @ThId=unit then
              ThId:=
              thread @ThId={Thread .this}
              proc {Loop}
                {Widget set(bg: @OldCol)}
                {Delay 200}
                {Widget set(bg: ETk.etk.color.'LightBlue')} 
                {Delay 200}
                {Loop}
                end
            in
              {Loop}
            end end
          try {Thread.terminate @ThId} catch _ then skip end
          {Widget set(bg: @OldCol cursor: arrow)}
          if @Side=none then
            {Migrate {Widget getRef($)} @Side}
          end
        end
      end
    end
  end
}
end

% show the window and assign the scrolling behaviour to the text box
{Window.top show}
{Scroll addYLink(TextHandle)}
This document summarizes the questionnaire used as a support for the users’ interviews during the evaluation phase.
Questionnaire

1. General impression
   - Did you like the gateway application? 1 - 2 - 3 - 4 - 5
   - Would you use the gateway application? 1 - 2 - 3 - 4 - 5
   
   If yes, which (other) services would you like to use with this application?
   1. ____________________________ 4. ____________________________
   2. ____________________________ 5. ____________________________
   3. ____________________________ 6. ____________________________
   - Where / under which circumstances would you make use of such an application?

   ____________________________________________________________

   - What benefit do you see in the application?

   ____________________________________________________________

   - Do you see any security issues?

   ____________________________________________________________

   - How easy do you find the application to use? 1 - 2 - 3 - 4 - 5
   - How clumsy do you find the application to use? 1 - 2 - 3 - 4 - 5
   - How useful do you find the application to use? 1 - 2 - 3 - 4 - 5
   - How attractive do you find the application to use? 1 - 2 - 3 - 4 - 5
   - How intuitive do you find the application to use? 1 - 2 - 3 - 4 - 5

2. Disturbing factor
   - In real life would you expect the automatic appearance of gateways interrupting? 1 - 2 - 3 - 4 - 5
   - Should the gateways attract more attention?
     □ no
     □ yes, they should ...
       □ ... be bigger
       □ ... be more colorful
       □ ... have flashier icons
     □
   - Should the gateways attract less attention?
     □ no
     □ yes, they should ...
       □ ... be smaller
       □ ... be less colorful
       □ ... have less flashy icons
       □ ... not have icons
     □
   - Did the gateways hide other important elements (e.g. the taskbar)?
     □ no
     □ yes: ____________________________
3. Button vs. drag-and-drop

There were two different ways of using the gateways
1. drag-and-drop a file to the gateway
2. click on the gateway to get more options (button)
   - How hard was it to figure this out?
   - Which one did you find more obvious: □ button □ drag-and-drop
   - Which one did you find more appealing: □ button □ drag-and-drop
   - Did you read the tooltip* message? □ yes □ no
     (* they appear when the mouse is over the gateway)
   - Were the message boxes helpful? 1 - 2 - 3 - 4 - 5
   - How much of the text in the message boxes did you read? ______ %
   - Do you have any suggestions on how to improve the design?

4. Conditional and Scanning Mode
   - Would you like to see gateways for all services in the room at all time? 1 - 2 - 3 - 4 - 5
   - Would you like the system to guess you need the service and pop up the gateways for you? 1 - 2 - 3 - 4 - 5
   - How often would you use the
     - Conditional mode: ______ %
     - Scanning mode: ______ %

   In the first part of the study, the gateways popped up under certain conditions.
   - Did you notice that?
   - What would you guess are those conditions in today’s prototype?

   - Which factors would you want the system to take into account in the future when deciding whether to show the gateways?

   - Is understanding these conditions important to you? 1 - 2 - 3 - 4 - 5

5. Comments

6. Personal information
   - Your age □ 21 - 25 □ 26 - 30 □ 31 - 40 □ 41 or older
   - Your sex □ female □ male
   - Your occupation □ student □ other
ASCII  American Standard Code for Information Interchange
API    Application Programming Interface
BCG    Boston Consulting Group
CHI    Conference on Human factors in computing systems
DHCP   Dynamic Host Configuration Protocol
DNS    Domain Name System
DNS SD DNS Service Discovery
EBL    Enhanced Binding Layer
EIS    Embedded Interactive Systems
EJB    Enterprise Java Beans
E2ESD  End-to-End Service Discovery
GUI    Graphical User Interface
GPS    Global Positioning System
HCI    Human Computer Interaction
HTML   Hypertext Markup Language
IDE    Integrated Development Environment
IR     InfraRed
JAR    Java ARchive
JVM    Java Virtual Machine
JSP    Java Server Page
LDAP   Lightweight Directory Access Protocol
**MSI** Mobile Spatial Interaction

**MVC** Model View Controller

**NAPTR** NAPTR Naming Authority Points

**NAT** Network Address Translation

**NFC** Near Field Communication

**OASIS** Organization for the Advancement of Structured Information Standards

**ONS** Object Name Service

**OOP** Object-Oriented Programming

**P2P** Peer-to-Peer

**PARC** Palo Alto Research Center (Xerox)

**PERMID** Pervasive Mobile Interactions Devices (workshop)

**POJO** Plain Old Java Object

**PHP** PHP Hypertext Preprocessor

**RAM** Random Access Memory

**RF** Radio Frequency

**RFID** Radio Frequency IDentification

**SCT** Spatial Context Toolkit

**SDP** Service Discovery Protocol

**SLP** Service Location Protocol

**SOA** Service-Oriented Architecture

**SSDP** Simple Service Discovery Protocol

**TO** Transfer Object

**Ubicomp** Ubiquitous Computing

**UDDI** Universal Description, Discovery and Integration

**UI** User Interface

**UPnP** Universal Plug and Play

**W3C** World Wide Web Consortium

**WSDL** Web Services Description Language

**XML** Extensible Markup Language
WoZ  Wizard of Oz

CSCW  Computer Supported Collaborative Work
CVS  Concurrent Versions System
FTP  File Transfer Protocol
OCL  Object Constraint Language
OCR  Optical Character Recognition
OMG  Object Management Group
D

Licenses

D.1. Documentation

Copyright (c) 2007 Dominique Guinard.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The full-text of the GNU Free Documentation Licence can be read from [11].

D.2. Software

Copyright The RelateGateways project team, Lancaster University (UK) Computing Department, Ubicomp/Eis Group http://ubicomp.lancs.ac.uk/relategateways

The RelateGateways framework is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

The full-text of the GNU LGPL can be read from [11].
A website hosted at Lancaster University, is dedicated to this project: http://ubicomp.lancs.ac.uk/relategateways. On this page you will find:

- The binaries and sources of this documentation.
- The RelateGateways project’s source code and Javadoc API.
- PDF version of the articles, posters and presentations and videos published for this work.

Figure E.1 provides a screenshot of this website.
RelateGateways: An Architecture Enabling Spontaneous Mobile Interactions With Pervasive Services

The RelateGateways Project

In the early vision of ubiquitous computing, mobile users were to be able to spontaneously use services available in their environment (e.g. printers, display, file-servers, etc.). As a matter of fact this time has not come yet. While current discovery mechanisms facilitate network-level connections, it remains quite difficult for users to find available services, configure and consume them. To enable spontaneous mobile interactions with pervasive services we propose the end-to-end discovery model and its implementation: the RelateGateways Architecture.

The User Discovery Layer introduces the concept of visual discovery. In order for the user interface to reflect the users' view of their environment, we will use the Relate ad-hoc sensor network which provides information on where to display small widgets representing services (gateways) on the mobile screen. Once the user found a suitable service, the second layer ensures a network connection between the service and the mobile client is established. Eventually, the Invocation and Interoperability Layer lets the mobile client consume the previously unknown service, using dynamic downloading of mobile code.

Project's Supervision

- Professor at Lancaster University (UK): Hans Gellersen
- Professor at the University of Fribourg (DIUF): Rolf Ingold
- Senior-assistant at the University of Fribourg (DIUF): Denis Lalanne
- MSc student: Dominique Guinard

Figure E.1.: Screenshot of the project’s official website
On the CD-ROM of the project you will find:

- The source code, Ant files and compiled binaries of the project.
- The APIs of the RelateGateways Architecture.
- The produced scientific articles, videos and posters.
- The binaries and sources of this documentation.
- Various documents that were of great use during this Master Thesis.

Most of the CD-ROM’s content can also be downloaded from the official website of the project (see E).
Figure F.1.: The CD-ROM of this project
References


References


Referenced Web Resources


Index

Authentication, 74

Business Architectures, 46

CHI, 5

Clients, 57

Compass, 21

Component Based Architectures, 46

Conditional mode, 21

Context, 29

Contract, 70

Desktop, 22

Directories, 56

Discovery

Classification, 63

End-to-End, 7

Network, 53

Service, 53

Taxonomy, 63

Visual, 18

Discovery Systems, 53

Distributed Computing, 46

Dynamic environments, 30

ebXML, 46

EESD, 7

EJB, 46

Electronic Product Code, 56

End-to-End Service Discovery, 7

Enterprise Pattern, 73

EPC, 56

EPCNetwork, 56

Evaluation, 109

Gateways, 21

Gateways Interaction Technique, 21

GUI, 25

HCI, 2

Human Computer Interactions, 2

Interaction, 22

Invisible Computing, 2

Lookup Service, 56

Mark Weiser, 2

Model, 7

Modes

Conditional, 21

Scanning, 21

MSI, 5

MVC, 27

Naming, 58

Naming Systems, 53

Network Discovery, 53

PERMID, 5

PERVATIVE, 5

Pervasive Computing, 2

Publications, 5

Query, 57

Registries, 56

Relate, 36

Scanning mode, 21

SCT, 29

Service, 47, 54

Service Contract, 70

Service Discovery, 53

Service Subject, 73

ServiceRequest, 73

ServiceResponse, 73

Software Architecture, 44

150
Index

Spatial Context, 29
Spatial Discovery, 18
Spontaneity, 7
Spontaneous Interactions
  Model, 7
Spontaneous Mobile Interactions, 7
Stateful, 70
Stateless, 70
Surrogate, 55

Toolbar, 28
Transfer Object, 73

Ubicomp, 2
Ubiquitous Computing
  Definitions, 2
UI, 24
Universal Requester, 70
User Interface, 21, 24
User Study, 109
UUID, 56

Visual Discovery, 18

Webservices, 46
Weiser, 2
Wizard of Oz, 28
WoZ, 28

Xerox PARC, 2