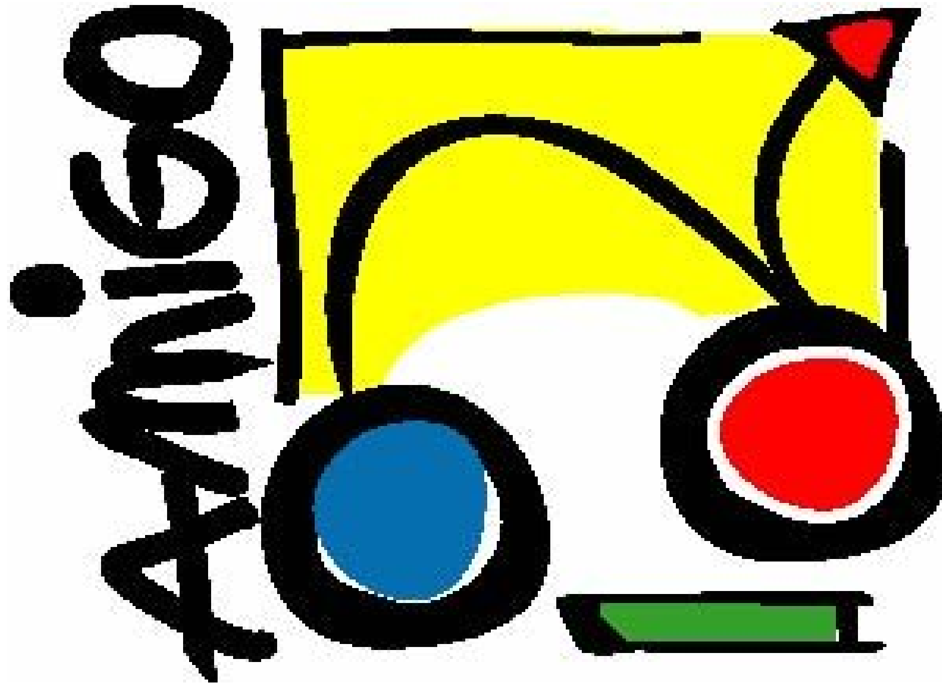


IST Amigo Project
Deliverable D3.5

Amigo overall middleware: Final prototype implementation & documentation

Annex: Qualitative & quantitative assessment of the middleware

IST-2004-004182
Public

Project Number	:	IST-004182
Project Title	:	Amigo
Deliverable Type	:	Report

Deliverable Number	:	D3.5
Title of Deliverable	:	Amigo overall middleware: Final prototype implementation & documentation – Annex: Qualitative & quantitative assessment of the middleware
Nature of Deliverable	:	Public
Internal Document Number	:	assesment_annex_amigo_d3.5_final
Contractual Delivery Date	:	30 November 2007
Actual Delivery Date	:	31 December 2007
Contributing WPs	:	WP3
Editor(s)	:	**INRIA:** Graham Thomson, Nikolaos Georgantas
Author(s)	:	**FT:** Anne Gérodolle, Mathieu Vallée **ICCS-NTUA:** Ioanna Roussaki, Dimitris Tsesmetzis, Yiannis Papaioannou, Miltiades Anagnostou **IMS:** Edwin Naroska **INRIA**: Graham Thomson, Sébastien Bianco, Nikolaos Georgantas, Sonia Ben Mokhtar, Valérie Issarny, Nicolas Palix, Charles Consel, Laurent Réveillère, Wilfried Jouve **Microsoft:** Ron Mevissen, Stephan Tobies, Rich Hanbidge **TELIN:** Pravin Pawar, Remco Poortinga **TID:** José María Miranda, Álvaro Ramos, David Cordón, Andrés Tuells **VTT:** Jarmo Kalaoja, Ilkka Niskanen, Toni Piirainen

Abstract

This document presents the assessment annex of the deliverable "D3.5 Amigo overall middleware: Final prototype implementation & documentation – Final integrated methodology ('how to') for employing the middleware". Included are a number of assessment results addressing the middleware components developed within WP3. We have performed a thorough assessment of the middleware, both qualitative – based on self-evaluation and a survey carried out among internal Amigo developers, and quantitative – based on experimental tests.

Keyword list

Assessment, ambient intelligence, networked home system, interoperability, mobile / personal computing / consumer electronics / domotic domain, semantic concept, ontology, service description vocabulary, service description language, semantic reasoning, service matching, service composition, service adaptation, service execution, middleware, service discovery protocol, service interaction protocol, programming and deployment framework, context, quality of service, multimedia streaming, content distribution, security, data storage.

Table of Contents

Table of Contents 2

1 Introduction 4

2 Amigo API Assessment Survey 5

3 Programming and Deployment Frameworks Assessment 6

3.1 Development, configuration, deployment, management aspects 6

3.1.1 Interoperability 6

3.1.2 Portability 6

3.2 Runtime aspects 6

3.2.1 Performance 6

3.2.2 Discovery 6

3.2.3 Service interaction 7

3.2.4 Results 8

3.2.5 Conclusion 10

4 Interoperability Framework Assessment 11

4.1 Development, configuration, deployment, management aspects 11

4.1.1 Service Discovery 11

4.1.2 Service Interaction 12

4.2 Runtime aspects 14

4.2.1 Service Discovery 14

4.2.2 Service Interaction 16

5 Security Framework Assessment 20

5.1 Development, configuration, deployment, management aspects 20

6 Semantic Service Framework Assessment 21

6.1 Complex service workflows 21

6.1.1 Development, configuration, deployment, management aspects 21

6.1.2 Runtime aspects 22

6.1.2.1 Experimental set-up 23

6.1.2.2 Discovery time measurements 23

6.1.2.3 Composition Time Measurements 25

6.2 Context Aware Services 27

6.2.1 Development, configuration, deployment, management aspects 27

6.2.1.1 Ease of learning 27

6.2.1.2 Additional development effort 27

6.2.1.3 Tooling 28

6.2.2 Runtime aspects 29

6.2.2.1 Performance 29

6.2.2.2 Resource consumption ... 31
6.2.2.3 Scalability ... 31
6.2.2.4 Persistent lookup ... 32
6.2.3 Conclusions ... 32
6.3 Quality of Service Aware Services ... 32
6.3.1 Development, configuration, deployment, management aspects ... 32
6.3.2 Runtime aspects ... 34
6.4 Event-based Services ... 34
6.4.1 Development, configuration, deployment, management aspects ... 34
7 Multimedia Content Framework Assessment ... 36
7.1 Development, configuration, deployment, management aspects ... 36
7.1.1 Content Adaptation DMS ... 36
7.1.2 Content Discovery ... 36
7.1.3 Content Distribution ... 36
7.2 Runtime aspects ... 37
7.2.1 Content Adaptation DMS ... 37
7.2.1.1 Performance ... 37
7.2.1.2 Resource Consumption ... 37
7.2.1.3 Scalability ... 37
7.2.1.4 Robustness ... 37
7.2.2 Content Discovery ... 38
7.2.2.1 Performance ... 38
7.2.2.2 Resource Consumption ... 38
7.2.2.3 Scalability ... 38
7.2.2.4 Robustness ... 38
8 Datastore Framework Assessment ... 39
8.1 Development, configuration, deployment, management aspects ... 39
9 Assessment of Home Configuration with VantagePoint ... 40
9.1 Development, configuration, deployment, management aspects ... 40
9.1.1 Introduction ... 40
9.1.2 Assessment Results ... 41
9.1.3 Discussion ... 45
10 Summary ... 47
11 Resources ... 48

1 Introduction

This document presents the assessment annex of the deliverable “D3.5 Amigo overall middleware: Final prototype implementation & documentation - Final integrated methodology (‘how to’) for employing the middleware”. Included are a number of assessment results addressing the middleware components developed within WP3. We have performed a thorough assessment of the middleware, both qualitative – based on self-evaluation and a survey carried out among internal Amigo developers, and quantitative – based on experimental tests.

A range of aspects are addressed by the assessment of each component. These include:

- Development, configuration, deployment, management aspects, such as: ease of learning and ease of use; the time and effort required for development, and the resulting efficiency; provided aid in dealing with complex, tedious, error-prone development tasks; and additional tool aid in system design and development; and portability/interoperability.
- Runtime aspects, such as: effectiveness; performance; resource consumption; scalability; and robustness.

The elicited assessment approach and target assessment aspects reflect our initial objectives in WP3 and the resulting nature of WP3 software. The development of the Amigo Base Middleware software focused on producing research prototypes that explored novel applications in a new domain, and exhibited extensive interoperability. Therefore, our evaluation effort focused on assessing the capacity of the Amigo middleware to enable promising perspectives, rather than on a complete assessment of a commercial product. Overall, the Amigo middleware software is judged sufficient to meet such expectations.

In the following, Chapter 2 presents the results of an assessment survey completed by internal Amigo developers on their experiences of using the Amigo software API, which includes the Base Middleware and Intelligent User Services API. Thus, this part of our assessment concerns the whole Amigo software. Chapters 3 to 8 present the results of our self-assessment, both qualitative and quantitative, of the components of the Base Middleware. Chapter **Error! Reference source not found.** describes the findings of a component-specific questionnaire conducted to assess the VantagePoint tool. Finally, Chapter 10 presents a summary of the assessment results.

2 Amigo API Assessment Survey

The full text and results of the Amigo API assessment survey completed by internal Amigo developers are included in the delivery of this annex. This chapter provides a summary and analysis of these results. Participants of the survey were asked to complete a questionnaire consisting of 27 questions concerning their experience with using the Amigo Middleware API, which includes the Base Middleware and Intelligent User Services APIs. Thus, this part of our assessment concerns the whole Amigo software. The questionnaire was proposed by Microsoft and reviewed by the rest of the WP3 partners. The survey was carried out with the technical support of Vanguard Software Corporation, an external software company specialized in interactive Web applications for e-business and desktop tools for quantitative analysis. This company also processed the input data of the survey and provided the full report document included in the delivery of this annex.

Over one third of the respondents used the Amigo Middleware to build an application or demo, while the two thirds used the middleware to build both a middleware component and to build an application or demo. Two thirds of the respondents had previous experience in distributed programming, while just under one third claimed to have very little previous experience.

Concerning the overall experience of using the Amigo Middleware, the feedback was positive, with 62% of the respondents rating the experience to be as expected, and the remaining 38% better than expected. Furthermore, the majority of respondents reports that they used just the right amount of classes, that the classes were at the expected level of abstraction, and that they were satisfied with the experience of learning how to use the different classes.

When asked about performing tasks with the Amigo Middleware, the majority of respondents replied that they felt they had to do an expected number things in order to accomplish the task, and that they had to keep track of additional information, such as writing something down on paper or committing something to memory, a reasonable number of times.

Few respondents found that they had to reverse a designing decision using the Amigo Middleware, nor had to remedy the negative consequences related to this. Furthermore, the majority of respondents felt that they only had to understand a reasonable amount of the implementation details of the middleware in order to use it successfully, and that they experienced no cases where they had to make changes to classes they had already written due to the way the middleware was implemented.

A large majority of the respondents found it easy to understand the role of the classes they used, that there was a high level of consistency throughout, and that almost all classes represented concepts they expected.

Almost all of the respondents felt that using the Amigo Middleware saved some or a lot of development time, that implementing an application or service with the middleware was easy or very easy, and that is was easy or very easy to learn how to use the middleware. Furthermore, a large majority (86%) felt that the tool support for the middleware available to them was as or more than expected.

A large majority of the respondent reported that they had no performance, quality, or interoperability issues with the Amigo Middleware, and almost all respondents agree that the Amigo Middleware enabled new scenarios that would not have been possible, or would have been difficult to do, without it.

Overall, this gives a very positive response for the use of the Amigo Middleware.

3 Programming and Deployment Frameworks Assessment

3.1 Development, configuration, deployment, management aspects

3.1.1 Interoperability

Both the OSGi and .NET Programming and Deployment Frameworks have been developed keeping in mind interoperability. The OSGi and .Net tutorials [OSGi/.NET] show how to develop applications in both frameworks, so that an application based on either the .Net or the OSGi framework can discover services running on either the .Net or the OSGi framework, and can interact with the discovered services, that is: place remote calls to the methods that these services expose; as well as subscribe to event sources and receive notifications.

Several applications and middleware components developed in the other Amigo work-packages have helped assessing interoperability.

3.1.2 Portability

Applications developed with the Amigo .Net framework should in principle run on any platform running .NET 2.0 (or .Net 2.0 compact framework). We have successfully checked the portability on various PCs running either Windows XP or Windows Vista, as well as on different PDAs and smart phones.

Applications developed with the Amigo OSGi framework should in principle run on any Java runtime J2SE (>= 1.4) or J2ME/Personal Profile. We have successfully tested the following configurations:

- various PCs or laptops, on Windows XP, Windows Vista or Linux, with the standard Java packages (1 .4, 1.5, 1.6) installed
- PDAs (Windows CE / Pocket PC) running IBM/J9
- NSLU2/ Linux, with the embedded Java runtime JamVM.

3.2 Runtime aspects

3.2.1 Performance

The results presented in this section must be considered with caution, as the performance of a networked application depends on many factors.

3.2.2 Discovery

We have not performed quantitative measurements on the discovery process. As the discovery protocol used in Amigo (WS-discovery) is based on IP Multicast, its “performances” depend highly on the network congestion. In highly congested networks, some services may even not be discovered at all, as there is no guarantee brought by the protocol that multicast packets will be received at all. To reduce the risk of a service not being discovered, the WS-discovery protocol specifies that each WS-discovery message must be sent 3 times with a random delay between each sending.

We have observed problems when running distributed applications on a network with no DNS server. This has lead to improvements in the discovery framework, so that the robustness is increased.

3.2.3 Service interaction

We have made a series of experiments to evaluate the time of a remote call between a client application and an Amigo service on a local network.

We have considered several networking conditions: wired (company network), wired (home network), wireless (home network) or mixed (one machine wired on the home gateway, the other machine on Wifi).

For both .Net and OSGi frameworks, we have developed an Amigo service with 3 methods, and a client that measures the time necessary for calling these methods. Two types of measurements were done: measuring the time between calling the service and receiving the result (this allowing to possibly observing discrepancies between several “identical” calls), and measuring the average time of a series of 10 successive calls.

For example, the code of the OSGi client looked as follows:

```
for (int i = 0; i < 10; i++) {
        System.gc();
        t0 = System.currentTimeMillis();
        library.helloString("hello");
        t1 = System.currentTimeMillis();
        long elapsed = t1 - t0;
        System.out.println("call time=" + elapsed);
}

System.gc();
t0 = System.currentTimeMillis();
for (int i = 0; i < 10; i++) {
        library.helloString("hello");
}
t1 = System.currentTimeMillis();
long elapsed = t1 - t0;
System.out.println("average call time=" + elapsed/10);
```

The server code was kept as simple as possible, so that the processing of the call was negligible (note that these are not realistic conditions for a middleware in use, but we wanted to isolate specifically the middleware).

We have deployed the clients and servers on different platforms, as follows:

	Hardware	OS	OSGi or .Net	Java runtime
PC Linux OSGi	Intel Pentium 4 CPU 3.20 GHz	Linux 2.6.18	OSGi	Sun J2SE 1.4 Sun J2SE 1.5 Sun J2SE 1.6
PC Win OSGi		Windows XP Windows Vista	OSGi	Sun J2SE 1.4 Sun J2SE 1.5 Sun J2SE 1.6
PC XP Net		Windows XP	.Net	n.a.
iPaq	Intel PXA 270 624 MHz (only Wifi)	Windows mobile	OSGi	IBM J9
NsLU2	Xscale-IPX42x 133 MHz (only wired)	Linux 2.6.18	OSGi	JamVM

Other PCs with different hardware characteristics, under Linux or Windows were also tested, with similar results.

3.2.4 Results

Generally, there was no significant difference between the average time of a call to a void method, and that of a method taking a few arguments.

When several Java runtimes were available on a machine, we varied the runtimes used in the tests. We did not notice any change in the results.

The following table summarizes the results (average call time observed during the different trials, in ms):

Client \ server	PC Linux OSGi (wired)	PC Win OSGi (wired)	PC Win OSGi (wifi)	PC Win Net (wired)	PC Win Net (wifi)	iPaq (Wifi)	NsLU2 (wired)
PC_Linux_OSGi (wired)	5-8	8-10	-	8-10	-	-	40
PC_Win_OSGi (wired)	6-10	6-10	8-12	6-10	8-12	50	40-50
PC_Win_OSGi (WiFi)	-	8-12	10-14	8-12	10-14	50	
PC_Win_Net	6-10	300-330	300-330	300-330		300-330	40-60
PC_Win_Net (Wifi)		300-330	300-330	300	300	-	40-60
iPaq (Wifi)		400-500		400-500	400-500	-	-
NsLU2 (wired)	100-120	100-120	-	100-120	-	-	120-150

We were surprised by the high average call time, compared to other test configurations, when using a Windows .Net client and a Windows (.Net or OSGi) server. After investigation, we found that the reason for low performance was related to the packet scheduling strategy on Windows, and particularly to the use of the Nagle algorithm that delays sending small packets. After Microsoft EMIC had made some changes to the client software and the .Net stack, the results were changed as follows:

Client \ server	PC Win Net (original)	PC Win OSGi (original)	PC Win Net (modified stack)
PC Win Net (original)	300	300	-
PC Win OSGi (original)	10-14	10-14	10-14
PC Win Net (modified client)	200	300	2-3

After the correction, the interaction time between .Net platforms was highly decreased. The interaction between a .Net client and an OSGi server on Windows remains the same. A reason could be that the OSGi server closes the connection after each call. However, interactions of OSGi clients with the same server are much faster (< 10ms), as are interactions of the .Net client with OSGi servers running on Linux platforms. At this time, we do not have a satisfactory explanation.

The following tests aim at measuring performances when the payload of the call is big. In the first two tests, we measure the average call time of a method with several parameters. The first parameter is a string containing 1000 or 10000 characters.

The server returns the concatenation of the 3 parameters.

In the third test, a byte array (size 10000) is passed as parameter. The server return is void.

Test with big payload (sting size=1000)				
Client \ server	PC Linux OSGi (wired)	PC Win OSGi (wired)	PC Win Net (wired)	NsLU2 (wired)
PC Linux OSGi (wired)	5	4-7	5-10	100-120
PC Win OSGi (wired)	4	<10	<10	110-125
PC Win Net	-	-	-	
NsLU2 (wired)	145	150-200	145-150	230-280

Test with very big payload (sting size=10000)				
Client \ server	PC Linux OSGi (wired)	PC Win OSGi (wired)	PC Win Net (wired)	NsLU2 (wired)
PC Linux OSGi (wired)	10-30	10-30	10-20	630-660
PC Win OSGi (wired)	10-15	10-15	10-15	640-660
PC Win Net	-	-	-	-
NsLU2 (wired)	680-800	710-7260	710-750	1150-1340

Test with big payload (byte array size=10000)				
Client \ server	PC Linux OSGi (wired)	PC Win OSGi (wired)	PC Win Net (wired)	NsLU2 (wired)
PC Linux OSGi (wired)	10	10-20	20-25	350-390
PC Win OSGi (wired)	15-20	15-20	15-20	380-400
PC Win Net	-	-	-	-
NsLU2 (wired)	780-800	830	860	1100-1120

Although the call times are increased compared with calls of methods with “small” arguments, they remain “low” when executed by clients and servers running on PCs. The call time increases to some 100 ms when clients and server are on nsLU2. In the worst case (string size 10000 characters, both clients and server on nslu2), the average duration was a little more than 1s.

3.2.5 Conclusion

Though these tests are limited to only time measures of method calls, they provide important insights about the performance developers can expect from the Amigo middleware. Also they allowed to spot some issues and to improve the middleware. To summarize:

Interaction between powerful platforms (PCs) is typically below 15 ms, and response time does not depend as much as could have been expected on networking conditions (wired or wireless). We did not make experiments on highly congested network.

Interaction between PCs and resourse-constrained devices is slower (as expected). However, the figures are not symmetrical: the time needed for a client on a constrained platform to call a server on a PC is much higher than the reverse (client on PC, server on constrained platform).

4 Interoperability Framework Assessment

In this chapter, we present our assessment of the Amigo Interoperability Framework that we carried out on two earlier prototypes of the service discovery and service interaction subsystems, as already reported in Deliverables D3.1b and D3.2 respectively. Even if this evaluation does not concern the final Interoperability Framework prototype, the produced results are representative of the qualities of the Interoperability Framework.

4.1 Development, configuration, deployment, management aspects

4.1.1 Service Discovery

The assessment reported herein concerns the first prototype of the service discovery interoperability subsystem of INMIDIO, the Amigo interoperable middleware core, which included a UPnP unit and a SLP unit. Although that prototype was not yet optimised, it was robust enough for assessing the performance of our approach in different use cases. The following discusses key elements of the prototype. We first discuss its small code footprint requirements compared to existing solutions. We then evaluate its performance by comparing supported response times with native service discovery.

Amigo middleware size requirements				
	Size (KB)	**Classes**	**NCSS**	**Overhead**
Core framework	44	15	789	-
UPnP Unit	125	18	1515	-
SLP Unit	49	6	606	-
Total	**218**	**39**	**2910**	-
SDP library size requirements				
OpenSlp Library	126	21	1361	-
Cyberlink UPnP	372	107	5887	-
Total	**498**	**128**	**7248**	-
Size requirements to provide interoperability with and without Amigo middleware				
SLP &UPnP Library + SLP & UPnP clients	514	-	-	-
UPnP client & Library + Amigo middleware	598	-	-	**14%**
SLP client & Library + Amigo middleware	352	-	-	**-31.5%**

Table 4-1: Footprint requirements in KBytes for known libraries and the Amigo middleware core.

The prototype is implemented in Java to take advantage of cross platform portability. We are, in particular, able to deploy our solution on any mobile device that embeds J2ME[1], which provides a Java virtual machine customized for devices with limited resources.

In Table 4-1, we compare the footprint requirements of the Amigo middleware core with the ones of common open-source libraries like *OpenSlp*[2] and *Cyberlink* for Java[3]. The overall Amigo middleware core consists of 39 Java classes and 2910 lines of Non-Commented Source Statement Classes (NCSS). The overall system size is 218 Kbytes. This includes

[1] http://java.sun.com/j2me/index.jsp

[2] http://www.openslp.org/

[3] http://www.cybergarage.org/net/upnp/java/

125Kbytes for the UPnP Unit and 49Kbytes for the SLP Unit. To be interoperable, nodes running UPnP (resp. SLP) applications need to host a native UPnP (resp. SLP) library plus the Amigo middleware core. This is to contrast with an interoperable device that is not equipped with our interoperable system, which needs: (i) to host both the full UPnP stack and the SLP library, and (ii) some engineering effort to develop and host an additional SLP (resp. UPnP) client that is equivalent in terms of functionalities to the UPnP (resp. SLP) client.

As further depicted in Table 4-1, the size requirements of a middleware that needs to be interoperable and does include the Amigo interoperable middleware core (includes both full SLP and UPnP) is 514Kbytes when hosting one simple service. In contrast, the size requirement for a middleware dedicated to UPnP (resp. SLP) equipped with the Amigo middleware core is 598Kbytes (resp. 352Kbytes). Then, the size requirements increase proportionally with the number of hosted services. The size requirements of an interoperable middleware without the Amigo interoperable middleware core increase faster than the ones of a middleware equipped with the Amigo interoperable middleware core, because, for the former, each time we add a service, we have to add two implementations of the service (e.g., SLP service + UPnP service). Thus, the small size overhead introduced by the Amigo interoperable middleware core with UPnP applications disappears when the number of hosted services increases.

Further, a middleware that needs to host different services, in terms of both functionalities and service discovery protocol (SDP) used, must have all the corresponding native libraries irrespectively of the use of Amigo middleware. However, in this case, the latter still provides efficient interoperability: it reduces drastically both the number of hosted services and, in the long term, the overall middleware size since we do not have to develop and deploy services for each existing SDP.

4.1.2 Service Interaction

The assessment reported herein concerns the second (but not final) prototype of the service interaction interoperability sub-system of INMIDIO. This INMIDIO prototype is implemented in ANSI-C. The C programming language has been chosen for several reasons: (i) it enables the deployment of INMIDIO without requiring any additional software (e.g., requirement of the Java virtual machine) as embedded system kernels are mainly developed in C, and (ii) it increases the execution speed, which is a key requirement. However, INMIDIO may be developed in any other programming language and/or dedicated to one specific software platform to increase further its efficiency.

INMIDIO provides 2 instances of the SUN compliant RMI stack (See RMI_1 and RMI_2, Table 4-2: The RMI stacks of INMIDIO vs. Sun JVM) through the use of 4 units developed in C: the Java Remote Method Protocol (JRMP), Java Object Stream Protocol (JOSSP), HTTP protocol and Java Mobile Code. As given in Table 4-2, the RMI stack of INMIDIO requires at most 636 Kb against about 3Mb for the Java Micro Edition environment with the additional packages to support RMI as a client. Note that we reuse existing non optimised HTTP library. In addition, through an adequate configuration of the protocol units, INMIDIO can act not only as an RMI client but also as an RMI service and is therefore able to generate dynamically Java proxy/stub code on the fly. This behaviour is, normally, only possible on the desktop Java runtime environment whose size is of 45 Mb. INMIDIO drastically reduces the size requirements to support the full features of the RMI specification, as it needs neither a JVM nor Java class libraries at all.

INMIDIO					SUN JVM	
Units		Size	RMI Stack 1	RMI Stack 2	JRE	J2ME
Mobile Code	*Parser* *Composer*	140	**-**	**X**	**-**	**-**
JOSSP	*Parser* *Composer*	56	**X**	**-**	**-**	**-**
JRMP	*Parser* *Composer*	40	**X**	**-**	**-**	**-**
HTTP	*Parser* *Composer*	164	**-**	**X**	**-**	**-**
IO abstraction		36	**X**	**X**		
Event Manager		200	**X**	**X**		
TOTAL in Kb		**636**	**332**	**540**	**45000**	**3000**

Table 4-2: The RMI stacks of INMIDIO vs. Sun JVM

To support the Web services communication protocol, the INMIDIO prototype builds on an existing SOAP library developed in C, to implement the required SOAP and HTTP units. Unfortunately, to the best of our knowledge, there does not exist any optimised SOAP library, developed in C, dedicated to resource constrained devices in the open source community. Consequently, we reuse the CSOAP[4] library, which has the severe constraint to be memory consuming, as given in Table 4-3. GSOAP [EG02] is known to be more appropriate for saving resources, but it does not provide the ability to create dynamically SOAP calls at run-time. It is interesting to note that some commercial SOAP versions require only 150Kb against the 1524Kb for CSOAP. Accordingly, it is very promising for the next INMIDIO prototypes in terms of memory cost. Nevertheless, although the current INMIDIO prototype is half optimised, its size is already less than the J2ME runtime, while providing interoperability.

INMIDIO			
Units		Size	Web services Stack
SOAP Unit	*Parser* *Composer*	1360	**X**
HTTP Unit	*Parser* *Composer*	164	**X**
Event Manager		200	**X**
TOTAL in Kb		**1724**	**1724**

Table 4-3: The CSOAP-based Web services stack of INMIDIO

4 http://csoap.sourceforge.net

4.2 Runtime aspects

4.2.1 Service Discovery

Again, this assessment concerns the first prototype of INDMIDIO – the Amigo interoperable middleware core – service discovery. We evaluate the performance of our interoperability mechanisms by investigating the response time of the Amigo interoperable middleware core when enabling a client dedicated to one SDP to discover a service based on another SDP. Specifically, the experiments consider the case where a SLP (resp. UPnP) client searches a SLP (resp. UPnP) service. We then compare the native client waiting time to get an answer from a native service with its waiting time to get an answer from an Amigo-interworked service. The impact of the Amigo middleware core on performance varies according to its location, either on the client or on the service side. Thus in the following, we consider the two cases. In addition, as interoperability is achieved without generating additional traffic, we have not evaluated the network bandwidth consumption. Indeed, the generated traffic is well known since we are neither providing a new service discovery protocol nor altering native protocols.

Although our solution is dedicated to various devices, including resource-constrained ones, all tests are performed on workstations equipped with 256Mbytes RAM on Intel PIV processor rated at 1.8GHz. In fact, currently, to the best of our knowledge, there does not exist any UPnP profile for J2ME devices in the open source community. Thus, the operating system, the Java virtual machine and the performance tools platform used are, respectively, Linux from Redhat Fedora Core 2, JDK1.4.2 from Sun, and the Hyades platform from the Eclipse Foundation. Moreover, the SLP (resp. UPnP) client and SLP (resp. UPnP) service are hosted on different hosts connected to a LAN at 10Mb/s. The SLP client and service are based on OpenSlp, whereas the UPnP client and service use Cyberlink for Java. The given measurements are in msec and are the median of 30 successful tests to avoid a mean skewed by a single high or low value.

	SLP -> SLP	UPnP -> UPnP
Median value (ms)	0.7	40

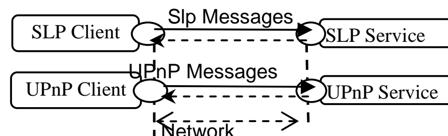
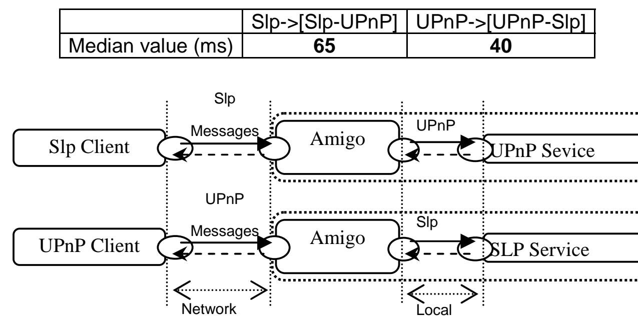


Figure 4-1: Native clients & services

In Figure 4-1, we first give the response time of a search request generated by a native client to get a successful answer from a native service: for SLP, we get 0.7 ms, whereas for UPnP, we get 40ms. It is clear that using SLP is much more efficient than UPnP, which is a higher-level protocol than SLP. These results are considered as references values to enable us to interpret the following results.

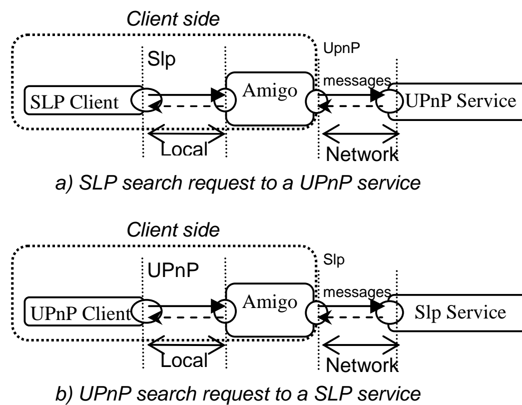


	Slp->[Slp-UPnP]	UPnP->[UPnP-Slp]
Median value (ms)	**65**	**40**

Figure 4-2: Performance with Amigo located on the service side

Consider now the case where the Amigo middleware core is located on the service side to enable the latter to be interoperable with any client independently of its SDP (Figure 4-2). In the context where the client is SLP and the service is UPnP, the client gets an answer in 65ms. The translation between SLP and UPnP is not direct. For instance, UPnP and SLP search responses are semantically different: a SLP client expects a direct reference to interact with the service discovered, whereas a UPnP client expects a reference to a description file corresponding to the service found. Consequently, the Amigo middleware core has translated the SLP request into two local UPnP requests to get the information that is necessary to generate on the network the corresponding SLP response. This means that the Amigo middleware core has waited and parsed successively two UPnP responses, thus increasing the SLP responsiveness latency. On the service side, it is clear that the Amigo middleware core simulates a UPnP client, and therefore we cannot interfere on the native time taken to get a UPnP response from the service. In this context, the Amigo middleware core result is pretty good.

Still in Figure 4-2, when the client is UPnP and the service is SLP, the response time to get an answer is 40ms. In fact, it corresponds exactly to a search request generated on the network from a native UPnP client to a native UPnP service. On the service side, the response time to a SLP request is negligible as the latter is generated locally.



	[Slp-UPnP]->UPnP	[UPnP-Slp]->Slp
Median value (ms)	80	0.12

Figure 4-3: Performance with Amigo located on the client side

When the Amigo interoperable middleware core is located on the client side (Figure 4-3a), the latter becomes interoperable and can discover any service whatever its SDP. If the client is SLP and the service is UPnP, the SLP client gets the answer to its search request in 80ms. It corresponds globally to two native UPnP responses from a native UPnP service. This is obvious, since, as previously, the Amigo interoperable middleware core has translated the SLP request into two network UPnP requests to get the necessary information to generate locally the corresponding SLP response. Once again, the Amigo interoperable middleware core result is encouraging. It is important to note that compared to the case depicted in Figure 4-2, the response time is higher than previously, simply because the UPnP traffic goes across the network between the Amigo interoperable middleware core and the UPnP service, increasing by 15 ms the response time. In the same context, the high response time inherent to the UPnP protocol is confirmed, as a UPnP client gets a response from a SLP service in only 0.12ms (Figure 4-3b). This is due to the fact that, first, the UPnP traffic is local and, then, the only traffic that goes across the network is SLP, which is particularly fast. In addition, the necessary information to generate a search response for UPnP is tiny. We can consider this case as the best case.

The above results show that the Amigo interoperable middleware core is particularly efficient in providing interoperability in all possible contexts.

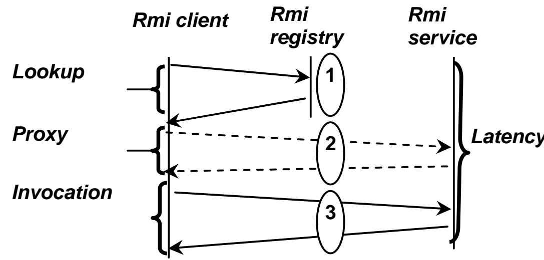
4.2.2 Service Interaction

Again, this assessment concerns the second (but not final) prototype of INMIDIO service interaction. We evaluate the performance of INMIDIO by investigating the latency required for a client to get an answer to its RPC request from a remote service based on a different RPC protocol. The latency does not include the time needed for the service to export its interface. Although the exporting step is mandatory, it is more related to the service/registry discovery process than the interoperable interaction mechanism. Accordingly, our experiments focus on the latency of remote service invocation, for which we implemented an *echo* service that echoes to the client the string given as an argument in the RPC request. We compare then the resulting latency with the one of a native RPC between a client and service based on an identical RPC protocol.

Although our solution is dedicated to various devices, including resource constrained ones, all tests are performed on a workstation equipped with 256Mbytes RAM on Intel IV processor rated at 1.8GHz as our focus is on assessing performance against native cases. Hence, the operating system is Linux Redhat Fedora Core 2. INMIDIO is compiled with the *gcc* compiler and the *glibc library* version 3.2.2. The Web services client and service are based either on the CSOAP library or Java Apache Axis[5], whereas the RMI client and service are based on JDK 1.4.2 from SUN. The given measurements are in ms and are the median of 15 successful tests to avoid a mean skewed by a single high or low value. Moreover, all the tests are run on a single host to avoid the network delays, as we want to measure the INMIDIO performance. Indeed, INMIDIO provides interoperability without affecting the existing protocols and therefore does not increase the network bandwidth consumption.

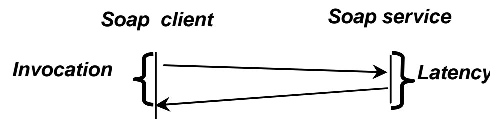
Figure 4-4 depicts a RMI request/response between a RMI client and service. If the client has already the proxy byte-code of its desired remote service, the overall latency (Figure 4-4, Steps ❶ & ❸), including both the RMI invocation and the RMI lookup request (i.e., to get the stub of the remote service from a regular RMI registry), is 201ms. However, if we consider exclusively the RMI invocation from the client perspective, the request/response latency takes only 1 ms against 8.08 ms or 20 ms for a similar SOAP interaction between a Web services client and service developed respectively in C or Java (See Figure 4-5).

[5] http://ws.apache.org/axis/.



	Elapsed time (ms)
❶ **RMI lookup**	200
❸ **RMI request/response**	1
Total with proxy	201

Figure 4-4: Native RMI RPC with and without mobile code

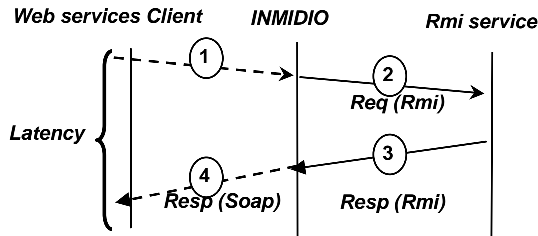


	Latency (ms)	
	CSOAP	**Java AXIS**
Total	**8**	**20**

Figure 4-5: Native SOAP invocation in C and Java

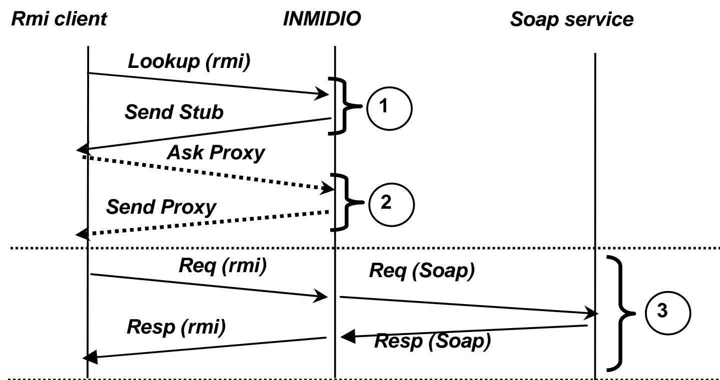
Since the RMI RPC is binary oriented, RMI invocations are obviously faster than SOAP ones. Furthermore, the latency difference between the C and Java SOAP native call hints at the impact of the C programming language on performance.

Consider now the case where the client and service are based on heterogeneous RPC protocols and rely on INMIDIO as a transparent intermediary that achieves interoperability. When the client is SOAP and the remote service is RMI-based, the overall latency of the SOAP interaction, from the client perspective, is of about 9 ms (see Figure 4-6). Comparing to the C-based SOAP native call, the latency of 1 ms overhead corresponds to the latency of a Java-based RMI interaction. In other terms, the interoperability between a SOAP client and a RMI service takes as much time as is needed for exactly both one C-based SOAP interaction and one Java-based RMI interaction. Comparing now the 9 ms with the 20 ms required for Java-based SOAP interaction, INMIDIO clearly performs better. However, if we compare solely with the RMI native case, INMIDIO performs poorly but this is inherent to the SOAP protocol.



	Latency (ms)
❶**SOAP Parser**	5
❷**RMI Composer**	0.2
❸**RMI Parser**	0.2
❹**SOAP Composer**	3
Total	9

Figure 4-6: Interoperable invocation between a Web service client and a RMI service with INMIDIO



	Latency (ms)
❶ **STUB Generation**	0.30
❷ **Mobile Code Generation**	0.85
❸ **Invocation**	9
Total with proxy	9.30
Total without proxy	10.15

Figure 4-7: Interoperable invocation between a RMI client and a Web service with INMIDIO

Consider next that the client is RMI-based and the service is SOAP-based, INMIDIO acts, from the client side, as both a compliant RMI registry and a RMI remote service (See Figure 4-7).

The mandatory lookup request from the client to get the stub of the service takes about 0.30 ms when INMIDIO acts as a RMI registry, whereas it takes 200ms with a standard java-based registry. In the case where the client does not have found in its JVM the proxy byte-code corresponding to the received stub, the latency increases of 0.85 ms. This overhead corresponds to the cost for the client to get from INMIDIO the proxy byte-code, which is dynamically generated from the interface exported by the Web services remote service (Figure 4-7, Step ❷). Moreover, excluding Steps ❶&❷, the latency of the client RMI invocation (Figure 4-7, Step ❸) is almost equal to the similar C-based SOAP invocation of the previous scenario. In fact, once clients have all the necessary information to perform their RPC call (i.e., endpoints, stubs, proxy byte-code), the cost of the interoperability processes between Web services and RMI entities is finally independent of the nature of the client/service (i.e., either RMI or SOAP based) and stays nearly constant: about 9 ms.

Summarising, for sending a lookup, the latency increases of 0.30 ms whereas for getting the proxy byte-code, the latency increases of 0.85 ms. Therefore the overall latency is, in the best case, of 9ms, and in the worse case, of 10.15 ms (See Figure 4-7) . It is clear that the latency required for an interoperable interaction between RMI and SOAP entities can not be smaller than the sum of the latency required for both a native RMI call and a native C-based SOAP call. Hence, the overhead of INMIDIO is negligible.

5 Security Framework Assessment

5.1 Development, configuration, deployment, management aspects

Security has been an essential component from the first design on in the Amigo Service Oriented Architecture. It was a challenging task to come up with a reliable and secure solution that had on one hand, the same security level as a traditional security solution but on the other hand was maintenance free and easy to use by regular Amigo home inhabitants.

The provided solution consists of 2 parts: a distributed server solution written in .Net and a client solution written in .Net and in Java. Application and Service developers use either the .Net or the Java client depending on their preferred platform. Securing a regular service is simply enabled by wrapping existing Web Service calls with methods provided by the client. In this way the effort for the developer is kept to a minimum. The server uses an automatic replication method to guarantee high reliability. If a security server is no longer available, another one in the Amigo infrastructure will seamlessly take over without causing any disruption in the security function.

From an Amigo home inhabitant point of view, the security solution is:

- Very user friendly when adding new devices/services to the home (no admin functionality/equipment) required
- Enables Single Sign On (user do not have to repeatedly enter their credentials)
- Guaranteed maintained security quality through a multiple role based configuration model. Adding new devices, services and users do not require thread analysis as long as the multiple role model is sufficient.
- Allows parental control by deferring authorization decisions to users with a higher authorization level (e.g. parents)
- Handles authentication as well as authorization (who is who and who is allowed to do what)

From an application/service developer point of view, the security solution is:

- Easy to embed since existing methods only have to be wrapped using standard wrappers from the client software.
- Uses standardized/interoperable XML files for data storage on the client
- Based on existing and proven security solutions (Kerberos)
- Fully interoperable between Java and .Net
- Leverages existing standards (Web Services, XML files) and therefore cross platform

The security solution is demonstrated in the extended home scenario that demonstrates that it can even be used to enable and ensure security across multiple homes.

6 Semantic Service Framework Assessment

6.1 Complex service workflows

6.1.1 Development, configuration, deployment, management aspects

Developing and using semantic services involves several stages: services must be given a semantic description and user tasks must be created; semantic services must be deployed and made available to Amigo applications; applications must discover the semantic services that are currently available in the environment; in the cases where no single service can match an application's request, a composition of services, adapted where necessary, may be constructed to satisfy the request; and finally, semantic services, whether a single service or composed, must be executed by the application. The SD-SDCAE middleware assists the developer at each of these stages of development.

The SD-SDCAE developer has several tools at his/her disposal to assist him/her in creating service descriptions. The Protégé ontology editor can be used to graphically edit descriptions. The Eyeball OWL checker tool, available from the Jena project website, is a lint-like command-line tool that can be used to check that service descriptions are free of certain common OWL errors [Eyeball]. Developers who prefer to work directly with Amigo-S XML can of course create service descriptions using their favourite text editor.

However, there is also a tool that has been specifically developed for SD-SDCAE, called *amigosgen*, that can significantly decrease the effort required to create semantic service descriptions. The amigosgen tool takes a service's WSDL description as input and creates an almost complete semantic service description. All the developer must do to complete the description is to replace a few placeholder tags with the semantic concepts to be used for each capability.

Figure 6-1 shows a snippet of the output of amigosgen for a coffee maker service, as developed in the SD-SDCAE User's Guide [SD-SDCAE]. The tags the developer must edit are surrounded by double ampersands.

```
<service:Service rdf:ID="@@ SEMANTIC SERVICE NAME @@">

  <lang:ServiceType rdf:datatype=
          "http://www.w3.org/2001/XMLSchema#string">
  @@ SERVICE TYPE SEMANTIC URI @@
  </lang:ServiceType>

  <service:presents>
    <capabilities:ServiceProfile rdf:ID=
          "@@ SEMANTIC SERVICE NAME @@Profile">

      <!-- Provided capabilities -->
      <lang:hasProvidedCapability>
        <capabilities:@@ switchOn CAPABILITY SEMANTIC @@
                rdf:ID="CoffeeMachineServiceSwitchOnCapability">
          <lang:hasConversation rdf:resource=
                "#CoffeeMachineServiceSwitchOnConversation"/>
          <lang:hasOutput rdf:resource=
                "#CoffeeMachineServiceSwitchOnOutput"/>
        </capabilities:@@ switchOn CAPABILITY SEMANTIC @@>
      </lang:hasProvidedCapability>

          <lang:hasProvidedCapability>
        <capabilities:@@ brew CAPABILITY SEMANTIC @@ rdf:ID=
```

```
            "CoffeeMachineServiceBrewCapability">
    <lang:hasConversation rdf:resource=
             "#CoffeeMachineServiceBrewConversation"/>
    <lang:hasOutput rdf:resource="#CoffeeMachineServiceBrewOutput"/>
  </capabilities:@@ brew CAPABILITY SEMANTIC @@>
</lang:hasProvidedCapability>
```

Figure 6-1: A snippet of the template generated by the amigosgen tool for an example coffee maker service.

The amigosgen tool affords a significant reduction in the effort required to create semantic service descriptions as, rather than have to write complete, often large OWL XML files, the user simply has to make $(C * O * I) + 2$ trivial edits to the template generated by amigosgen, where C is number of capabilities of the service, O the average number of unique outputs, and I the average number of inputs.

On the other hand, deploying services in the Amigo home environment is straightforward, and performed either graphically using the VantagePoint tool or programmatically using the standard mechanisms from the OSGi Programming and Deployment Framework. Using VantagePoint affords a visual representation of the semantic services, making registration of services with the semantic service repository simple for users by offering an intuitive drag-and-drop interface. Using the OSGi Programming and Deployment Framework affords the developer programmatic control over the semantic service repository, allowing precise control over the registration and further manipulation of services.

Furthermore, the discovery, composition and adaptation facilities of the semantic service repository are easily accessed through a single, common interface. The user simply has to supply the description of the abstract task he/she requires to be realized from the services in the environment in a single method call on the repository, and all matching, composed, and/or adapted services will be returned.

By describing a task in an abstract way, we are not bound to any particular remote service in terms of the capabilities provided or the specific orchestration of these capabilities, thus increasing the availability and promoting interoperability of the potentially matching services. Creating a semantic service description for a basic service allows the service to be discovered via semantic matching at both the service and capability levels, thus increasing the service's availability and promoting its interoperability. Furthermore, describing a semantic service's provided capabilities as conversation-based workflows allows the expression of data and control dependencies between the service's capabilities. Complex conversations can be automatically and reliably composed, while offering fine-grained control over the placement of capabilities in the task, and guaranteeing that the data and control dependencies of each of the provided capabilities are preserved.

Once the abstract task has been realised, an Amigo application developer simply calls the public methods, that is, the required capabilities of the task as he/she would call a normal web service from application code, and the SD-SDCAE ensures that the execution of the workflow of capabilities it contains is performed automatically and transparently by the ActiveBPEL execution engine.

6.1.2 Runtime aspects

In this section, we present an evaluation of the runtime performance of the SD-SDCAE middleware. Specifically, we focus on semantic service discovery and composition response times over a variety of repository and service configurations.

6.1.2.1 Experimental set-up

Figure 6-2 shows the topology and Table 6-1 the specification of the machines that were used to perform the measurements given below. The names of the machines are “Sas” and “Chico1”.

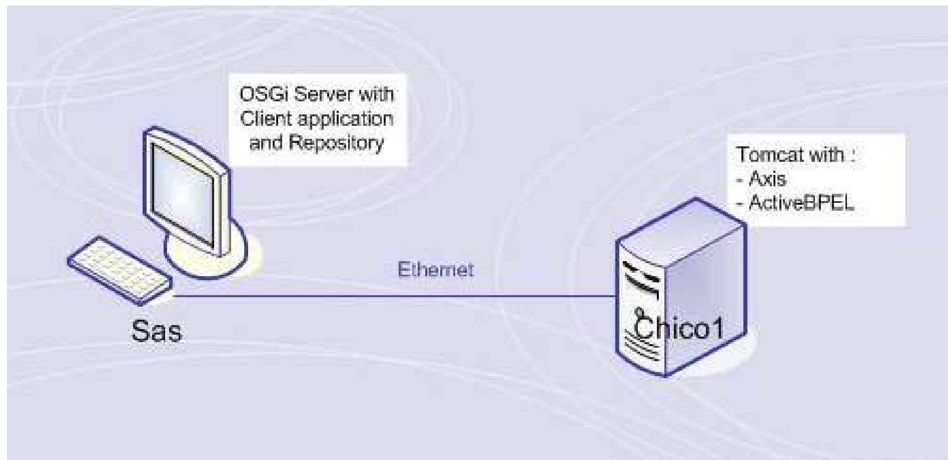


Figure 6-2: The network set up.

Sas	Chico1
Dell Precision 380 Intel Pentium 4 – 3.4 GHz 2 GB RAM	Dell Precision M60 Intel Pentium M – 1.7 GHz 1 GB RAM
Run under eclipse 3.2.2 with memory parameters: -Xms768M -Xmx1024M	JDK1.5.0_06, Tomcat 5.5, Axis 1.4, ActiveBPEL 4.1

Table 6-1: The machine specifications.

6.1.2.2 Discovery time measurements

This presents the results of measuring the average response time for discovering a semantic service as the number of services registered with the semantic service repository increases.

For these measurements, the repository was filled with synthetic semantic services generated using the JavaServiceGenerator tool available in the SD-SDCAE source bundle. This tool provides a useful aid for measuring the performance of a service repository set-up. All of the services were deployed on the Chico1 machine.

Each generated service had 5 atomic, provided capabilities, with each capability having 3 inputs and 1 output. For each service that matched the task, the semantics used for each capability, input, and output for both the service and the task were the same. For every other service that did not match the task, the semantics used for each capability, input, and output were distinct.

The WSDL description of a service was retrieved from Axis by issuing the HTTP request http://urlOfTheService?wsdl. Using this WSDL description, a semantic description template was created for each service using the *amigosgen* tool. The declarations of each of the

capability, input, and output semantics for a non-matching service were then completed using the following substitutions:

- Capability Semantic Type → "http://www.inria.fr/myGeneratedOntology.owl" + "#GeneratedCapabilitySem_"+ serviceID + "_" + methodID
- Input Semantic Type → "http://www.inria.fr/myGeneratedOntology.owl" + "#GeneratedInputSem_" + serviceID + "_" + methodID + "_" + inputID
- Output semantic type → "http://www.inria.fr/myGeneratedOntology.owl" + #GeneratedOutputSem_" + serviceID + "_" + methodID;

The declarations were generated in similar fashion for the matching services, though here, the ID parameters were fixed to have suitably matching values.

The task used for these measurements was created manually, and contained 5 required capabilities. Testing for a match is performed on a per-capability basis. First, the semantic of a required capability of the task is compared with the semantic of a provided capability of a service. If a match occurs, the semantics of each of the inputs of the required capability is compared with those of the provided capability. If the inputs also match, then the required and provided capabilities' outputs are also compared. If these 3 comparisons are successful, then the task's required capability is said to be matched by the service's provided capability. For these measurements, the services were constructed such that a single service would provide each of the 5 capabilities required by the task.

# Services	Task Parsing	SD Single	SD Single FR	SD All	SD All FR
1	2000	35	578	41	680
10	2000	35	592	46	774
20	2000	36	601	49	818
30	2000	38	639	49	820
40	2000	41	683	51	843

Table 6-2 presents the results. The standard error of each of the timings shown here, as in the following tables, was less than 1%. The first column shows the number of services registered with the repository. The second column shows the time taken to parse the task description. The following columns show two different extremes of the service discovery performance. The thirds and fourth columns shows the time taken by service discovery when the task matches a single service out of all of the services registered with the repository. The fifth and sixth columns show the time taken by service discovery when the task matches all of the services registered with the repository. The first service discovery request received by the semantic repository will take longer than subsequent requests, as it incurs the additional cost of parsing and inferring relationships from all of the necessary ontologies. The fourth and sixth columns show the longer first request times, while the third and fifth columns show the shorter subsequent request times, for matching a single and all services, respectively.

From

# Services	Task Parsing	SD Single	SD Single FR	SD All	SD All FR
1	2000	35	578	41	680
10	2000	35	592	46	774
20	2000	36	601	49	818
30	2000	38	639	49	820
40	2000	41	683	51	843

Table 6-2 we can see that the discovery time increases gradually and linearly as the number of services increases. Furthermore, based on the subsequent request times, matching all services takes on average only 10 milliseconds (27%) longer than matching a single service. In both cases, the cost of semantic discovery is significantly shorter than the time required to parse the task's Amigo-S XML description.

# Services	Task Parsing	SD Single	SD Single FR	SD All	SD All FR
1	2000	35	578	41	680
10	2000	35	592	46	774
20	2000	36	601	49	818
30	2000	38	639	49	820
40	2000	41	683	51	843

Table 6-2: Service discovery times. All times are shown in milliseconds.

# Services	Task Parsing	SD Single	SD Single FR	SD All	SD All FR
1	2000	35	578	41	680
10	2000	35	592	46	774
20	2000	36	601	49	818
30	2000	38	639	49	820
40	2000	41	683	51	843

Table 6-2 showed results for service discovery where full semantic reasoning was employed. That is, all inferred relationships, in addition to those explicitly stated, are included when considering concept equality. However, not all Amigo applications will require this level of power. The semantic service repository also supports simple semantic reasoning, where only explicit relationships are considered when comparing semantic concepts for equality.

# Services	SD Simple	SD Full
1	4	35
10	5	35
20	6	36
30	8	38
40	9	41

Table 6-3: A comparison of service discovery times using simple semantic matching and full semantic matching.

# Services	SD Simple	SD Full
1	4	35
10	5	35
20	6	36
30	8	38
40	9	41

Table 6-3 shows the results of repeating the single match measurements, though this time using simple, rather than full, semantic reasoning. We can see that this provides a dramatic increase in service discovery performance, with response times typically taking only 17% of the times of previous measurements.

6.1.2.3 Composition Time Measurements

Figure 6-3 shows the results of measuring the average response times of requesting a task that requires a number of services to be composed. Results of using both simple and full semantic reasoning are presented.

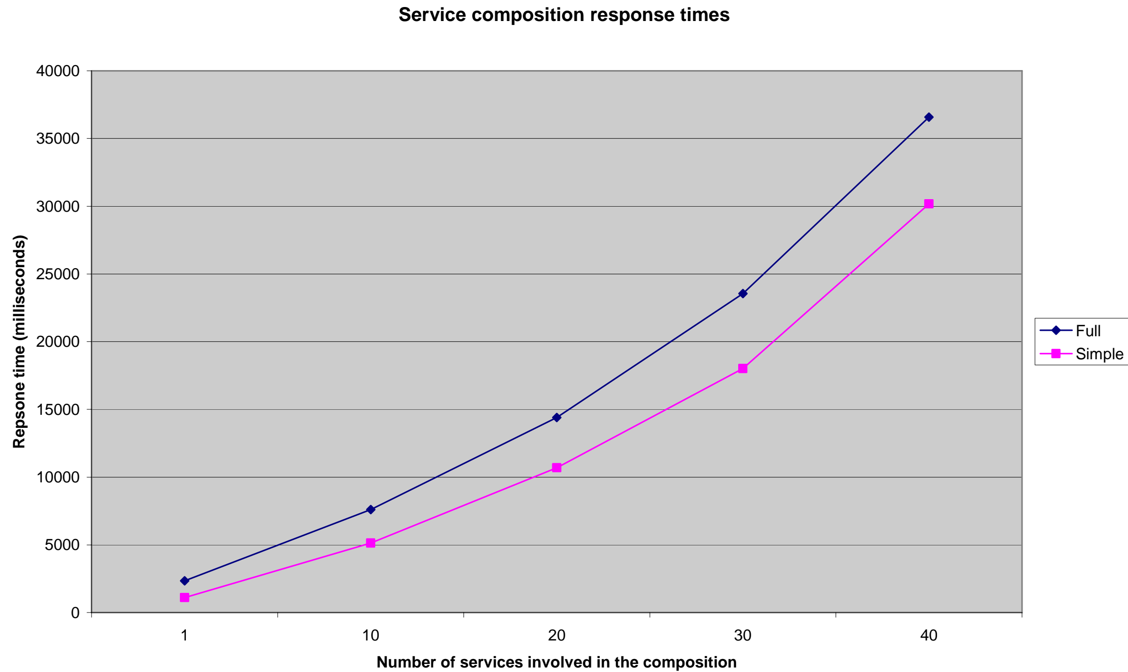


Figure 6-3: A comparison of service composition response times using full and simple semantic reasoning.

For these measurements, 40 distinct semantic services were registered with the repository. Each service had 5 atomic, provided capabilities each with 1 output and no inputs. The semantics used for each capability and output were distinct. The number of services involved in a composition reflects the number of capabilities that featured in the task's sequential conversation. Each capability required by the task would be matched by a single provided capability from a distinct service. So when a task contained a conversation featuring 40 capabilities, 40 distinct services would be involved in the composition.

From Figure 6-3 we see that cost of composition increases approximately linearly with the number of services involved in the composition. On average, an additional 133% increase in response time is incurred for each doubling of the number of services involved in a composition. Furthermore, we can see that the response times are faster using simple reasoning over full, approximately 30% faster on average.

A breakdown of the relative consumption of the different stages of the composition process for each of the different composition sizes is shown in

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	744	31.7%	77	3.3%	1301	55.4%	227	9.7%	2349
10	2645	34.7%	165	2.2%	2874	37.7%	1929	25.3%	7612
20	5686	39.5%	233	1.6%	4612	32.0%	3867	26.9%	14397
30	9788	41.6%	309	1.3%	7543	32.0%	5906	25.1%	23546
40	15294	41.8%	377	1.0%	12641	34.6%	8267	22.6%	36579

Table 6-4 and

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	802	72.7%	16	1.4%	18	1.6%	269	24.3%	1104
10	2732	53.2%	22	0.4%	178	3.5%	2201	42.9%	5132
20	5779	54.0%	35	0.3%	720	6.7%	4159	38.9%	10691

30	9857	54.7%	43	0.2%	2329	12.9%	5791	32.1%	18020
40	15672	51.9%	51	0.2%	6514	21.6%	7938	26.3%	30174

Table 6-5. The columns show the number of services involved in the composition, the time and percent of the total response time spent parsing the task, the time and percent of the total response time spent in service discovery (excluding first request times), the time and percent of the total response time spent in service composition, the time and percent of the total response time spent generating the BPEL file that represents the resulting composed service, as well as the total response time.

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	744	31.7%	77	3.3%	1301	55.4%	227	9.7%	2349
10	2645	34.7%	165	2.2%	2874	37.7%	1929	25.3%	7612
20	5686	39.5%	233	1.6%	4612	32.0%	3867	26.9%	14397
30	9788	41.6%	309	1.3%	7543	32.0%	5906	25.1%	23546
40	15294	41.8%	377	1.0%	12641	34.6%	8267	22.6%	36579

Table 6-4 provides an analysis of the measurements using full semantic reasoning, and

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	802	72.7%	16	1.4%	18	1.6%	269	24.3%	1104
10	2732	53.2%	22	0.4%	178	3.5%	2201	42.9%	5132
20	5779	54.0%	35	0.3%	720	6.7%	4159	38.9%	10691
30	9857	54.7%	43	0.2%	2329	12.9%	5791	32.1%	18020
40	15672	51.9%	51	0.2%	6514	21.6%	7938	26.3%	30174

Table 6-5 provides an analysis of the measurements using simple semantic reasoning. Note that in both cases, the total response time is dominated by XML processing, which involves parsing Amigo-S abstract task descriptions and generating BPEL concrete task descriptions. BPEL generation involves combining the internal automata-based model of a service composition with the in-memory representations of the services it composes, to produce the resulting XML-based BPEL representation. Using full semantic reasoning, abstract task parsing consumes on average 38% of the total processing time, while BPEL generation of the composed concrete task consumes 22%, giving a total of 60% for XML processing. This dominance is even greater when using simple semantic reasoning, where abstract task parsing consumes on average 57% of the total processing time, while BPEL generation of the composed concrete task consumes 33%, giving a total of 90% for XML processing.

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	744	31.7%	77	3.3%	1301	55.4%	227	9.7%	2349
10	2645	34.7%	165	2.2%	2874	37.7%	1929	25.3%	7612
20	5686	39.5%	233	1.6%	4612	32.0%	3867	26.9%	14397
30	9788	41.6%	309	1.3%	7543	32.0%	5906	25.1%	23546
40	15294	41.8%	377	1.0%	12641	34.6%	8267	22.6%	36579

Table 6-4: Semantic composition response times using full semantic reasoning. All times are shown in milliseconds.

# Services	Parsing	%	Disc.	%	Comp.	%	BPEL	%	Total
1	802	72.7%	16	1.4%	18	1.6%	269	24.3%	1104
10	2732	53.2%	22	0.4%	178	3.5%	2201	42.9%	5132
20	5779	54.0%	35	0.3%	720	6.7%	4159	38.9%	10691
30	9857	54.7%	43	0.2%	2329	12.9%	5791	32.1%	18020
40	15672	51.9%	51	0.2%	6514	21.6%	7938	26.3%	30174

Table 6-5: Semantic composition response times using simple semantic reasoning. All times are shown in milliseconds.

6.2 Context Aware Services

This section provides an assessment of Context Aware Service Discovery (CASD), developed as part of the Amigo Semantic Services Framework. The assessment will cover several aspects of CASD, such as a comparison between the Basic Service Discovery (BSD) (as supported by the Amigo Programming and Deployment Frameworks) and CASD, development effort needed for using CASD, performance, and estimated memory footprint, covering both development and runtime aspects. All assessments will be done relative to Basic Service Discovery (BSD), since that is the service discovery mechanism normally used for Amigo services and because it is used as the basis for Context Aware Service Discovery (CASD) as well.

6.2.1 Development, configuration, deployment, management aspects

Next to the performance aspects of using CASD there are also development aspects; which relate to the effort of implementing additional functionalities to clients or services in order to be able to use CASD. In other words: what extra time and effort is needed for using CASD instead of basic service discovery?

6.2.1.1 Ease of learning

For determining how easy it is to learn to use CASD relative to basic service discovery, we assume developing for BSD to be the baseline; that is: developers are assumed to be skilled already in developing Amigo services using Amigo BSD. The difference is then in the additional effort to learn to use CASD.

The biggest difference between regular service discovery and context aware service discovery is in the use of context sources for both services and clients. If the developer is unfamiliar with context management, context source development, and/or the use of ontologies in general, then there is a steep learning curve, since he or she first has to become familiar with (the use of) context and ontologies. If the developer has already some familiarity with the subject then the learning curve will be gentler, since one of the tutorials provided by Amigo is the CMS tutorial for developing and using context and context sources. The CMS tutorial does assume some basic understanding of ontologies.

6.2.1.2 Additional development effort

Discovery Client

For the client side, the additional effort is in creating a Context Source (CS) that provides (at least) the context needed for the specific type of context aware discovery (such as location, if the client wants to find the ‘CLOSEST’ service of a certain type).

For creating a CS, helper libraries and bundles as well as tutorials are available for both OSGi and .NET deployment environments[6].

For using CASD itself, a helper bundle is available as well. The helper bundle will take care of administrative tasks such as discovering the CASD service in the network, so that the client can work with CASD in a similar fashion as with the basic service discovery.

[6] Available at https://gforge.inria.fr/frs/?group_id=160.

If we take a look at the example CASD Helper client, the following code does the actual call to CASD, which is very similar to a regular service discovery call.

```
String[] services = new String[0];
try {
  services = casdHelper.lookup(
      "AmbulanceService",
      new String[] { clientcs.getAmigoReference().toString() },
      "CLOSEST");
} catch (CASDException e1) {
  e1.printStackTrace();
}
```

The same holds for persistent CASD, comparable to 'passive discovery' of regular service discovery. Instead of implementing the ServiceListener interface of regular service discovery to receive information about (dis)appearing services, the client has to implement the ICASDServiceChanged interface. The CASD helper will call this interface with the initial results of the context aware service discovery and call it again whenever the results of the discovery change. This is again comparable with the regular (passive) service discovery approach (and needed effort).

Discoverable Service

Every service potentially discoverable by CASD also has to implement a Context Source providing relevant context for CASD to use in refining the results of BSD. The additional effort for services is bigger than for clients, since the CS of a service has to be able to provide every type of context for which it wants to be considered by CASD. So, if a service wants to be part of the matching process done by CASD for finding the 'CLOSEST' service, then it should provide location context information. If it wants to be part of matching for the cheapest service, it should also provide pricing information, etc.

Apart from that, creating and exporting a service for discovery is very similar to regular services.

```
AmigoExportedService myService;
AmbulanceServiceImpl service;

service = new AmbulanceServiceImpl("Amigo Ambulance Service ");
try {
  logger.debug("Trying to export the Ambulance service");
  myService=serviceExporter.createService(server[i]);
  myService.addProperty("oid","AmbulanceService"+i);
  myService.addProperty("ServiceType","AmbulanceService");
  myService.addProperty("Scope","urn:amigo");
  myService.addProperty("ContextSourceURL", acs[i].getReference().getUrl());
  myService.exportInterface(AmigoReference.DEFAULT, IAmbulance.class);
} catch (AmigoException e) {
  logger.info("Exception when exporting object",e);
  e.printStackTrace();
}
```

6.2.1.3 Tooling

As stated before the main difference between regular service discovery and context aware service discovery is the use of context sources. For the Context Management Service (CMS) of Amigo, support is available in the form of helper libraries and/or bundles as well as a set of test tools such as the Context Source Tester for dynamically trying out context sources. Since the context sources used by CASD are regular CMS context sources, the libraries and tooling support for CMS can also be used for testing context sources created for CASD.

6.2.2 Runtime aspects

6.2.2.1 Performance

The most important run time aspect of CASD is the performance with respect to BSD. In order to estimate the relative performance, a test setup was created. This test setup consisted of a CASD client, the CASD service itself, and a number of services, all with associated context sources (See Section 6.3 in [D3.5]). The number of services to be discovered and processed by CASD was variable, for the test 1, 15, and 45 services were used.

The type of selector (i.e. what context and rating algorithm) is flexible with CASD, but for this test the typically most used 'CLOSEST' was chosen. Note that the selector used does influence the performance, but, since CLOSEST is the most used and also not optimised for performance, it is a good indicator nonetheless.

To avoid network performance to skew the results, all the tests were run on one computer, although no optimisation was done to take advantage of the fact that all services and endpoints were running on the same machine. In other words: the ordinary web service calls and mechanisms were used, just as if the services would have been running on different machines.

All tests were run on the same machine, a laptop with an Intel Core 2 Duo T5600 processor at 1.83 GHz, with 1.5 GB of RAM, running Windows XP Professional SP2. The JVM used was 1.6.0_03-b05 from Sun.

At a number of places in the CASD code, the time was taken and logged to the screen; other than that no changes were made to the standard CASD implementation.

The following times were measured using this technique; these correspond to the columns in Table 6-6:

- The time it takes the basic service discovery to discover the X services (*BSD*)
- The time it takes to get all needed context from the services (*Ctxt get*)
- The average time per service (calculated), by dividing the previous number by the number of services X (*Ctxt/svc*)
- The time it takes to retrieve the client context (*Client Ctxt*)
- The time it takes to run the matching algorithm; 'CLOSEST' in this case (*Matching*)
- The total time from the start of the call to CASD to the point where the services are returned (*Total*)

The tests were done with a varying number of services to be discovered and matched in 3 batches of 1, 15, and 45 services. Every batch was run 5 times to even out the measurements and influence of the first run (although the first run was taken into account for determining the average/min/max times).

The Oscar setup used for the tests is shown in Figure 6-4 below. Note that bundle#25 provides X number of Ambulance services, each with a different location, for the CASD Helper Client (performing the actual test) to discover.

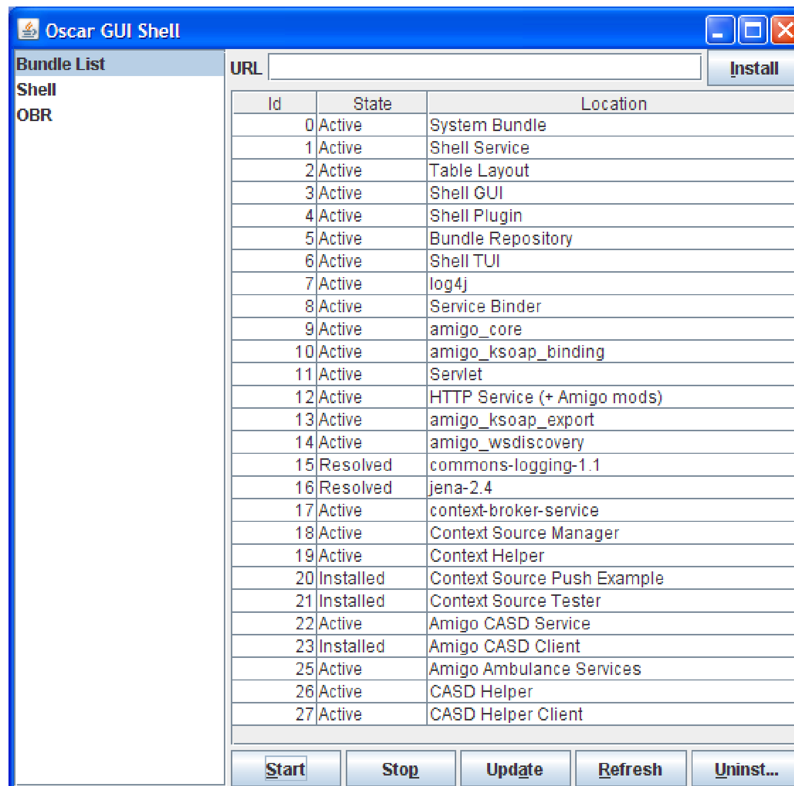


Figure 6-4: Oscar setup used for the tests.

An example output from a test run is shown below:

```
2008-01-30 20:06:22,625 DEBUG - *** Finding services took: 3266ms
2008-01-30 20:06:22,843 DEBUG - *** Getting context took:203ms
2008-01-30 20:06:22,843 DEBUG - *** 4 ms/service
2008-01-30 20:06:22,843 DEBUG - *** Finding services+context took: 3484ms
2008-01-30 20:06:22,875 DEBUG - *** Getting client context took: 32ms
2008-01-30 20:06:22,937 DEBUG - *** Matching services+context took: 62ms
```

The results from the measurement are shown in Table 6-6 below. For every batch, the minimal, average and maximum time is mentioned per sub-item of the test individually. Please note that this means that e.g. the max total time does not necessarily coincide with the max Ctxt Get time for example.

The Basic Service Discovery time is the time it would take a client anyway when discovering multiple services with basic service discovery. As can be seen in the results, this time is mostly determined by the time spent waiting for answers, specified by the client, to the Web Service discovery request. In this case the waiting time was set to 3 seconds, which was enough in all cases to discover every service active during the test. From the table, one can clearly see that processing the answers to discovery requests takes more time as more services are available, ranging from practically nothing (not significant) to about 0.5 seconds in the case of 45 services.

The total additional time added by CASD to the BSD time can be determined by subtracting the first column from the last column, since the total time includes the time spent using BSD. The additional overhead induced by CASD varies from 0.2 s in the case of 1 service, to 0.7 seconds (on average) in case of 45 services.

# services	Min/Average/Max	BSD	Ctxt get	Ctxt/Svc	Client Ctxt	Matching	Total
1	Min	3 s	0 ms	0 ms	47 ms	0 ms	3.1 s
	Average	3 s	78 ms	78 ms	98 ms	12 ms	3.2 s
	Max	3 s	297 ms	297 ms	203 ms	31 ms	3.4 s
15	Min	3.2 s	78 ms	5 ms	31 ms	31 ms	3.3 s
	Average	3.2 s	290 ms	19 ms	66 ms	39 ms	3.6 s
	Max	3.2 s	890 ms	59 ms	172 ms	62 ms	4.2 s
45	Min	3.4 s	0.2 s	4 ms	16 ms	47 ms	3.8 s
	Average	3.5 s	0.6 s	14 ms	20 ms	63 ms	4.2 s
	Max	3.5 s	1.94 s	43 ms	31 ms	93 ms	5.6 s

Table 6-6: Performance measurement results for CASD.

Most time of this additional time is taken by retrieving the context from every context source associated with the services, ranging from approximately 0.08 seconds for one service to 0.6 seconds for 45 services. The average time it takes per service to get the context decreases slightly with an increasing number of services.

The least amount of overhead is, perhaps surprisingly, caused by the matching algorithm; increasing only slightly from 12 ms in the case of 1 service to approximately 60 ms in the case of 45 services. Note that the matching algorithm, at least in this case of 'CLOSEST', retrieves the needed information (per service) from a semantic description of the context of that service, which shows that semantic processing, using Jena, is not as big an overhead as is sometimes assumed.

6.2.2.2 Resource consumption

No exact measurements were done to determine the memory footprint of CASD, but to get a feeling for the memory footprint, the test setup was slightly changed so that the services to be discovered and the client of CASD ran on a different computer. The original laptop ran only the CASD service. The same test batches were run again, but this time only once for every batch, doing a clean Oscar start before every batch. After every batch was run, the total amount of memory taken by Oscar was estimated by reading it from the process list of the Task Manager. Just before every test run, the memory taken was 41 MB; after running it for 15 services, the memory occupied was 48 MB, increasing to 49 MB for 45 services, which suggests that the additional increase in memory consumption is not likely to cause problems for an even larger number of services.

6.2.2.3 Scalability

When looking at the result for performance and resource consumption, for the normal number of services to be expected in the home, using CASD should not pose problems; neither with respect to performance nor with respect to memory usage. If the number of services increases substantially, say to hundreds of (similar!) services, the performance may no longer be sufficient. However, in those circumstances it is questionable whether ordinary service

discovery would still suffice, and indeed whether the whole architecture and distribution of functionalities over the different services is well thought out for that particular situation.

6.2.2.4 Persistent lookup

CASD also offers functionality to do a persistent lookup. With a persistent lookup the discovery request is stored by CASD. CASD keeps track of changes in the availability of the potential candidate services and changes in the context of those services. Whenever one of these changes would lead to a different answer to the original discovery request, the client will be informed of these changed results. The performance aspects of a persistent lookup are the same as for the 'standard' context aware service discovery, since CASD will perform the same operations whenever the availability of services or their context changes. Therefore no separate measurements of this type of lookup are necessary.

6.2.3 Conclusions

The difference in run-time performance between regular service discovery and CASD is relatively minor, especially when compared with the waiting time, for which 3 seconds is a realistic minimum to be able to discover all services.

Creating context sources for both the client and the services is fairly easy to do, if the developer is familiar with Amigo CMS. If not, then tutorials are available to get acquainted with CMS. The extra amount of code needed is only the additional Context Sources created; which can be limited by using the provided CMS helper bundles and libraries; the actual calls themselves have a similar amount of required lines of code.

All this does not mean that CASD should be used regardless. There is a performance impact and additional required coding effort, however small or easy they may be. If the situation does not ask for taking context into account when trying to discover services, then there is no need to use CASD in the first place. Also if the regular service discovery can be used to achieve the same goals without jumping through hoops, then that should be used. For example: for discovering all services in one room, CASD could be used. However, the same could be achieved by using different scoping properties (set to room names for example) with regular service discovery. The basic rule should be: choose the (technically) most straightforward solution for any given problem. So, if the information influencing the clients' service discovery decisions is static, then consider regular service discovery; whenever the information (about services or clients) becomes dynamic (in other words: context) then CASD should be considered.

6.3 Quality of Service Aware Services

6.3.1 Development, configuration, deployment, management aspects

This section aims to thoroughly assess the QoS-aware Service Selection Tool (QASST). QASST provides a mechanism for filtering a list of services and selecting the most appropriate one that addresses specific QoS requirements set by an Amigo User. Several parameters will be examined to assess the QASST. These parameters are: correctness, robustness, required resources, scalability, usability, portability, debugging, extensibility and complexity (performance) of the selection algorithm.

Concerning the correctness of the tool, it has been examined whether QASST addresses in full the requirements that have been set. In the course of the project, these requirements have been modified to make the QASST compliant with the Amigo platform. More specifically, QASST was supposed to filter services that were available either inside or outside the Amigo home. However, the services available outside the Amigo Home were eventually not discoverable by the discovery mechanisms provided by the Amigo system. Thus, as QASST is

based on these discovery mechanisms, the requirements had to be changed and the filtering of services is restricted to services that are offered only inside the home from other Amigo components. All the other requirements have been addressed in full.

Concerning the robustness of the QASST, several tests have been conducted in order to ensure that the code does not fail on run-time in any case. Thus, it is ensured that the QASST is completely reliable, as no QASST crashing has been observed after the final code updates.

With regards to the required resources, it can be stated that the QASST is a very lightweight component. Taking into account programmatic techniques for creating extensible and efficient code, the QASST has been designed and implemented by 6 classes of approximately 600 lines of code. Thus, the compilation time of the QASST code is extremely fast. Nevertheless, the libraries that are mandatory for the QASST to run properly require a considerable amount of space (approximately 10MB). Also, the frameworks (.NET v2.0, Java v1.5, Oscar) that have to be installed in order for QASST to run successfully are also very demanding in resources (approximately 210MB). However, once the Amigo platform is up and running, starting-up the QASST requires minimal time.

The scalability of the QASST is not of critical importance for the Amigo platform. This is due to the fact that it is not expected that a large number of users will be concurrently using the QASST in the Amigo home. The QASST is designed so that one instance is required per Amigo user, if QoS-aware service selection is requested. Therefore, the maximum number of QASST instances running in parallel in the Amigo home is equal to the maximum number of persons located in the Amigo home. The QASST has been tested for 10 users in parallel (admittedly not a demanding scalability requirement) and no problems have occurred.

In order to test the usability of the QASST, its source code accompanied with all the required software components and libraries, as well as the relevant documentation (User's and Developer's Guides) were given to 3 students of the Electrical and Computers Engineering School of the National Technical University of Athens (NTUA). The students were assigned with two tasks. First, they were asked to configure and install the QASST on the Oscar platform. All students succeeded in this task, the completion of which required from them maximum 1 working day. After installing and running the QASST tool, their second task was to understand and experiment with the code, as well as to use the functionality of the QASST. The students' general opinion was that it was easy to alter certain parts of the code, as it's described in detail in the developer's guide. Furthermore, two of the students found quite easy and straightforward the process of adding services with QoS parameters to the repository, adding users with QoS preferences for these services, changing these values and observing the discovery and selection of different services when these values are updated. The only drawback reported by the students concerns the fact that, each time a new service was added to the repository, the QASST had to be restarted. In total, it took the students 3 working days in average to interpret the code, configure it, install it, start the QASST and use it successfully.

As already mentioned, the .NET framework is required for the QASST to run. Thus, portability of QASST is restricted to Windows-based platforms.

With regards to debugging, the implementation of the QASST tool aimed to facilitate the discovery of potential bugs. This is achieved via the provision of detailed logging information in core parts of the source code. Thus, the developer was able to quickly identify the location of the bugs and easily correct them. Furthermore, the QASST has been designed in order to keep all the provided core functionalities distinct and separate, which also accelerates the debugging process.

Concerning the extensibility of QASST, this tool can be extended in two possible ways. The first one is to enhance it so that support for filtering and selection of services that are offered outside the Amigo home is also provided. In order to achieve this, new discovery mechanisms have to be implemented. Once these discovery mechanisms are in place, the QASST can be easily extended to exploit these new discovery facilities. The second way to extend QASST

would be to use an alternative service selection algorithm. This might be useful in cases where specific priorities in selection of services have to be considered. This is easily achieved in QASST due to the reason that the provided core functionalities are distinct and separate. Thus, the developer has to modify only the method that implements the selection of services (more details are available in the developer's guide).

6.3.2 Runtime aspects

The assessment of QASST is completed with the study of the performance / complexity of the selection algorithm. The algorithm is used in order to select a service from a provided list of available services of a specific service type with criteria based on the QoS service properties and the QoS preferences of the user that requested the service. This list of services includes services that are offered only inside the Amigo home via other Amigo components. Thus, the number of these services is expected to be quite low (i.e. not more than 20), which allows for the application of the exhaustive solution algorithm for the service selection problem, even though the problem is NP-complete. This did not cause any noticeable performance deterioration for the algorithm, as the number of services and users is low. Several tests have been conducted with 20 services (a relatively large number) for a specific service type in each of them and 5 users. All the services had several QoS properties with different values and the same held also for the users' QoS preferences. The performance of the selection algorithm was really fast: all tests conducted required far less than 1 sec to run and select the service that addressed best all users' requirements. Of course, in case the QASST is used outside the Amigo home, in cases where thousands of users and services exist, then the exhaustive selection algorithm will have to be replaced by an approximation algorithm of polynomial complexity.

6.4 Event-based Services

6.4.1 Development, configuration, deployment, management aspects

By using the Amigo Event-based Semantic Services Framework, programmers can focus on the service interface in terms of typed high-level operations, namely command and event. Our system manages the event throughout its lifecycle and uses the command operation as is. While commands are used for synchronous communication between services, events can be used for asynchronous communication. It is important to provide both because, in a ubiquitous environment, deployed services use heterogeneous communication modes (*i.e.*, synchronous and asynchronous). The event design pattern for asynchronous communication relies on several low-level synchronous operations, *i.e.*, subscription and notification. Providing a high-level viewpoint prevents programmers from dealing with these low-level operations when it is unnecessary. Semantic services and their instances can then be separately developed. Indeed, developers only express the communication means, command or event, of semantic services in a typed manner. Typed event enables a safer service composition. Valid composition can then be ensured, *i.e.* event consumers handle properly received events from event producers.

In our approach, each event is uniquely associated with a type (and vice versa). Consider the following types: *Luminosity*, *Temperature* or *Availability*. One can define an event for each of these types; the defined event then inherits from the semantics of the corresponding type.

From a more quantitative viewpoint, our approach frees developers from painful, error-prone and repetitive development. Table 6-7 illustrates the benefits of the approach in terms of conciseness for a small example that consists of 3 services. Further, the more services the developer declares, the more bundles the compiler generates. In the example considered, a manager subscribes to an event produced by a sensor service. The manager then controls an actuator according to the event value received. Thanks to the framework, the application

written by a developer only consists of three classes, one for each service instance. The total number of lines, for these three classes, is 175.

	Domain Description in Amigo-S (XML)		Generated Amigo bundles (Java)		Ratio
	3 Services	1 Data type	1 API bundle	3 bundle skeleton	
# of bytes	3 567	335	28 536	27 229	14
# of words	110	22	2 076	2 077	31
# of lines	86	11	974	958	19

Table 6-7: Expansion factor for the Light Manager example

The above Light Manager example is a very simple example. The domain compiler has also been used with the Bluetooth Presence Manager example. This example involves four services and four data types. Its goal is to coordinate Bluetooth readers with a database. The Bluetooth Presence Manager follows Bluetooth tags throughout a building and publishes *Presence* information about the tag owner. In this example, the expansion factor rises to at least 23 (for the size of the files). The domain compiler generates over 26 more lines from the verbose OWL domain description in XML. Note that the empty lines have been ignored, as they are meaningless. Finally, the Bluetooth Presence Manager domain has been written in a couple of hours by a developer who knows the Amigo-S syntax. The skeletons have been filled and debugged in a few hours.

	Domain Description in Amigo-S (XML)		Generated Amigo bundles (Java)		Ratio
	4 Services	4 Data types	1 API bundle	4 bundle skeleton	
# of bytes	4 391	1 084	73 351	54 689	23
# of words	142	61	5 726	3 336	44
# of lines	102	35	2 126	1 479	26

Table 6-8: Expansion factor for the Bluetooth Presence Manager example

To summarize, our approach enables high-level development, thanks to the event abstraction and the code generation, and separation of concerns, thanks to the separation between semantic service description and service instance. A semantic discovery process enables to find event-based service instances by checking their event-based semantic descriptions.

7 Multimedia Content Framework Assessment

The Amigo Multimedia Content Framework package is composed of three components: Content Distribution Service or Interface, the Content Adaptation (Enabled) Digital Media Server (DMS) and Content Discovery. Each of these components is assessed separately.

7.1 Development, configuration, deployment, management aspects

7.1.1 Content Adaptation DMS

The Content Adaptation DMS is thoroughly and successfully used in the Home Information and Entertainment applications. The CADMS enables storage of content in a dynamic interoperable environment by implementing UPnP Digital Media Server interfaces. Furthermore, it provides enhanced adaptation functionalities via a configurable plugin-based content adaptation framework. These functionalities are published via UPnP and are therefore discoverable.

7.1.2 Content Discovery

The Content Discovery allows discovering content dynamically, independently of its physical location. Similarly it can discover renderers. Besides, Content Discovery is in charge of metadata aggregation and semantic functionalities. Content Discovery is configurable and is developed as a bundle within the OSGi framework.

7.1.3 Content Distribution

The content distribution component is a middleware component that enables the discovery and playback of multimedia on an Amigo infrastructure.

It is based on the UPnP standard and hence enables standard UPnP and DLNA equipment to be seamlessly used by services and applications.

The key features of this component are:

- It is the single point of contact for applications and services that want to do something with multimedia content in an Amigo home
- Automatic discovery and embedding of multiple media sources (Digital Media Servers, either embedded in Hardware or in Software)
- Automatic discovery of multiple media renderers (Digital Media Renderers, either embedded in Hardware or in Software)
- Automatic adaptation of content streams for media renderers in order to save bandwidth and guarantee optimal user perceived quality, even on resource constrained devices. New and unknown Media Renderers can be integrated into the Amigo home by using standardized profiles that describe device capabilities.

Like all Amigo middleware components, it provides a simple but powerful interface towards applications and services while using standard Web Services, WS-Discovery and WS-Eventing. Besides the basic functionality of starting, stopping and pausing multimedia sessions, the interface allows advanced functionality like:

- Setting extended properties on multimedia content (e.g. setting metadata like actor name on a movie or modifying the tags on an audio file)
- Supports advanced semantically controlled conversions by adding the appropriate codecs and transcoders (e.g. Text->Speech or Video->Slideshow).

- Allows multimedia files to be relocated within the home (move from a server to another server)

The power of the content distribution middleware component is currently being demonstrated by the MediaManager demo application.

7.2 Runtime aspects

7.2.1 Content Adaptation DMS

7.2.1.1 Performance

The CADMS presents the following average performance measures of 100 tests on a Pentium D 3.40 Ghz/ 3.50 GB RAM.

Measure	Value
Startup time ($T_{startup}$)	4.234 seconds + $T_{synchronizeDB}$
Database synchronization ($T_{synchronizeDB}$)	0.25 sec/file *
Adaptation: Time to respond with a proposal after invoking the *NegotiateAdaptation* method.	0.017 seconds
Time to perform adaptation	Depends highly on plugin performance.
***Search* invocation response time**	0.077 seconds for 3 items 0.189 seconds for 30 items
***Browse* invocation response time**	0.109 seconds for 3 items 0.216 seconds for 20 items
***UpdateObject* invocation response time.**	0.312 seconds

* New files not detected before by the component on previous runs. Average file size 6 MB

7.2.1.2 Resource Consumption

Requires at least hardware with the following characteristics:

RAM consumption by component	22 MB
Processor	Pentium III

7.2.1.3 Scalability

UPnP/SOAP is not exactly a lightweight protocol. Thus, recursive invocations to Browse actions may have scalability problems due mainly to network bandwith consumption and associated delay.

7.2.1.4 Robustness

In its current version, the CADMS has been running for weeks in the integrated HIE demonstrator without any crashes.

7.2.2 Content Discovery

7.2.2.1 Performance

The Content Discovery presents the following average performance measures of 100 tests on a Pentium D 3.40 Ghz/ 3.50 GB RAM.

*average file size 6 MB

Measure	Value
Startup time ($T_{startup}$)	3.823 seconds + $T_{synchronizeDB}$
Database synchronization ($T_{synchronizeDB}$)	0.25 sec/file *
***Search* invocation response time**	0.171 seconds for 3 items 0.656 seconds for 30 items
***Browse* invocation response time**	0.062 seconds for 3 items 0.102 seconds for 30 items
***UpdateObject* invocation response time.**	0.765 seconds

7.2.2.2 Resource Consumption

Requires at least hardware with the following characteristics:

RAM consumption by component	30 MB
Processor	Pentium III

7.2.2.3 Scalability

UPnP/SOAP is not precisely a lightweight protocol. Thus, recursive invocations to Browse actions may have scalability problems due mainly to network bandwith consumption and associated delay.

7.2.2.4 Robustness

In its current version, the Content Discovery has been running for weeks in the integrated HIE demonstrator without any crashes.

8 Datastore Framework Assessment

8.1 Development, configuration, deployment, management aspects

The Datastore is a .Net middleware component in the Amigo architecture available to any other service or component that needs to store some form of data.

The Datastore abstracts a concrete database implementation and adds extra functionality to it following the standard Amigo Service Oriented Architecture. This means that it can be accessed using standard Web Services, can be located through WS-Discovery and that notifications are send through WS-Eventing. This enables interoperable and cross platform usage of this component.

The abstracted interface is intentionally kept simple to enable a short learning curve for Service or Application developers. Developers do not need to know low-level details or learn specific languages like for example SQL in order to use the component, yet they still benefit from the full performance and reliability of the backend. The volume of data that can be stored is only dependent on the capabilities of the backend, which are usually only limited by available hard disk capacity.

The Datastore was designed with a zero maintenance effort in mind. This means that the Datastore itself will automatically backup its contents, and, on the other side, detect any defects and automatically repair its contents.

The Datastore is currently being used by the content distribution middleware and indirectly by the MediaManager demo application.

9 Assessment of Home Configuration with VantagePoint

9.1 Development, configuration, deployment, management aspects

9.1.1 Introduction

VantagePoint is a tool for the developer of Amigo applications and services. It offers views to semantic context data and a graphical user interface to the Context Broker and Semantic Service Repository, thus making the management of both components easier in the application development phase. This chapter presents the assessment of VantagePoint.

The assessment of VantagePoint was carried out with a qualitative and empirical approach. The evaluators had to work with VantagePoint and perform given tasks, and respond to a questionnaire afterwards. From the results of both oral and written feedback, we gathered the results compiled in Section 9.1.2.

VantagePoint was to be assessed more from the developer and usability viewpoints, since it is not an actual middleware component nor does it affect the middleware efficiency at all. The ease of use and added value in Amigo application and service development as well as middleware component management are more interesting aspects to assess in the context of VantagePoint. Hence the runtime and performance evaluation of VantagePoint is discarded in this assessment.

Most of the time, developers using VantagePoint do not use it at the API level but through its GUI. Indeed, developers deal with the Amigo middleware API when developing their applications and use the VantagePoint user interface to observe context sources and service registries. This suggests that API level assessment can also be considered not as interesting as evaluating the actual tool.

While VantagePoint assessment falls into the category of evaluating a software tool like a product, we stress that VantagePoint is software created entirely within a research project. Creating professional user interfaces is another research topic in its own. Nevertheless, in our questionnaire, we asked multiple questions about the usability of the GUI. Even if we had predicted that most of the critique would deal with the shortcomings or complexity of the user interface, regardless of how the user interface applies the guidelines of usability, we wanted to see how useful the evaluators see the features of VantagePoint. Is the whole idea of having such a tool worthwhile? Does it offer any help in stepping into the world of semantic context data, context sources and service registries when developing applications on top of the Amigo middleware?

The questionnaire used a 5 level grading system where 5 is the best and 1 the worst grade, 3 being the average. After each question, there was the possibility to add free comments, and the evaluators did use this opportunity. The phase where each evaluator worked with VantagePoint was monitored, but the questionnaires were returned anonymously.

The tasks that the evaluators had to do included: starting up VantagePoint, creating semantic house models and registering them as context sources in the Amigo middleware, creating own item libraries, registering services with the semantic service repository, and testing context sources and service discovery. The evaluators were given written instructions of how to perform the tasks, but these were not very detailed: this was to help in assessing the usability and ease of learning of VantagePoint.

The evaluators were VTT staff of various backgrounds and Amigo developers. The number of evaluators that performed tasks with VantagePoint was 9. Even though the amount of

participants was not that high, they were rich in comments, and the results indicate clear high and low points.

9.1.2 Assessment Results

In Figure 9-1, the average grades are given from all the questions. The questions with zero grades did not have grading possibility and are handled together with the comments from all questions.

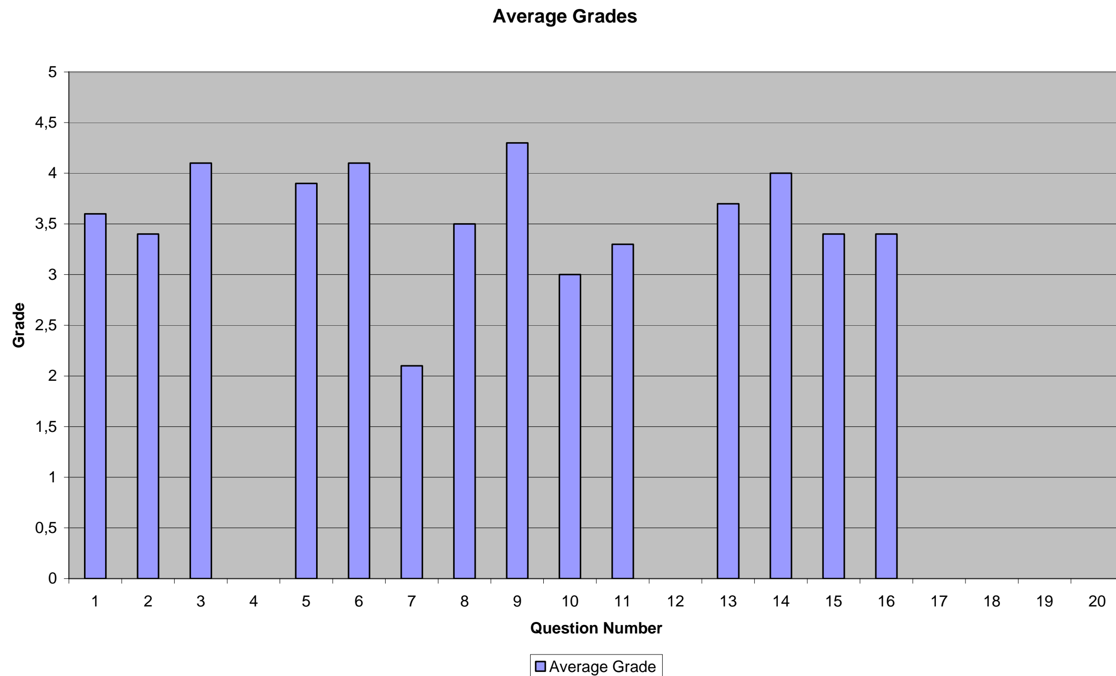


Figure 9-1: The average grades from all questions of the VantagePoint questionnaire.

Below is a list of all the questions and some of the most interesting comments that the evaluators gave during the assessment. Notice that comments were given also to questions with grading possibility.

1. How would you rate your overall experience with VantagePoint?

Comments:

“Overall experience was a bit confusing.”

“Some difficulties, but learning was fast.”

“Nice GUI, at first a bit confusing.”

2. How easy was the graphical user interface to follow?

Comments:

“Easy enough, at least with the instructions”

“Some difficulties to understand the meaning of some fields and finding the right buttons etc.”

“Too cryptic messages in the information area”

3: How easy it was to install VantagePoint?

Comments:

"Pretty straightforward OSGi bundle installation"

4: Have you worked with OSGi bundles before?

Most respondents had at least some kind of experience in working with OSGi bundles

Comments:

"Yes, I have developed some and used many"

5: How easy it was to create a house model?

Comments:

"Logical operations"

"I wish I could drag 'n drop items straight from the list"

"I'd like to have different kind of floors"

"Once I learned how to use the GUI, it was very easy."

6: How easy it was to register context sources?

Comments:

"It was easy"

"Just press the button"

"Output could be clearer and delays shorter"

7: How easy it was to build items?

Comments:

"You need to know RDF/OWL"

"There is no GUI support for this task"

"Why not an integrated tool?"

"I did not find the MyItems.owl"

8: How easy it was to register services?

Comments:

"More difficult than context sources with those descriptions and groundings"

"Output was not clear to non-expert like me"

9: How useful do you find the possibility to test context source and service discovery with the graphical user interface?

Comments:

"GUI is always nice compared with command prompts"

"This is honey for the developer"

"Useful, but not intuitive for a layman"

"This is a very powerful idea"

10: How useful do you find the VantagePoint console logging all the method calls for context sources and service registries?

Comments:

"Useful if you understand what you are doing…"

"Very useful but mystical"

"Could be useful for a Guru user"

"Good for debugging"

11: How good was the installation and quick start guide for VantagePoint?

Comments:

"Guide needs some editing"

"A little confusing in some parts"

12: Have you used VantagePoint to test your context-aware Amigo application? If yes, how have you used it?

Most of the evaluators had not used VantagePoint to test Amigo applications.

Comments:

"To receive context events"

"I will use it as a location context source for the user"

13: How easy it was to learn how to use VantagePoint?

Comments:

"Pretty easy, some difficulties in understanding fields in the GUI."

"GUI helps a lot but is not yet completely self-explanatory."

"A how-to would be a great help"

14: Estimate the overall added efficiency in the development process of Amigo context-aware applications when using VantagePoint

Comments:

“It is very handy indeed but quite complex.”

“VantagePoint clarifies the process.”

15: If you were to develop a context-aware service or an application to Amigo middleware, do you feel that you could save time by using VantagePoint? If so, how much?

Comments:

“I feel that I wouldn’t save much time because I’d play around with it too much.”

“I guess a lot, because it helps the testing work so much.”

“The time of developing my own location (or other) context source.”

16: How useful do you consider VantagePoint in general when developing with the Amigo Middleware?

Comments:

“Works great with the middleware and is a concrete example of an application that deploys the services it provides.”

“It’s good for the CMS and semantic service repository parts only.”

17: Did you encounter any performance issues?

Most evaluators encountered some kind of problems

Comments:

“Yes, some delays were too long.”

“A couple of short waiting periods”

“A little delay with the semantic service repository when adding items with services”

18: Did you encounter any quality issues? (e.g. the number of bugs encountered, responses to bugs found, etc.)

Few evaluators encountered problems

Some comments:

“One with Oscar”

“Some bundle crashed in the beginning.”

“The interface is not very intuitive”

19: Did you encounter any interoperability issues? (e.g. between different operating systems, types of host machine, etc.)

No one did.

Comments:

“Worked even with an abnormal resolution”

20: This is the last one. Here you can give free feedback, comments, suggestions or whatever about VantagePoint.

Comments:

“More test work in order to find all the bugs.”

“Actions that cause delays should be put into their own threads so they won’t block the GUI”

“It would be nice to see easily to which item a service is attached to”

“Nice tool, it is a pity we didn't have it at the beginning of the project”

“I didn’t fully understand what I was doing but otherwise the application seemed cool. GUI was easy to use”

“A nice piece of software that could be refined to commercial product class with a bit more work and extra developers”

9.1.3 Discussion

In general the assessment was very useful and revealed some extremely interesting points about the VantagePoint tool. With the help of this assessment, we discovered which features of VantagePoint users find useful and easy to use and which parts of the application still need to be improved. The assessment consisted of two parts: the user testing and the questionnaire. The results produced by these two phases are further discussed in this section.

To start with, the overall experience of working with VantagePoint was considered as positive. The graphical user interface was functional and easy to learn, at least with instructions. The technologies and topics covered in Amigo were a bit unfamiliar for a big part of the respondents, which caused confusion. For example, the messages produced by the application were in many cases found to be too cryptic. Also the true meaning of some functions and/or buttons was hard to understand for some respondents. But in general, the graphical appearance and the user interface of the application received positive feedback.

The test situation included various tasks. Creating new models, adding areas and items, and registering a model as a context source to the Amigo network were considered as easy tasks in general. On the contrary, building one’s own items was judged complicated by many users, because they had to edit RDF descriptions with a text editor. A number of respondents commented that there should be a graphical editor within VantagePoint to support this task, which is valuable feedback for the developer team. Finally, adding services was considered more difficult than adding context sources.

Besides questions about the actual use of VantagePoint, the questionnaire included questions about the experienced usefulness of the tool. In general, VantagePoint was considered to support well the process of creating context aware services and applications on top of the Amigo middleware. The time taken to create one’s own context sources is noticeably decreased, and the monitoring of the context sources and service registries was found to ease debugging. Nevertheless, many respondents still remarked upon the complexity of the tool. In addition, one respondent thought that using VantagePoint would not save much time because he/she would play around with it too much.

VantagePoint itself did not cause any quality issues during the user tests. However, most respondents reported some performance issues, especially when adding new services into the model. This is a matter that most certainly needs attention in the future development work on VantagePoint. In a nutshell, VantagePoint was considered as a useful and nice-looking but a bit complex tool.

10 Summary

This document presented the results of a number of assessment efforts mainly of the Amigo Base Middleware, and to a small extent, of the Amigo Middleware (Base Middleware and Intelligent User Services) as a whole.

Chapter 2 described the findings of the internal Amigo developer survey, which showed that the majority of developers felt that the Amigo Middleware as a whole was easy to use, they did not experience performance or quality problems, and that the Amigo Middleware enabled new scenarios that would not have been possible, or would have been difficult to do, without it.

In Chapter 3, the assessment of the Amigo Programming and Deployment Frameworks exhibited the high rate of interoperable platform interaction the frameworks afford, as well as resolving performance issues on the Windows platform.

Chapter 4 presented the Amigo Interoperability Framework, demonstrating not only its utility, but also that the performance overhead of using the framework is negligible.

The ease of use and inter-home capabilities of the Amigo Security Framework were discussed in Chapter 5.

Chapter 6 presented assessment of both development aspects and runtime aspects of the complex service workflows, context aware, quality of service aware, and event-based parts of the Amigo Semantic Service Framework, while Chapter 7 provided similar assessment for the Amigo Multimedia Content Framework.

The utility of the Amigo Datastore was examined in Chapter 8.

And finally, Chapter **Error! Reference source not found.** presented the results of the VantagePoint-specific questionnaire, which showed that VantagePoint was considered useful for visualising and configuring services in the Amigo home, and was also pleasant to use.

Overall, this assessment annex has demonstrated that the prototype implementations developed for the Amigo Base Middleware provide a comprehensive, efficiently usable, effective, and sufficiently performing platform for developing novel applications for the Amigo networked home.

11 Resources

[D3.5] Amigo D3.5 Amigo overall middleware: Final prototype implementation & documentation - Final integrated methodology ('how to') for employing the middleware. Available on-line at: http://www.hitech-projects.com/euprojects/amigo/deliverables.htm.

[EG02] R. van Engelen, K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In Proc. IEEE International Symposium on Cluster Computing and the Grid, 2002.

[Eyeball] Eyeball: a tool for checking RDF/OWL for common problems. See: http://jena.sourceforge.net/Eyeball/.

[OSGi/.NET] The OSGi and .Net Programming and Deployment Framework User's Guides are available at: https://gforge.inria.fr/frs/?group_id=160.

[SD-SDCAE] The SD-SDCAE User's Guide is available at: https://gforge.inria.fr/frs/?group_id=160.