Spasiba: A Context-Aware Adaptive Mobile Advisor

by

Alexey Rudkovskiy
B.Sc., Thompson Rivers University, 2007

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisory Committee

Spasiba: A Context-Aware Adaptive Mobile Advisor

by

Alexey Rudkovskiy
B.Sc., Thompson Rivers University, 2007

Supervisory Committee

Dr. Hausi A. Müller, (Department of Computer Science)
Supervisor

Dr. Alex Thomo, (Department of Computer Science)
Departmental Member

Abstract

Supervisory Committee
Dr. Hausi A. Müller, (Department of Computer Science)
Supervisor
Dr. Alex Thomo, (Department of Computer Science)
Departmental Member

This thesis presents the design and analysis of Spasiba, a context-aware mobile advisor. We argue that current context-aware mobile applications exhibit significant flaws with respect to (1) limited use of context information, (2) incomplete or irrelevant content generation, and (3) low usability. The proposed model attempts to tackle these limitations by advancing the usage and manipulation of context information, automating the back-end systems in terms of self-management and seamless extensibility, and shifting the logic away from the client side. A distinguishing characteristic of Spasiba is the proactive approach to notifying the user of information of interest. In this proactive approach, the user subscribes to the service and receives content updates as the context changes. This proposed model is realised in a proof-of-concept prototype that uses a Nokia Web Runtime widget as the client application. The widget, which sports an elegant, touch-optimised interface, collects multiple context parameters to deliver high-quality results. The server-side architecture employs the publish/subscribe paradigm for managing the active users and Comet—for proactively notifying the clients of updated information of interest. IRS-III, a Semantic Web Services broker, handles the process of content generation. The prototype employs nine data sources, seven of which are open API web services and two of which are regular web pages, to deliver diverse and complete results. A simple autonomic element, implemented with the help of aspect-oriented programming, ensures partial self-management of the back-end systems. Spasiba is evaluated by means of a case study that involves a tourist couple visiting Victoria. The application assists the tourist couple with finding attractions, relevant stores, and places serving food.

Table of Contents

List of Figures

Acknowledgments

This work would have been impossible without immense assistance from my supervisor Dr. Hausi A. Müller. He has provided the most valuable advice and moral support. Also, I would like to thank all members of the Rigi research group at the University of Victoria for their suggestions and corrections.

In addition, I am endlessly grateful to Alexey Kulakov, Georgy Korablev, and Alexander Elnikov for sharing their ideas and being my friends at all times. In particular, Alexey played a significant role in devising and implementing the case study described in Chapter 8.

Chapter 1 Introduction

This chapter introduces the reader to the work described in this thesis. The recent trends of the Web form the main basis for the motivation. The problem statement elaborates on how the shortcomings of previous and current context-aware mobile applications served as an incentive to undertake this research. Then, the approach description presents a solution that leverages the aforementioned trends to improve the user experience of a context-aware mobile application. Finally, a thesis outline and a brief mention of contributions provide the reader with an idea of how this thesis is organised.

1.1 Motivation

The Web has now existed for almost two decades. We have observed many trends and breathtaking innovations come and change the way we live. The end of the Web's second decade can be marked by a continuing migration to mobile platforms. According to International Telecommunications Union (ITU),[1] there are currently over 4 billion active cellular subscriptions in the world. Another statistic, now from Gartner,[2] informs us that in 2008 there were 140 million smartphones sold; in 2009, despite the alleged financial crisis, this number is likely to exceed 160 million. Smartphones are mobile multimedia computers that have increasingly rich functionality. Their key features are powerful hardware, advanced connectivity, multiple input methods, and many sensors. Modern smartphones often sport high-resolution touch screens, gigabytes of memory, built-in Global Positioning System (GPS) receivers, Wi-Fi and 3G connectivity. Current market leaders—Nokia, RIM, and Apple—have opened their smartphone platforms enabling developers to create third-party applications that harness all of the device's capabilities. The context information collected from the user's device unveils unprecedented levels of software intelligence and, thus, user satisfaction. Location information, battery charge, available connections, roaming information, and many other sensors enable mobile applications to generate the most relevant results in the most efficient way.

[1] http://www.itu.int/ITU-D/icteye/Reporting/ ShowReportFrame.aspx (retrieved 08/24/2009).

[2] http://www.gartner.com/it/page.jsp?id=910112 (retrieved 08/28/2009).

In parallel, the Web is undergoing a major architectural change with the adoption of web services and semantics. Both are designed to support interoperable machine-to-machine interaction (T. Berners-Lee, Hendler, & Lassila, 2001; Booth et al., 2004). With the use of web services, an application can publish its function or message to the rest of the world in an open, standardized mode. In the modern Web, web services are often represented by an Application Programming Interface (API), presenting an opportunity to easily integrate data from disparate sources in a dynamic mode. Semantics, as in the Semantic Web, enrich the content with meaning and form the Web of Data as opposed to today's (and yesterday's) Web of Documents. Semantically enriched data can be queried in a precise and efficient manner, generating only the most relevant results. A remarkable paradigm that integrates semantics into web services is called Semantic Web Services (SWS). Such web services exhibit enhanced discovery and composition characteristics (Cabral, Domingue, Motta, Payne, & Hakimpour, 2004). These characteristics are prevalent in the fluid system architecture that is required to accommodate emerging business needs.

The synergy of contextual information available on smartphones and semantically enriched web services presents an opportunity for pleasurable user *and* developer experience. Users receive an intelligent, adaptive mobile application. Developers enjoy an easy to expand and maintain software system.

1.2 Problem Statement

In my opinion, modern context-aware applications for smartphones present three key opportunities for improvement: (1) limited use of the context information, (2) irrelevant or incomplete content generation, and (3) low usability.

As can be observed in numerous mobile applications, location information and date are the two most exploited context parameters leveraged in content generation. There is definitely more to context than that—this is the main sentiment in (Wright, 2009) and (A. Schmidt, Beigl, & Gellersen, 1999). Consider battery life, network information (e.g., is the user roaming?), locale settings (e.g., language and time zone), available connections (e.g., EDGE, 3G, Wi-Fi), screen orientation, and others. My proposition is that every one of those parameters can be extremely useful in content generation. Some of these

parameters may only affect the presentation of results, while others serve as the very basis for generating the results. The former can be represented by battery life, data connections, and screen orientation, while the latter—by network information, locale settings, and, of course, location and date. What if we could employ at least some of these sensors together in a mobile application?

Irrelevant or incomplete content generation is really frustrating for anyone who needs to obtain information quickly and accurately on a mobile device. On "big" computers users have the opportunity to really search the Web and select the options they like the most. On a mobile device, we the developers, have to care about the user more. We have to make sure the user receives only the most relevant content in a blink of an eye in the most usable format. There are plenty of context-aware mobile applications. What is their foremost limitation? The main problem is, what I would call, their restricted area of service. Even such great applications as Google Maps Mobile[3] or Yelp Mobile[4] fail in areas where their data sources have no data. Such areas are mostly countries where English is not the first language and where the Web culture is such that content is unavailable to Google or any popular English-oriented data collector. As an example, consider Facebook,[5] which is enormously popular in the United States, the United Kingdom, and Canada, but almost nobody has heard of it in Brazil or Russia, where Orkut[5] and VKontakte[5] rule the social network realm (Cosenza, 2009). While Orkut was acquired by Google some time ago and thus can probably be easily accessible, VKontakte forbids robots indexing its content. The importance of social networks as content providers can hardly be overestimated, since they are the few resources that have up-to-date information about physical locations. Therefore the question is how can we create a mobile application that can be easily extendable to accommodate new, independent content sources?

Low usability is another cause of frustration for users of mobile applications. Some applications are not well-suited for touch interactions, some have connectivity issues,

[3] http://www.google.com/mobile/products/maps.html (retrieved 08/24/2009).

[4] http://www.yelp.com/yelpmobile (retrieved 08/24/2009).

[5] http://www.{facebook,orkut,vkontakte}.com (retrieved 08/24/2009).

others overload the Random Access Memory (RAM) and force themselves to quit. What if we write a simplistic client application with a minimum user interface?

This work sets out to provide a working prototype of an application that aims to partially solve the aforementioned problems of modern context-aware mobile applications.

1.3 Approach

The objective of this research was to formulate a model for an adaptive, user-oriented, context-aware mobile application that would have the following idiosyncratic features:

1) Employs the full power of context information to enhance presentation and quality of generated content.
2) Ensures extensibility and autonomicity of back-end architecture in order to accommodate seamless addition of new, heterogeneous data sources.
3) Enhances the usability by sheer simplicity. The client application is represented by a thin client that acts as a terminal and context collector.

I realised such a model in a prototype called Spasiba[6] for Nokia S60 5th Edition platform. Spasiba is an adaptive mobile advisor for exploring a city. It delivers dining, shopping, and sightseeing suggestions based on the user's contextual information. What are some of the key features that make Spasiba stand out and achieve the proposed requirements?

- *Lightweight client application with an extremely simple and clean user interface.* The client app is represented by a Nokia widget that consumes little resources, while still providing the user with immense capabilities. The user interface consists of an input field, content area, and a submit button—as simple as it gets.
  Client application collects seven contextual parameters and submits them in a Spasiba envelope along with the user's request. The seven parameters are location, date and time, locale, battery status, available data connections, network status (the user is in home network or roaming), and model of device. In addition, the client application also collects certain information in real time: location updates and screen orientation. All of those parameters are used to improve the

[6] Spasiba is a Russian word (*Спасибо*) that means "Thank you". This name attributes to the background of the author of this thesis and the emotion the user should feel when the application truly does its thing.

application's intelligence and, thus, the user's experience. This prototype's user interface is demonstrated in a screenshot below.

Figure 1. Demonstration of Spasiba's user interface

- *Dynamically generated filters that allow the user to narrow down her request.* A user's request is categorised using a number of ontologies. Then, in accordance with the selected category, a filter is created and loaded into the client application.
- *Database-less server application with a Semantic Web Services composition engine.* Having received an envelope from a client application, the system assesses available context parameters as well as the user's request and matches them against a number of preregistered web services that have been semantically annotated. These services are, then, composed and queried to generate results for the user. This server application employs Internet Reasoning Service III (IRS-III) (Cabral et al., 2006) for brokerage and composition of web services.

- *Data sources represented by open API web services and regular web pages.* Regular web pages can be converted into a web service by means of an adapter tailored specifically to them. The addition of new data sources is relatively simple, because all that is needed is to provide a semantic annotation for the web service.

- *Multiple feedback loops embedded into both client and server components of Spasiba.* These feedback loops enhance the intelligence of the application, providing a degree of autonomicity. Client application uses feedback loops to monitor output from sensors. An autonomic manager on the server side enforces policies and handles exceptions. Some policies deal with the presentation of content on the user's device. For instance, if battery is low, optimize the map's size or do not load it at all—depending on user's preferences. Others participate in content generation. For example, if location coordinates change beyond a certain threshold, start the services composition engine to seek new results.

- *Proactive approach to notifying the user of places of interest.* Spasiba can function in two modes: normal and dynamic. In normal mode, the user submits a request and then receives results. In dynamic mode, the user submits a request, selects the dynamic mode option, and then receives results dynamically as location and time change. This is achieved with the help of Comet (Crane & Mccarthy, 2008) architecture.

1.4 Contributions

The contributions can be divided into two categories: developer's experience and application model. Developer's experience of designing and implementing the Spasiba application is documented in this thesis. In particular, a tutorial for students who would like to create a similar application is presented in Chapter 9. The application model is discussed in detail in Chapter 3. This model is comprised of a conceptual foundation and a system architecture.

1.5 Thesis Outline

The first two chapters form the foundation for the rest of the thesis. Chapter 2 introduces background and touches on smartphones, context and context-awareness, service-oriented architecture (SOA) and web services, the Semantic Web and Semantic Web Services, and self-adaptive systems and autonomic computing.

Chapters 3-7 present the implementation of Spasiba application. Chapter 3 introduces the application model with detailed description of each component. Chapter 4 discusses the client application with special emphasis on context collection, communication techniques, and adaptive functionality. Chapter 5 is concerned with control of flow and dynamism—it describes the static and dynamic behaviours of Spasiba. Chapter 6 discusses how content is generated and refined; attention is drawn toward data sources and brokerage of web services. In Chapter 7 the self-adaptive features of Spasiba are presented in terms of autonomic policies and error-handling.

The work is evaluated in a case study introduced in Chapter 8. Chapter 9 presents a tutorial for computer science students that wish to create an application similar to Spasiba.

Chapter 10 concludes this thesis with a summary and an outlook. A glossary of used abbreviations is given in Appendix A.

Chapter 2 Background

This chapter introduces the reader to the most prominent concepts and technologies employed in this project. The discussion begins with an overview of smartphones and, in particular, the Symbian operating system. Then, the emphasis moves to context and context-awareness. This is followed by a brief discussion of web services and the service-oriented architecture (SOA). Then, Semantic Web and Semantic Web Services (SWS) along with relevant technologies are described. Next, the reader's attention is drawn to Self-Adaptive Systems and Autonomic Computing. The chapter concludes with a summary.

2.1 Smartphones

Encyclopaedia Britannica (Hosch, 2008) defines a smartphone as a "mobile telephone with a display screen (typically a liquid crystal display, or LCD), built-in personal management programs, such as an electronic calendar and address book, and an operating system (OS) that allows other computer software to be installed. Smartphones may be thought of as a merger between the traditional telephone and a PDA (personal digital assistant)." In basic terms, a smartphone is a mobile phone with advanced capabilities that offers PC-like functionality. The pioneering smartphones were Simon[1] by IBM and BellSouth and the Nokia 9000[2] released in 1993 and 1996, respectively. At that time the devices looked and weighed like bricks and were targeted at a very narrow niche of customers, mostly early adopters and business executives. Now smartphones offer rich capabilities, are affordable and increasingly popular. According to a statistical projection from Gartner,[3] in 2009 there will be 160 millions of smartphones sold. Intuitive touch interfaces coupled with convenient ways of distributing third-party applications allow the new generation of smartphones to conquer the hearts and minds of consumers. On the next page are some of the capabilities that modern smartphones offer, based on the devices released in 2009.[4]

---

[1] http://en.wikipedia.org/wiki/Simon_(phone) (retrieved 08/24/2009).

[2] http://en.wikipedia.org/wiki/Nokia_9000_Communicator (retrieved 08/24/2009).

[3] http://www.gartner.com/it/page.jsp?id=910112 (retrieved 08/28/2009).

[4] Nokia N97, iPhone 3GS, Palm Pre, Samsung i7500, Samsung Omnia II, HTC Hero, HTC Magic

Common capabilities of modern smartphones:

- Telephone
- Messaging (SMS and MMS)
- Data connectivity: GPRS, EDGE, 3G/3.5G, Wi-Fi, Bluetooth
- GPS/A-GPS
- CPU at 400 MHz+, RAM of 128 MB+
- Touch or multi-touch screen with 3 inches+, 24 bit colours+, 480 x 320 pixels+
- Photo/video camera
- Multiple sensors such as accelerometer, compass, and ambient light sensor
- Advanced operating system that often includes multitasking, support for third-party applications, multimedia and Internet capabilities

The smartphone market overview performed by Canalys in 2009 (Canalys, 2009) and depicted in Figure 2 illustrates that Symbian[5] is the most popular mobile operating system in the world. Other widely used platforms include RIM Blackberry,[6] Windows Mobile,[7] iPhone OS,[8] and a number of Linux-based operating systems such as Android,[9] WebOS, Maemo, and, recently, Moblin.

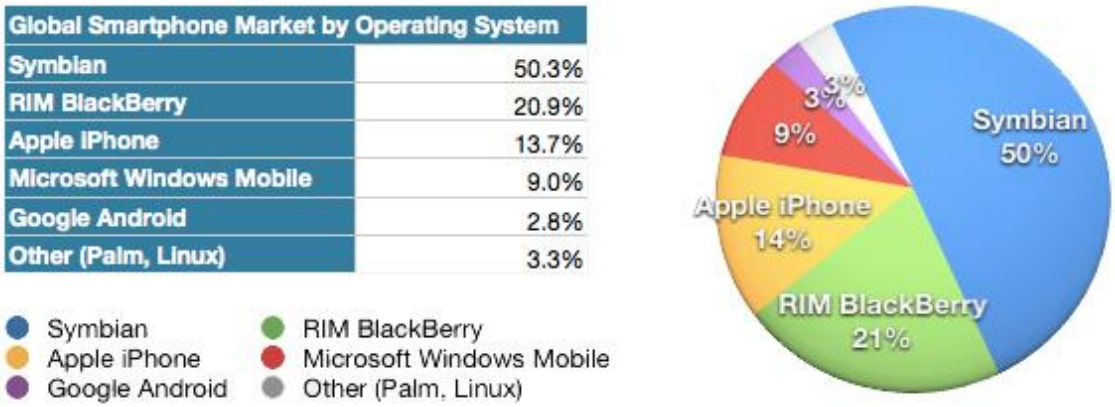

Figure 2. Global smartphone market by operating system (Canalys, 2009)

[5] http://developer.symbian.org (retrieved 09/28/2009).

[6] http://na.blackberry.com/eng/developers (retrieved 09/28/2009).

[7] http://developer.windowsphone.com (retrieved 09/28/2009).

[8] http://developer.apple.com/iphone (retrieved 09/28/2009).

[9] http://developer.android.com (retrieved 09/28/2009).

Thanks to Symbian's dominating position in the industry and my intimate personal ties with Nokia's products, I attempted to develop an application for the latest generation of Nokia smartphones. These smartphones are based on Symbian 9.4. Some of the released devices that support Spasiba are Nokia N97, Nokia 5800, and Nokia X6.

2.1.1 Symbian and S60

Symbian is an operating system designed for mobile devices and smartphones, with associated libraries, user interface, frameworks and reference implementations of common tools, developed by Symbian Ltd. It was a descendant of Psion's EPOC and runs exclusively on ARM processors. In 2008 a new, independent non-profit organization called the Symbian Foundation was established and the former Symbian Software Limited was acquired by Nokia (Nokia Press Release, 2008). Symbian OS has an impressive range of technology features—even in its earliest versions. Its memory-management and multitasking features allow for safe and efficient operation under conditions of constrained resources that typify mobile devices. Nokia's platform built upon Symbian is called S60; a figure illustrating its architecture is given below.

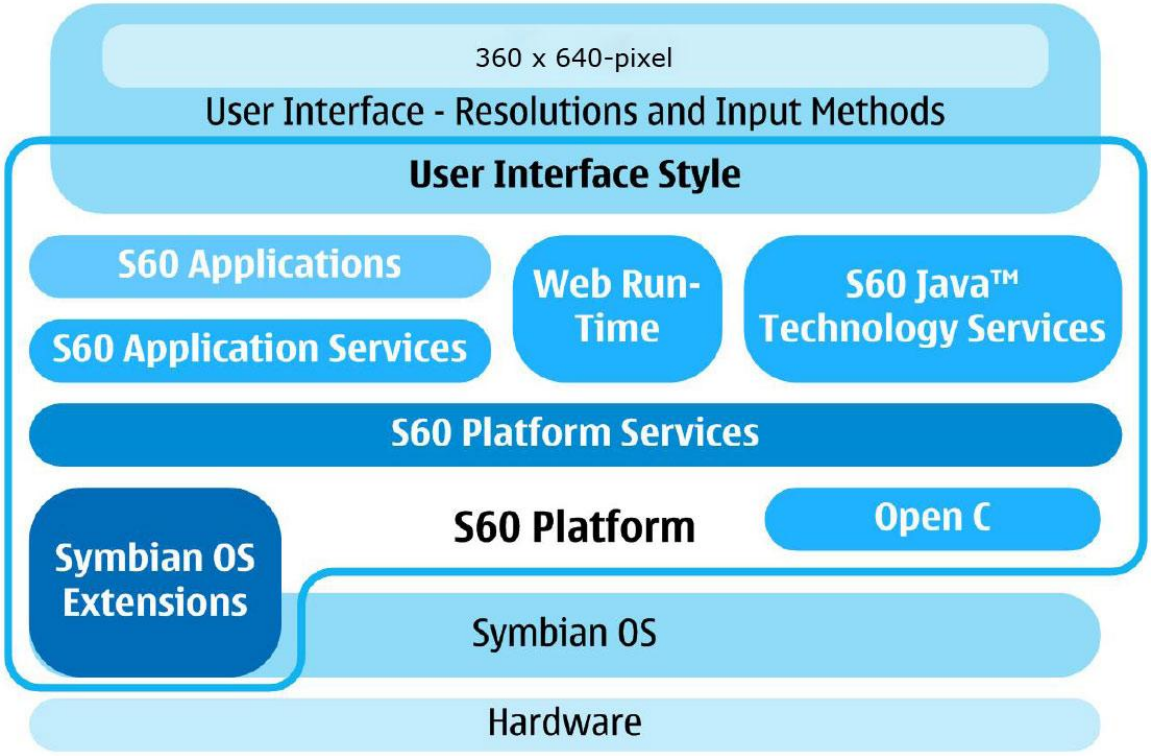

Figure 3. A schematic diagram of the S60 platform architecture (Nokia, 2008b)

Let us briefly discuss each component of the S60 platform architecture:

- Symbian OS Extensions are a set of capabilities that allow the S60 platform to interact with device hardware functions such as vibration alert, device lights, and battery charge status.
- Open C is an extension of the POSIX libraries for Symbian OS. It provides a subset of functions from nine well-known standard POSIX and middleware C libraries. Open C enables developers to port middleware and application engines from a desktop environment to the S60 platform.
- S60 Platform Services are the fundamental services offered by the S60 platform. These include the Application Framework Services, UI Framework Services, Graphics Services, Location Services, Web-Based Services, Multimedia Services, and Communication Services.
- S60 Application Services are a set of capabilities that provide certain basic functionality for S60 applications. These services are used by the embedded S60 Applications and also are available for use in third-party applications. They include Personal Information Management (PIM) Application Services, Messaging Application Services, and Browser Application Services.
- S60 Java Technology Services provide support for Java Micro Edition (ME). Further, the S60 Java Technology Services support a range of additional APIs to enable access to the S60 file system, access to PIM data, use of Bluetooth technology, messaging, audio, video, web services, security and trust services, location information, Session Initiation Protocol (SIP), and 3D graphics.
- Web Run-Time (WRT) is a runtime environment that enables S60 devices to run web widgets. Introduced in S60 3rd Edition, Feature Pack 2, WRT is powered by technology from the WebKit Open Source Project[10]—the same technology that is used by the Web Browser for S60.
- S60 Applications are applications embedded within the platform that are available to a device's user, including PIM, messaging, media applications, and profiles.

[10] http://webkit.org (retrieved 09/28/2009).

- S60 platform specification includes a UI style and UI libraries, thereby promoting a consistent look and feel; however, the specification does not mandate a particular UI.

From the developer's point of view, S60 presents a number of options for the development environment. Here I would like to emphasise the five predominant ones:

C++ Symbian OS was built using C++, and this development language provides the fullest access to the feature-rich S60 platform. All versions of the S60 platform support C++ development, and various integrated development environments (IDEs) are available for each version.

Open C Open C is an extension of the POSIX libraries for Symbian OS. The implementation of Open C allows developers to reuse software assets and thereby increase productivity.

Java ME As with C++ development, Java development for the S60 platform requires a suitable development environment and a suitable Software Development Kit (SDK). Nokia provides SDKs that support the development of MIDlets using NetBeans with NetBeans Mobility Pack and Eclipse with EclipseME.

Web Widgets Web widgets are lightweight applications created using the standard web technologies that are used to create web pages, such as HTML, Cascading Style Sheets (CSS), JavaScript, and Asynchronous JavaScript and XML (AJAX). S60 device users can download and install web widgets as they would any other S60 application or content item. Once installed, web widgets appear as standard S60 applications and provide S60 device users with a full web experience, in a way that allows them to personalise the content and services they access. Spasiba was implemented using this technology.

Flash Lite S60 platform is the reference platform for Flash Lite. Development is supported by Flash CS 3 Professional.

To summarise, Spasiba was implemented as a widget application using the Nokia WRT and, thus, is supported on all Nokia S60 5th Edition smartphones. What are the special features of these smartphones and how do they look like?

Figure 4. Left to right: Nokia 5800, Nokia N97, and Nokia X6

S60 5th Edition is built on Symbian OS 9.4. The features worth of mention are listed below, in accordance with (Nokia, 2008a):

- Improvements to demand paging so that this feature is used for all items in a device's internal memory. This offers faster device boot-up and application start-up times, and reduces the likelihood of out-of-memory situations.
- UI has been updated to support 640 x 360-pixel screens.
- Touch screen capability with tactile feedback through the device's vibrate function. This enables device users to interact with their devices in a more natural and expressive way. To support touch interaction, S60 5th Edition delivers on-screen full keyboards, as well as handwriting-recognition input.
- A new sensor framework allows for the inclusion of accelerometers, magnetometers, and tap sensors. These sensors enable the creation of applications that respond to a device's motion and orientation.

- WRT has been integrated with S60 Platform Services in S60 5th Edition. This means that widgets can access device data and information through new JavaScript extensions. This feature is instrumental to Spasiba's existence, as it enables such Platform Services as device location, system information, and sensors to be accessed in a widget. Nokia believes that "together, the UI and platform-access enhancements enable web developers to create a completely new breed of context-aware applications and services that provide users with information unique to their own experiences." (Nokia, 2008a)

2.2 Context-Awareness in Ubiquitous Computing

Ubiquitous computing is a post-desktop model of human-computer interaction in which information processing has been thoroughly integrated into everyday objects and activities (M. Weiser, 1991; M. Weiser, 1993). Ubiquitous Computing is also known as ubicomp, pervasive computing, and everyware. Smartphones are one of the most illustrative examples of ubicomp. Nokia researcher Jan Chipchase's investigation into the ways we interact with technology has led him from the villages of Uganda to the insides of our pockets (Blom, Chipchase, & Lehikoinen, 2005). He made some unexpected discoveries along the way. Consider this excerpt from his talk[11] at a TED conference:

> *"So I've probably done about five years research looking at what people carry. I go in people's bags. I look in people's pockets, purses. I go in their homes, and we do this worldwide, and we follow them around town with video cameras. It's kind of like stalking with permission. And we do all this—and to go back to the original question: What do people carry?*
>
> *And it turns out that people carry a lot of stuff, OK. That's fair enough. But if you ask people what the three most important things that they carry are—across cultures and across gender and across contexts—most people will say keys, money, and if they own one, a mobile phone."*

Mobile phones and, in particular, smartphones are vital to a modern person's existence. They allow us to delegate many everyday tasks to intelligent software agents. However,

[11] http://www.ted.com/talks/jan_chipchase_on_our_mobile_phones.html (retrieved 09/28/2009).

our interaction with these agents remains inefficient. Context-awareness, or more specifically how to create applications that are context-aware, is a central issue to research in ubicomp.

Context and, in particular, the concept of context-awareness in mobile applications have recently become a hot topic in the ubicomp community. While there are a few definitions of context—the classical one is given below, one may simply define context as a state a given system is in at a particular moment. The concept of a context-aware computing system was first introduced by Schilit in 1994 (Schilit, Adams, & Want, 1994) as a system that "adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time." Unfortunately, this concept did not catch on until the beginning of the new millennium. Thanks to Anind Dey, Carnegie Mellon University and Albrecht Schmidt, University of Bonn, who devoted their PhD dissertations ((A. K. Dey, 2000) and (A. Schmidt, 2003), respectively) and numerous publications to context-awareness, the novel concept received a strong theoretical basis and thus an impulse to develop. In 2000, Anind Dey proposed the following definitions of context and context-awareness that quickly became widely accepted. Both are found in (A. K. Dey, 2001).

Context

"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

Context-awareness

"*A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."*

In recent years, context-aware systems have been investigated in a number of fields. Some of these fields are ubicomp (Baldauf, Dustdar, & Rosenberg, 2007; A. K. Dey, 2001; A. Schmidt et al., 1999), sensor networks (may be regarded as ubicomp) (A. K. Dey, 2000), web services (Fujii & Suda, 2009; Kapitsaki, Prezerakos, Tselikas, & Venieris, 2009; Keidl & Kemper, 2004), computer-supported collaborative work

(CSCW) (Abowd et al., 1999), and human-computer interaction (HCI) (Hong & Landay, 2001; A. Schmidt, 2003).

Generally, context-aware systems are concerned with the acquisition of context, the abstraction and understanding of context, and application behaviour based on the recognized context. What does it mean for a software engineer? To start with, context must be modelled. Of course, it is infeasible to capture all context parameters present in a system—this may be due to equipment limitations or specific use case requirements. Thus, such a model must be devised that deals with a subset of a given system. Then, one would consider the ways of acquiring and harnessing the context.

In context acquisition, the selection of context parameters is crucial. Context parameters, which are basically units of context information, may be characterised by source (where a parameter originates from), classification (what universe it belongs to), relationships (ties with other entities), and lifetime (when a parameter—or its value—becomes obsolete). Unfortunately, there is no widely accepted taxonomy for classifying or organizing context parameters. A context parameter may be organised within a variety of dimensions—for instance, static versus dynamic. The selected context parameters can be (1) extracted from the environment with the help of sensors, (2) inferred from user preferences or user input, (3) derived from a web service.

Once the context information is acquired, the context processing occurs. At this point the context parameters are filtered, composed, and reasoned upon. A number of context management paradigms are available: model-driven approach, message-oriented approach, or control-oriented approach (such as described in (Mejias, B. and Vallejos, J., 2007)). Spasiba employs a mixed architecture for leveraging context, where a control-oriented approach is coupled with a model-driven approach. The control-oriented approach represented by feedback loops provides a practical abstraction for managing individual context information units. A macro model governs sets of context parameters and ensures the satisfaction of global context-related policies.

Context-awareness is regarded as an enabling technology for ubiquitous computing systems. In particular, context-awareness has proved to be a handy paradigm in the realm of smartphone applications. Location-based services and augmented reality applications

are some of the popular examples. Particularly interesting design guidelines and models for developing context-aware mobile applications are proposed in (Froehlich, Chen, Consolvo, Harrison, & Landay, 2007; Hakkila & Mantyjarvi, 2006; Verkasalo, 2009). As can be observed in numerous mobile applications, location information and date and time are the two most exploited context parameters leveraged in content generation. There is definitely more to context than that—this is the main sentiment in (Wright, 2009) and (A. Schmidt et al., 1999). In fact, modern smartphones are capable of collecting dozens of context parameters both from the environment and from the user. Spasiba, the prototype described in this document, collects eight environment parameters and a variable number of user parameters—such as preferences and immediate interest.

2.3 Service-Oriented Architecture and Web Services

A web service is a software system designed to support interoperable machine-to-machine interaction over a network. With the means of a web service an application can publish its function over the web in a standard way—and then this function can easily be consumed by a disparate, decoupled client. Generally speaking, web services can be broken down into two categories: SOAP-based and RESTful.

2.3.1 SOAP-based Web Services

SOAP-based web services adhere to a number of standards and that is why they are popular with traditional enterprises. Most often these web services provide an interface described in a machine-processable format called Web Services Description Language (WSDL) (Curbera et al., 2002). Other systems interact with the web service in a manner prescribed by its description using Simple Object Access Protocol (SOAP) messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards (Booth et al., 2004). Thanks to its simplicity and extensibility, SOAP provided an attractive alternative to traditional proprietary protocols, such as CORBA and DCOM. Web services are catalogued in a registry known as Universal Description Discovery and Integration (UDDI). Figure 5 illustrates how one may conceptually map the aforementioned web service paradigms to those of the classical Web.

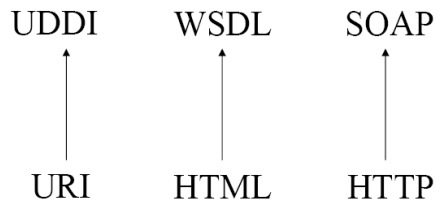

Figure 5. Mapping of paradigms: Web Services and the Web

To sum up, SOAP is used for transport, WSDL—for service description, UDDI—for locating a service, and, of course, XML is the glue that holds all of those together. These technologies along with WS-I Basic Profile[12] characterise the first generation of web services (Erl, 2009). As one may notice, this first generation of web services exhibits a number of gaps in the realm of Quality of Service (QoS) (Wang, Huang, Qu, & Xie, 2004).

The second generation of web services attempts to close these gaps and implement a number of other useful extensions. Numerous specifications, generally listed under the umbrella of WS-*, build upon the fundamental first-generation messaging framework (Erl, 2009). These specifications provide a rich set of features, but they also impose a high degree of complexity in both technology and design—thus their implementation is a tremendous challenge. Some of the notable WS-* extensions include: WS-Addressing, WS-Reliable Messaging, WS-Security, WS-Coordination, and WS-Choreography. Moreover, special emphasis is placed on service management that is supported by such specifications as WS-Resource Framework and WS-Distributed Management. Many of WS-* and management extensions are in early stages of acceptance.

2.3.2 RESTful Web Services

RESTful web services build upon the Representational State Transfer (REST) architecture for distributed hypermedia systems, proposed by Roy Fielding in his doctoral dissertation in 2000 (Fielding, 2000). In contrast to SOAP-based web services, there is no accepted standard for RESTful web services. This is because REST is a type of architecture, unlike SOAP, which is a protocol. Nevertheless, most RESTful implementations of web services employ HTTP for transport, URI for addressing, and a number of standards for messaging—such as XML and JSON.

[12] http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html (retrieved 10/21/2009).

A RESTful web service is often a simple web service implemented using HTTP and the principles of REST. Such a web service exposes a collection of resources that can be accessed and modified using the regular HTTP methods: POST, GET, PUT, or DELETE. Thanks to a better integration (as compared to SOAP) with HTTP and web browsers, RESTful web services have recently been regaining popularity among Internet companies (Richardson & Ruby, 2007). Such a phenomenon as Web API, enabled by RESTful web services, allows multiple web services to be combined into a new application known as mashup (Benslimane, Dustdar, & Sheth, 2008). This phenomenon has been regarded as one of the pillars pertinent to Web 2.0 (O'Reilly, 2007).

2.3.3 Service-Oriented Architecture

As business needs evolve, the IT infrastructure should adjust accordingly in a timely and cost-effective manner. The industry has gone through numerous computing architectures designed to allow fully distributed processing, programming languages designed to run on any platform, and a multitude of connectivity products designed to allow better and faster integration of applications (Arsanjani, 2004). Service-oriented architecture (SOA) is being promoted in the industry as the next evolutionary step in software architecture to help IT organizations meet their complex set of challenges. SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains (Erl, 2009). It provides a uniform means to offer, discover, interact with, and use capabilities to produce desired effects consistent with measurable preconditions and expectations (Josuttis, 2007). By following SOA, a computer system or software is componentized as a service. One of the most important points of SOA is interoperability; the services should be accessed from other services easily. SOA is still an emerging approach and it lacks concrete implementations.

SOA implementations rely on a mesh of software services. Services comprise unassociated, loosely coupled units of functionality that have no calls to each other embedded in them. Each service implements one action, such as viewing an online bank-statement or placing an airline ticket order. Instead of services embedding calls to each other in their source code, they use defined protocols that describe how services pass and parse messages, using description metadata. SOA developers associate individual SOA

objects by using orchestration. In the process of orchestration the developer associates software functionality (the services) in a non-hierarchical arrangement.

According to (Balzer, 2004), the following guiding principles define the ground rules for development, maintenance, and usage of the SOA:

- Reuse, granularity, composability, componentisation and interoperability
- Standards-compliance (both common and industry-specific)
- Services identification and categorization, provisioning and delivery, and monitoring and tracking

IBM SOA Foundation

IBM has been actively promoting SOA by generating and disseminating knowledge about the paradigm within the IT and business communities. In particular, IBM provides a number of guidelines towards successful implementation and deployment of a SOA-driven infrastructure in an enterprise environment. One of the key documents is the SOA Reference Architecture (Arsanjani, 2004) that substantiates the highlights of service-oriented modeling and architecture. Based on the input from clients, IBM compiled a representation of the SOA lifecycle.

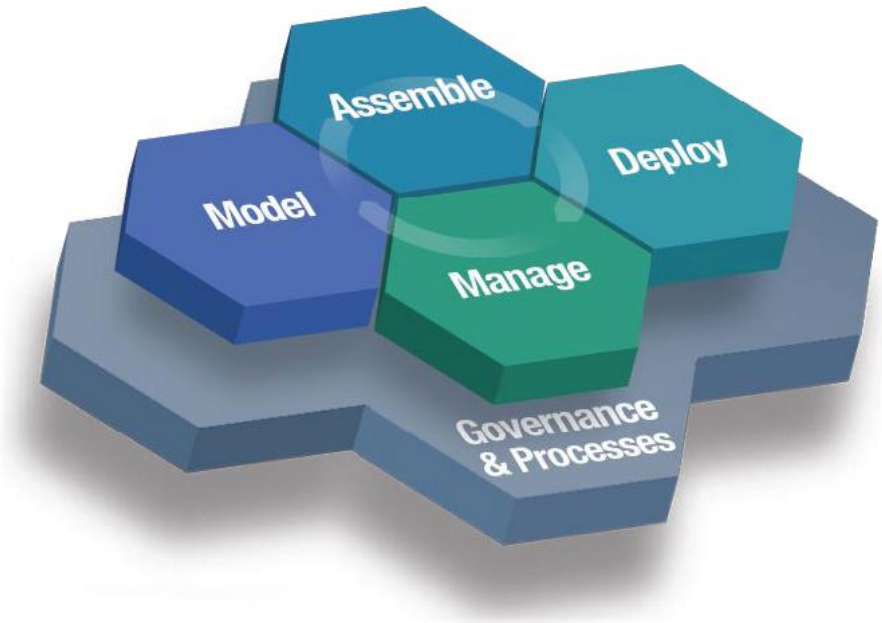

Figure 6. The SOA Lifecycle (IBM Corporation, 2005)

As Figure 6 illustrates, the proposed SOA lifecycle consists of four stages—Model, Assemble, Deploy, and Manage—and the underpinning—Governance & Processes—that provides guidance and oversight. This is how the flow of this lifecycle proceeds, according to (IBM Corporation, 2005):

1. Start in the model phase by gathering business requirements and designing their business processes.
2. After processes are optimized, implement them by assembling new and existing services to form these business processes.
3. Then deploy these assets into a highly secure and integrated services environment.
4. After the business processes are deployed, manage and monitor these business processes from both an IT and a business perspective.
5. Information gathered during the manage phase is fed back into the life cycle to enable continuous process improvement.
6. Overarching all of these life-cycle stages are governance and processes that provide guidance and oversight for the SOA project.

2.4 Semantic Web and Semantic Web Services

In the late 1990s, Tim Berners-Lee, the creator of the Web, substantiated his vision of the Web as a universal medium for data, information, and knowledge exchange (T. Berners-Lee, 1998; T. Berners-Lee, 2000). The Web has now existed for two decades. Over this period of time the medium has miraculously transformed from simple static web pages to sophisticated and highly-dynamic web applications. What yet appears to be a cause of frustration is the integration among disparate resources. Massive amounts of data are being generated by the social Web phenomenon. All of this data is practically useless unless there is a way to integrate it (Figure 7 comically illustrates it). The intention behind the Semantic Web is to facilitate this integration—of all the data available on the Web—by means of semantics (T. Berners-Lee et al., 2001). The semantics enrich an entity with meaning that can be comprehended by machines and human beings alike.

The Semantic Web links data by associating one idea to another. This way each item on the Web is supported by the entire Web and, thus, being resolved at the scope of the

entire Web (T. Berners-Lee, 2009). Hence, the Semantic Web is practically a database of things or knowledge; one may also call it the Web of Data. The notion of Linked Data is used to describe a method of exposing, sharing, and connecting data via dereferenceable URIs on the Web. Tim Berners-Lee outlined four principles of Linked Data in (T. Berners-Lee, 2009) as follows:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names
3. When someone looks up a URI, provide useful information, using the standards
4. Include links to other URIs, so that more things can be discovered

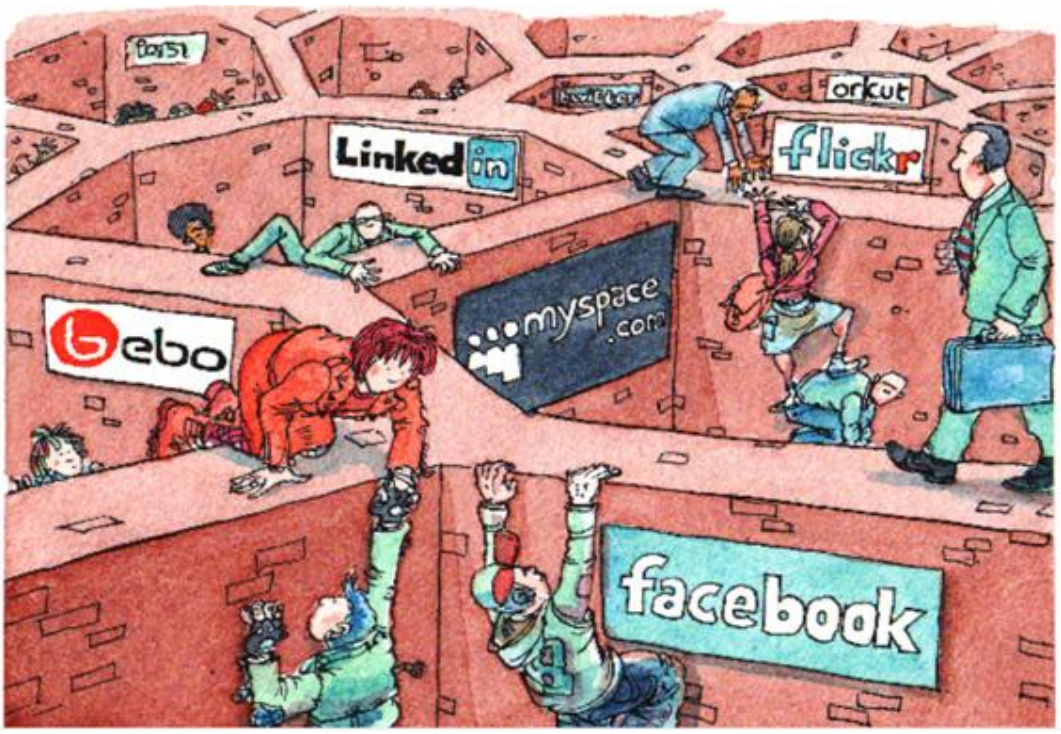

Figure 7. Integration challenge in the social Web. Courtesy of David Simonds, *The Economist*

At its core, the Semantic Web is comprised of a set of design principles, collaborative working groups, and a variety of enabling technologies. Some elements of the Semantic Web are expressed as prospective future possibilities that are yet to be implemented or realised. Others are expressed in formal specifications. The architecture of the Semantic Web is depicted in Figure 8—this is the so-called Semantic Web Stack or Layer Cake (Horrocks, Parsia, Patel-Schneider, & Hendler, 2005).

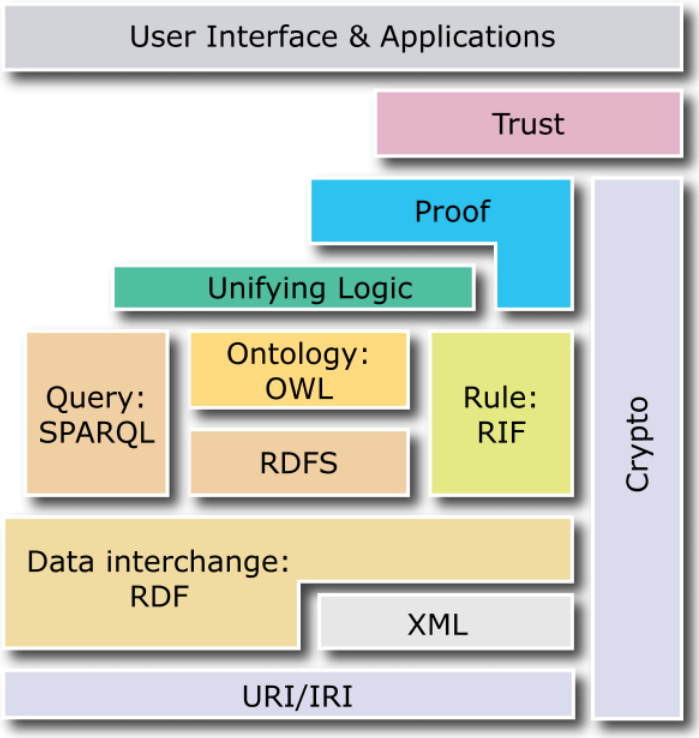

Figure 8. The Semantic Web Stack, also known as the Layer Cake (Horrocks et al., 2005)

From this stack, one can emphasise the key enabling technologies that are specific to the Semantic Web. Those are RDF, RDFS, RIF, OWL, and SPARQL. Let us briefly discuss what they are. Note that all of them, except for RIF, have been standardised.

- Resource Description Framework (RDF) is a simple language for expressing data models, which refer to objects ("resources") and their relationships. An RDF-based model can be represented in XML syntax.
- RDF Schema is a vocabulary for describing properties and classes of RDF-based resources, with semantics for generalized-hierarchies of such properties and classes.
- Web Ontology Language (OWL) adds more vocabulary for describing properties and classes: among others, relations between classes, cardinality, equality, richer typing of properties, characteristics of properties, and enumerated classes.
- SPARQL Protocol and RDF Query Language (SPARQL) is a protocol and query language for Semantic Web data sources.
- Rule Interchange Format (RIF) is the Rule Layer of the Semantic Web Stack.

The Semantic Web has been mainly supported and promoted by the World Wide Web Consortium (W3C). Specifically, the Linking Open Data Community Project[13] is of special interest. The project's goal is to extend the Web with a data commons by publishing various open datasets as RDFs on the Web and by setting RDF links between data items from different data sources. As of May 2009, datasets consisted of 4.2 billion RDF triples, interlinked by around 142 million RDF links (Heath, 2009).

2.4.1 Semantic Web Services

Semantic Web Services (SWS) are a relatively new research paradigm that can be generally defined (Payne & Lassila, 2004) as "the augmentation of Web Service descriptions through Semantic Web annotations, to facilitate the higher automation of service discovery, composition, invocation, and monitoring in an open, unregulated, and often chaotic environment (i.e., the Web)." SWS are to deliver an improvement over the regular web services by enabling access to web resources by content rather than keywords. With the help of SWS, one can specify the meaning of and semantic constraints on the data.

Various approaches have been investigated to enrich web services with semantics. Some of these attempt to provide an upgrade to a single component of the existing web services stack; others promote a complete solution with multiple components (Cabral et al., 2004). The table below provides a brief overview of the more prominent approaches to SWS.

WSMO[14] Web Service Modeling Ontology (WSMO) is a conceptual model for relevant aspects related to Semantic Web Services. It provides an ontology based framework, which supports the deployment and interoperability of Semantic Web Services.

The WSMO has four main components (Roman et al., 2005):

- *Goals*: The client's objectives when consulting a Web Service.
- *Ontologies*: A formal Semantic description of the information used by all other components.

[13] http://linkeddata.org (retrieved 08/14/2009).

[14] http://www.wsmo.org (retrieved 08/14/2009).

- *Mediators*: Connectors between components with mediation facilities. Provides interoperability between different ontologies.
- *WebServices*: Semantic description of Web Services. May include functional (Capability) and usage (Interface) descriptions.

The WSMO working group is supported by the European Union's research funding frameworks. A multitude of tools are available for developing Semantic Web Services using WSMO. Some of these are WSMO Studio, WSMO4J, and IRS-III.

OWL-S[15] OWL-S is an ontology built on top of OWL by the DARPA DAML program. It replaces the former DAML-S ontology. "OWL-S is an ontology, within the OWL-based framework of the Semantic Web, for describing Semantic Web Services. It will enable users and software agents to automatically discover, invoke, compose, and monitor Web resources offering services, under specified constraints." (Martin et al., 2004)

The OWL-S ontology has three main parts: the service profile, the process model, and the grounding (Martin et al., 2004).

- The *service profile* is used to describe what the service does. This information is primary meant for human reading, and includes the service name and description, limitations on applicability and quality of service, publisher and contact information.
- The *process model* describes how a client can interact with the service. This description includes the sets of inputs, outputs, pre-conditions and results of the service execution.
- The *service grounding* specifies the details that a client needs to interact with the service.

OWL-S complements existing web service specifications such as WSDL and WSRF.

Unfortunately, OWL-S lacks mature implementations and tools.

[15] http://www.w3.org/Submission/OWL-S (retrieved 08/14/2009).

SWSF[16] Semantic Web Services Framework (SWSF) contains two major components (Battle, Bernstein, Boley, Grosof, & Gruninger, 2005):

- Semantic Web Services Language (SWSL) is a highly expressive language used to specify formal characterizations of web service concepts and descriptions of individual services.
- Semantic Web Services Ontology (SWSO) presents a conceptual model by which web services can be described, and an axiomatization, or formal characterization, of that model.

SAWSDL[17] Semantic Annotations for WSDL and XML Schema (SAWSDL) defines how to add semantic annotations to various parts of a WSDL document such as input and output message structures, interfaces and operations (Farrell & Lausen, 2007). SAWSDL does not specify a language for representing the semantic models. Instead, it provides mechanisms by which concepts from the semantic models that are defined either within or outside the WSDL document can be referenced from within WSDL components as annotations. These semantics when expressed in formal languages can help disambiguate the description of web services during automatic discovery and composition of the web services.

METEOR-S is a framework for Semantic Web Services and Processes that officially supports SAWSDL.

Spasiba employs WSMO, thanks to its continuously evolving state, supportive community, and a number of mature tools. IRS-III was used as the framework for more rapid implementation.

IRS-III

Internet Reasoning Service III (IRS-III)[18] is a framework and infrastructure that supports the creation of semantic web services according to the WSMO ontology. This framework builds upon the previous version, IRS-II, and, in addition, includes a Java API

[16] http://www.w3.org/Submission/SWSF (retrieved 08/14/2009).

[17] http://www.w3.org/TR/sawsdl (retrieved 08/14/2009).

[18] http://technologies.kmi.open.ac.uk/irs/#irsiii (retrieved 08/14/2009).

and a browser for visual editing. According to (Cabral et al., 2006), IRS-III has four distinguishing features:

1. It supports one-click publishing of standard programming code. In other words, it automatically transforms programming code (Java and Lisp are supported) into a web service, by automatically creating the appropriate wrapper. Hence, it is very easy to make existing standalone software available on the net, as web services.
2. Secondly, by extending the WSMO goal and web service concepts users of IRS-III directly invoke web services via goals. That is, IRS-III supports *capability-driven* service execution.
3. IRS-III is programmable. IRS-III users can substitute their own semantic web services for some of the main IRS-III components—for example, how web services are selected from a goal request or how complex services are executed.
4. IRS-III services are web service compatible—standard web services can be trivially published through the IRS-III and any IRS-III service automatically appears as a standard web service to other web service infrastructures.

The IRS-III architecture is composed of three main components—the IRS-III Server, Publisher, and Client—that communicate through a SOAP-based protocol. The authors of the framework describe these components in the following way in (Cabral et al., 2006) and (Hakimpour, Sell, Cabral, Domingue, & Motta, 2005):

- The IRS-III Server is based on an HTTP server written in lisp and extended with a SOAP handler. Separate modules handle SOAP-based requests from the browser, the publishing platforms, and the invocation client. Messages result in a combination of queries to or changes within the entities stored in the WSMO library.
- Publishing with IRS-III entails associating a deployed web service with a WSMO web service description. Within WSMO a web service is associated with an interface that contains orchestration and choreography. Orchestration specifies the control and dataflow of a web service that invokes other web services (a composite web service). Choreography specifies how to communicate with a web service. When a web service is published in IRS-III, all of the information necessary to call the service—host, port, and path—is stored within the

choreography associated with the web service. Additionally, updates are made to the appropriate publishing platform. IRS-III contains publishing platforms to support the publishing of standalone Java and Lisp code and of web services. Web applications accessible as HTTP GET requests are handled internally by the IRS-III server.

- A key feature of IRS-III is that web service invocation is capability-driven. The IRS-III Client supports this by providing a goal-centric invocation mechanism. The user simply asks for a goal to be achieved and the IRS-III broker locates the semantic description of an appropriate web service and then invokes the underlying deployed web service.

2.5 Self-Adaptive Systems and Autonomic Computing

As computer systems become more complex and dynamic, it appears to be infeasible to hardwire all functionality at design time. The field of self-adaptive and self-managing systems attempts to tackle this challenge by adopting the concepts known from other disciplines, such as Control Theory, Biology, and Artificial Intelligence (Brun et al., 2009). The "self" prefix denotes that such systems operate autonomously, with no or little human intervention. However, often the system operator has to specify a policy or policies that must be complied with. These policies may convey high-level objectives in terms of both functional and non-functional requirements (Dawson, Desmarais, Kienle, & Müller, 2008).

Self-adaptive systems adjust their behaviour in response to the environment. They monitor their internal state and surrounding environment, assess quality criteria, and self-tune themselves to improve the operations. In order to control dynamic behaviour, design decisions in self-adaptive systems are commonly moved towards runtime, where an individual system is capable of reasoning about itself—this is known as introspection (Overgaard, 2006). Self-adaptive systems can be characterised using a number of dimensions (Brun et al., 2009):

- Centralisation (centralized or decentralized)
- Organisation (top-down or bottom-up)
- Feedback latency (slow or fast)
- Environment uncertainty (low or high)

Top-down self-adaptive systems are often centralized around a controller or policy. They evaluate their own global behaviour and correct it when the evaluation indicates that they are not accomplishing what they were intended to do, or when better functionality or performance is possible. Such systems often operate with an explicit internal representation of themselves and their global goals. The behaviour of the whole system can be deduced by analyzing the components of a top-down self-adaptive system (Brun et al., 2009).

Another breed of self-adaptive systems is composed bottom-up of a large number of components that are decentralised and whose behaviour is governed by a set of simple rules. Such systems are called self-organizing systems (Hellerstein, 2007). The global behaviour of such a system emerges from local interactions between its components. In contrast to top-down self-adaptive systems, self-organizing systems do not use internal representations of global properties or goals.

In practice, actual engineered or natural systems incorporate techniques from both of these two cardinally different approaches (Brun et al., 2009). It is sufficient to consider the example of the Web, where the overall decentralised organisation is complemented with inner centralised components.

Feedback Loops

Müller et al. argue that feedback loops provide the generic mechanism for self-adaptation and that is why they must become first-class citizens in adaptive systems (Müller, Pezzè, & Shaw, 2008). The feedback loop, also known as control loop, is a circular pathway of cause and effect. Essentially, in context of adaptive systems a feedback loop monitors some resource (software or hardware component) and autonomously tries to keep its parameters within a desired range. As Figure 9 depicts, a feedback loop typically involves four key activities: collect, analyse, decide, and act. First, with the help of sensors, data about the current state of the system are collected, filtered, and stored. Second, trends and symptoms are inferred from the refined data. Third, based on the inferred symptoms, the system plans a set of actions to regulate its state. And, finally, the planned actions are executed with the use of actuators. Such a

feedback loop is employed—and appears to be very effective—in the IBM's Autonomic Computing Reference Architecture (ACRA) (IBM Corporation, 2006).

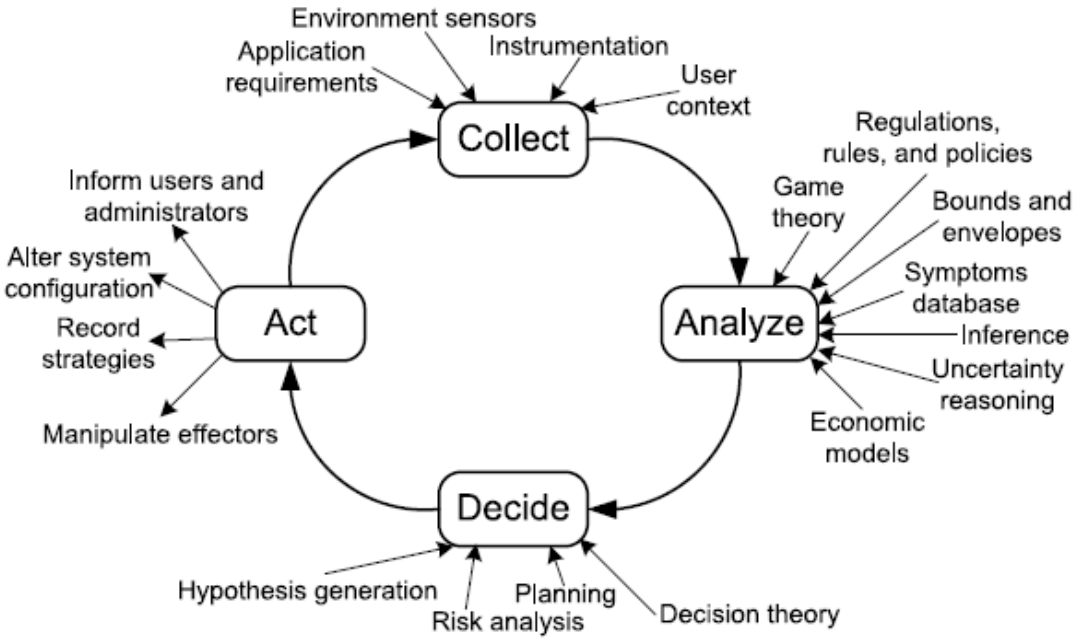

Figure 9. Autonomic control loop from (Dobson et al., 2006)

2.5.1 Autonomic Computing

Autonomic Computing (AC) is an initiative started by IBM in 2001. Its ultimate aim is to develop computer systems capable of self-management by overcoming the rapidly growing complexity of computing systems management (Kephart & Chess, 2003). In other words, AC refers to the self-managing characteristics of distributed computing resources, adapting to unpredictable changes while hiding intrinsic complexity to system operators. An autonomic system makes decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to changing conditions. AC is inspired by the autonomic nervous system of the human body (Kephart & Chess, 2003). This nervous system controls important functions of the organism without any conscious intervention.

In a self-managing system, instead of controlling the system directly, the human operator defines general policies and rules that serve as an input for the self-management process. For this process, IBM has defined the following four functional areas (Kephart & Chess, 2003):

- *Self-Configuration* (Automatic configuration of components)
- *Self-Healing* (Automatic discovery and correction of faults)
- *Self-Optimization* (Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements)
- *Self-Protection* (Proactive identification and protection from arbitrary attacks)

Driven by IBM's Autonomic Computing vision, a variety of architectural frameworks based on self-regulating autonomic components have been proposed ((IBM Corporation, 2006), (Garlan, Cheng, Huang, Schmerl, & Steenkiste, 2004), (Kramer & Magee, 2007)). Among these frameworks, Autonomic Computing Reference Architecture (ACRA) is one of the most widely applicable and effective (Müller, Kienle, & Stege, 2009).

Autonomic Computing Reference Architecture (ACRA)

ACRA introduces the notion of the autonomic system that consists of arrangements of interdependent, collaborative autonomic elements. An autonomic element is a fundamental building block for designing self-adaptive and self-managing systems (Kephart & Chess, 2003).

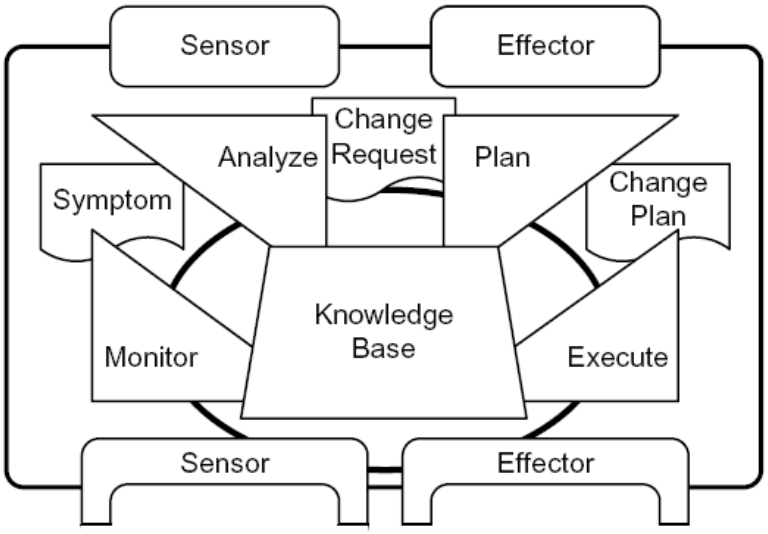

Figure 10. Autonomic Manager with a MAPE-K loop (Müller et al., 2009)

An autonomic element is comprised of an autonomic manager, a managed element, and two manageability interfaces (IBM Corporation, 2006). The autonomic manager collects various sensory data from the managed element and augments this data with information from a multitude of knowledge sources accessed via a service bus. Then, the

autonomic manager adjusts the managed element, if necessary, with the help of manageability interface—sensors and effectors—according to the control objective. An important nuance to note is that the autonomic manager can itself be a managed element—this is what the second manageability interface is for.

As Figure 10 illustrates, in the heart of an autonomic manager resides a feedback loop, called Monitor-Analyse-Plan-Execute-Knowledge (MAPE-K). Similarly to the generic autonomic control loop depicted in Figure 9, the MAPE-K loop operates in four phases over a knowledge base to assess the current state of the managed element, reason about future states, and enforce the desired behaviour (IBM Corporation, 2006):

- *Monitor*. Reads, filters, and stores data gathered from sensors.
- *Analyse*. Compares detected events against patterns in the knowledge base; diagnoses and stores symptoms.
- *Plan*. Interprets the symptoms and devises a plan to execute.
- *Execute*. Realises the devised plan through effectors.

In the MAPE-K loop, knowledge is exchanged between all four phases. An autonomic manager maintains its own knowledge and has access to knowledge which is shared among collaborating autonomic managers (Müller et al., 2009).

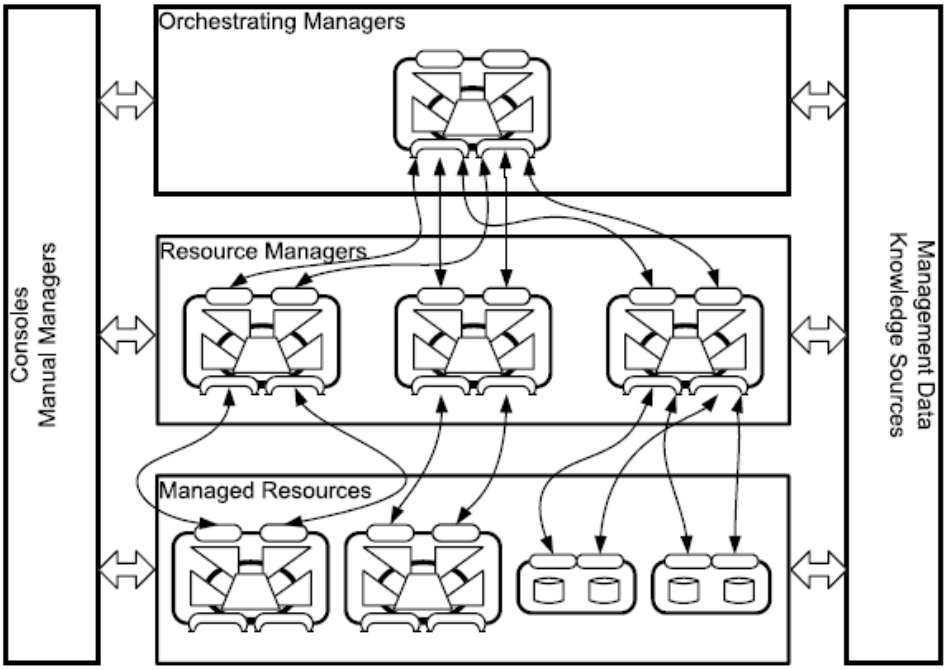

Figure 11. Autonomic Computing Reference Architecture (ACRA) (Müller et al., 2009)

ACRA is a three-layer hierarchy of orchestrating managers, resource managers, and managed resources. All of these layers exchange management data across an enterprise service bus, as depicted in Figure 11 (IBM Corporation, 2006). Moreover, ACRA includes manual managers or operators, who have access to all levels of the architecture. These hierarchical arrangements of autonomic managers promote separation of concerns (Müller et al., 2009).

When to Consider Self-Adaptive and Self-Managing Systems

Müller et al. proposed a set of problem attributes that suggest considering self-adaptive and self-managing solutions (Müller et al., 2008):

- *Uncertainty in the environment*. In particular, uncertainty that leads to substantial irregularity or may arise from external perturbations, rapid irregular change, or imprecise knowledge of the external state.
- *Nondeterminism in the environment*. Specifically, when significantly different responses at different times are expected.
- *Requirements satisfied by regulation*. In particular, extra-functional requirements, which are effectively satisfied through regulation of complex, decentralized systems.
- *Incomplete control of system components*. For instance, when the system incorporates embedded external components or when the human factor is present.

Spasiba's architecture, as will be discussed in the following chapters, exhibits uncertainty, nondeterminism, and incomplete control. Spasiba operates dynamically by invoking and composing a set of external web services. Because there is little control over these web services, uncertainty in the environment is high. Thanks to the diverse context information, the degree of nondeterminism is significant as well—hardcoded use cases are not an option. Moreover, Spasiba incorporates an external component in its architecture—IRS-III, lowering the control over the system itself. Hence, self-adaptive and self-managing solutions have been considered and applied.

2.6 Summary

This chapter introduced the reader to smartphones, context and context-awareness, service-oriented architecture and web services, the Semantic Web and Semantic Web Services, and self-adaptive systems and autonomic computing. In the discussion, the key goal was to emphasise the conceptual basis of each technology and identify overlapping points among different technologies. It proved to be a substantial effort to build an "abstraction layer" over the technical detail required to introduce a technology.

Chapter 3 Application Model

The key contribution of this work is a generic model proposed for an adaptive mobile advisor. This model and its subset that was implemented within the Spasiba prototype are described in this chapter. The chapter begins with an overview of key features, continues with a detailed description of the generic model and the actually implemented model, and finally concludes with a summary.

3.1 Key Features

The introduction mentions that modern context-aware applications for smartphones present three key opportunities for improvement: (1) limited use of the context information, (2) irrelevant or incomplete content generation, and (3) low usability. The objective of this research was to formulate a generic model for an adaptive, user-oriented, context-aware mobile application that could tackle those limitations with the following means:

1) Leverage multiple context parameters to enhance presentation and quality of generated content.
2) Ensure extensibility and autonomicity of back-end architecture in order to accommodate seamless addition of new, heterogeneous data sources.
3) Enhance the usability by simplicity: the client application should be a thin client that merely acts as a terminal and context collector.

After a considerable effort to research how these means could be substantiated within one model, I found that a myriad of technologies and paradigms would have to be incorporated in a complex, highly-distributed system. Yet, one of my goals was to create a model that is simple and easily-comprehensible. Let us take a look at some of the points that served as pillars of the proposed model.

- *Loosely-coupled architecture*. Each component is self-contained and interfaces with fellow components in a standard manner. This feature is instrumental in improving extensibility and autonomic control of the overall system.
- *Publish/Subscribe interaction model*. Within this paradigm, clients subscribe to a service and receive notifications when those become available. This provides an

elegant, effective, and highly-scalable solution to managing a multitude of clients that wish to receive dynamic updates of some information of interest.

- *Semantic Web Services*. Semantic Web Services (SWS) deliver an improvement over regular web services by employing semantical descriptions of web services, thus allowing us to manipulate these web services with greater efficiency and accuracy. Not only SWS generate more accurate results, but they also contribute to a higher degree of autonomicity of the system, thanks to automated discovery, composition, and capability-driven invocation.
- *Simplistic client application*. It is hard to deny the importance of paying utmost attention to usability of the client application. A generic model must take into account numerous platforms that could potentially be used by target users. The simplistic client application that can easily be ported is a key to accommodating these many platforms.
- *Proactive approach to interacting with users*. We are used to the software that responds to our requests, but why not let the software perform the first step and be the initiator of the interaction? In such a fashion, the software suggests the information that could possibly be helpful to the user based on the collected context.
- *Context is first class*. Context, being the aggregate of multiple environment, user, and device parameters, undoubtedly deserves to be a first class citizen in the proposed model. This is because context can be leveraged to dramatically enhance the presentation and generation of results.
- *Stateless architecture*. By stateless architecture, I express the notion of a system that only exists at run-time. Such a system assumes no persistence; all the actions are executed dynamically on the go. This stateless architecture benefits the proposed model with savings in terms of database management, I/O access, and relevance of results.
- *Self-adaptive system*. Self-adaptive and autonomic systems are such that adapt to the changing environment and manage themselves according to predefined policies and objectives. Self-adaptive functionality enables the proposed model to

automatically handle exceptions in a manageable manner, mitigate context conflicts, and generally sustain a healthy status of the system.

Of course, many of these points are interrelated and often are immediately dependent on each other. For instance, the publish/subscribe interaction model is the implementation of the proactive approach to communicating with the users. These points serve as a glimpse of the proposed model.

3.2 Model for a Mobile Web Application

Inspired by the work described in (Froehlich et al., 2007), (Weissenberg, Gartmann, & Voisard, 2006), (Becker & Bizer, 2008), and (Amendola et al., 2004) the proposed model reflects a generic aggregation of components intended to deliver a highly-satisfactory user experience to any client wishing to receive information of interest updates based on context changes. Since context plays a crucial role in this proposed framework, many important design decisions were affected by it—specifically, in accordance with the design guidelines presented in (Hakkila & Mantyjarvi, 2006). In order to leverage the context information and elevate the degree of autonomicity, feedback loops are extensively employed in the model.

In the beginning of the discussion of the proposed model, let us observe the manner in which the system operates. Figure 12 illustrates the main scenario that the system is expected to accommodate.

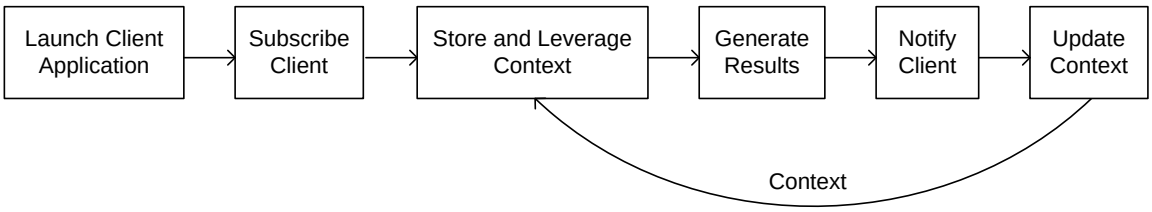

Figure 12. The illustration of flow of control in the proposed model

This scenario is initiated by the launch of the client application on a user's device. As the client application launches, it subscribes itself to the service that delivers some information of interest. In the process of subscription, the service receives the data needed for future communication (e.g., IP-address, HTTP request) and the context information. Then, the service stores and analyses the context in order to determine what the user could be interested in at a given moment. In accordance with the outcome of the

context analysis, the service compiles a query to a number of web services that finally deliver results. These results are, in turn, submitted to the client. Once the client receives new results, context information is updated and the results generation takes place again. In this scenario, as we observed, the service proactively notifies the user of the information of interest based on the changing contextual data.

A system that could effectively implement the above scenario in real life settings would have to deal with many architectural and operating challenges. Among the architectural challenges are the interaction model, the communication protocol, the realisation of separation of concerns and autonomic behaviour. The main operating challenges may be identified as performance, scalability, and error-handling. The proposed generic model aims to mediate these proclaimed challenges. However, in the process of implementing the actual prototype, I realised that it was inevitably impossible for one programmer to fully implement an example of the generic model within the available amount of time. That is why a sub-model was created that reflected the essence of the generic model, yet was intended to be easier to implement.

3.2.1 Generic Model

The presented generic model is an abstract representation of a distributed system whose objective is to deliver context-dependent information of interest to a subscribed user armed with a smartphone device proactively.

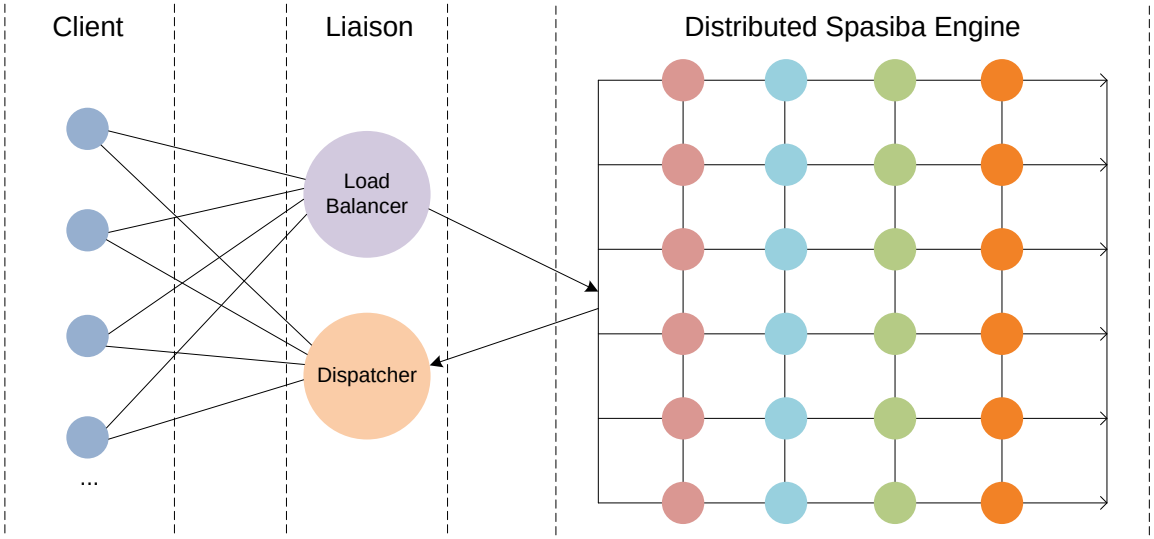

Figure 13. An overview of the proposed generic model

Figure 13 demonstrates an overview of the generic model. The model involves three loosely-coupled layers: Client, Liaison, and Distributed Spasiba Engine. The Client layer contains all smartphone devices that intend to or already interact with the Spasiba service. Each of these smartphone devices has a Spasiba client application installed and running on it. This application collects a number of context parameters and submits them to the Spasiba service, thus creating a new subscription. Before client requests reach the Spasiba service, they are filtered in the Liaison layer.

The Liaison layer is a utility layer that orchestrates and encapsulates all communication processes. This layer has two major components: Load Balancer and Dispatcher. The Load Balancer accepts incoming requests from clients and routes them to the least loaded Spasiba Engine Vector (described below). The Dispatcher handles outgoing communication with the clients.

The heart of the whole model lies in the Distributed Spasiba Engine layer. This layer is comprised of a number of distributed, yet interconnected Spasiba Engine Vectors (SEV). Each vector is a collection of consecutively invoked components that constitute the Spasiba Engine. In the generic model, all vectors exchange information among each other using the blackboard paradigm (Corkill, 1991). Blackboard is an advanced collaborative mechanism in which there is a shared knowledge base (called blackboard), updated by a multitude of knowledge sources as their internal constraints match the blackboard state. This collaboration between the SEVs enables to build a real-time social web of clients, where suggestions generated by the Spasiba service for one client may feature other clients.

Now let us discuss the Spasiba Engine Vector in more detail. As Figure 14 depicts, an SEV involves four components—Publish/Subscribe Manager (PSM), Context Handler (CH), Suggestion Creator (SC), and Semantic Web Services Broker (SWSB)—monitored and managed by the Autonomic Manager (AM). An SEV also includes two registries—Publish/Subscribe Registry and Semantic Web Services Registry—and a knowledge base with users' context history. Components in an SEV are invoked consecutively, but the registries and the context history can be accessed at any stage by any component and even by other SEVs. The Autonomic Manager operates outside of the sequence, by cross-cutting each component's operation.

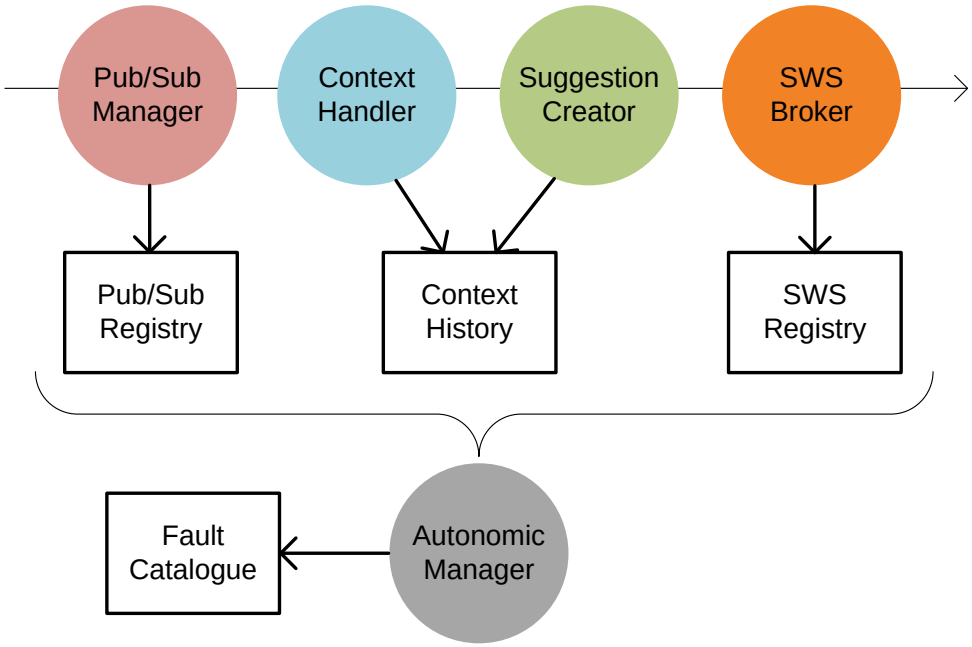

Figure 14. An illustration of the Spasiba Engine Vector (SEV)

The Publish/Subscribe Manager is the first component that is invoked in an SEV. It extracts the communication data required for future interaction and creates a record in the Publish/Subscribe Registry. Also, the PSM handles all other subscription related requests, such as update and delete directives.

The Context Handler is the next component that is invoked. The received context is processed in three stages. The first stage is concerned with identifying the user's surrounding environment and smartphone device capabilities—for instance, location coordinates and battery life. In the second stage, the outcome of the first stage is augmented with additional data acquired from the Web—for example, weather information and actual address. Then, in the third stage, the CH attempts to establish what the user might be interested in at this particular moment by analysing the current context. Finally, the results of the three stage processing are stored in the context history knowledge base.

The Suggestion Creator comes next in the sequence. It elaborates on what the CH produced in the third stage of its operation. The SC assesses all previous context states, extracts themes developed by the CH, collates these themes in accordance with a heuristic, and formulates a sensible query for the SWS Broker. During the execution of

the algorithms, the SC consults the context history and possibly modifies the record in question.

The Semantic Web Services Broker finalises the sequence of an SEV by querying semantically annotated web services listed in the SWS Registry. The SWS Broker specifies a goal-to-achieve in line with a query formulated by the SC in the previous step. Semantic descriptions allow the SWS Broker to perform capability-driven web service invocation and automated composition. After the invocation, the results obtained from the web services are sorted and refined in order to be immediately used in the presentation on a client's device.

The overall operation of an SEV is monitored and managed by the Autonomic Manager that cross-cuts each component at certain predefined execution points. Key goals of the AM are to enforce autonomic policies and handle errors that may arise. Generally speaking, an autonomic policy specifies a set of if-else relationships intended to maintain the desired system behaviour. Autonomic policies are governed by high-level objectives that allow the AM to resolve possible inter-policy conflicts. A number of knowledge bases are used to assist the AM in decision making. Among these knowledge bases are the already mentioned context history and registries as well as a fault catalogue that holds all exceptions and errors that have occurred.

3.2.2 Implemented Model

Though theoretically valid, the generic model proved to be impractical in view of the constrained resources (in particular, time) available to me. That is why the Spasiba prototype is based on a simplified model, where multiple SEVs are downgraded to a single SEV. This simplified model preserves the overall nature of the system, yet leaves the blackboard collaboration mechanism and load balancing for future work. Figure 15 illustrates the actually implemented model. Now all communication is performed via the Dispatcher (noted as D in the figure) that handles both incoming and outgoing requests. Also, there is only one SEV that is, however, almost identical to that depicted in Figure 14. One significant simplification is imposed on the context history knowledge base and, consequently, the Suggestion Creator component—instead of storing and reasoning upon all past context states, the system only keeps track of the last context state.

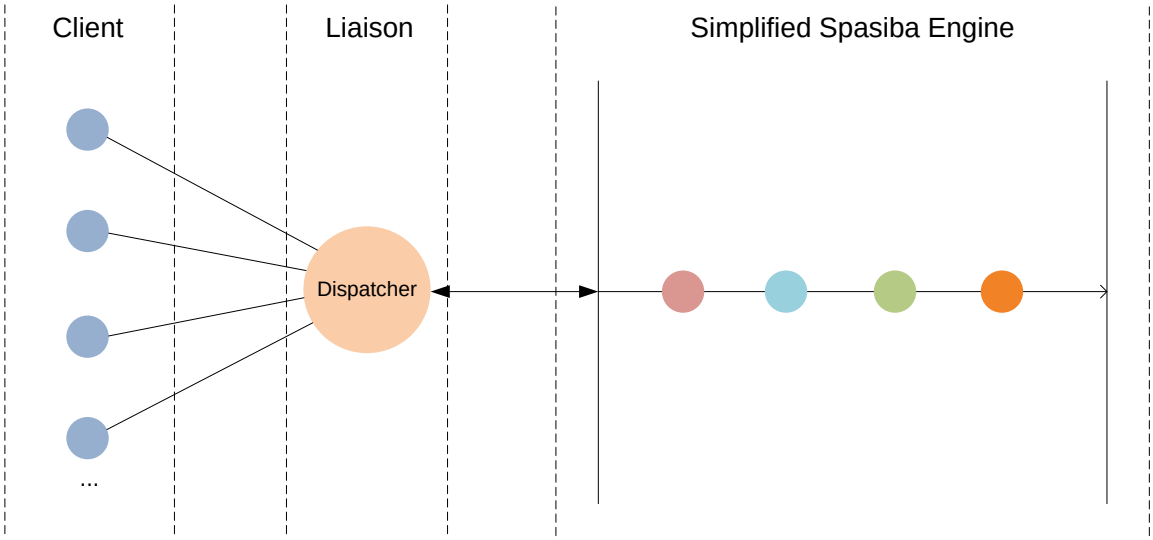

Figure 15. An overview of the simplified model

The key difference between the two models is the lack of the distributed collection of SEVs in the simplified model. This results in the elimination of need for orchestrating the collaboration between SEVs and load balancing. What are the consequences for the system? In terms of operating challenges, the system will not sustain real-life performance, scalability, and fault-tolerance challenges. Without load balancing and distributed nature of the engine, these challenges are uneasy to cope with. In terms of functionality, the users lose the awareness of fellow users interacting with the Spasiba service.

3.3 Summary

In this chapter the reader was familiarised with key architectural and functional features of the proposed generic model for a context-aware mobile application. The presented generic model reflects an abstract representation of a distributed system whose objective is to deliver context-dependent information of interest to a subscribed user armed with a smartphone device proactively. While theoretically valid, the generic model was found to be excessively hard to implement with available resources. That is why a simplified model was created especially for the implementation of the Spasiba prototype described in the following chapters. The simplified model downgrades the Distributed Spasiba Engine to a standalone Spasiba Engine and imposes somewhat trivial function on the context history and, thus, the Suggestion Creator component of the SEV.

Chapter 4 Client Application

This chapter presents the client application developed for the Spasiba prototype. To start with, the reader is requested to consult the background chapter to become familiar with the Symbian OS and Nokia S60 platform on which the client application is run. In this chapter, only a brief mention of their features is given. Then, the chapter concisely reviews the Nokia WRT—the framework that the client application was developed in. This is followed by the discussion of components and main features of the Spasiba Widget and, in particular, the context collection process. And, finally, the chapter is concluded with the known limitations and a summary.

4.1 Overview of Symbian OS and Nokia S60

This section only provides an extremely brief overview of Symbian OS and Nokia S60. Please consult Chapter 2 for extended treatment of these topics.

Symbian is an operating system designed for mobile devices and smartphones, with associated libraries, user interface, frameworks and reference implementations of common tools. Its memory-management and multitasking features allow for safe and efficient operation under conditions of constrained resources that typify mobile devices. Nokia's platform built upon Symbian is called S60; Figure 3 depicts the architecture of S60. Prominent components of this architecture are Platform Services (fundamental services offered by the platform), S60 Applications (embedded applications), Java Technology Services (support for Java ME), UI libraries, and the Web Runtime (a runtime environment that enables S60 devices to run web widgets).

From the developer's point of view, S60 presents a number of options for the development platform: C++, Open C, Java ME, Web Widgets, and Flash Lite. The Spasiba prototype was implemented as a widget application using the Nokia Web Runtime (WRT) and, thus, is supported on all Nokia S60 5th Edition smartphones. In addition to the Nokia WRT, these smartphones feature a touch screen capability, improved UI, a new sensor framework, and enhanced on-demand paging.

4.1.1 Nokia WRT

Web Runtime is a browser-based engine that allows a programmer to apply web development skills to create full mobile applications that are simple, powerful, and optimised for mobile devices.

Web widgets are lightweight applications created with the help of standard web technologies that are used to create web pages, such as HTML, Cascading Style Sheets (CSS), JavaScript, and Asynchronous JavaScript and XML (AJAX). Widgets run in the Web Runtime environment, which is supported by Nokia S60 3rd Edition, Feature Pack 2 devices and later. Users can download and install web widgets as they would any other S60 application or content item. Once installed, web widgets appear as standard S60 applications and provide the users with a full web experience, in a way that allows them to personalise the content and services they access. Integration with S60 Platform Services enables widgets to access information—such as a device's location—and share it with a web server to offer unprecedented levels of relevance in web content. All available Platform Services are presented below according to Nokia (2009):

AppManager	Access and launch applications.
Calendar	Access, create, and manage calendars and calendar entries.
Camera	Launch the device's native camera application and retrieve information about pictures taken with the application.
Contacts	Access and manage information about contacts.
Landmarks	Access and manage information about landmarks and landmark categories.
Location	Retrieve information about the geographic location of the device and perform location-based calculations.
Logging	Access and manage logging events such as call logs, messaging logs, and data logs.
Media Management	Retrieve information about the media files stored in the device's public folders.
Messaging	Send, retrieve, and manage messages using the Message Store.
Sensors	Access data provided by the physical sensors of the device.

System Information	Access and modify system information. System information is represented as a set of system attributes that are grouped into the following categories: Battery, Connectivity, Device, Display, Features, General, Memory, and Network.

In addition to the functionality provided by the Platform Services, web widgets also have built-in JavaScript objects that implement the following features:

Self-updating widgets	Widgets can update themselves by communicating directly with a web service or server.
Navigation	Mobile device users can navigate the UI using tabs or a cursor.
Language-specific versions	The S60 platform supports multiple languages and allows mobile device users to select the language that their device uses for UI texts.
Dynamically modified web pages	Widgets can inspect or modify web pages dynamically through the use of Document Object Model (DOM) Level 2.

Widget development is simplified with plugins for Aptana Studio, Adobe Dreamweaver, and Microsoft Visual Studio. These plugins enable developers to create, edit, test, validate, package, and deploy widgets.

A widget consists of two mandatory and a number of optional component files. These component files are listed below:

info.plist	Mandatory	An XML-formatted file that contains the property and configuration information of a widget, such as the HTML file used as well as the identifier of the widget.
<name>.html	Mandatory	A standard HTML file that mainly contains information for structuring a widget. The name of the HTML file must be predefined within the *info.plist* properties. A widget package contains only one HTML file.
icon.png	Optional	A custom icon file that represents a widget on a mobile device when it has been installed.

**.css*	Optional	External CSS file that defines the style and layout of widget contents. A widget can have as many individual CSS files as needed.
**.js*	Optional	External JavaScript source code that implements the logic of a widget's operations, such as the construction of the widget UI elements, UI interaction, and communication. A widget can have as many individual *js* files as needed.
**.jpg/gif/png*	Optional	A custom image file that can be used in a widget. A widget can have as many individual image files as needed.

4.2 Components of the Spasiba Widget

The client application for the Spasiba prototype was developed using the Nokia WRT framework and, thus, is a widget. The Nokia WRT proposes a clever separation of concerns in which logic, view, layout, and graphics are decoupled from each other. The Spasiba widget fully followed this model in the fashion illustrated in Figure 16.

The centerpiece of the widget is *spasiba.html* that encapsulates the layout of the widget. In a web widget there is only one layout file that is composed of a number of widget states controlled by the logic enclosed in JavaScript scripts.

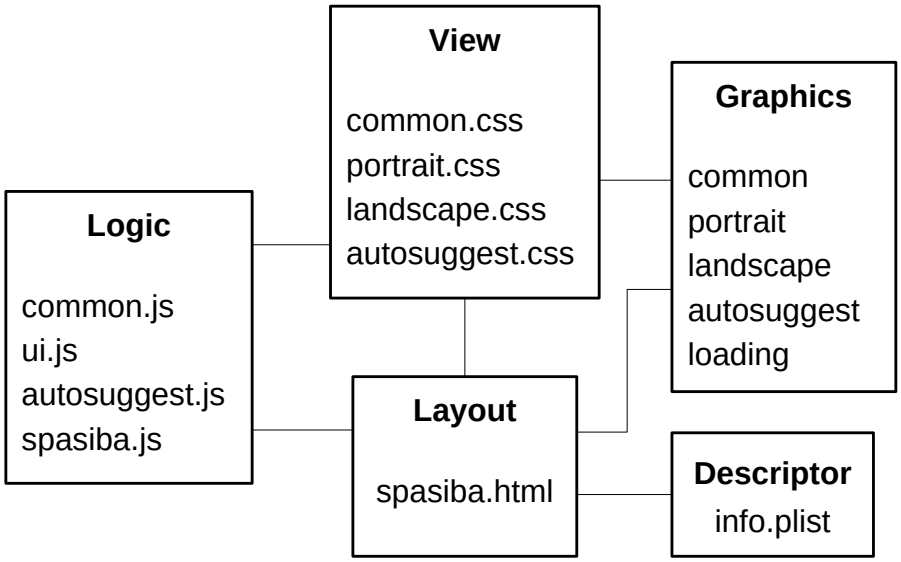

Figure 16. Components of the Spasiba widget by purpose

The presentation is described by the styles encoded using the Cascading Style Sheets. When designing a widget, two separate style sets (along with pertinent graphics) must be created for the two screen orientation modes: landscape and portrait. Styles are swapped and modified at runtime with the help of JavaScript.

The logic of the widget is distributed over four files. Common utility functions reside in *common.js*. Functions related to the user interface are encapsulated in *ui.js*. The auto-suggest functionality (a feature that auto-completes the user's request) is enclosed in *autosuggest.js*. Last, but definitely not least, *spasiba.js* provides the AJAX communication, context collection, and application control functionality.

4.3 Features of the Spasiba Widget

The key considerations in the design and implementation of the Spasiba widget were high usability, portability, context collection, and communication with the Spasiba service. The high usability aspect is accommodated by user interface features and adaptive functionality. Portability is inherent in the design of the client application—that is why the interface was to be simplistic and programming logic was to be minimised. The less there is to port between platforms, the easier is the migration. Context collection is dedicated a special treatment in the Spasiba widget. This process is uneasy thanks to the imperfect APIs of the Nokia WRT and the volatile nature of some of the collected parameters (e.g., battery status, location coordinates). Other context-related issues include selecting context parameters and polling period. Communication with the Spasiba service is performed solely via AJAX requests. In particular, in order to implement the proactive notification of information of interest, a technique known as Comet (Crane & Mccarthy, 2008) was employed. Let us start the detailed discussion of the Spasiba widget with the user interface.

4.3.1 User Interface

- Simple yet elegant interface that is optimised for touch interaction. Usability and absence of a learning curve are supported by an intuitive interface that consists of an input field, context area, and a submit button. Figure 17 demonstrates these three elements. Each element of the interface occupies a substantial portion of the screen, thus making it easy to navigate the widget with a finger. The Spasiba

widget requires no configuration. It can be installed on any Nokia S60 5th Edition device at just one touch. In addition to the enhanced usability, the simplicity of the interface decreases the widget's memory footprint to almost nothing.

Figure 17. Interface elements of the Spasiba widget

Figure 18. The auto-suggest functionality of the Spasiba widget

- Support for both touch and keyboard-based input. The user has the freedom of interacting with the widget using the on-screen keyboard as well as the actual keyboard (when present).
- Auto-suggest functionality. This feature auto-completes the user's request or, in case the user misspelled a word, corrects it. The suggested words are drawn from a list of the most popular requests. The auto-suggest functionality is largely based on a JavaScript library developed by Timothy Groves.[1] Figure 18 illustrates the auto-suggest feature in action.
- Dynamically generated filters. These filters are created on the fly, according to the semantic category of a user's request. They assist the user in narrowing down the

[1] http://www.brandspankingnew.net/specials/ajax_autosuggest (retrieved 4/21/2009).

request's scope, thus increasing the accuracy of results. All filters allow the user to specify the desirable distance and mode of operation. The Spasiba widget can operate in two modes: static and dynamic. In the static mode, the user submits the request and immediately receives the response—the communication is ended at this point. In the dynamic mode, the user subscribes to the Spasiba service and receives updates in accordance with context changes. These modes will be described in much more detail in the communication subsection and in the next chapter. Some filters—for instance, when request is "food" or "restaurant"—also allow the user to select a subtype of the entity (shown in Figure 19).

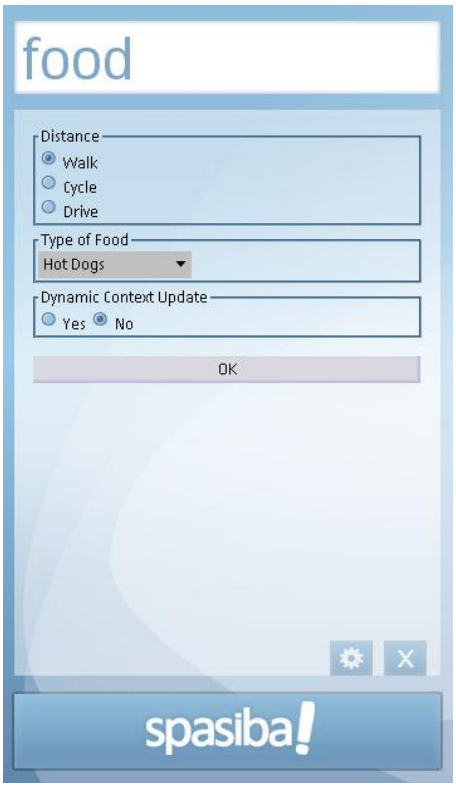

Figure 19. A dynamically generated filter for request "food"

- Two views—portrait and landscape—with identical feel and usability. The widget features two style sets for each view. With the help of a feedback loop that monitors the screen orientation, these style sets are swapped in and out to present the content to the user in the most usable and aesthetically pleasing manner. Figure 20 and Figure 21 depict portrait and landscape views, respectively.

Figure 20. Portrait view of the Spasiba widget

Figure 21. Landscape view of the Spasiba widget

4.3.2 Context Collection

Obviously, in a context-aware mobile application context plays one of the most important roles. When dealing with context on a mobile device, a number of steps must be taken prior to the successful operation. First, context parameters must be selected out of all available device, environment, and user attributes. Then, these selected context

parameters are collected according to a certain policy. This policy specifies when the context is to be updated. The update may be triggered by a significant change in one of the parameters (e.g., if the user has moved 200 metres, then update the overall context), expiration of a predefined time interval, or by an order of the Spasiba service. Defining an appropriate policy is not the only issue in the process of context collection. Unfortunately, at the time when the Spasiba widget was being developed, the Nokia WRT API was somewhat unstable. In addition, this problem was aggravated by the volatile nature of several attributes. That is why a major programming effort was required to ensure all selected context parameters could actually be acquired from the device.

Let us examine the process of context selection in more detail. To start with, I would like to list all attributes available on a Nokia S60 5th Edition device. As noted earlier, there are three types of attributes that can potentially be selected: user, environment, and device. Below are listed attributes by each of these types, based on the Nokia WRT API documentation.[2]

User Attributes

- Calendar entries
- Media (e.g., pictures, music, videos)
- Contacts
- Logs (e.g., calls, sent and received messages, data usage)
- Messages

Environment Attributes

- Landmarks (i.e., places of interest)
- Location (i.e., coordinates, velocity, elevation)
- Camera

Device Attributes

- Sensors (e.g., accelerometer, ambient light sensor, proximity sensor)
- Battery (i.e., strength and charging status)
- Connectivity (e.g., active connections, MAC address)

[2] http://library.forum.nokia.com/index.jsp?topic=/Web_Developers_Library/GUID-B796D072-4E51-4BC7-9259-84530DB3539D.html (retrieved 4/21/2009).

- Device (e.g., firmware version, platform version, phone model, IMEI)
- Display (e.g., brightness, user inactivity, display orientation, display resolution)
- Features (e.g., available hardware features)
- Language (e.g., input language, supported languages)
- Memory (e.g., available memory in RAM, flash, or memory card)
- Network (e.g., signal strength, network mode, current network, cell ID)

Even though a system that could cleverly incorporate all of these potential context parameters could exhibit unprecedented intelligence, the feasibility of such a system is improbable. During the collection of the attributes, the consumption of resources is exceptionally high in terms of battery usage, CPU load, and Input/Output interruptions. Hence, only a limited number of attributes can be collected at once. The Spasiba widget collects the following attributes:

- Language (i.e., input language)
- Date and Time
- Battery (i.e., strength and charging status)
- Location (i.e., coordinates and velocity)
- IMEI (i.e., unique identification number)
- Active connections (i.e., GPRS, EDGE, 3G, Wi-Fi, LAN)
- Network mode (i.e., whether the user is in home network or roaming)

Since Spasiba is a prototype, I deemed these seven context parameters to be sufficient. Let us briefly review the purpose of each parameter. The input language parameter allows the service to provide a localised response translated to the user's language with relevant cultural adjustments (e.g., convert metres to feet, if language is EN-US). The service uses the date and time parameter during the suggestion creation and as a filter in results generation (e.g., if it is 1 PM on a Sunday, suggest a nearby bistro for a special deal brunch; or if it is 4 AM and the request is "food", suggest the closest 7/11 store). Battery, active connections, and the network mode are all employed to adapt the presentation of results prior to submitting them to the client (e.g., if battery strength is less than 10% or user is roaming, omit or optimise a heavy map). IMEI is used as a primary key in the dynamic storage of context history. Last, but not least, location coordinates are, of course,

employed in the results generation and refinement by distance; velocity is used to deduce whether the user is driving, walking, or cycling.

In addition to these seven parameters collected using the Nokia WRT Platform Services, the Spasiba widget acquires a number of user preference parameters that complement the overall context. These are the request specified by the user (e.g., "store") and information obtained from a dynamically generated filter (e.g., desired distance from the user's location, subtype of an entity ("souvenir store")).

Now that the context parameters are selected, they must be collected. The only point that deserves the reader's attention in this process is specifying when to update context. As was mentioned in one of the previous sections, the Spasiba widget can operate in two modes: static and dynamic. In both modes, the context is collected once the communication with the service is initiated. However, in the static mode the interaction ends after the results are received, while in the dynamic mode a special policy must be in place in order to continuously supply updated context information to the service. I identified three types of such update policies: *threshold*, *timer*, and *order*.

In a *threshold policy*, the system monitors one of the context parameters and when this parameter changes beyond a certain value, the update is triggered. For example, if the location changes over 400 metres, context is updated. The advantage of this policy is an ability to perform a justified update—that is, an update that is actually needed. The disadvantage is the overhead required for constantly monitoring one of the parameters and calculating whether its value exceeds a threshold.

In a *timer policy*, context updates are initiated at a predefined interval. The advantage of such a policy is in its simplicity and low overhead. The disadvantage is that the context changes must be modelled in order to choose a favourable update interval; otherwise, unreasonable context updates may take place, thus wasting resources.

In an *order policy*, the Spasiba service (or even the user) initiates the update on its behalf. This policy cannot be compared to the other two, because it involves a forced update.

In demonstrations and presentations, the Spasiba widget uses a timer policy with an interval that depends on the hard-coded route. However, the policy that was selected for the prototype is a threshold policy based on the location change over 50 metres. This approximation is derived from the fact that the main use case of the Spasiba widget is a tourist assistant. In this use case, the tourist predominantly walks or cycles in the downtown of an unfamiliar city.

4.3.3 Communication

All communication with the Spasiba service is performed via AJAX requests, since this is the only available option for a web widget application. There are two types of HTTP requests that are initiated by the Spasiba widget: GET and PUT. The GET request is used for obtaining the auto-suggest results, while the PUT request delivers a so-called Spasiba envelope to the service.

A Spasiba envelope, formatted in XML, is the transport unit employed for interacting with the Spasiba service. Figure 22 depicts the structure of a Spasiba envelope. It is comprised of two parts: control and body. The control part contains two subparts: action and client type. Action specifies the purpose of the envelope. Four types of actions are possible: *get filter*, *get results*, *subscribe*, and *update*. While these types are self-explanatory, it must be emphasised that the latter two belong to the dynamic mode operation (described below). In particular, *update* is used when updated context needs to be communicated to the Spasiba service. The client type is a place holder that will become useful once more platforms are supported. The body of a Spasiba envelope carries the actual payload that is represented by either collected context or encoded filter values.

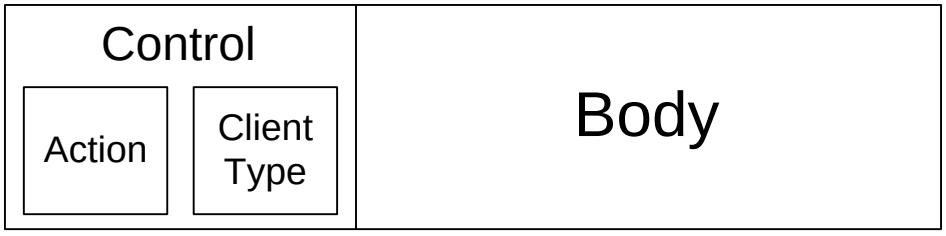

Figure 22. The structure of a Spasiba envelope

The Spasiba prototype supports two modes of communication: *static* and *dynamic*. In the static mode, the interaction is terminated once the user receives the first results. In contrast, in the dynamic mode the user continues to receive updates as the context changes. That is, in the dynamic mode the user subscribes to the Spasiba service. Then, the service not only delivers results relevant to the user's initial request but also provides suggestions or recommendations in accordance with the acquired context information. The user has the freedom of modifying the subscription at any point by invoking the options screen and changing areas of interest (shown in Figure 23).

Figure 23. Modifying a subscription in the dynamic mode

The implementation of the dynamic mode in terms of communication is a challenge, thanks to the limitations of the HTTP protocol. The technique known as Comet or Reverse AJAX (Crane & Mccarthy, 2008) must be used in order to realise the proactive behaviour of the Spasiba service. The proactive behaviour involves server-initiated updates directed to the client. As far as implementation goes, there are a number of options to implementing a Comet solution. These options include: HTTP streaming, hidden IFrame, long-polling, and short-polling.

HTTP streaming features a single persistent connection to the server. Each time the server sends a new event, the browser interprets it; neither side closes the connection. Even though HTTP 1.1 has brought improvements in terms of long-lived connections, this option is unreliable—in particular, when a mobile device is in play.

Hidden IFrame is a technique where an invisible IFrame is sent as a chunked block and, thus, is declared infinite. When events occur, <*script*> blocks are sent by the server and executed right away on the client side. This is an error-prone and bulky solution that may be considered a hack. In addition, the Nokia WRT browser does not support IFrames.

Short-polling is a simple simulation of the dynamic push functionality. In this technique, the client polls the server at a predefined interval using AJAX requests. This method is favourable when updates are likely to happen constantly and frequently and when immediate notification is desirable, but not required. The interval at which the server will be polled must be chosen cleverly. If polling occurs too often, the service may become overloaded. If polling is performed seldom, the relevance of notifications may be reduced.

Long-polling is a mechanism similar to short-polling. However, instead of polling the server at a predefined interval, a more sophisticated technique is employed. The client commences the interaction by sending an AJAX request to the server. The server keeps the connection open until an update is available and only then responds to the client. Once the client receives an update, it immediately sends a new AJAX request to the server. Long-polling is a simple yet very effective mechanism that ensures the updates are instantly propagated to the client. A drawback of this mechanism is that in some cases the connection may remain idle for a while and be terminated, causing an error.

The Spasiba widget employs a context-driven model for Comet that may be classified as long-polling or short-polling depending on a context collection policy. In this context-driven model, the Spasiba service is polled when new, updated context becomes available. It may be classified as short-polling when a timer policy is in place and long-polling—when an order policy is in effect. When a threshold policy is chosen, the

dynamic push is initiated by the client; however, to the user it will appear as an update initiated by the Spasiba service.

4.3.4 Adaptive Functionality

Adaptive functionality of the Spasiba widget is three-fold. First, the Spasiba widget adapts its UI to changes in screen orientation and resolution by hot-swapping predefined style sets. Also, the UI exhibits adaptive behaviour in terms of adjusting the presentation in accordance with the seven collected parameters—however, these adaptations take place on the server side. Second, the Spasiba widget features an extensive error-checking mechanism. Error-checking is crucial in such an application, because it is common to encounter problems collecting or monitoring context parameters—often a zero or null value is returned. Third, an adaptive solution is exploited in the process of context collection—a feedback loop ensures the context is updated once location or another selected attribute changes beyond a threshold. All of these features are implemented with the tremendous help of feedback loops that not only present an intuitive abstraction, but are also easy to realise programmatically.

4.4 Limitations

Although the Spasiba widget is a fully functional application, it has a number of limitations or areas for improvement. These are: a lack of support for older Nokia devices, a socket recycle failure when in dynamic mode, a lack of a draggable map, and no results history.

The Nokia WRT was introduced in Nokia S60 3rd Edition, Feature Pack 2. However, the widget is only supported on Nokia S60 5th Edition devices. This is because Nokia S60 3rd Edition devices exhibit limited support of the Platform Services, significantly lower screen resolution, and absence of a touch screen.

A socket recycle failure occurs when the widget operates in the dynamic mode, receiving updates from the server at a low interval (e.g., every 5 seconds). When plain text results are loaded without a map, this problem does not occur. After a thorough investigation, I discovered that this is a Nokia WRT bug that will hopefully be resolved in future versions of the framework.

The user interface of the Spasiba widget can be improved with the help of a draggable map (currently the map is an image). It is possible to load a regular Google Map or Yahoo Map in the widget, but their performance is highly unsatisfactory. In addition, they are not suited for touch interaction. At the time of this writing, the Nokia/Ovi Maps have become free and integrated in the platform—that is, the user may click/touch the address and the Maps application will launch. Since Symbian OS and Nokia S60 support multitasking, the user can consult the map and then return to the Spasiba widget.

Another potential improvement in terms of the user interface is tab-based results history. Currently, when new results are received from the server, the old results are discarded. The usability and functionality will be enhanced if previous results are stored, hidden, and easily accessible with one touch on a tab.

4.5 Summary

The client application of the Spasiba prototype is a Nokia WRT widget. A widget is a lightweight application that is developed using regular web technologies. The Spasiba widget features a simplistic user interface that is optimised for touch interaction on the go. One of the key goals of the client application is context collection. The widget collects seven thoroughly selected context parameters that are complemented with the values obtained from a dynamically generated filter. These dynamically generated filters are created at runtime in accordance with the category of the user's request. The Spasiba widget is capable of operating in two modes: static and dynamic. In the static mode, the user submits the request and immediately receives the response—the communication is ended at this point. In the dynamic mode, the user subscribes to the Spasiba service and receives updates as the context changes. The dynamic mode is implemented with the help of Comet-based techniques that simulate the server-initiated push functionality. Feedback loops are employed to provide adaptive functionality in terms of UI adjustments, error-checking, and context monitoring. Even though the widget is fully functional, there are several limitations and areas for improvement that should be accounted for.

Chapter 5 Control of Flow and Dynamism

This chapter discusses the interaction model introduced in the Spasiba prototype. The discussion begins with a general overview of the interaction model and the transport unit employed for communication—the Spasiba envelope. Then, the two interaction modes, static and dynamic, are described more extensively. In description of the dynamic mode, special attention is drawn to the publish/subscribe paradigm, push notifications, and suggestion policies. The chapter concludes with known limitations and a summary.

5.1 Interaction Model

Chapter 3 introduced the generic model in which there is a single mode of interaction. In this mode, the client subscribes to the Spasiba service and receives push notifications as its context changes. The implemented Spasiba prototype also adds a static mode of interaction. In the static mode, Spasiba operates as a regular local search engine by merely answering a user's request. Hence, the interaction model included in the Spasiba prototype is comprised of two modes: static and dynamic.

In order to structure and systematize the requests sent by the client, a special transport unit was created. This unit is the Spasiba envelope that was described in the previous chapter. To reiterate, the Spasiba envelope, depicted in Figure 22, consists of two parts: control and body. The control section assists the Spasiba service in identifying the purpose of the request and the device from which the request originates. Thus, the control part contains two directives: action and client type. Four types of actions exist: *get filter*, *get results*, *subscribe*, and *update*. These actions are elaborated upon below. The body section of the Spasiba envelope is devoted to either context information or filter values. In terms of implementation, the Spasiba envelope is encoded in XML.

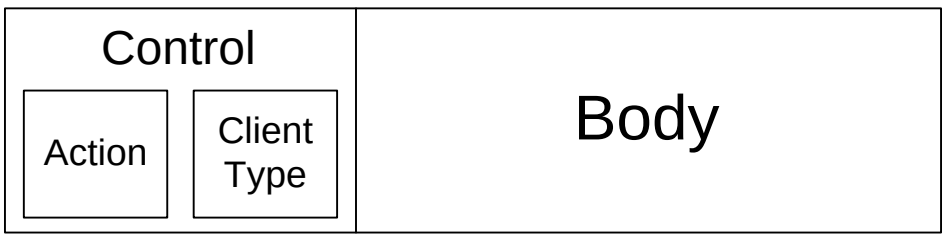

Figure 22. The structure of a Spasiba envelope (from Chapter 4)

Figure 24 illustrates an overview of the interaction model implemented in the Spasiba prototype. Conceptually, the interaction with the Spasiba service commences with the user's initial intention. The user's intention may involve either a single local search or a continuous search in which updates are delivered constantly. The initial request submitted by the client is always *get filter*, under both intentions. The *get filter* request transmits the client's context to the service as well as the user's search string and demands a dynamically generated filter to be sent in response. The filter (which is an HTML form) tailored especially for the user's search string allows the user to impose additional constraints on the search, such as distance and subtype of an entity, but also to choose the interaction mode. If the user selects *static*, the client submits a *get results* request that delivers the filter values to the service and indicates that the static mode of interaction should be followed. If the user's choice is *dynamic*, a *subscribe* request is sent to the service. A *subscribe* request delivers the filter values to the service and specifies the dynamic mode of interaction.

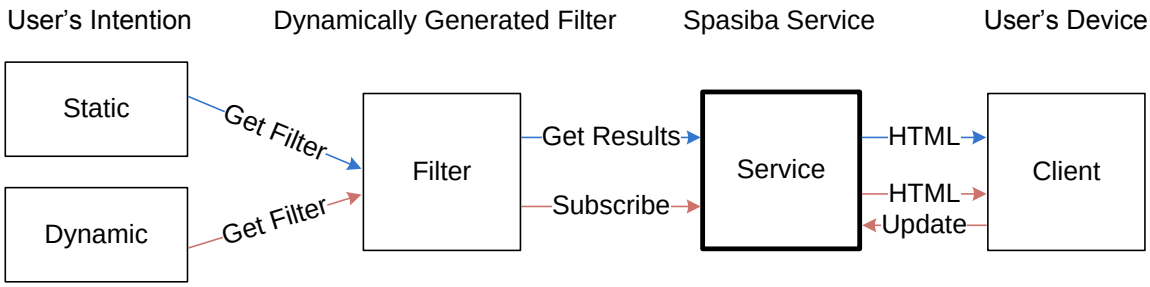

Figure 24. An overview of the interaction model introduced in the Spasiba prototype

Then, in the static mode, the Spasiba service synthesises and analyses the context, filter values, and user's search string to formulate a query for the web services broker. When the web services respond, the search results are refined and propagated to the user in display-ready HTML. In the dynamic mode, the Spasiba service executes two additional steps: (1) adding the client to the publish/subscribe registry and (2) consulting the suggestion engine prior to formulating a query for the web services broker. In the static mode, the interaction is ended after receiving the results. Clients that operate in the dynamic mode continue the interaction with the service beyond receiving initial results by constantly submitting context updates (using the *update* request); the service analyses the updated context and provides new results and suggestions.

5.2 Static Interaction

The static mode of interaction allows the user to communicate with the Spasiba service in the traditional way. In this mode, the Spasiba service behaves like a local search engine that aggregates data from various sources and provides a single delivery of information of interest to the user. As was noted earlier, the generic model proposed in this thesis assumes only dynamic interaction in the publish/subscribe fashion. However, in the process of implementation I concluded that it would be beneficial for the user and for testing to include a simplified interaction model as well as a full-blown proactive notification model. Thus, the static mode conveys a long-established user-experience, which helps the user to feel more comfortable and encourages him or her to engage in further exploration of the application's features. In addition, the static mode played an instrumental role in testing many components of the Spasiba Engine Vector (SEV), such as the Context Handler and the Semantic Web Services Broker.

In terms of the user experience, using the Spasiba client application in the static mode is a trivial, intuitive task. The user inputs a search string (e.g., "sushi") and presses the submit button. Next, the Spasiba service assesses the search string and user's context to build a filter tailored specifically to the user's request. The filter allows the user to specify the distance, mode of interaction, and, possibly, other parameters that depend on the request. Although the dynamic mode is the main communication model, the static mode is selected by default. This is because the user is stimulated to try the static mode first and only then to explore the proactive behaviour—in other words, this is a feature that eases the learning process. After the submission of the filter, the user receives local search results, which are generated and presented in accordance with the user's context and request.

In regard to the implementation of the static mode, two Spasiba envelope actions are employed for orchestration of communication: *get filter* and *get results*. All requests originating from the client are encoded in XML, while all data submitted by the service back to the client are formatted in display-ready HTML. This way the service can easily parse the Spasiba envelope, extract the required control information, and inflate a context or filter bean. And, the client need not be concerned with parsing the results and adding mark-up, since HTML-formatted results acquired from the service can be placed in a *div*

block and displayed to the user immediately. In general, the implementation of the static mode does not involve any sophisticated communication protocols or paradigms—the flow of control is predictable and closed-ended.

5.3 Dynamic Interaction

One of the key features of the generic model is the proactive approach to notifying the user of potential information of interest. This proactive approach is an interaction model in which the service acts as the initiator of communication. The only action that is required from the user is subscribing to the service. Once subscribed, the user continuously receives suggestions containing local places of interest. The suggestions may either be based on the initial user's search string or a heuristic that deduces from the context what the user might be interested in.

The manner in which the communication is started with the service is identical to the static mode described above, except for the interaction mode selection in the dynamically generated filter (which must be switched to *dynamic*). However, after the submission of the filter, the user experience drastically differs from that imposed by the static mode. The client continues to monitor, collect, and communicate context to the Spasiba service, which, in turn, constantly produces suggestions for the client. At any point of the continuous notification, the user has the freedom of changing the subscription by modifying the areas of interest (three areas are supported: Food, Stores, and Attractions).

In terms of implementation, the dynamic mode involves a much more sophisticated (as compared to the static mode) set of technologies. Among the enabling technologies are the publish/subscribe paradigm and Comet. The former provides an elegant, effective, and highly-scalable solution to managing a multitude of clients that wish to receive dynamic updates of some information of interest. The latter is a paradigm that builds on existing web technologies and protocols to enable server-initiated push notifications. Comet was introduced and described in limited detail in the previous chapter. In addition to the two enabling technologies, the realisation of the suggestion mechanism requires a number of algorithms and heuristics that are capable of recommending useful information with limited knowledge of the user. With respect to the Spasiba envelope actions, two of them are used in coordinating the dynamic mode: *subscribe* and *update*.

5.3.1 Publish/Subscribe Model

Publish/subscribe is an asynchronous messaging paradigm that decouples senders from receivers. Published messages are characterised into classes, without knowledge of what subscribers there may be. Subscribers express interest in one or more classes, and only receive messages that are of interest, without knowledge of what publishers there are. In the publish/subscribe model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called filtering. There are two common forms of filtering: *topic-based* and *content-based*. In a *topic-based* system, messages are published to topics or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe. In a content-based system, messages are only delivered to a subscriber if the attributes or content of those messages match predefined constraints.

As Figure 25 illustrates, a typical architecture implementing the publish/subscribe model involves the following components (Box et al., 2006): subscriber, event source, subscription manager, event publisher, event dispatcher, and event sink. Subscriber is an entity that sends subscription requests, which generally can be of four types: *subscribe*, *get status*, *renew*, and *unsubscribe*. Event source is a service that sends notifications and accepts requests to create subscriptions. Subscription manager is a component that binds itself to an event source in order to manage, manipulate, and store subscriptions. Information about subscriptions is stored in a registry. Event publisher is a special mediator that retrieves subscription information from the event source and directs the event for dispatch. Event dispatcher delivers events to event sinks that possessed matching subscriptions. Finally, event sink is an entity that receives notifications.

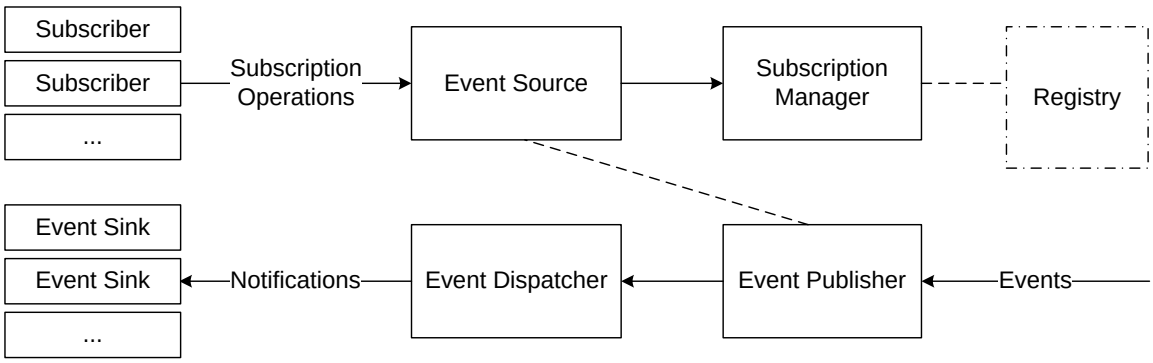

Figure 25. An example of an architecture implementing the publish/subscribe model

Conceptually, the publish/subscribe model implemented in the Spasiba prototype closely follows the described typical architecture. Subscriber and event sink are integrated into one entity—the client application. The client application is capable of producing three types of subscription-related requests: *subscribe*, *unsubscribe*, and *modify subscription*. The *subscribe* request is delivered via a Spasiba envelope action with identical name. The *unsubscribe* request is implicit—it is recognised by the service when a new *get filter* request is received (that is, when the same client creates a new subscription or performs a search in the static mode). The *modify subscription* request occurs when the user changes an area of interest using the screen in Figure 23.

Event source and subscription manager are represented by one component—the Publish/Subscribe Manager (PSM). The PSM handles all subscription-related requests, manages, manipulates, and stores the subscription data. In terms of storage, an in-memory registry is employed in the Spasiba prototype (realised with the help of a *ConcurrentHashMap*). In addition, the PSM provides an access point for the event publisher in order to perform the matching of subscriptions.

In the Spasiba prototype, event publishing, depicted in Figure 26, begins with a context update received by the service. Then, operations pertinent to the Spasiba Engine Vector occur: context analysis, suggestion creation, and, finally, results generation. The produced results are classified under two topics: interest topic and suggestion topic. Interest topic denotes the user's initial registered interest, designated by the search string. Suggestion topic is a wildcard that is unique to a particular client. The results along with their respective topic form two events that are handed to the event publisher. The event publisher performs the mediation of events by matching the topics against existing subscriptions. After the mediation, the events are propagated to their receivers by the dispatcher. The same dispatcher is used for all incoming and outgoing communication within the Spasiba service.

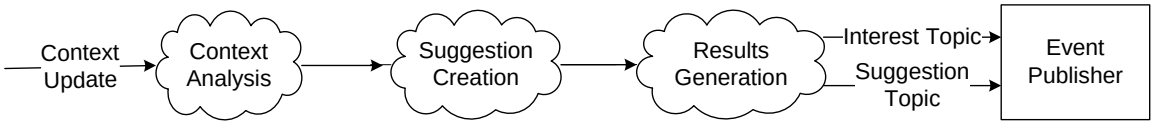

Figure 26. Event publishing in the implemented publish/subscribe model

5.3.2 Comet and Push Notifications

The proactive approach to notifying the user of information of interest requires server-initiated push technology. Since the client application is basically an HTML page rendered by a web browser engine, these push notifications cause an implementation challenge. Unfortunately, the original model of the Web, in particular the HTTP protocol, does not support server-initiated communication with the client. Instead, a page-by-page interaction is assumed with the client always acting as the initiator. However, in recent years, the Web has shifted further towards web applications rather than web pages (Zepeda & Chapa, 2007). One of the features of these web applications is high responsiveness achieved by means of AJAX and Comet. AJAX is a technique that employs the *XMLHttpRequest* object of JavaScript to communicate with the server without reloading the page—that is, the web page has a capability of sending requests to the server seamlessly, in a separate channel. AJAX is the only communication model available to Nokia WRT widgets. Comet, also known as Reverse Ajax, is an umbrella term for various technologies that provide the server push functionality. These technologies may be broken into three categories (Bozdag, Mesbah, & van Deursen, 2007): polling, streaming, and piggybacking. Polling is basically a simulation technique where the client continuously sends AJAX requests to the server, ensuring there is always a connection the server could use. Polling techniques are the easiest to implement, but they impose a substantial challenge with respect to scalability. Streaming involves a long-lived HTTP connection, which is often achieved with the help of an invisible, chunked IFrame. Currently available streaming techniques belong to the realm of hacks and are extremely unreliable—especially, when a nomadic user is in play. Piggybacking is an even worse technique that exploits an unrelated server message to piggyback a push notification. It is expected that HTML 5 will realise the so-called web sockets, thus providing an elegant solution for the push notifications challenge. Alas, as far as Comet implementation goes at present, there are the following options: HTTP streaming, hidden IFrame, long-polling, and short-polling.

HTTP streaming features a single persistent connection to the server. Each time the server sends a new event, the browser interprets it; neither side closes the connection.

Even though HTTP 1.1 has brought improvements in terms of long-lived connections, this option is unreliable—in particular, when a mobile device is in play.

Hidden IFrame is a technique where an invisible IFrame is sent as a chunked block and, thus, is declared infinite. When events occur, <*script*> blocks are sent by the server and executed right away on the client side. This is an error-prone and bulky solution that may be considered a hack. In addition, the Nokia WRT browser does not support IFrames.

Short-polling is a simple simulation of the dynamic push functionality. In this technique, the client polls the server at a predefined interval using AJAX requests. This method is favourable when updates are likely to happen constantly and frequently and when immediate notification is desirable, but not required. The interval at which the server will be polled must be chosen cleverly. If polling occurs too often, the service may become overloaded. If polling is performed seldom, the relevance of notifications may be reduced.

Long-polling is a mechanism similar to short-polling. However, instead of polling the server at a predefined interval, a more sophisticated technique is employed. The client commences the interaction by sending an AJAX request to the server. The server keeps the connection open until an update is available and only then responds to the client. Once the client receives an update, it immediately sends a new AJAX request to the server. Long-polling is a simple yet very effective mechanism that ensures the updates are instantly propagated to the client. A drawback of this mechanism is that in some cases the connection may remain idle for a while and be terminated, causing an error.

The Spasiba client application employs a context-driven model for Comet that may be classified as long-polling or short-polling depending on a context collection policy. In this context-driven model, the Spasiba service is polled when new, updated context becomes available. It may be classified as short-polling when a timer policy is in place and long-polling—when an order policy is in effect. When a threshold policy is chosen, the dynamic push is initiated by the client; however, to the user it will appear as an update initiated by the Spasiba service.

5.3.3 Notification Policies and Heuristics

The suggestion mechanism used in the Spasiba prototype employs a number of simple notification policies and heuristics to produce recommendations for the client. These policies specify potential information that may be helpful to the user under the circumstances described by the current context and the registered interest of the user. For instance, if the user's registered interest is "food" and the current time is 12:30 PM, send a notification containing the closest cafés and bistros. Unfortunately, because of the limited attributes available for business listings provided by the chosen data sources, sophisticated policies are impossible to implement at the moment. Such a sophisticated policy could involve sales and promotions information, menu items, working hours, Wi-Fi and parking availability, and payment options. All implemented policies are based on time of day, date, weather, device's network status, and exploration of newly opened businesses. Below are presented the notification policies that have been included in the Spasiba prototype:

Registered Interest	Selected Context	Suggestion
Any	Time of day: 12:00 PM -2:00 PM	Closest places serving food
Any	Day of week: Friday Time of day: 8:00 PM -11:00 PM	Restaurants, lounges, and night clubs
Any	Day of week: Saturday, Sunday	Attractions
Any	Weather: Rain/Shower Velocity: < 10 km/h	Places of interest within 5 minutes of walking distance
Any	Network status: Roaming	Attractions
Any	Any	Newly added/opened places of interest
Food	Time since subscription: > 2 hrs	Closest stores & attractions
Attractions	Time since subscription: > 2 hrs	Closest places serving food

In case two or more policies may apply at the same time, the suggestions are provided for both policies, unless there is an explicit conflict. For instance, if the user's registered interest was "attractions", three hours have passed since the subscription, it is raining, the

user is walking, and network status is roaming, deliver places serving food within a 5 minute walking distance. In the example, three policies apply: one for expiration of the registered interest, one for the network status, and one—for the current weather conditions. Two of these three conflict with each other because one assumes that "attractions" is an improbable interest (as three hours have passed since the user was interested in it), whereas the other produces a suggestion involving attractions (because network status is roaming). Such conflicts are resolved with the help of a meta-policy that dictates: "If the user's registered interest has expired, provide suggestions only from other categories." Obviously, all of the mentioned policies are somewhat basic and they may not suit the needs of some users—they are rather rules of thumb that set a foundation for future investigation.

5.4 Limitations

With respect to the topics discussed in this chapter, four main known limitations may be stated: HTML may not be suitable for results delivery to other platforms, suggestion mechanisms are too basic, push functionality is realised in an inelegant manner, and transport of subscription requests is inconsistent.

The Spasiba service delivers all results in display-ready HTML. This is a great solution when a Nokia WRT widget is the client application, since the received HTML can be immediately displayed to the user—there is no overhead for parsing and adding mark-up. However, if new platforms are added (such as iPhone OS or Android), HTML may not be suitable anymore. A clever solution will be to produce results in XML uniformly and introduce an adaptation policy that will translate the results to HTML only if client application is a widget.

The suggestion mechanism is clearly trivial. One of the goals of the Spasiba prototype was to investigate how all components of the system would work in unison. Thus, little attention was paid towards sophisticated recommender algorithms. As noted earlier, all of the employed policies are rules of thumb that may not satisfy some users. In future work, the emphasis will be placed on introducing a persistent user history along with clever recommender algorithms.

The server-initiated push notifications are delivered to the client with the help of such techniques as short-polling and long-polling. Both of these are inelegant solutions that impose a substantial scalability challenge. At the moment, the only way to tackle this challenge is to use a specialised web server, such as Caucho or Jetty. In the near future, HTML 5 is said to realise the support for web sockets that will enable web applications to maintain bidirectional communications with server-side processes.

Current implementation of the Spasiba prototype employs a different way to identify each of the subscription operations. The *subscribe* request is delivered via an action indicated in the Spasiba envelope. The *unsubscribe* request is implicit—it is recognised when a new *get filter* request is received by the service. And, the *modify subscription* request is recognised via the assessment of a context update. In order to render these requests consistent, all of them should be delivered via a Spasiba envelope action. That is, the Spasiba envelope should be extended to include two more actions: *unsubscribe* and *modify subscription*.

5.5 Summary

The implemented interaction model is comprised of two modes: static and dynamic. In the static mode, Spasiba operates as a regular local search engine by merely answering a user's request. In the dynamic mode, the client subscribes to the Spasiba service and receives push notifications as its context changes. In terms of realisation, both modes employ the Spasiba envelope for transport and coordination of requests. Unlike the static mode, the dynamic mode presents a number of implementation challenges. These challenges are tackled by means of publish/subscribe model for managing clients, Comet for handling push notifications, and rules of thumb—for producing suggestions. Although significant effort was invested into the realisation of the interaction model, there are several opportunities for improvement.

Chapter 6 Content Generation

This chapter presents the process of content generation. The discussion begins with an overview of the data sources selected as information providers. In order to diversify the results and supply as many listings as possible, two types of data sources are employed: web services with open APIs and regular web pages. Then, the reader's attention is drawn to the Semantic Web Services Broker, IRS-III. Next, the chapter examines the content refinement and merging techniques. The discussion is concluded with the known limitations and a summary.

6.1 Data Sources

One of the key limitations of current context-aware mobile applications is the incomplete or irrelevant content generation. That is why the utmost effort was applied towards selecting and leveraging the information providers. For a data source to qualify and be included in the Spasiba's web services registry, the following requirements needed to be fulfilled:

- Free access (preferably unlimited)
- Solid reputation
- Low latency/fast response
- No or rare downtime
- Availability of required topics: food, stores, or attractions
- Presence of location-based search (by address or coordinates)
- Structured nature of listings (each attribute is extractable and accessible)
- Classification attribute for each listing (e.g., restaurant, cafe, store, museum)

After an extensive search, the candidates for inclusion in the web services registry were selected. These candidates met the above requirements to a varying extent, but all of them provided a decent API:

- Local search providers: Google Local, Yahoo! Local, Windows Live Local
- Semantic knowledge bases: DBPedia, True Knowledge
- Reviews: Yelp, Qype

The three local search providers deliver listings of local businesses, similarly to the Yellow Pages. Yelp and Qype are not used for obtaining reviews or ratings, but rather to complement the listings provided by the three local search engines. However, the ratings and reviews will be employed for ranking the results in the future work. Geographically, Yelp covers the North American region, whereas Qype focuses mainly on Europe. Unfortunately, neither the local search nor the reviews providers deliver historical and cultural places of interest. This is the reason why two more data sources are employed. These data sources are both members of the emerging Semantic Web: DBPedia and True Knowledge. With their help, it is possible to investigate what historical or cultural attractions are near a given location. Moreover, thanks to rich semantical annotations of data, advanced queries can be directed to these two sources. Such advanced queries can potentially allow Spasiba to search for specific types of historical or cultural attractions.

As an experiment to expand the food category, I decided to add several static HTML catalogues to the already selected data sources. Of course, in order to convert them to web services, an intermediary needs to be introduced. This intermediary is an adapter that parses the HTML page and exposes its content as a web service. Since the adapter must be aware of the HTML structure of a web page, adding new data sources of this kind is a cumbersome process. Tentatively, Spasiba includes two catalogues that list restaurants situated in British Columbia: FoodPages.ca and BCRFA.ca. Both of these catalogues were chosen because they possessed a multitude of unique listings and their HTML was well-formed, thus simplifying the parsing procedure.

With respect to the structure of listings, each listing is generally comprised of a name, address or location coordinates, and phone number. The majority of selected data sources also provide a classification of the listing. The classification specifies a place’s type that is normalised by the Spasiba engine to one of the three categories: food, store, or attraction. In addition to these attributes, some listings include a subtype—for instance, type of cuisine for restaurants. Initially, one of the key goals of Spasiba was to possess the knowledge of which place of interest are open at a given time. Sadly, I was unable to discover any data sources that were capable of providing working hours for listings. Other attributes that were investigated, but not included in the content generation were Wi-Fi availability, payment options, and parking information.

6.2 IRS-III as a Web Services Broker

Perhaps, the most critical component of the Spasiba Engine Vector is the Semantic Web Services Broker that is represented by an experimental tool, IRS-III, developed at the Knowledge Media Institute of the Open University, United Kingdom. Internet Reasoning Service III (IRS-III) is a framework and infrastructure that supports the creation of semantic web services according to the WSMO ontology. The IRS-III architecture is composed of three main components that communicate via a SOAP-based protocol: server, publisher, and client (Cabral et al., 2006). The server handles requests from the browser, the publishing platforms, and the invocation client. The received requests result in a combination of queries to or changes within the entities stored in the WSMO library. The publisher associates a deployed web service with a WSMO web service description. Within WSMO a web service is granted an interface that contains orchestration and choreography. The client provides a goal-centric invocation mechanism, with the help of which the user simply asks for a goal to be achieved and the IRS-III broker locates the semantic description of an appropriate web service and invokes the underlying web service.

When a web service is published in IRS-III, all of the information necessary to call the service—host, port, and path—is stored within the choreography associated with the web service. Additionally, updates are enforced to the appropriate publishing platform. IRS-III contains publishing platforms to support the publishing of standalone Java and Lisp code and of web services. Web applications accessible as HTTP GET requests (RESTful web services) are handled internally by the IRS-III server.

Figure 27 demonstrates the fashion in which IRS-III is integrated within the Spasiba service architecture. Context Handler and Suggestion Creator synthesise various context parameters, user's registered interest, and derived recommendations into a goal-to-achieve. A goal-to-achieve is described by input parameters, type of desired output, and associated mediators. The IRS-III infrastructure supports all the activities required for fulfilling a goal request. Having accepted a goal-to-achieve, IRS-III will (1) discover a candidate set of web services, (2) select the most appropriate one, (3) resolve any mismatches at the ontological level, and (4) invoke the relevant set of web services satisfying any data, control flow, and invocation requirements. To accomplish this,

IRS-III utilises a set of Semantic Web Service descriptions which are composed of goals, mediators, and web services—all supported by relevant domain ontologies. The web services that are invoked are represented by the data sources described in the previous section. For the two HTML catalogue data sources, an adapter is introduced as an intermediary, with which IRS-III interacts via a SOAP-based protocol. When results are returned from the web services, IRS-III completes the fulfillment of its responsibilities. Then, the obtained results are refined and annotated with HTML mark-up. Content generation is finalised at this point.

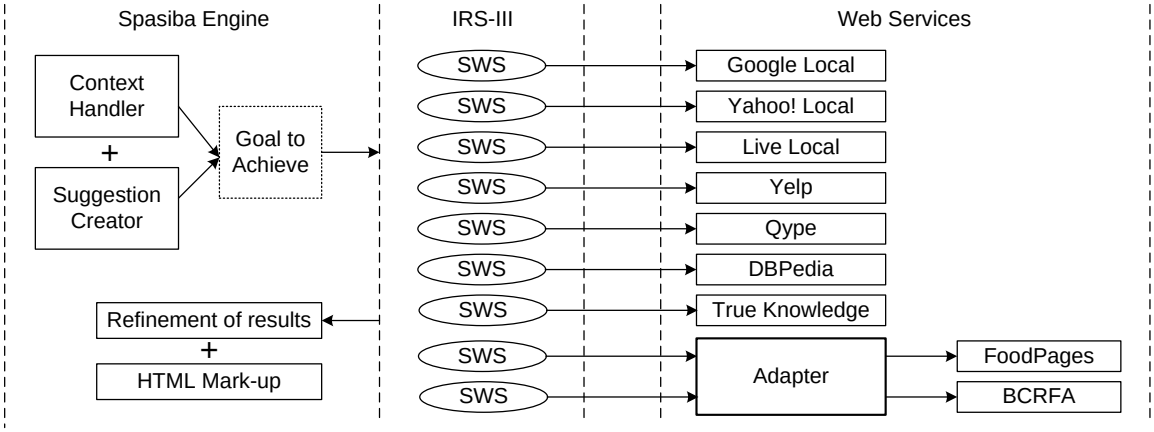

Figure 27. An overview of the process of content generation

Semantic Description of Deployed Web Services

A semantic web service description uses concepts from domain ontologies to represent the types of inputs and outputs of services and in logical predicates for expressing applied restrictions. This description can also include many other aspects, such as orchestration and choreography (Cabral et al., 2006; Hakimpour et al., 2005):

- *Non-functional properties*. These properties can range from information about the provider and the service to execution requirements.
- *Goal-related information*. A goal represents the user perspective of the required functional capabilities.
- *Web service functional capabilities*. They represent the provider perspective of what the service performs in terms of inputs, output, pre-conditions and post-conditions. Pre-conditions and post-conditions are expressed by logical predicates that constrain the state or the type of inputs and outputs.

- *Choreography*. The choreography specifies how to communicate with a web service.
- *Grounding*. The grounding is associated with the web service choreography and describes how the semantic declarations are mapped to a syntactic specification, such as WSDL.
- *Orchestration*. The orchestration of a web service specifies the decomposition of its capability in terms of the functionality of other web services.
- *Mediators*. A mediator defines which elements are connected and which type of mismatches can be resolved between them.

6.3 Refinement of Results

When IRS-III retrieves the results from web services in response to a goal request, a set of raw data is returned to the Spasiba Engine. Before content may be propagated to the client, a number of refinements should take place.

First, the acquired listings are merged in accordance with the following policy: "If two or more listings possess the same address and the same name, they are said to be the same." When listings fall under two different topics (this occurs when suggestions are included), a general rule followed in merging the results is: "Out of ten results, at least six shall fall under the user's registered interest."

Second, listings are sorted by distance and by relevance. How does sorting by relevance work? Highest relevance rank is granted to a listing that has the most attributes and belongs to the topic of the user's registered interest. Listings that appear to be suggestions generated by the Spasiba Engine are given a lower rank.

Third, the content is normalised to adhere to a consistent look. All location coordinates are converted to textual addresses and phone numbers are formatted according to a generic template.

Fourth, required adaptations are enforced in terms of presentation policies. Distances are converted to either feet or metres, depending on the locale settings. A map with marked listings is to be included in the results only if the user is not roaming, the battery is strong, and a fast connection is active.

Finally, the last step prior to submitting the generated content to the client is annotating the listings with HTML tags. The tags merely provide the content with a structural skeleton that can then be employed by the client to style the listings in any desired fashion.

6.4 Limitations

In terms of content generation, known limitations include the following aspects: limited listing attributes, no ranking by reviews, and a small number of data sources.

As was discussed earlier in this chapter, such listing attributes as working hours, Wi-Fi availability, accepted payment methods, and parking information would be extremely valuable additions to the content generation process. Unfortunately, at this time it appears to be problematic (if not impossible), to systematically obtain these attributes for each listing. Perhaps, collaboration with an organisation like BCRFA (British Columbia Restaurants and Food Association) could lead to a partial solution—however, on a global scale this would remain a difficult task until a pervasive tool for recording these attributes emerges.

Another area for improvement lies in the realm of sorting or ranking the listings. Currently, ranking is only performed by distance and by relevance. It would be sensible to provide the user with an ability to sort the listings by rating based on reviews. This feature can be implemented with the help of data retrieved from Yelp and Qype.

Since Spasiba is an experimental prototype targeted at the British Columbian audience, the number of data sources is relatively small. That is why in other geographical areas the content generation may be unsatisfactory. Ideally, several local data sources need to be added for each region.

6.5 Summary

Content generation is a vitally important process that enjoys a significant emphasis in the proposed model. The Spasiba prototype employs nine data sources, two of which are regular HTML pages. Brokering of web services is facilitated by IRS-III, an experimental tool for manipulating deployed web services using semantic annotations. The acquired results are merged, sorted, and enriched with HTML tags before being sent to the client.

Chapter 7 Self-Adaptivity

This chapter discusses the self-adaptive behaviour embedded in the Spasiba prototype. First, the chapter provides an overview of how the self-adaptive features are built-in in the overall Spasiba engine architecture. Then, the reader's attention is directed towards the autonomic policies that govern the operation of Spasiba. This is followed by a discussion of the error-handling features implemented with the help of self-adaptive technologies. Finally, the chapter concludes with a summary.

7.1 Self-Adaptive System

Spasiba's architecture exhibits uncertainty, nondeterminism, and incomplete control of system components. Spasiba operates dynamically by invoking and composing a set of external web services. Because there is little control over these web services, uncertainty in the environment is high. Thanks to the diverse context information, the degree of nondeterminism is significant as well—hardcoded use cases are not an option. Moreover, Spasiba incorporates an external component in its architecture—IRS-III, lowering the control over the system itself. Hence, a self-adaptive solution has been designed and implemented.

The self-adaptive system implemented in the Spasiba prototype is a simple aspect-based architecture with a single autonomic manager and a number of knowledge bases. The inspiration for the realised architecture was prevalently drawn from two papers: a recent paper on applying aspect-oriented solutions in self-adaptive systems (Truyen & Joosen, 2008) and the fundamental paper on autonomic computing by IBM (IBM Corporation, 2006).

The aspect-oriented approach permits to manage a particular concern within an application elegantly. Aspects[1] crosscut an application by means of *pointcuts, advice,* and *inter-type declarations*. Similarly to classes, aspects encapsulate methods, fields, and initialisers as well as crosscutting members. A pointcut signifies a certain join point in the

[1] The Spasiba prototype employs AspectJ for implementation of the aspect-oriented approach. That is why the presented description of aspects and their crosscutting members is relevant to the AspectJ framework only.

program flow. To actually implement crosscutting behaviour, advice is used. Advice brings together a pointcut (to address an execution point) and a body of code (to perform an action at a given join point). Inter-type declarations are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes.

In 2006, IBM introduced the notion of an autonomic element with its inherent MAPE-K loop (IBM Corporation, 2006). An autonomic element is comprised of an autonomic manager, a managed element, and two manageability interfaces. An autonomic manager collects various sensory data from the managed element and augments this data with information from a multitude of knowledge sources accessed via a service bus. Then, the autonomic manager adjusts the managed element, if necessary, with the help of the manageability interface—sensors and effectors—according to the control objective. In the heart of an autonomic manager resides a feedback loop, called Monitor-Analyse-Plan-Execute-Knowledge (MAPE-K). The MAPE-K loop operates in four phases over a knowledge base to assess the current state of the managed element, reason about future states, and enforce the desired behaviour. *Monitor* reads, filters, and stores data gathered from sensors. *Analyse* compares detected events against patterns in the knowledge base; diagnoses and stores symptoms. *Plan* interprets the symptoms and devises a plan to execute. *Execute* realises the devised plan through effectors. In the MAPE-K loop, knowledge is exchanged between all four phases. An autonomic manager maintains its own knowledge and may have access to external knowledge sources.

As Figure 28 illustrates, the implemented self-adaptive system resembles an autonomic element to a great extent. The top part of the figure depicts the managed element, represented by the Spasiba Engine. The Spasiba Engine is comprised of four key components: Publish/Subscribe Manager, Context Handler, Suggestion Creator, and Semantic Web Services Broker. These components along with the Autonomic Manager constitute the so-called Spasiba Engine Vector (SEV). SEV and the four components are elaborated upon in Chapter 3. In addition to the four mentioned components, the top part of the figure also features three knowledge sources that may be leveraged by the autonomic manager: Pub/Sub Registry, Context History, and SWS Registry. Thanks to the aspect-oriented approach in facilitating the communication between the managed

element and the autonomic manager, none of the Spasiba Engine components need to be aware of the fact that they are being monitored and managed.

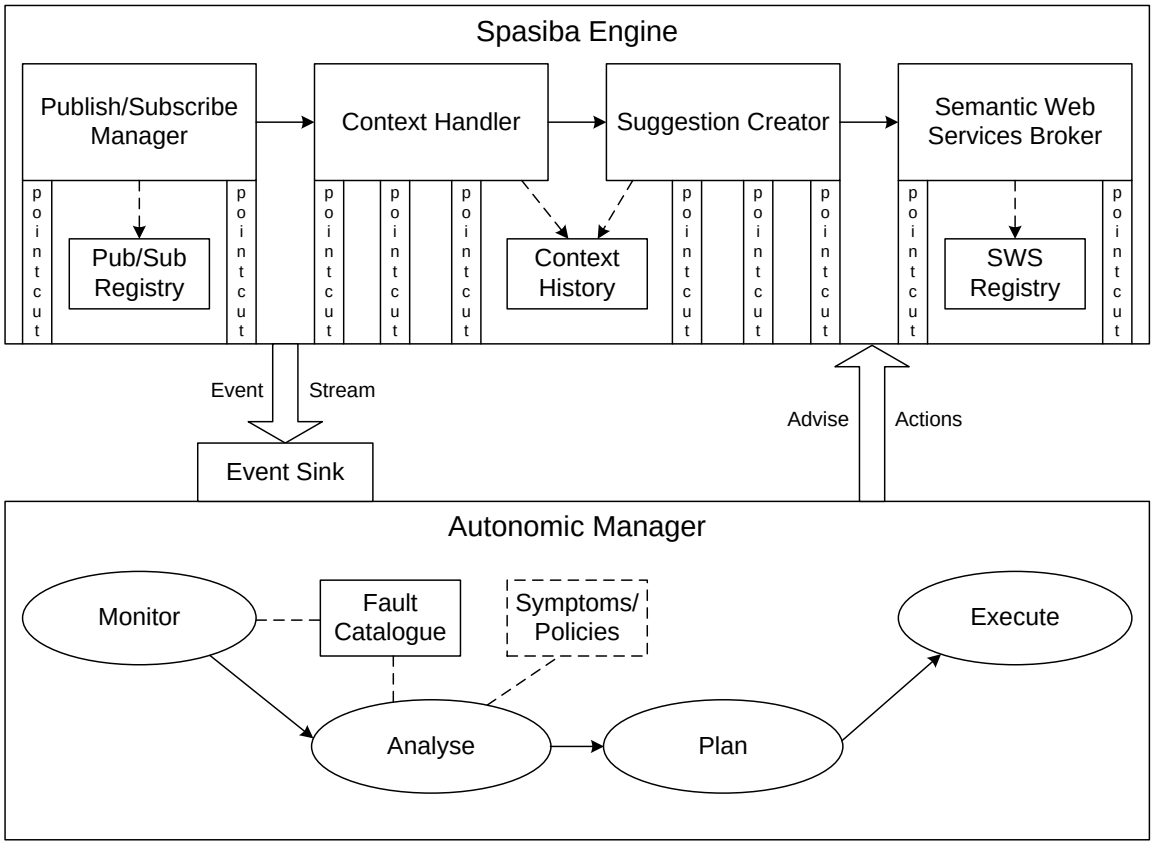

Figure 28. An overview of the self-adaptive system employed in Spasiba

The bottom part of the figure is devoted to the Autonomic Manager that is realised with the help of a variation of the MAPE-K loop. The Autonomic Manager, implemented as an aspect, defines a multitude of pointcuts in each of the Spasiba Engine components. Generally, the pointcuts occur at the beginning and end of core methods and when I/O operations take place. With each pointcut an advice block is associated that directs a status event to the event sink attached to the Autonomic Manager. The monitoring phase examines the event sink and if events of interest are present, they are added to the fault catalogue. Next, the analysis phase assesses the detected events by matching them against predefined symptoms and synthesising them with the already encountered events. At this point, the external knowledge sources—namely, Pub/Sub Registry, Context History, and SWS Registry—may be consulted with as well. An example of such a consultation is when an unknown exception has occurred in the Suggestion Creator and the context

history can provide an insight at what context parameters may have caused it. In the realised autonomic manager, the planning phase is an extension of the analysis, rather than a self-contained phase. The planning phase formulates a number of steps required to stabilise the Spasiba Engine. Often, these steps are formulated automatically in compliance with a hard-coded rule. Finally, the execution phase employs the advice that generated the initial event to effect an action in accordance with the devised plan.

Two key goals of the Autonomic Manager are (1) to enforce autonomic policies (i.e., governance) and (2) to handle errors or exceptions that may arise (i.e., self-healing). An important nuance to note is that, to a certain extent, error-handling is a procedure pertinent to enforcing policies, but its significance is so high that it deserves a separate mention. Below both of these will be discussed.

7.2 Autonomic Policies

Generally speaking, an autonomic policy specifies a set of if-else relationships intended to maintain the desired system behaviour. Autonomic policies are governed by high-level objectives that allow the Autonomic Manager to resolve possible inter-policy conflicts. Autonomic policies employed in the Spasiba prototype attempt to ensure the fulfilment of Quality of Service (QoS) requirements and effective usage of context information.

An essential QoS requirement is responsiveness. That is, Spasiba should never leave a user's request unanswered. For instance, when a critical exception occurs, Spasiba will, at the very least, send an error message to the client, or more likely attempt to restore a previous state from the context history and generate updated results based on the restored state. Another fundamental QoS requirement is performance. In order to ensure satisfactory performance, the back-end components must be well-maintained. Memory-management and recycling of network resources are two aspects that are emphasised in the back-end maintenance of Spasiba.

Effective usage of context information is enforced via autonomic policies as well. Context information is not only useful in generation of fruitful results, but it is also crucial in tailoring the presentation of content to the user's projected needs. Autonomic policies specify what a particular context parameter or a set of context parameters may

indicate. For instance, locale settings permit Spasiba to improve presentation of results by adhering to certain language or cultural canons; battery status, active data connections, and network status help advance the delivery of results onto the device.

7.3 Error-Handling

The Autonomic Manager collects error-reports and exceptions from all of the components of the Spasiba Engine. The errors[2] are classified by origin, severity, and affected application function. In addition, the application may encounter unknown errors—that is, exceptions that do not convey any comprehensible (by the Autonomic Manager) descriptive information. In case an unknown error occurs, the Autonomic Manager consults the knowledge sources and attempts to reverse the application to the last known healthy configuration. Most often, this is implemented by ignoring the new context that may have caused the problem, and restoring a previous context state from the context history.

When expected errors occur, the Autonomic Manager exploits their meta-information to perform matching against the fault catalogue and devise actions required to remedy the affected application function. The meta-information is provided by a thrown exception itself. Origin is specified similarly to a real address: “component, class, method”. For example, an exception that occurred at the context handling stage may have the following origin: “Context Handler, ContextAugmenter, getWeatherInformation”. Severity is assigned on a scale from “Informational” to “Critical”, with “Minor”, “Moderate”, and “Major” in between. The last meta-information parameter, denoting the affected application function, indicates which element of the Spasiba Engine will be impaired if the error is not remedied. An important aspect to note is that the meta-information provided by the error does not state the root cause or the actually impaired application function—the meta-information only conveys locally available facts. That is why error-handling requires analysis and decision-making from the side of the Autonomic Manager.

Analysis of errors involves statistical tools and matching against predefined rules. Statistical tools include basic counts, means, and deviations as well as clustering techniques and probabilities. Matching is performed by means of regular expressions.

[2] Hereafter errors and exceptions are both called errors.

7.4 Summary

Since Spasiba's architecture exhibits uncertainty, nondeterminism, and incomplete control, a self-adaptive solution has been considered and applied. The self-adaptive solution implemented in the Spasiba prototype is a simple aspect-based solution with a single autonomic manager and a number of knowledge bases. The aspect-oriented approach permits to elegantly manage a particular concern within an application. In Spasiba, the aspect-oriented approach facilitates the communication between the managed element and the autonomic manager. As a result, none of the Spasiba Engine components need to be aware of the fact that they are being monitored and managed. Two key goals of the Autonomic Manager are (1) to enforce autonomic policies and (2) to handle errors or exceptions that may arise. An autonomic policy specifies a set of if-else relationships intended to maintain the desired system behaviour in accordance with high-level objectives. When errors occur, the Autonomic Manager exploits their meta-information to perform matching against the fault catalogue and devise actions required to remedy the affected application function.

Chapter 8 Evaluation

This chapter presents a qualitative evaluation (Creswell, 2003) of the implemented Spasiba prototype. This evaluation is performed by means of a case study that addresses the three limitations of current context-aware mobile applications (as discussed in Chapters 1 and 3): (1) low usability, (2) limited usage of context, and (3) incomplete results generation. The chapter begins with the description of the case study. Then, assessment criteria are introduced. This is followed by the evaluation of each mode of interaction (static and dynamic). Finally, the chapter concludes with a discussion of evaluation results and a summary.

8.1 Case Study Description

The Spasiba prototype was devised as an application with varying domains of usage. However, perhaps the key use case was a tourist assistant. To reiterate, currently Spasiba is capable of delivering information of interest under three categories: food, stores, and attractions. In the case study presented in this chapter, a tourist couple arrives to an unfamiliar city with a Nokia S60 5th Edition smartphone. For the sake of clarity, let us assume that an American senior couple (Jim and Angela) arrives by ferry to Victoria, British Columbia with the purpose of sightseeing and spending a wonderful summer weekend. On this occasion, there are no whales observed while the ferry is sailing and the couple is feeling bored. Jim, who is tech-savvy, starts playing with his Nokia smartphone and comes across an interesting application in the Nokia's Ovi Store. This application is called Spasiba and it is said to deliver dining, shopping, and sightseeing suggestions to the user. Since the application is free, Jim decides to give it a try and installs Spasiba. It takes 5 seconds to load the package, because its size is only 160 kilobytes; and, 10 seconds more to install and launch the application. Jim attempts to use the application, but it always claims that no results can be found; he gives up and decides to come back to it after he and his wife set their feet on the island. Once the couple arrives to the ferry terminal and boards a bus, Jim pulls out his smartphone and attempts to use Spasiba once again. To his joy, now Spasiba becomes much friendlier and delivers some results. The couple heads downtown to their hotel, Fairmont Empress. They spend an hour in the

hotel room and then at about noon go out for a stroll around the harbour area. Victoria truly charms the two retirees and they stroll till dusk. The following sections describe Jim's experience using Spasiba, as he and his wife explore Victoria.

8.2 Assessment Criteria

From the point of view of the end user, usability is the key assessment criteria. Usability is a property depending on multiple factors, which gives an overall measure of how an application can be used to achieve specific goals. These factors include (ISO/IEC, 1999):

- *Effectiveness* (degree of accuracy of the reached goal, compared to the planned one),
- *Efficiency* (time and steps required to reach the goal), and
- *User satisfaction* (an overall measure of the user's feelings about the application's usage).

Effectiveness and user satisfaction are generally qualitative measures, whereas efficiency can easily be quantified—for instance, in terms of number of clicks performed.

In the assessment of Spasiba, effectiveness is defined by the accuracy of produced results—be it search results or suggestions. More precisely, in case of suggestions, the effectiveness is measured by the utility or extra value brought to the user. The accuracy of produced results and utility of suggestions are assessed using the following scale: *low, moderate,* and *high.* Effectiveness evaluates the quality of context handling, suggestions creation, and results generation.

In measuring efficiency, the following two quantitative parameters are employed: number of clicks required and time spent to reach a goal. Efficiency addresses the evaluation of the user interface and the communication model.

User satisfaction is measured with the help of the following emotional scale: *disappointed*, *dissatisfied*, *indifferent*, *satisfied*, and *impressed*. User satisfaction is a composite factor that evaluates all components and processes of Spasiba with a single subjective measure. For instance, the adaptive behaviour of Spasiba is a feature that is intended to increase the user satisfaction by meeting current or predicting future needs.

8.3 Evaluation of Static Interaction

When in the hotel room, Jim and Angela decide to find a cozy Italian restaurant for a light lunch. Jim uses Spasiba to assist them with finding a restaurant nearby. However, on the way to the restaurant, Angela recalls that she had left behind her medication in the hotel room in Vancouver (where they stayed for a day). Jim again employs Spasiba, now to search for a nearby pharmacy. Below is the evaluation of the Spasiba's performance with respect to the two requests imposed by Jim.

Notes

- Actual results include a map and more precise distance values.
- The number of clicks does not include the clicks needed to launch the application.
- Time spent to achieve a goal is counted from the first click on the input field.

Goal I

Find a nearby Italian restaurant.

Context

- *Date and time*: 11:45 AM, Saturday, July 11, 2009
- *Language*: EN-US
- *Battery*: 67%, not charging
- *Location*: 48.4218, -123.3680, 0 km/h
- *Network status*: Roaming
- *Data connectivity*: 3G
- *Distance*: Walk
- *Mode of interaction*: Static
- *Search string*: Italian restaurant

Results

- The Old Spaghetti Factory, 703 Douglas Street, 200m, (250) 381-8444
- True North Gelato, 910 Government St, 200m, (250) 383-5303
- Pagliacci's, 1011 Broad Street, 300m, (250) 386-1662
- Valentino's Fresh Pasta & More, 1002 Blanshard Street, 400m, (250) 386-3223
- Fiamo Italian Kitchen, 515 Yates Street, 600m, (250) 388-5824
- Il Terrazzo, 555 Johnson Street, 600m, (250) 361-0028

- Topos Ristorante Italiano, 1218 Wharf Street, 600m, (250) 383-1212
- Zambri's, 110-911 Yates St, 800m, (250) 360-1171

Evaluation

Effectiveness: High. Jim felt that all of the results were useful, except for True North Gelato, which seemed more appropriate for dessert and not lunch.

Efficiency: 3 clicks, 27 seconds.

User satisfaction: Satisfied. Jim's goal was achieved quickly and with little effort. He was able to book a table at Fiamo Italian Kitchen by placing a call right from the widget. However, Jim did not like the fact that one of the listings, Il Terrazzo, was actually closed for lunch on the weekends—he thought it would be great if Spasiba could return only places that are open at the moment.

Goal II

Find a nearby pharmacy.

Context

- *Date and time*: 12:20 PM, Saturday, July 11, 2009
- *Language*: EN-US
- *Battery*: 62%, not charging
- *Location*: 48.4242, -123.3680, 4 km/h
- *Network status*: Roaming
- *Data connectivity*: 3G
- *Distance*: Walk
- *Mode of interaction*: Static
- *Search string*: Pharmacy

Results

- Rexall Drug Stores, 649 Fort Street, 100m, (250) 384-1195
- Shoppers Drug Mart, 1222 Douglas Street, 400m, (250) 381-4321
- Peoples Drug Mart, 867 View Street, 500m, (250) 361-3773
- Peoples Drug Mart, 715 Douglas Street, 500m, (250) 388-5824
- View Street Pharmacy, 867 View Street, 600m, (250) 361-3773
- London Drugs Ltd, 911 Yates Street, 700m, (250) 360-0880

Evaluation

Effectiveness: High. All of the generated results were pharmacies or drug stores.

Efficiency: 3 clicks, 31 seconds.

User satisfaction: Satisfied. Jim was able to find a pharmacy just within 100 metres and help out his wife almost without delaying the lunch.

8.4 Evaluation of Dynamic Interaction

Impressed by the service that Spasiba provided, Jim decides to try out the dynamic mode immediately after lunch. Jim enters "attractions", selects the dynamic mode, and waits for something that could catch his interest as he and Angela take a stroll around the harbour.

Notes

- Actual results include a map and more precise distance values.
- The number of clicks does not include the clicks needed to launch the application.
- Time spent to achieve a goal is counted from the first click on the input field.
- Actual context updates feature all context parameters.
- Only two most interesting context updates are described. In reality, context updates occur every 50 metres or 5 minutes.

Goal

Explore the attractions located in the downtown of Victoria.

Initial Context

- *Date and time*: 1:40 PM, Saturday, July 11, 2009
- *Language*: EN-US
- *Battery*: 47%, not charging
- *Location*: 48.4269, -123.3697, 1 km/h
- *Network status*: Roaming
- *Data connectivity*: 3G
- *Distance*: Walk
- *Mode of interaction*: Dynamic
- *Search string*: Attractions

Results

- The Maritime Museum of BC, 28 Bastion Square, 100m, (250) 385-4222
- Ocean Explorations, 602 Broughton Street, 300m, (250) 383-6722
- Victoria Bug Zoo, 631 Courtney Street, 400m, (250) 384-2847
- Miniature World, 649 Humboldt Street, 500m, (250) 385-9731
- Pacific Opera Victoria, 1815 Blanshard Street, 600m, (250) 385-0222
- Pacific Undersea Gardens, 490 Belleville Street, 700m, (250) 382-5717
- Royal British Columbia Museum, 675 Belleville Street, 700m, (250) 356-7226
- Royal London Wax Museum, 470 Belleville Street, 700m, (250) 388-4461
- Victoria Reptile Zoo, 1420 Quadra Street, 700m, (250) 885-9451

Evaluation

Effectiveness: High. All of the generated results were attractions of some sort.

Efficiency: 4 clicks, 38 seconds.

User satisfaction: Indifferent. Jim and Angela did not think any of the suggested attractions were suitable for them.

Context Update I

- *Date and time*: 4:15 PM, Saturday, July 11, 2009
- *Battery*: 27%, not charging
- *Location*: 48.4175, -123.3785, 0 km/h

Results

- Spinnaker Sips, 425 Simcoe Street, 300m, (250) 590-3519
- Heron Rock Bistro, 435 Simcoe Street, 350m, (250) 383-1545
- Crepes N Cream, 201 Menzies Street, 400m, (250) 590-6588
- Cup O Joe, 1-230 Menzies Street, 400m, (250) 380-2563
- The Bent Mast, 512 Simcoe Street, 450m, (250) 383-6000
- Cafe Mulatta, 281 Menzies Street, 450m, (250) 385-9616
- Barb's Place, 310 St. Lawrence St, 500m, (250) 384-6515
- James Bay Tearoom & Restaurant, 332 Menzies Street, 500m, (250) 382-8282
- Ogden Point Cafe, 199 Dallas Road, 700m, (250) 386-8080

Evaluation

Effectiveness: Moderate. Because over two hours have passed since Jim subscribed to "attractions", Spasiba assumed that now having experienced enough attractions, he might be interested in food. The generated results are places serving food.

Efficiency: 0 clicks, N/A.

User satisfaction: Satisfied. Jim and Angela were having some rest on a bench in the MacDonald Park and craving for a cup of coffee with a cake.

Context Update II

- *Date and time*: 6:01 PM, Saturday, July 11, 2009
- *Battery*: 9%, not charging
- *Location*: 48.4247, -123.3653, 4 km/h
- *Weather*: Raining

Results

- Cactus Club Cafe, 1125 Douglas Street, 100m, (250) 361-3233
- The Bay Centre, 1150 Douglas Street, 100m, (250) 952-5690
- Chapters, 1212 Douglas Street, 150m, (250) 380-9009
- Peacock Billiards, 1175 Douglas Street, 150m, (250) 384-3332
- Dolce Vita Coffee Art, 1213 Douglas Street, 200m, (250) 386-7732
- Dutch Bakery & Coffee Shop Ltd, 718 Fort Street, 200m, (250) 385-1012
- Smitty's Family Restaurant, 850 Douglas Street, 250m, (250) 383-5612
- Golden Chopsticks Chinese Restaurant, 627 Fort Street, 300m, (250) 388-3148

Evaluation

Effectiveness: Low. The only new suggestion policy that was applied is the one based on the rainy weather—to only provide results within a 5 minute walking distance. Thus, the generated results included stores and places serving food within 300 metres.

Efficiency: 0 clicks, N/A.

User satisfaction: Indifferent. Jim followed the update, but the generated results did not suit any of his or Angela's desires. Jim was pleasantly surprised at the fact that Spasiba did not load the map and has switched to less frequent updates, specifying that it is saving the device's low battery.

8.5 Discussion

The described case study demonstrated both positive and negative sides of the Spasiba prototype. In the evaluation of the static mode, thanks to a diverse number of data sources and semantic annotations of web services, it can be observed that Spasiba performs excellently. The service produces accurate results in a timely manner, thus leaving the user satisfied. Most often only three clicks are required for a static search. In addition, the auto-suggest feature ensures the input of the search string is as quick and easy as possible. Therefore, in terms of the user interface usability and results generation, the evaluation of the static mode indicates a success. One area for improvement in the results generation is filtering out the places of interest that are closed at a given time; however, in order to implement such a feature, data sources with richer listing attributes are needed.

In the evaluation of the dynamic mode, a number of pitfalls of the Spasiba prototype can be identified. First of all, regular context updates badly drain the battery. Second of all, suggestion mechanisms are so basic that often they cannot produce any useful recommendations. On the positive side, there is some exhibited intelligence that can be noticed by the user: (1) Spasiba assumes that after two hours of exploring attractions, the user may become hungry, (2) if it rains, Spasiba constrains the search distance to 300 metres, and (3) when the battery is low, Spasiba does not load the map and lowers the frequency of context updates. Hence, it can be concluded that leveraging multiple context parameters is indeed instrumental in increasing the intelligence of an application and, consequently, the user satisfaction. Most importantly, this evaluation indicates that clever recommender algorithms and efficient push notification techniques are required for the success of the dynamic mode.

Aside from the case study, I would like to devote a couple of words to the evaluation of the process of adding new data sources. As described in Chapter 6, when a new data source is discovered, the system operator needs to provide a semantic annotation for it and enlist it in the web services registry. This procedure can often be trivial, but the knowledge of a pertinent ontology may be required. Thus, the process of adding new data sources requires learning. The positive aspect is that the new data source is integrated seamlessly at runtime.

8.6 Summary

The evaluation of the Spasiba prototype was performed qualitatively, with the help of a case study. The case study introduced a tourist couple visiting Victoria and using a smartphone with the Spasiba widget installed on it to explore the unfamiliar city. The two interaction modes, static and dynamic, were evaluated separately. The assessment was facilitated by the following usability criteria: effectiveness, efficiency, and user satisfaction. The evaluation of the static mode indicates a success in terms of the user interface and results generation, whereas the evaluation of the dynamic mode points out both negative and positive sides of the Spasiba prototype. Among the negative sides are limited suggestion mechanisms and battery-intensive context updates, while the positive side is denoted by a degree of intelligence fostered by extensive usage of context.

Chapter 9 Project for Students

The implementation of the Spasiba prototype required a colossal effort with respect to learning new technologies and adapting the familiar ones to the mobile development environment. This effort would be vain without sharing at least some experience that I have gained. This chapter presents a small but engaging project for first- and second-year computer science or software engineering students. In this project, students will develop their own context-aware mobile application and connect it to a famous REST-style web service. Most of the chapter is devoted to a tutorial that demonstrates how a simple context-aware mobile application can be created using the Nokia WRT and REST API of the Google Local Search. After the tutorial, the chapter outlines the project requirements.

9.1 Tutorial

9.1.1 Introduction

Goal

Develop a context-aware mobile application that interacts with an existing web service.

Prerequisites

- Some knowledge of HTML, CSS, JavaScript
- Basic understanding of AJAX and HTTP
- Basic understanding of web services

Recommended Software

- Aptana Studio with a Nokia WRT Plug-in

Learning Outcomes

- Understand the design requirements for a touch-based mobile application
- Learn to create a simple mobile application with the help of the Nokia WRT
- Appreciate the importance of context information
- Learn to interface with an existing web service using a REST API
- Practice simple AJAX-based communication

Audience

First- or second-year computer science or software engineering students and all interested individuals who meet the prerequisites.

Assumptions

- Only very basic error-handling is introduced
- Visual enhancements are omitted, but encouraged
- Application will only run on Nokia S60 5th Edition devices and later

Useful Resources

- Forum Nokia Library: http://library.forum.nokia.com
- Forum Nokia Wiki: http://wiki.forum.nokia.com
- Google Local Search API: http://code.google.com/apis/ajaxsearch/documentation
- Complete code for tutorial: http://webhome.csc.uvic.ca/~alessio/LocalSearch.zip

Description

In this tutorial you will create a context-aware mobile application that will deliver local search results to the user. The purpose of the application is to assist the user in finding a place of interest that is nearby.

In order to implement the mobile application, you will use the Nokia Web Runtime (WRT). The Nokia WRT is a browser-based engine that allows a programmer to apply web development skills to create full mobile applications that are simple, powerful, and optimised for mobile devices. The applications that run in the Nokia WRT are called web widgets. Web widgets are lightweight applications created with the help of standard web technologies that are used to create web pages, such as HTML, Cascading Style Sheets (CSS), JavaScript, and Asynchronous JavaScript and XML (AJAX). You will be able to run your widget on an actual Nokia smartphone.

What is context and what are context-aware applications? To paraphrase the definition of Anind Dey (A. K. Dey, 2001), context is any information that can be used to characterise the situation of an entity. Then, context-aware applications are such applications that use context to provide relevant information to the user, where relevancy depends on the user's task. Our simple application will collect only one (but extremely important and popular) context parameter—location coordinates. This will be achieved by the Nokia WRT Platform Services that allow us to access most of the device features.

Having collected the context and user's input, what is next? Next we need to send a query to the information provider. Normally such an information provider is a web

service. In this tutorial we will use the publicly available Google Local Search web service. This web service delivers local business listings in response to a query that involves a search string and a so-called center point (location in which you are searching). Of course, there are many more query parameters, but we will only use these two. Google Local Search provides a REST API that significantly simplifies web service invocation from a JavaScript script. A RESTful web service is often a simple web service implemented using HTTP and the principles of Representational State Transfer (REST). Such a web service exposes a collection of resources that can be accessed and modified using the regular HTTP methods: POST, GET, PUT, or DELETE. We will use AJAX to send a GET request to the Google Local Search web service.

9.1.2 Step-by-Step Tutorial

Preparation

Before starting the development, analyse and define the requirements, scope, and functionality of the widget to ensure efficient operation and a smooth user experience. Design the application for a single purpose and analyse how it can best serve its users. Mobile devices have been designed for use when mobile. Keep the characteristics of mobile devices in mind when you create applications for them.

Usability guidelines for mobile applications

These guideline help you design and develop highly usable mobile applications for devices with varying characteristics, such as screen size and support for input methods:

- Know your users
- Design for small screens
- Design for multiple screen sizes
- Design for changing screen orientation
- Design intuitive ways of moving within widgets
- Design for limited input methods
- Keep response times short
- Save battery time
- Consider network issues
- Remember the processing limits of the device

Development environment

You can use a regular text editor or any Integrated Development Environment (IDE) that is intended for working with web technologies. In order to preview and test the widget, you will need the Nokia Emulator. To make more advanced WRT widgets, it is recommended to install an IDE such as Aptana Studio, Adobe Dreamweaver, or Microsoft Visual Studio. Nokia offers plug-ins for all of these web development environments. The plug-ins provide code auto-complete functionality and, most importantly, a preview possibility, which makes the testing easier and faster (no emulator is needed). Below are provided links for download of the recommended software packages:

- Aptana Studio: http://www.aptana.org/studio/download
- Nokia WRT plug-in: http://tools.ext.nokia.com/wrt/prod/aptana/plugin

Development process

The development process involves a number of steps:

1) Define the purpose and scope of your service and design the widget.
2) Create the widget functionality and all the required component files.
3) Create the widget installation package.
4) Test the widget.

Tutorial

Step 1: Download and install the IDE. Create a project.

1) Download and install Aptana Studio from the link given above.
2) In Aptana Studio, navigate to “Help => Install Aptana Features” and find Nokia WRT Plug-in. Select and install it. Restart Aptana.
3) Now you will create a Nokia WRT project. Go to “File => New => Project”. Select “Nokia Web Runtime => New Nokia Web Runtime Widget => Basic Widget Project”.
4) Type in the name of your widget: “LocalSearch”. Study but leave unchanged all parameters in the wizard. Click “Next” twice and then “Finish”. The project is now created. You can explore it in the “File” view (not the menu, but a sub-screen) under “Projects”.

5) Note each file that is listed in the project. Files marked with a red circle are Aptana helper files that are needed for previewing and testing your widget—thus do not modify or delete these files. All other files are components of your future widget. Let us briefly discuss what they are:

- *Info.plist* is a mandatory file that carries the description of the widget. This is how your *Info.plist* should look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Nokia//DTD PLIST 1.0//EN"
        "http://www.nokia.com/DTDs/plist-1.0.dtd">
<plist version="1.0">
<dict>
    <key>DisplayName</key>
    <string>LocalSearch</string>
    <key>Identifier</key>
    <string>com.LocalSearch.basic.widget</string>
    <key>Version</key>
    <string>1.0</string>
    <key>AllowNetworkAccess</key>
    <true/>
    <key>MainHTML</key>
    <string>index.html</string>
    <key>MiniViewEnabled</key>
    <false/>
</dict>
</plist>
```

- *index.html* is a mandatory file that specifies the layout of the widget. A widget may only have one HTML file. This file can be renamed, provided the name change is noted in Info.plist.
- *basic.css* is an optional file that defines the appearance of the widget with the help of Cascading Style Sheets (CSS). This file may have any name. Any number of CSS files is allowed.
- *basic.js* is an optional file that defines the behaviour of the widget with the use of JavaScript. This file may have any name. Any number of JavaScript files is allowed.

Step 2: Define the layout of the widget.

In this step you will work with *index.html* to describe the layout of the widget. The LocalSearch widget will have five user interface elements: a header with the widget's title, an input field, a search button, a content pane for results, and an exit button. Each element should be enclosed in its own * * and given an *id* attribute, so that it can be easily referenced in CSS and JavaScript.

1) Open *index.html* in Aptana Studio and start adding code inside of the <*body*> element. Let us begin with the widget's title:

```
<div id="title">
    <h1>Local Search</h1>
</div>
```

2) Now add the mark-up for the input field and the search button in one <*div*> element. Note that each of the <*input*> elements has an *id* attribute.

```
<div id="searchForm">
    <form>
        <input type="text" id="searchStr" />
        <input type="button" value="Search!" id="submit" />
    </form>
</div>
```

3) Create an empty <*div*> that will hold the generated results:

```
<div id="results">

</div>
```

4) Finally, add mark-up for the exit button:

```
<div id="closeApp">
    <input type="button" value="X" id="exitButton" />
</div>
```

5) All required layout has now been specified. Switch from source-editing mode to Nokia Web Runtime mode in order to preview the widget. Ensure in the settings of the preview that the selected resolution is 360 by 640. This is how your widget should look like:

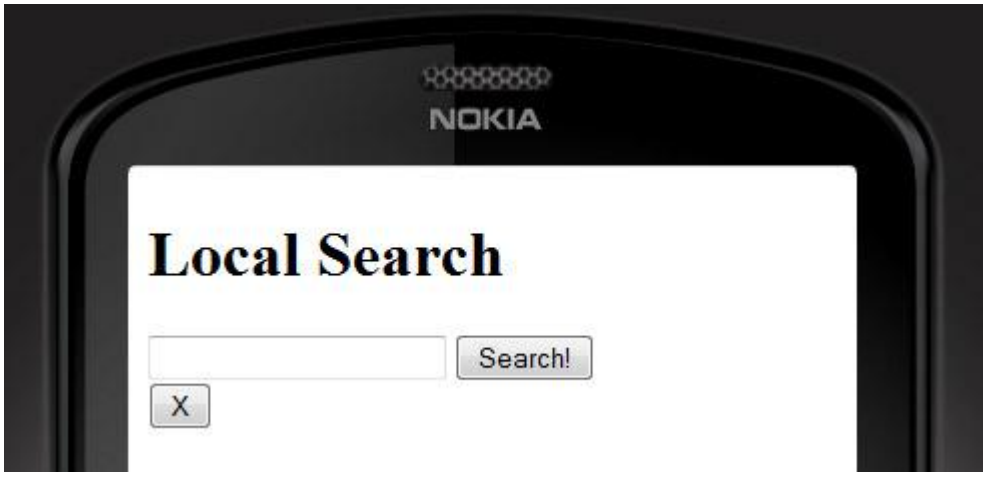

Figure 29. LocalSearch widget with no CSS applied

Step 3: Style the widget with CSS.

The next step after specifying the layout is styling each element of the interface with the help of CSS. First, you will define a few styles for generic HTML elements. Then, you will style and position each element of the layout. You are strongly encouraged to enhance the visual appeal and usability of the widget as you find appropriate.

1) Begin by adding styles for <*body*> and <*html*>. Specify the background of your widget and reset padding and margins under *html*. Style the default font and enforce central alignment of the widget elements under *body*:

```
html {
    background-color: rgb(255, 255, 255);
    padding: 0;
    margin: 0;
}

body {
    font-family: sans-serif;
    font-size: 18px;
    color: rgb(0, 0, 0);
    text-align: center;
}
```

2) Next, specify the left alignment of text for paragraph elements, no outline for input elements, and no outline and no underlining for links:

```
p {
    text-align: left;
}

input {
    outline: none;
}

a {
    text-decoration: none;
    outline: none;
}
```

3) Style your input field and search button. Use the *id* attributes to reference these two elements. Keep in mind the touch-based input and the fact that the user will likely be looking at the screen while walking. Ensure the input field and search button are large enough for an average finger. Adjust the width, height, and margins, in order to accommodate both viewing modes: portrait and landscape. Increase the font-size to match the size of the two interface elements.

```
#searchStr {
    width: 290px;
    height: 30px;
    font-size: 24px;
    margin: 0 5px 10px 5px;
}

#submit {
    width: 296px;
    height: 38px;
    font-size: 24px;
}
```

4) Add substantial margins to the content pane:

```
#results {
    margin: 20px 30px;
}
```

5) Increase the height and width of the exit button to enhance its usability, increase the font size accordingly, and position the button in the top right corner of the screen:

```
#closeApp {
    position: absolute;
    right: 0px;
    top: 0px;
}

#exitButton {
    width: 35px;
    height: 35px;
    font-size: 24px;
}
```

6) Now return to *index.html* and preview your widget. This is how the widget should look like in portrait and landscape mode (both images are cropped):

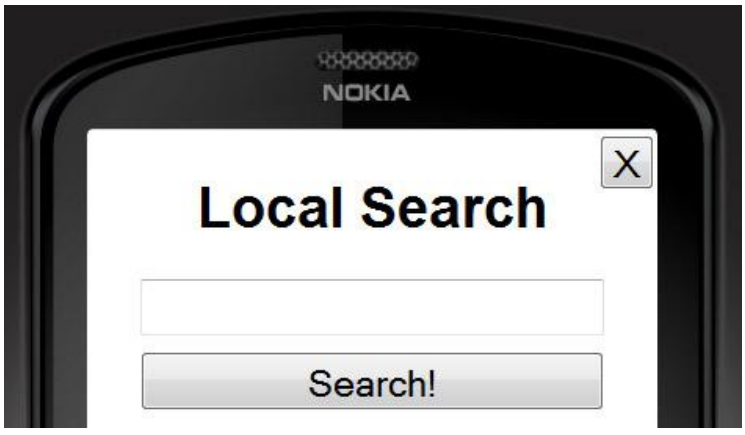

Figure 30. LocalSearch widget with CSS applied in portrait mode

Figure 31. LocalSearch widget with CSS applied in landscape mode

Step 4: Add behaviour to the widget.

The most intricate step in widget development is adding behaviour with the use of JavaScript. In this step you will write functions in *basic.js* that will handle click events, context collection, AJAX communication, and parsing of the search results. Prior to beginning this step, navigate to http://code.google.com/apis/ajaxsearch/signup.html in order to obtain an API key. Also, study the Google Local Search API documentation to find out what query parameters are available and in what format the results are delivered to the client.

1) Before writing any functions, declare and assign two constant variables: one for the API key and one for the URL of the Google Local Search web service. Assign a *window.onload* event handler that will be called when the widget is launched.

```
window.onload = init;

var API_KEY = "ABQIAAAAn3fMadECzH9ocpAzoJ03thSq_tdJI6Rj" +
            + "iCzCFselW7A-XrUyTRRV-gTF9r_m-fdHqba5JIg9Ym3U-w";
var GOOGLE_URL = "http://ajax.googleapis.com/ajax/services/" +
            + "search/local?v=1.0&key=" + API_KEY;
```

2) Next, write function *init* that will be called when the widget is launched. This function should check whether the user's device is supported, hide soft keys that occupy the precious space on the screen, and assign *onclick* event handlers to the two buttons used in the widget.

```
function init()
{
    checkDeviceSupport();

    if(menu != undefined)
        menu.hideSoftkeys();

    document.getElementById("exitButton").onclick = closeApp;
    document.getElementById("submit").onclick = doSearch;
}
```

3) Create function *checkDeviceSupport* that will notify the user and close the widget if the screen resolution is not 360 by 640:

```
function checkDeviceSupport()
{
    if(window.innerWidth != 360 && window.innerWidth != 640) {
        alert("Your device is not currently supported!");
        window.close();
    }
}
```

4) Create function *closeApp* that will handle *onclick* events originating from the exit button. This function will seek the user's confirmation and, if positive, close the widget.

```
function closeApp()
{
    var answer = confirm("Close widget?");
    if(answer)
        window.close();
}
```

5) Create function *doSearch* that will be called when the user presses the search button. This function will retrieve the input string and device's location, build a URL to query the Google Local Search web service, and call another function to send an AJAX GET request.

```
function doSearch()
{
    var query = document.getElementById("searchStr").value;
    if(!query)
        return;
    var point = getLocation();
    if(!point)
        return;

    var url = GOOGLE_URL + "&q=" + escape(query)
            + "&sll=" + point.latitude + "," + point.langitude;
    sendAjaxRequest(null, handleResults, "GET", url);
}
```

6) Next, write function *getLocation* that will retrieve the device's location coordinates using the Nokia WRT Platform Services. You will specify that location information need not be guaranteed—this means that if the location cannot be obtained, the widget will not be blocked. The coordinates will be rounded to four decimals and packaged in a JavaScript object.

```
function getLocation()
{
    var servObj = null;

    try {
        servObj = device.getServiceObject("Service.Location",
                                                    "ILocation");
    }
    catch (ex) {
        return null;
    }

    // Specify that location information need not be guaranteed.
    var updateOptions = new Object();
    updateOptions.PartialUpdates = true;

    // Initialise the criteria for the GetLocation call.
    var trackCriteria = new Object();
    trackCriteria.LocationInformationClass = "GenericLocationInfo";
    trackCriteria.Updateoptions = updateOptions;

    // Attempt to retrieve the location.
    var result = null;
    try {
        result = servObj.ILocation.GetLocation(trackCriteria);
    }
    catch (ex) {
        return null;
    }

    var point = new Object();
    point.latitude = result.ReturnValue.Latitude.toFixed(4);
    point.langitude = result.ReturnValue.Longitude.toFixed(4);
    return point;
}
```

7) One of the most involving and hard functions is *sendAjaxRequest* that will be in charge of performing AJAX communication with the Google Local Search web service. You will write a generic function that can be reused in any other widget or web application. This function will take four arguments: *data*, *callback*, *method*, and *url*. The *data* argument will only be used when method is POST; that is, when method is GET, the first argument will be null. The *callback* argument specifies the function that will be called once the response from the server is received. Since the widget preview in Aptana Studio is based on the engine of Mozilla Firefox, you will have to explicitly enable cross-domain AJAX requests.

```
function sendAjaxRequest(data, callback, method, url)
{
    // Create an XMLHttpRequest object.
    var xhr = null;
    try {
        xhr = new XMLHttpRequest();
    }
    catch(e) {
        callback(null);
        return;
    }

    // Allow cross-domain Ajax requests.
    // Required for testing in Aptana.
    try {
        netscape.security.PrivilegeManager
                        .enablePrivilege("UniversalBrowserRead");
    }
    catch(e) {
    }

    // Open connection.
    method = method.toUpperCase();
    xhr.open(method, url, true);

    // Google Local Search requires a Referer header.
    xhr.setRequestHeader('Referer', 'http://www.csc.uvic.ca');

    // If method is POST, set content request headers.
    if (method == "POST") {
        xhr.setRequestHeader("Content-type",
                                "application/xml;charset=UTF-8");
        xhr.setRequestHeader("Content-length", data.length);
    }

    // Define state change handlers.
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4) {
            if (xhr.status == 200 || xhr.status == 500) {
                if (xhr.responseXML != null)
                    callback(xhr.responseXML);
                else {
                    callback(xhr.responseText);
                }
            }
            else
                callback(null);
        }
    };

    // Send a request.
    xhr.send(data);
}
```

8) The last function that you will write is *handleResults*, which will be invoked when the response from the server is received. This function will accept one argument, *data*, representing the JSON results provided by the Google Local Search web service. The purpose of *handleResults* is to evaluate the received JSON text, parse the listings, and display them in the content pane. Even though listings provided by Google involve many attributes, you will only use title, street address, and phone number. With the help of the special *tel:* directive placed in the *href* attribute of a link, you can make a phone number callable right from the widget.

```
function handleResults(data)
{
    var resultsDiv = document.getElementById("results");
    resultsDiv.innerHTML = "";

    if(data == null)
        resultsDiv.innerHTML = "<p>An error has occurred.</p>";
    else {
        var jsonResults = eval('('+ data +')');
        jsonResults = jsonResults.responseData.results;

        for(var result in jsonResults) {
            var resultHTML = "<p><em>" + jsonResults[result]
                                    .titleNoFormatting + "</em>";

            var address = jsonResults[result].streetAddress;
            if(address)
                resultHTML += "<br />" + jsonResults[result]
                                              .streetAddress;

            var phoneNums = jsonResults[result].phoneNumbers;
            if(phoneNums && phoneNums[0]) {
                resultHTML += "<br /><a href='tel:";
                resultHTML += phoneNums[0].number + "'>";
                resultHTML += phoneNums[0].number + "</a></p>";
            }
            resultsDiv.innerHTML += resultHTML;
        }
    }
}
```

Step 5: Test and deploy the widget.

The last step of the widget development is testing and deployment. You will test the widget in Aptana Studio using the Nokia Web Runtime preview. The retrieved location coordinates will likely be dummy values representing a place in India or Finland. It is a

good idea to hard-code your own values in function *getLocation*, once you have thoroughly tested this function.

Your widget can be deployed to a Nokia S60 5th Edition device, a Nokia Emulator, or to a Nokia Remote Device Access (RDA) device. You can modify the deployment settings in Aptana Studio by right-clicking on your project and selecting "Widget Deployment Settings". This is how the LocalSearch widget looks like on the screen of a Nokia 5800 XM:

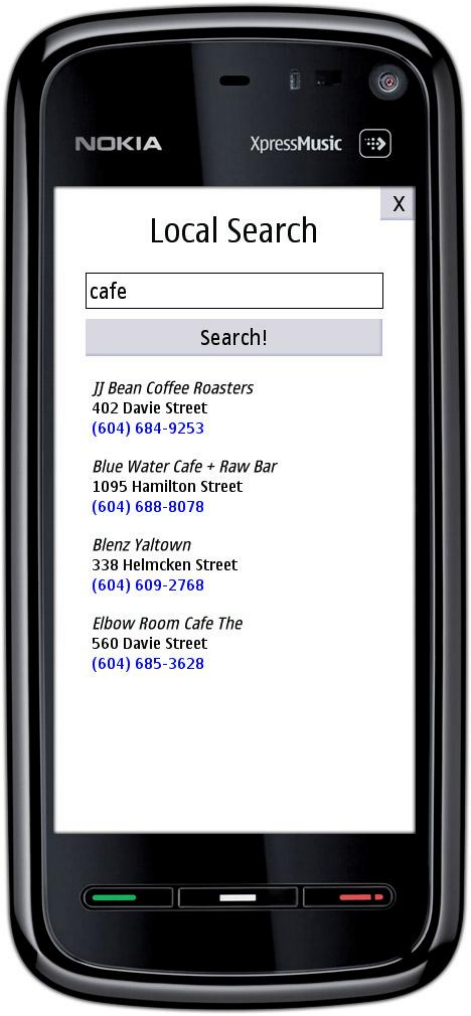

Figure 32. LocalSearch widget with generated results on a Nokia 5800 XM

If you would like to publish and distribute your widget, you will need to package it in a wgz archive. You can achieve this in Aptana Studio by right-clicking on your project and selecting "Package Widget". Alternatively, you can enclose all your component files in a zip archive and change the archive's file extension from *.zip* to *.wgz*.

9.2 Project Requirements

1) Design and describe a context-aware mobile application that would truly help you in your everyday life.

- Model the user-device-environment interaction.
- Identify and select the required context parameters.
- Investigate what publicly available APIs there exist on the Web and identify which ones you could use.

2) Implement your context-aware mobile application. Use the Nokia WRT or any environment you are comfortable with. If you were unable to find a needed web service, consider creating your own web service for a substantial bonus mark.

Bonus: Set up your own Java web service with the help of Axis2.

9.3 Summary

This chapter attempted to share my experience of developing the Spasiba prototype in the form of a tutorial. The beginning computer science or software engineering students should find this tutorial interesting and easy-to-follow. The tutorial demonstrates that a mobile application can be created with regular web technologies and almost no specific knowledge of the mobile platform. With a moderate effort, the project requirements can be met by most of the students. I believe it is of great importance to stimulate the computer science or software engineering students to explore mobile development and, in particular, context-aware mobile applications that interact with the cloud.

Chapter 10 Conclusions and Outlook

This chapter concludes the thesis by summarising its contents, stating the contributions, and outlining future directions.

10.1 Summary

This thesis presented Spasiba, a context-aware mobile advisor, in terms of a theoretical model and concrete implementation details of key functionality aspects. Having identified from a literature survey and personal observations the fact that currently available context-aware mobile applications exhibit significant flaws, I proceeded to formulate my own generic model for such an application. The three major flaws of current context-aware mobile applications that were emphasised can be stated as follows: (1) limited use of context information, (2) incomplete or irrelevant content generation, and (3) low usability. The proposed generic model attempts to tackle these limitations by advancing the usage and manipulation of context information, automating the back-end systems in terms of self-management and seamless extensibility, and shifting the logic away from the client side. This model reflects an abstract representation of a distributed system whose objective is to deliver context-dependent information of interest to a subscribed user armed with a smartphone device proactively. While theoretically valid, the generic model was found to be excessively hard to implement with available resources. That is why a simplified model was created especially for the implementation of the Spasiba prototype. The simplified model downgrades the Distributed Spasiba Engine to a standalone Spasiba Engine and imposes somewhat trivial function on the context history and, thus, the suggestion creation mechanism.

The client application of the Spasiba prototype is a Nokia WRT widget. A widget is a lightweight application that is developed using regular web technologies. The Spasiba widget features a simplistic user interface that is optimised for touch interaction on the go. One of the key goals of the client application is context collection. The widget collects seven thoroughly selected context parameters that are complemented with the values obtained from a dynamically generated filter. These dynamically generated filters are created at runtime in accordance with the category of the user's request. The Spasiba

widget is capable of operating in two modes: static and dynamic. In the static mode, the user submits the request and immediately receives the response—the communication is ended at this point. In the dynamic mode, the user subscribes to the Spasiba service and receives updates as the context changes. The dynamic mode is implemented with the help of Comet-based techniques that simulate the server-initiated push functionality. Feedback loops are employed to provide adaptive functionality in terms of UI adjustments, error-checking, and context monitoring.

Content generation is a vitally important process that enjoys a significant emphasis in the proposed model. The Spasiba prototype employs nine data sources, seven of which are public API web services and two of which are regular HTML pages. Brokering of web services is facilitated by IRS-III, an experimental tool for manipulating deployed web services using semantic annotations. The acquired results are merged, sorted, and enriched with HTML tags before being sent to the client. Spasiba is intended to generate content under three categories: dining, shopping, and sightseeing.

Since Spasiba's architecture exhibits uncertainty, nondeterminism, and incomplete control, a self-adaptive solution has been considered and applied. The self-adaptive solution implemented in the Spasiba prototype is a simple aspect-based solution with a single autonomic manager and a number of knowledge bases. The aspect-oriented approach permits to elegantly manage a particular concern within an application. In Spasiba, the aspect-oriented approach facilitates the communication between the managed element and the autonomic manager. As a result, none of the Spasiba Engine components need to be aware of the fact that they are being monitored and managed. Two key goals of the Autonomic Manager are (1) to enforce autonomic policies and (2) to handle errors or exceptions that may arise. An autonomic policy specifies a set of if-else relationships intended to maintain the desired system behaviour in accordance with high-level objectives. When errors occur, the Autonomic Manager exploits their meta-information to perform matching against the fault catalogue and devise actions required to remedy the affected application function.

The evaluation of the Spasiba prototype was performed analytically, with the help of a case study. The case study introduced a tourist couple visiting Victoria and using a

smartphone with the Spasiba widget installed on it to explore the unfamiliar city. The two interaction modes, static and dynamic, were evaluated separately. The assessment was facilitated by the following usability criteria: effectiveness, efficiency, and user satisfaction. The evaluation of the static mode indicated a success in terms of the user interface and results generation, whereas the evaluation of the dynamic mode pointed out both negative and positive sides of the Spasiba prototype. Among the negative sides were limited suggestion mechanisms and battery-intensive context updates, while the positive side was denoted by a degree of intelligence fostered by extensive usage of context.

My experience of developing the Spasiba prototype is shared in the form of a tutorial. The beginning computer science students should find this tutorial interesting and easy-to-follow. The tutorial demonstrates that a mobile application can be created with regular web technologies and almost no specific knowledge of the mobile platform. With a moderate effort, the project requirements can be met by most of the students. I believe it is of great importance to stimulate the computer science students to explore mobile development and, in particular, context-aware mobile applications that interact with the cloud.

10.2 Contributions

In my opinion, the key contributions of this work are:

1. A model for a context-aware mobile application that is intended to operate over multiple context parameters, varying client platforms, in a semi-autonomic and extensible fashion
2. Proof of feasibility of such a model by means of a fully-functional prototype
3. Application of Semantic Web Services in a mobile setting
4. Empirical proof of the claim that Nokia WRT can be a viable alternative to regular development environments when a context-aware mobile application is in question
5. A tutorial for beginning computer science and software engineering students that encourages them to explore mobile development
6. A simulation and debugging solution for the implemented prototype
7. A point of reference for future research in context-aware mobile applications

10.3 Future Directions

10.3.1 Areas for Improvement

As was stated in the sections discussing known limitations, the utmost effort was applied toward designing and realising Spasiba, yet there are a number of aspects that could be improved. Arranged by priority, these aspects are as follows:

- Keep user history as additional context.
- Enhance the suggestion mechanism.
- Improvements in current client application: fix the socket recycle failure, add results history, and add a draggable map.
- Ranking of results by reviews.
- Ensure the consistency of the transport of subscription requests.
- Add more data sources.
- Output results in XML instead of HTML. Introduce an adaptation to convert results formatted in XML to HTML for Nokia WRT widgets only.
- Support for more client platforms.

10.3.2 Public Release Potential

At the moment, Spasiba is rather a proof-of-concept prototype, than a final product. In order to release Spasiba publicly, a number of substantial enhancements would be required. First of all, these enhancements will include the aspects discussed above. Second of all, Spasiba will require a solid scalable solution for its back-end systems—the implementation of the full-blown generic model presented in Chapter 3 will probably be a good starting point. The proposed generic model introduces the Distributed Spasiba Engine that exhibits advancements in terms of scalability and fault-tolerance. Third of all, special arrangements with content providers will be needed, as most of the open API web services impose constraints on the maximum number of requests per day for a given application.

10.3.3 Commercial Potential

If Spasiba is ever released publicly, monetisation techniques will include contextual advertisements and sponsored listings as well as selling custom-tailored solutions based on Spasiba’s chassis to tour agencies or any interested parties.

Bibliography

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., Steggles, P., et al. (1999). Towards a Better Understanding of Context and Context-Awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing* (pp. 304-307). London, UK: Springer-Verlag.

Amendola, I., Cena, F., Console, L., Crevola, A., Goy, A., Modeo, S., et al. (2004). UbiquiTO: a Multi-Device Adaptive Guide. In *Proceedings of Mobile Human-Computer Interaction – MobileHCI 2004: 6th International Symposium* (Vol. 3160, pp. 13-16).

Arsanjani, A. (2004). Service-oriented modeling and architecture. Retrieved from http://www.ibm.com/developerworks/webservices/library/ws-soa-design1

Baldauf, M., Dustdar, S., & Rosenberg, F. (2007). A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, *2*(4), 263-277.

Balzer, Y. (2004). Improve your SOA project plans. Retrieved from http://www.ibm.com/developerworks/webservices/library/ws-improvesoa

Battle, S., Bernstein, A., Boley, H., Grosof, B., & Gruninger, M. (2005). Semantic Web Services Framework (SWSF) Overview. Retrieved from http://www.w3.org/Submission/SWSF

Becker, C., & Bizer, C. (2008). DBpedia Mobile: A Location-Aware Semantic Web Client. In *Proceedings of the Semantic Web Challenge* (pp. 13-16).

Benslimane, D., Dustdar, S., & Sheth, A. (2008). Services Mashups: The New Generation of Web Applications. *Internet Computing, IEEE*, *12*(5), 13-15.

Berners-Lee, T. (1998). The Semantic Web Road Map. Retrieved from http://www.w3.org/DesignIssues/Semantic.html

Berners-Lee, T. (2000). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web*. Collins.

Berners-Lee, T. (2009). Linked Data - Design Issues. Retrieved from http://www.w3.org/DesignIssues/LinkedData.html

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American*, *284*(3), 34-43.

Blom, J., Chipchase, J., & Lehikoinen, J. (2005). Contextual and cultural challenges for user mobility research. *Communications of the ACM*, *48*(7), 37-41.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., et al. (2004). Web Services Architecture. Retrieved from http://www.w3.org/TR/2004/NOTE-ws-arch-20040211

Box, D., Cabrera, L. F., Critchley, C., Curbera, F., Ferguson, D., Graham, S., et al. (2006). Web Services Eventing (WS-Eventing). Retrieved from http://www.w3.org/Submission/WS-Eventing

Bozdag, E., Mesbah, A., & van Deursen, A. (2007). A Comparison of Push and Pull Techniques for AJAX. In *Proceedings of the 9th IEEE International Workshop on Web Site Evolution* (pp. 15-22). IEEE Computer Society.

Brun, Y., Di, M. S., Gacek, C., Giese, H., Kienle, H., Litoiu, M., et al. (2009). Engineering Self-Adaptive Systems through Feedback Loops. In *Software Engineering for Self-Adaptive Systems* (pp. 48-70). Berlin, Heidelberg: Springer-Verlag.

Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Tanasescu, V., Pedrinaci, C., et al. (2006). IRS-III: A Broker for Semantic Web Services Based Applications. In *Proceedings of the 5th International Semantic Web Conference* (pp. 201-214). Athens, GA, USA: Springer.

Cabral, L., Domingue, J., Motta, E., Payne, T., & Hakimpour, F. (2004). Approaches to Semantic Web Services: an Overview and Comparisons. In *The Semantic Web: Research and Applications* (pp. 225-239). Berlin, Heidelberg: Springer-Verlag.

Canalys. (2009). Smart phones defy slowdown. Retrieved from http://www.canalys.com/pr/2009/r2009081.htm

Corkill, D. D. (1991). Blackboard Systems. *AI Expert*, *6*(9), 40-47.

Cosenza, V. (2009). World Map of Social Networks. Retrieved from http://www.vincos.it/world-map-of-social-networks

Crane, D., & Mccarthy, P. (2008). *Comet and Reverse Ajax: the Next-Generation Ajax 2.0*. Berkeley: Academic Press.

Creswell, J. W. (2008). *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches* (3rd.). Sage Publications, Inc.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S., et al. (2002). Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, *6*(2), 86-93.

Dawson, D., Desmarais, R., Kienle, H. M., & Müller, H. A. (2008). Monitoring in adaptive systems using reflection. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems* (pp. 81-88). New York, NY, USA: ACM.

Dey, A. K. (2000). *Providing Architectural Support for Building Context-Aware Applications*. Doctoral dissertation, Georgia Institute of Technology.

Dey, A. K. (2001). Understanding and Using Context. *Personal Ubiquitous Computing*, *5*(1), 4-7.

Dobson, S., Denazis, S., Fernandez, A., Gaiti, D., Gelenbe, E., Massacci, F., et al. (2006). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, *1*(2), 223-259.

Erl, T. (2009). *SOA Design Patterns (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR.

Farrell, J., & Lausen, H. (2007). Semantic Annotations for WSDL and XML Schema. Retrieved from http://www.w3.org/TR/sawsdl

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-Based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Froehlich, J., Chen, M. Y., Consolvo, S., Harrison, B., & Landay, J. A. (2007). MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services* (pp. 57-70). New York, NY, USA: ACM.

Fujii, K., & Suda, T. (2009). Semantics-based context-aware dynamic service composition. *ACM Transactions on Autonomous and Adaptive Systems*, *4*(2), 1-31.

Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., & Steenkiste, P. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, *37*(10), 46-54.

Hakimpour, F., Sell, D., Cabral, L., Domingue, J., & Motta, E. (2005). Semantic Web service composition in IRS-III: the structured approach. In *CEC '05: Proceedings of the Seventh IEEE International Conference on E-Commerce Technology (CEC'05)* (pp. 484-487). Washington, DC, USA: IEEE Computer Society.

Hakkila, J., & Mantyjarvi, J. (2006). Developing design guidelines for context-aware mobile applications. In *Mobility '06: Proceedings of the 3rd international conference on Mobile technology, applications & systems* (p. 24). New York, NY, USA: ACM.

Heath, T. (2009). Linked Data - Frequently Asked Questions (FAQs). Retrieved from http://linkeddata.org/faq

Hellerstein, J. (2007). Engineering Self-Organizing Systems. In *Self-Organizing Systems*. Berlin, Heidelberg: Springer-Verlag.

Hong, J., & Landay, J. (2001). An Infrastructure Approach to Context-Aware Computing. *Human- Computer Interaction*, *16*, 2-3.

Horrocks, I., Parsia, B., Patel-Schneider, P., & Hendler, J. (2005). Semantic Web Architecture: Stack or Two Towers? In *Proceedings of the 3rd International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)* (pp. 37-41). Springer.

Hosch, W. L. (2008). Britannica Online Encyclopedia: Smartphone. Retrieved from http://www.britannica.com/EBchecked/topic/1498102/smartphone

IBM Corporation. (2005). IBM SOA Foundation: providing what you need to get started with SOA. Retrieved from ftp://ftp.software.ibm.com/.../SOA_g224-7540-00_WP_final.pdf

IBM Corporation. (2006). An Architectural Blueprint for Autonomic Computing. Retrieved from http://www.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

ISO/IEC. (1999). *13407 Human-Centred Design Processes for Interactive Systems*. ISO, Geneva, Switzerland.

Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media.

Kapitsaki, G. M., Prezerakos, G. N., Tselikas, N. D., & Venieris, I. S. (2009). Context-aware service engineering: A survey. *Journal of Systems and Software*, *82*(8), 1285-1297.

Keidl, M., & Kemper, A. (2004). A Framework for Context-Aware Adaptable Web Services. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (p. 826–829). Heraklion, Crete, Greece: Springer.

Kephart, J. O., & Chess, D. M. (2003). The Vision of Autonomic Computing. *Computer*, *36*(1), 41-50.

Kramer, J., & Magee, J. (2007). Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering 2007* (pp. 259-268). Washington, DC, USA: IEEE Computer Society.

Martin, D., Burstein, M., Lassila, O., Paolucci, M., Payne, T., Sycara, K., et al. (2004). OWL-S: Semantic Markup for Web Services. Retrieved from http://www.w3.org/Submission/OWL-S

Mejias, B., & Vallejos, J. (2007). Implementing Self-Adaptability in Context-Aware Systems. In *Proceedings of the 6th ECOOP Workshop on Multiparadigm Programming with OO Languages (MPOOL 2007)*. Berlin, Germany.

Müller, H., Kienle, H., & Stege, U. (2009). Autonomic Computing: Now You See It, Now You Don't: Design and Evolution of Autonomic Software Systems. In A. De Lucia & F. Ferrucci, *International Summer School on Software Engineering (ISSE) 2006–2008* (pp. 32-54). Berlin, Heidelberg: Springer-Verlag.

Müller, H., Pezzè, M., & Shaw, M. (2008). Visibility of control in adaptive systems. In *ULSSIS '08: Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems* (pp. 23-26). New York, NY, USA: ACM.

Nokia. (2008a). S60 5th Edition: What's New for Developers. Retrieved from http://www.forum.nokia.com/S60_5th_edition_Whats_New_In_5th_Edition.html

Nokia. (2008b). S60 Platform: Introductory Guide. Retrieved from http://www.forum.nokia.com/S60_Platform_Introductory_Guide_v1_6_en.pdf.html

Nokia. (2009). Forum Nokia Library - Using Platform Services. Retrieved from http://www.forum.nokia.com/library/platform_services.html

Nokia Press Release. (2008). Nokia to acquire Symbian Limited to enable evolution of the leading open mobile platform. Retrieved from http://www.nokia.com/press/press-releases/showpressrelease?newsid=1230415

O'Reilly, T. (2005). What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. Retrieved from http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

Overgaard, M. (2006). Introspection in Science. *Consciousness and cognition*, *15*(4), 629-633.

Payne, T., & Lassila, O. (2004). Guest Editors' Introduction: Semantic Web Services. *Intelligent Systems, IEEE*, *19*(4), 14-15.

Richardson, L., & Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media, Inc.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., et al. (2005). Web Service Modeling Ontology. *Applied Ontology*, *1*(1), 77-106.

Schilit, B., Adams, N., & Want, R. (1994). Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*. Santa Cruz, CA, US.

Schmidt, A. (2003). *Ubiquitous Computing - Computing in Context*. Doctoral dissertation, Lancaster University. Retrieved from http://www.comp.lancs.ac.uk/~albrecht/phd

Schmidt, A., Aidoo, K. A., Takaluoma, A., Tuomela, U., Laerhoven, K. V., de Velde, W. V., et al. (1999). Advanced Interaction in Context. *Lecture Notes in Computer Science*, *1707*, 89-94.

Schmidt, A., Beigl, M., & Gellersen, H. (1999). There is more to context than location. *Computers & Graphics*, *23*(6), 893-901.

Truyen, E., & Joosen, W. (2008). Towards an aspect-oriented architecture for self-adaptive frameworks. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software* (pp. 1-8). New York, NY, USA: ACM.

Verkasalo, H. (2009). Contextual patterns in mobile service usage. *Personal Ubiquitous Computing*, *13*(5), 331-342.

Wang, H., Huang, J. Z., Qu, Y., & Xie, J. (2004). Web services: problems and future directions. *Web Semantics: Science, Services and Agents on the World Wide Web*, *1*(3), 309-320.

Weiser, M. (1991). The computer for the 21st century. *Scientific American*, *265*(3), 94-104.

Weiser, M. (1993). Ubiquitous Computing. *Computer*, *26*(10), 71-72.

Weissenberg, N., Gartmann, R., & Voisard, A. (2006). An Ontology-Based Approach to Personalized Situation-Aware Mobile Service Supply. *GeoInformatica*, *10*(1), 55-90.

Wright, A. (2009). Get smart. *Communications of the ACM*, *52*(1), 15-16.

Zepeda, J. S., & Chapa, S. V. (2007). From Desktop Applications Towards Ajax Web Applications. In *Proceedings of the 4th International Conference on Electrical and Electronics Engineering* (pp. 193-196). Mexico City.

Appendix A Glossary

3.5G	High-Speed Downlink Packet Access (HSDPA)
3G	Third Generation is a family of standards for mobile telecommunications, which includes UMTS, WCDMA, and several others. 3G allows simultaneous use of speech and data services at a high speed.
AC	Autonomic Computing
ACRA	Autonomic Computing Reference Architecture
A-GPS	Assisted Global Positioning System is a carrier network dependent system which can, under certain conditions, improve the start-up performance of a GPS satellite-based positioning system.
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ARM	A 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) developed by ARM Limited.
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DAML-S	DARPA Agent Markup Language for Services
DARPA	Defence Advanced Research Projects Agency
DCOM	Distributed Component Object Model
DOM	Document Object Model
EDGE	Enhanced Data rates for GSM Evolution
EPOC	A family of graphical operating systems developed by Psion for portable devices, primarily PDAs.
GPRS	General Packet Radio Service
GPS	Global Positioning System
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output

IDE	Integrated Development Environment
IMEI	International Mobile Equipment Identity
IP	Internet Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
LCD	Liquid Crystal Display
MAC	Media Access Control
MAPE-K	Monitor-Analyse-Plan-Execute-Knowledge
MIDlet	A Java application framework for the Mobile Information Device Profile (MIDP) that is typically implemented on a Java-enabled cell phone.
MMS	Multimedia Messaging Service
OS	Operating System
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
PDA	Personal Digital Assistant
PIM	Personal Information Management
POSIX	Portable Operating System Interface for Unix
QoS	Quality of Service
RAM	Random Access Memory
RDF	Resource Description Framework
RDF-S	Resource Description Framework Schema
REST	Representational State Transfer
RIF	Rule Interchange Format
S60	Series 60 is a software platform for mobile phones that runs on Symbian OS. S60 is currently amongst the leading smartphone platforms in the world.
SAWSDL	Semantic Annotations for WSDL
SDK	Software Development Kit
SEV	Spasiba Engine Vector
SIP	Session Initiation Protocol
SMS	Short Messaging Service

SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SWS	Semantic Web Services
SWSF	Semantic Web Services Framework
SWSL	Semantic Web Services Language
SWSO	Semantic Web Services Ontology
UDDI	Universal Description, Discovery, and Integration
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
Wi-Fi	Wireless Fidelity is a brand assigned to a class of wireless local area network (WLAN) devices based on the IEEE 802.11 standards.
WRT	Web Runtime
WSDL	Web Service Definition Language
WSMO	Web Service Modelling Ontology
WSRF	Web Services Resource Framework
XML	Extensible Markup Language