

IST Amigo Project Deliverable D4.7

Intelligent User Services

2 - Context Management Service

Software Developer's Guide

IST-2004-004182
Public



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

Deliverable Number	:	D4.7 (CMS contribution)
Title of Deliverable	:	2 – Context Management Service - Software Developer's Guide
Nature of Deliverable	:	Public
Internal Document Number	:	amigo_2_d4.7_final
Contractual Delivery Date	:	30 November 2007
Actual Delivery Date	:	14 January_2008
Contributing WPs	:	WP4
Author(s)	:	Peter Vink (Philips, editor), Fano Ramparany (France Telecom), Remco Poortinga (Telin), Dirk-Jan van Dijk (Telin), Jörg Schmalenströer (University Paderborn), Volker Leutnant (University Paderborn) Christophe Cerisera (LORIA), Xabier Mardaras (Ikerlan), Sanna Kallio (VTT), Elena Vildjiounaite (VTT), Timo Urhema (VTT), Maja Stikic (Fraunhofer IPSI)

Abstract

This document is the updated developers guide for the Context Management Service components, it describes how to install, use and edit the components of the CMS

Keyword list

Context, Context management

Table of Contents

Abstract.....	1
Keyword list	1
Table of Contents	2
1 Component Overview	6
1.1 Context Source / Context Wrapper	7
1.2 Context Broker	9
1.3 Context Interpreter.....	10
1.4 Context History	12
1.5 Acoustic Position Estimation Sensor	13
1.6 RF Positioning Estimation	15
1.7 Topic Recognition Sensor	16
1.8 Domotic Sensor Information.....	18
1.9 Configuration Information.....	19
2 Deployment	21
2.1 Context Broker	21
2.1.1 System requirements.....	21
2.1.2 Download.....	21
2.1.3 Install	21
2.1.4 Configure	21
2.1.5 Compile.....	21
2.2 Context Source / Context Wrapper	22
2.2.1 System requirements.....	22
2.2.2 Download.....	22
2.2.3 Install	22
2.2.4 Configure	22
2.2.5 Compile.....	22
2.3 Context Interpreter.....	22
2.3.1 System requirements.....	22
2.3.2 Download.....	22
2.3.3 Install	22
2.3.4 Configure	22
2.3.5 Compile.....	23
2.4 Context History	23
2.4.1 System requirements.....	23

2.4.2	Download	24
2.4.3	Install	24
2.4.4	Configure	24
2.4.5	Compile.....	24
2.5	Acoustic Position Estimation Sensor	24
2.5.1	System requirements.....	24
2.5.2	Download.....	24
2.5.3	Install	24
2.5.4	Configure	24
2.5.5	Compile.....	24
2.5.6	Deployment.....	24
2.6	RF Positioning Estimation	25
2.6.1	System requirements.....	25
2.6.2	Download.....	25
2.6.3	Install	25
2.6.4	Configure	25
2.6.5	Compile.....	25
2.7	Topic recognition sensor	26
2.7.1	System requirements.....	26
2.7.2	Download.....	26
2.7.3	Install	26
2.7.4	Configure	26
2.7.5	Compile.....	26
2.8	Domotic sensor information	27
2.8.1	System requirements.....	27
2.8.2	Download.....	27
2.8.3	Install	27
2.8.4	Configure	27
2.8.5	Compile.....	27
3	Component Architecture.....	28
3.1	Component interface	28
3.1.1	Context Information	28
3.1.2	Describing Context Source capabilities	28
3.1.3	Discovering Context Sources	29
3.1.4	Querying context information.....	30
3.2	Context Interpreter interface.....	30
3.2.1	AND, OR and NotAND rules.....	31
3.2.2	FindFact function	32
3.2.3	FindBestFact function	32
3.2.4	getUsersAvailableTime function	33
3.3	Mechanisms of interaction.....	33
3.4	Overview and reference to internals	34

3.4.1	Context Source helper bundle	34
3.4.2	Context Consumer helper bundle	39
3.5	Detailed documentation	42
4	Tutorial.....	43
4.1	Introduction	43
4.2	Software requirements	44
4.3	Oscar.....	45
4.3.1	Installation.....	45
4.3.2	GUI	45
4.3.3	Installing bundles	45
4.3.4	Repositories.....	47
4.3.5	Stopping Oscar	47
4.4	CMS Context Broker	48
4.4.1	Installation by Oscar	48
4.4.2	Building the CMS broker.....	48
4.4.3	Building the Amigo Context Client / Context Source example	49
4.5	Example Context Source and Client	49
4.6	Writing a Context Source	51
4.6.1	Exporting the AmigoService	52
4.6.2	Context Source Control behaviour.....	55
4.6.3	The Ontology Model	56
4.7	Writing a Context Client	60
4.7.1	Finding a Context Broker	61
4.7.2	Finding the Context Sources	61
4.7.3	Subscribing to a Context Source	62
4.7.4	Receiving and Processing Context Information.....	62
4.7.5	Unsubscribing from a Context Source	63
4.8	Tutorial Context Source and Client.....	64
4.8.1	Preparing the Projects for being imported into Eclipse	64
4.8.2	Importing the Projects into the Eclipse Workspace	65
4.8.3	Setting Project Variables for Oscar.....	65
4.8.4	Completing the Context Source.....	66
4.8.5	Completing the Context Client.....	70
4.8.6	Installing our Bundles	72
4.9	Writing Context Sources and Client using Helper components.....	74
4.9.1	OSGi helper component for Context Sources	74
4.9.2	.NET helper component for Context Sources	78
4.9.3	OSGi Helper component for Context clients.....	81
4.9.4	.NET helper component for Context clients	84
5	Deploying and using T4.1 Context Sources.....	88
5.1.1	Context History	88

5.1.2	Acoustic Position Estimator	89
5.1.3	RF Positioning	91
5.1.4	Topic Recognition	92
5.1.5	Domotic Sensors	93
5.2	Debugging with eclipse.....	94
5.3	Troubleshooting.....	94
6	Appendix	96
6.1	Description of context capabilities	96
6.1.1	Identification of entities	96
6.1.2	Modeling of context.....	96
6.1.3	ContextParameter (Generic).....	97
6.1.4	ContextParameter (Specific).....	97
6.1.5	Summary	100
6.2	Context Interpreter data and syntax of function calls.....	101
6.2.1	CI function input/ output format.....	103
6.2.2	List of predefined terms and symbols:	103
6.3	Description of tools/languages provided by the CMS.....	104
6.3.1	Test tool for testing newly developed context sources and SPARQL queries..	104
6.3.2	Test tool for the RF positioning monitor	104
6.3.3	Configuration tool for the RF Position monitor	105
6.4	Domotic / Sensor information Demonstrator	108
6.4.1	Context Sources	108
6.4.2	Context Capabilities.....	108
6.4.3	Context Queries and Events.....	109
6.4.4	Domotic execution process.....	109
6.5	Topic recognizer	110
6.5.1	Quick guide to test the topic recognizer.....	110
6.5.2	How to adapt the system to your own topics and needs	111
6.5.3	Guide to create an Amigo service that uses the topic recognizer.....	112
6.5.4	How to adapt the system to your voice and microphone (Implicit speech part)	113
6.5.5	Workplan for the next versions	114
	References	115

1 Component Overview

The Context Management Service consists of the following main components; all these components are services by themselves.

Context Source. This is an abstract definition for any service that provides context information using the Amigo interface *IContextSource*. Several explicit context sources have been developed (see below),

Context Broker. This service is responsible for the registration and discovery of context sources. Context Sources can register their capabilities using the Amigo interface *IContextBroker*, while context consumers can use the same interface to discover appropriate context sources by specifying their context source needs.

Context Wrapper. This is a generic term for a context source that wraps raw context information (such as, for example, sensors) and provides to the CMS that information with an *IContextSource* interface.

Context Interpreter. This is a generic term for a component that combines information coming from (possibly multiple) other Context Source(s) and provides higher level information as a result. A Context Interpreter may rely on reasoning techniques.

Context History. The Context History component handles recorded histories of the users' interactions in context, which is context data that is needed as time series by a substantial number of applications and services and gathered from context sources.

The Context Wrapper, Context History and Context Interpreter can all be regarded as context sources. Furthermore, the Context Management Service has implemented a number of specific context sources, including an Audio Positioning Estimator, a RF Positioning Estimator, a Topic Recogniser, a Domotic Sensor Information service and a Configuration Service.

These services have been implemented using context wrappers, both OSGi and .NET Amigo frameworks (see [Ami06c]) have been used for implementing context sources.

Furthermore, the Context Management Service also provides helper libraries that can be used by a client application to facilitate communication with Context Sources as well as helper libraries facilitating the development of Context Sources themselves. These helper libraries are available as separate bundles for the OSGi framework and as link libraries (DLLs) for the .NET framework. The helper libraries for client applications also provide support for dealing with SPARQL results.

Further details about the helper libraries can be found in section 3.4 and in the tutorial chapter.

Finally, a close relation exists to the work in Amigo deliverable D3.x, "Location Management" and D6.y. In the location management task, various context sources from several other projects (including Awareness <http://awareness-freeband.nl>) and Amiloc [Amiloc04]) have been wrapped and added to the Amigo Context network, these include: **Device based Positioning Estimation** (e.g. via Bluetooth devices, badges, WIFI), **Device status** (e.g. power) and **Scheduled activity** (e.g. via agenda).

The management service adds the following: **Configuration** and **Service status**

The services that were introduced in the first two paragraphs are further treated in this document.

1.1 Context Source / Context Wrapper

Provider

TELIN

Introduction

The context source (as a base class) provides both synchronous and subscription-based interfaces. For developing Context Sources two helper libraries are available, one for every deployment framework within Amigo. Additionally there are also helper libraries for developing Context Clients, also for both deployment frameworks.

Development status

Final version available for .NET and OSGi deployment frameworks.

Intended audience

Developers

License

Open source, LGPL license

Language

Java, C# (.NET CLR)

Environment (set-up) info needed if you want to run this sw (service)

For the OSGi version: PC, JVM, OSGi-Based Programming & Deployment Framework from WP3, context broker

For the .NET version: PC, .NET deployment framework, context broker.

Platform

JVM, OSGi-Based Programming & Deployment Framework from WP3

.NET CLR, .NET deployment framework.

Tools

Oscar, Java IDE such as Eclipse, or Visual Studio (Express) for .NET based Context Sources.

Files

The base functionalities for context sources and clients are available from the following location in the gforge repository: [amigo]\ius\context_mgmt\common\trunk

The list of available components and their function is shown below

NAME	PLATFORM	FUNCTION
ContextSourceManager	OSGi	Context Source helper bundle, takes care of the common tasks of a context source.
ContextSourceHelper.NET	.NET	Context Source helper library, takes care of the common tasks of a context source.
ContextHelper	OSGi	Context Client helper bundle, takes care of the common tasks of a context client.
ContextHelper.NET	.NET	Context Client helper library, takes care of the common tasks of a context client.
ContextSourcePushExample	OSGi	Example context source bundle, making use of the helper components
ContextSourcePullExample	OSGi	Example context source bundle, making use of the helper components
ExampleContextSource.NET	.NET	Example context source executable, making use of the helper components
ContextSourceTester	OSGi	Example context client bundle, making use of the helper components. The Context Source Tester allows developers to fill in their own rdfNeeds fields and specify SPARQL queries. This can be used for testing (new) context sources or testing SPARQL queries on existing ones.
ContextSourceManager.NET	.NET	Management application onto which .NET context sources can be dropped (if they are developed with the .NET helper libraries), allowing multiple context sources to coexist in one run-time application (keeping the desktop clean and providing one logging window).

Note that the example Context Sources provided are for tutorial purposes only, and have no direct practical value as a Context Source. A practical Bluetooth Context Source which can also be used as an example is provided at:

[amigo]/ius/context_mgmt/bluetooth

Note that the OSGi bundles are also available from the CMS OBR at <http://core.lab.telin.nl/~amigo/obr/repository.xml> This CMS OBR is automatically included in the OBR window of Oscar when the main Amigo repository is configured (<http://amigo.gforge.inria.fr/obr/v2/repository.xml>). See also the tutorial section for more information.

Documents

Design can be found in D4.1 and D4.2, [Ami05g] [Ami06b].

Further information is available in the form of a tutorial, also included in this document.

Tasks

Bugs

None known at the moment.

Patches

None

1.2 Context Broker**Provider**

FT

Introduction

The context broker is a directory service, which is aware of context sources that are actually available. Based on this information it is able to connect context providers to context consumers. Conceptually, this information could be viewed as a table of context sources identifiers indexed by context source capabilities. When multiple context brokers are deployed within the same network domain, discovery requests from clients will be passed to other context brokers as well. In other words: it is possible to run multiple context brokers within the same (network) domain.

Development status

Final version available.

Intended audience

Context aware application developers and context source developers

License

LGPL

Language Java**Environment (set-up) info**

PC, Jena.

Platform

JVM

Tools:

OSCAR

Files**Documents**

Javadoc

Tasks

Development.

Bugs

None yet

Patches

None yet

1.3 Context Interpreter

Provider

TELIN, FT, VTT

Introduction

A context Interpreter is a component that takes (low-level) input from various context sources and combines the information into higher-level context information. An example of this is a user location management context source that combines information from low level sensors, such as Bluetooth, RFID readers, RF and acoustic positioning, into user locations. Other examples of context reasoning are recognition of user activity (e.g., sleeping) or the need to find the most suitable for the task at hand person or device. Depending on a task, “most suitable” can be nearest device with loudspeakers (e.g., if the task is to deliver an audio message to a user) or a person whose free time before the nearest appointment is longer than free time of other family members (e.g., if it is needed to drive a child somewhere, or to supervise some long process like child’s homework or software installation). Context Interpreter provides a set of fairly generic functions which can be used with application-specific parameters, for example, applications can create the rule “if the user is in room where light level is low and sound level is low, then the user is sleeping”; or applications can specify that such context data means “sleeping” activity only at night (during daytime family members can play hide-and-seek)

As default, reasoning is performed on current and future context, that is, on the most recent data from Context Sources and on user appointments ahead of current time.

Development status

Final version for .NET available.

Indented audience

Context aware application developers and context source developers

License

LGPL

Language

User Location: Java.

Rule-Based reasoning : C# and C++ dll

Environment (set-up) info

User Location: PC, Jena, Oscar.

Rule-Based reasoning : Microsoft .NET Framework v2.0, Amigo .NET programming framework

Platform

User Location: OSGi/Java, Oscar.

Rule-Based reasoning : Microsoft .NET Framework v2.0, Amigo .NET programming framework

Tools

User Location: Oscar, Eclipse.

Rule-Based reasoning : Microsoft Visual Studio 2005

Files

User Location: -

Rule-Based reasoning : The C# source codes and C++ dll can be downloaded from the gforge repository under the [amigo] / ius / context_mgmt / context_interpreter /

Documents

The general description of the Context Interpreter module is given in deliverable D4.2.

Tasks

User Location: The implementation of the first version

Rule-Based reasoning : implemented and provides a set of functions which were named as desirable by Amigo application workpackages.

.

Bugs

N/A.

Patches

N/A

1.4 Context History**Provider**

Fraunhofer IPSI

Introduction

The context history component contributes to the context management intelligent user service. Based on context histories, the context management service will provide data that helps AMIGO applications to get to know users over time and if possible do predictions based on patterns found. Reasoning based on context histories, as opposed to ontology-based reasoning, will especially take into account an enhanced understanding of the users' interactions over time and be embedded into and related to their other interactions happening in parallel, before, or afterwards (time-based reasoning).

Development status

Pre-release version for .NET available.

Indented audience

Project partners.

License

The software itself is under LGPL license, but it might make use of proprietary binaries/libraries for which no source code is provided.

Language

C#

Environment (set-up) info

PC, Microsoft .NET Framework v2.0, Amigo .NET programming framework (from WP3), Context Broker component (from WP4/Context Management Service)

Platform

Microsoft .NET Framework v2.0, Amigo .NET programming framework (from WP3)

Tools

Microsoft Visual Studio 2005

Files

The following location in gforge repository contains the source code for the pre-release version:

[amigo] / ius / context_mgmt / context_histories

Documents

The Context History component architecture is described in D4.2 [Ami06b]

Tasks

Implementation of more complex functionality, such as asynchronous subscriptions based on the WS-Eventing mechanism.

Improving the documentation.

Bugs

None yet.

Patches

None

1.5 Acoustic Position Estimation Sensor

Provider

University of Paderborn

Introduction

The acoustic position estimation sensor evaluates the speech signals captured by multi-channel acoustic sensors. These sensors or microphones are linearly arranged in a microphone array. With each array it will be possible to estimate the “direction-of-arrival”, i.e. the angle towards an active (currently speaking) person. It is not possible to track a non-speaking person with these sensors, since then no input data is available. A group of speaking persons is handled by the sensor, by detecting the loudest speaker. The combination of two microphone arrays allows a position estimate in Cartesian coordinates, if the placement of the microphone arrays and the shape of the room are known. Favourable is the use of two microphone arrays on two different walls, for example on top of two ambient displays. Sensor data is provided through a context source wrapper, which registers to the context broker service. The software is split in two parts, which is the closed-source part called “Spark” and the open source part of the context wrapper. Spark (speech processing and recognition kit) is a software toolbox for speech/acoustic signal processing, developed by the University of Paderborn. The download contains a subset of Spark specifically designed for acoustic position estimation.

Development status

First version of closed-source software part is available. First version of open-source context wrapper is available.

Indented audience

Project partners at first and then later also other developers of custom AMIGO services and applications.

License

This software is proprietary software by PAE. It will not be made open source since it is not middleware software. Special arrangements can be made with Amigo partners.

Language

Close-source software: C/C++

Context Source Wrapper: Java

Environment (set-up) info

Hardware: Linux PC, Multi-Channel soundcard

Software: SuSe Linux 9.3 or higher, Jack Audio Connection Kit, Java Runtime Environment

Platform

SuSe Linux 9.3 or higher

Tools

Java compiler for compiling the source code of the wrapper

Files

One executable (asreng – main engine), modular based shared libraries and special configuration files.

Context wrapper consists of one jar-file, which has to be deployed on the Oscar platform.

Documents

Design can be found in D4.1 and D4.2 [Ami05g] [Ami06b], Designers guide is this document.

Tasks

Tasks to improve/further develop software under development

Bugs

No known bugs

Patches

Intermediate critical patches (as we only make full releases every 6 months).

1.6 RF Positioning Estimation**Provider**

Philips

Introduction

Provides location information of users and (portable) objects based on active RF tag. Intended as example context source for C# environment

Development status

First working version available, room accurate,

Intended audience

Whoever reads this.

License

Philips license, BSD style

Uses *Semweb* (*Joshua Tauberer*), which is released under Creative Commons Attribution License

Semweb uses *Sparql engine for Java* (*Ryan Levering*) converted from java to C# with *IVKM* (*Jeroen Frijters*), which is released under GNU classpath license.

These libraries have not been changed.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

This service only works with specific hardware:

Sensite solutions HBL100 (at least one, typically 3-4)

Sensite solutions tags

PC with free serial port

Windows

Platform

.NET Amigo framework + Microsoft webserver

Tools

A small test tool for positioning monitor was developed (explained in Appendix)

A configuration tool is planned (to assign unique Amigo ID to Tags, rooms etc and calibrate the system) (TBD)

Files

Add list

Documents

Design can be found in D4.1 and D4.2 [Ami05g] [Ami06b], Designers guide is this document.

Tasks

Adapt software to privacy guidelines from T4.6

Use configuration context source to fill config files (create config tool)

Bugs

Known bugs

Patches

Feature: Position monitor returns exception info when a sparql query that was send was not accepted.

1.7 Topic Recognition Sensor

Provider

INRIA

Introduction

This module provides contextual information about the topic of discussion between users. Obviously, such information will be available only when someone (the user) is talking. A typical scenario is when several people are talking together, either face to face or via teleconferencing or phone. However, the information provided by this module can also be computed when the user is listening to the radio, or when he is listening to a recorded talk.

Development status

Second version implemented: it is now fully integrated as a CMS OSCAR bundle, with RDF description and SPARQL support

Intended audience

Project partners

License

LGPL

Language

JAVA and C

Environment (set-up) info

Requires a real-time large vocabulary continuous speech recognizer; a default lightweight keyword spotting component, as part of the UIS implicit speech subtask, is shipped within the main topic recognition bundle. This keyword recognizer is based on speaker-independent native English acoustic models. Some tools are available to train speaker-dependent acoustic models in the implicit speech repository: please refer to the User Guide for a detailed description of the related process.

.

Platform Hardware

PC with Windows XP and Cygwin, JDK version 1.5, OSCAR with all the required CMS bundles.

Tools

Some tools are available to train speaker-dependent acoustic models in the implicit speech repository: please refer to the User Guide for a detailed description of the related process.

Files Source code

Available for download at GForge

Documents

User guide available for download at GForge

Tasks

Simplify the personalization process

Bugs

No bugs reported for now

Patches

No patches created for now

1.8 Domotic Sensor Information

Provider

Ikerlan

Introduction

Provides information from domotic appliances and sensors. A wrapper (Gateway) sends/receives all the domotic commands from physical devices using different communication interfaces (RS-232, Bluetooth,...). A specific Context Source is generated for each detected device.

Development status

First working version available, need more work to comply with all the Amigo interfaces and ontologies.

Intended audience

Project partners.

License

This software is proprietary software by Ikerlan. It will not be made open source since it is not middleware software. Special arrangements can be made with Amigo partners.

Language

C#

Environment (set-up) info

Hardware: Windows PC Software: Windows XP, .NET Framework 2.0

Fagor domotic controller PC version + RS232, to communicate with devices

Platform

.NET Amigo framework + Microsoft webserver

Formatted: English (U.S.)

Tools

Visual C# .NET 2005 for compiling the source code

Files

They will be available for download at GForge

Documents

Design can be found in D4.1 and D4.2 [Ami05g] [Ami06b], Designers guide is this document.

Tasks

- Build a complete, running example
- Test, debug, validate with real devices, now a simulator is used.

Bugs

No bugs reported for now

Patches

No patches created for now

1.9 Configuration Information

Provider

Philips /Telin

Introduction

Provides configuration information that enables applications to refer to the same instances. Examples are: the names (ids) of the users in a home (and the known guests), the names of the rooms (areas) in the home, the names and positions of (fixed) objects in the home such as displays, refrigerator, etc.. Data will be obtained from Amigo Control Center (WP3, task 10) and Amigo VantagePoint tool (WP3, task 1)

Development status

Available, linked to 3.10 and 3.1, partial link to 3.1

Intended audience

Whoever reads this.

License

Philips license, BSD style

Uses *Semweb* (Joshua Tauberer), which is released under Creative Commons Attribution License

Semweb uses *Sparql engine for Java* (Ryan Levering) converted from java to C# with *IVKM* (Jeroen Frijters), which is released under GNU classpath license.

These libraries have not been changed.

Language

C#

Environment (set-up) info needed if you want to run this sw (service)

Needs an owl description of the home, generated in VantagePoint.

Platform

Formatted: English (U.S.)

.NET Amigo framework + Microsoft webserver

Tools

No tools

Files

Add list

Documents

Configuration context source is a standard context source as described in D4_2 Designers guide is this document.

Tasks

Use helper components

Create link to 3.10

Work out link to 3.1

Bugs

Context data from Vantage Point file is not translated correctly to contextparameter ontology

Patches

Feature: Configuration Service returns exception info when a sparql query that was send was not accepted.

2 Deployment

An operational context management service typically consists of at least one context broker, one or more specific context sources, and one or more context clients. The Context Interpreter and Context history components can be added to any setup to add richer context information.

Each component type within the Context Management Service has its dedicated requirements.

For more detailed step-by-step instruction on setting up a CMS environment, please also refer to the tutorial sections of this document.

2.1 Context Broker

2.1.1 System requirements

Context broker runs in OSCAR environment, which exists for various platforms, including windows, refer to <http://oscar.objectweb.org>

The following dependencies exist (will be solved by OSCAR)

Amigo OSGI framework (amigo_core, amigo_ksoap_binding, amigo_ksoap_export, amigo_wsdiscovery)

Oscar/OSGI framework (System bundle, Shell service, Table layout, Shell GUI, Shell plugin, Bundle repository, Shell TUI, HTTP Service, log4j, Service Binder, Servlet), note that GUI is not required. but available with Oscar.

Jena bundle

2.1.2 Download

broker.jar

2.1.3 Install

Install under oscar, updated version can be found at

http://gforge.inria.fr/projects/amigo/ius/context_mgmt/context_broker/trunk/dist/broker.jar
(internal version, beta, available for Amigo developers)

and

<http://amigo.gforge.inria.fr/obr/v2/repository.xml> (external version, released)

2.1.4 Configure

2.1.5 Compile

The Context Broker has been implemented as a java project under eclipse, Using the eclipse(.project file) file you can compile a new version of context broker.

2.2 Context Source / Context Wrapper

2.2.1 System requirements

For OSGi bundles the same requirements hold as for the Context Broker. For the .NET components the .NET framework has to be installed.

See also the tutorial sections in this document.

2.2.2 Download

Downloadable versions are available from the CMS OBR at <http://core.lab.telin.nl/~amigo/obr/repository.xml>.

2.2.3 Install

Use the normal Oscar (local) bundle installation procedures.

2.2.4 Configure

No configuration is needed for the tutorial or Bluetooth Context Source.

2.2.5 Compile

A new version of the various examples can be compiled either by Eclipse or command-line by using the provided build.xml file.

2.3 Context Interpreter

2.3.1 System requirements

Context Interpreter queries different Context Sources, so all requirements for running necessary Context Sources and Context Broker have to be fulfilled. Additionally, as Context Interpreter is written in C#, Thus, Amigo .NET programming framework and Microsoft .NET Framework v2.0 have to be installed.

2.3.2 Download

Downloadable versions are available from the gforge?

2.3.3 Install

Context Interpreter executable and referenced libraries can be located wherever needed, but under the same folder; C++ dll should be saved in C:\WINDOWS\system32.

2.3.4 Configure

Configuration file e.g. "ci_config.xml" should list Context Sources required for tasks of Context Interpreter. Context Interpreter has Graphical User Interface (GUI). Needed configuration file can be loaded by File – Open menu item. GUI shows only Context Sources found by Context Broker. If configuration file contains Context Sources which are not registered to Context Broker, Context Interpreter can't use them. Following example shows content of configuration file:

```

<contextInterpreter>
  <contextSourcesToDiscover>
    <contextSource name="HomeAgendaContextSource">
      <query>
        <![CDATA[
          PREFIX cal:<http://www.owl-ontologies.com/Ontology1170405390.owl#>
          PREFIX p:<http://www.owl-ontologies.com/Ontology1180340524.owl#>
          PREFIX cv:<http://amigo.gforge.inria.fr/owl/ContextVocabulary.owl#>
          PREFIX ct:<http://amigo.gforge.inria.fr/owl/ContextTransport.owl#>

          SELECT ?summary
          WHERE {
            ?res cal:hasSummary ?Summary .
            ?Summary cal:isSTRING ?summary .
          }
        ]]>
      </query>
    </contextSource>
    <contextSource name="UserLocation">
      <query>
        <![CDATA[
          PREFIX amigo: <http://amigo.gforge.inria.fr/owl/Amigo.owl#>
          PREFIX context: http://amigo.gforge.inria.fr/owl/ContextTransport.owl#>
          PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
          PREFIX iccs: <http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#>
          SELECT ?who ?room WHERE {
            ?p rdf:type amigo:Person .
            ?p context:identifier ?who .
            ?ul rdf:type context:UserLocation .
            ?ul context:isLocationOf ?p .
            ?ul context:isLocatedIn ?r .
            ?r context:identifier ?room
          }
        ]]>
      </query>
    </contextSource>
  </contextSourcesToDiscover>
</contextInterpreter>

```

Context Interpreter start running after configuration file is loaded and Context Sources are discovered.

2.3.5 Compile

Context Interpreter can be compiled with Microsoft Visual Studio 2005.

In Appendix, section 6.2 the basic principle that will be used in the context reasoning module is explained.

2.4 Context History

2.4.1 System requirements

Amigo .NET programming framework

Microsoft .NET Framework v2.0

2.4.2 Download

"ContextHistories.exe", located at (non-public):

[amigo] / ius / context_mgmt / context_histories / trunk / ContextHistories / bin / Debug

2.4.3 Install

Run the executable.

2.4.4 Configure

The configuration files connection.conf (for connection with the Context Histories database) and port.conf (where the Context Histories Web Service is running) are located at (non-public):

[amigo] / ius / context_mgmt / context_histories / trunk / ContextHistories / bin / Debug

2.4.5 Compile

Using Microsoft Visual Studio 2005 you can compile a new version of the Context History component.

2.5 Acoustic Position Estimation Sensor

2.5.1 System requirements

SuSe Linux 9.3 operating system

Amigo OSGI framework (various jars)

OSCAR platform (only Linux)

Refer to [<http://oscar.objectweb.org/>]

2.5.2 Download

Downloadable versions of the wrapper will be available through gforge.

2.5.3 Install

Use the normal Oscar (local) bundle installation procedures.

2.5.4 Configure

No configuration is needed for the context source wrapper.

2.5.5 Compile

Wrapper: A new version can be compiled either by Eclipse or command-line by using the provided build.xml file.

2.5.6 Deployment

First start the Spark engine with the corresponding configuration file. Keep in mind to configure correctly the files for your room setup (e.g. position of microphones, distance between microphones, etc.), before starting the spark engine. After starting the spark engine you can deploy the context source wrapper on the Oscar platform. It then connects to the spark engine, registers the new context source to the broker and handles the communication with the context consumers for the spark engine.

2.6 RF Positioning Estimation

2.6.1 System requirements

For running: PC running windows, amigo framework (.net based), one free serial port, sensite hardware

For debugging: Visual studio, .net 2.0

Formatted: English (U.S.)

RF position monitor has a dependency on Semweb libraries which in their turn depend on IVKM libraries

2.6.2 Download

Positioning service is delivered as windows installer file.

Located at http://gforge.inria.fr/projects/amigo/ius/context_mgmt/rf_positioning/trunk/dev/

(internal version, beta, available for Amigo developers)

and

http://amigo.gforge.inria.fr/home/components/wp4/Context_Management/download/positioning_monitor.msi

(external version, released)

There is also a small Test application that can be used to query the RF service, this tool is described in Appendix 5.2.1

2.6.3 Install

Run the executable, make sure that dll libraries are available (see below)

2.6.4 Configure

Position monitor can be configured by using the file "ReaderLocationInformation.xml", "readerdata.xml" and "Tagdata.xml" which can be found in the project directory

The values that are found in this file have to comply to the unique Ids as they are provided by the configuration context service. It is the intention to facilitate this with a small tool in the future.

2.6.5 Compile

Software was developed as visual studio project, open solution file (positioning service.sln)

Make sure that references exist to:

Amigo .net framework (see D3.x)

Semweb libraries (semweb.dll, semweb_sparql.dll and sparql-core.dll, can be found at <http://razor.occams.info/code/semweb/> copy can be found on website)

Semweb libraries in their turn depend on IKVM libraries (IKVM.GNU.Classpath and IVKM runtime)

2.7 Topic recognition sensor

2.7.1 System requirements

- Real-time large vocabulary continuous speech recognition service, or real-time keywords recognition service. When such services are not available, a fallback solution is shipped within the topic recognition bundle: it is a lightweight keyword spotting module dedicated to implicit speech recognition implemented in sub-task 4.5.1.
- Windows XP, cygwin, OSCAR OSGI platform
- As any CMS context source, it further requires the main CMS bundles described in this document

2.7.2 Download

Topic_recognition.jar under GForge, or via the OSCAR CMS OBR

2.7.3 Install

- Install under oscar for the topic recognition part (without keyword spotting)
- Unzip and run a .bat script for the keyword spotting part (please refer to the User Guide for the detailed description of the process)

2.7.4 Configure

Configuration of the topic recognition sensor is realized via a configuration file that respectively defines:

- the number of topics to be recognized
- the number of relevant words or keywords
- the list of topics
- the list of relevant words or keywords
- for each word, a multinomial probability distribution over the list of topics

An example configuration file, named "default.topics", is given in the topic_recognition repository at GForge. This file will be used by default.

The keyword spotting or speech recognition module that delivers a text stream for the topic recognition sensor shall also be configured accordingly. Please refer to the corresponding section of the *User Interface Service (UIS) Software Developer's Guide* for more details.

2.7.5 Compile

Using eclipse (.project file) you can compile a new version of the topic recognition sensor

You can also compile a new version by command-line using the ant software and the build.xml file.

Note that Topic recognizer has a separate section on how to use, this has been added in Appendix D

2.8 Domotic sensor information

2.8.1 System requirements

For running: PC running windows, one free serial port

For debugging: Visual studio, .net 2.0

Formatted: English (U.S.)

Amigo Web Service discovery

Amigo OSGI framework

Context Broker

Fagor Domotic Controller

2.8.2 Download

MD_Gateway.exe is the executable, will be uploaded to the gforge.

There is also a small Test application that can be used to query the context information related with the connected devices.

2.8.3 Install

Run it.

2.8.4 Configure

Different communication interfaces can be added to the Gateway. A specific driver (connector) is needed for each interface. This driver will include the communication protocol.

2.8.5 Compile

Software was developed as visual studio project, open solution file (MD_Gateway.sln)

Make sure that references exist to:

Amigo .net framework (see D3.x)

Semweb libraries (semweb.dll and semweb_sparql.dll)

Connector libraries

3 Component Architecture

3.1 Component interface

By definition all Context Source components have the following interface:

IContextSource

Containing the following methods:

```
public String query(String contextQueryExpression);
public String subscribe(String contextSubscriptionCharacterisation, String contextSubscriptionReference);
public boolean unsubscribe(String contextSubscriptionID);
```

and the broker has the following interface:

IContextBroker

```
public String discoverContextSource(String contextInfoDesc);
public String registerContextSource(String contextInfoDesc, String cSRef);
public String deregisterContextSource(String cSRef);
```

3.1.1 Context Information

Context Information is described using an ontology from the Amigo vocabulary (which was developed in Work Package 3). A small example is shown below, for a more detailed description refer to Amio5f] and [Ami06a]

3.1.2 Describing Context Source capabilities

Capabilities of a context source are modeled using RDF and used when the context source registers at the context broker. This RDF model is matched a description of the needs specified by a context consumer when it is looking for context (sources) at the broker. Context consumer model their needs using SPARQL expressions. The context capabilities have also been based on the context ontology, using the concept of *ContextParameter*

In the Table below, some of the most common capabilities are presented, Appendix 5.2 explains the structure in more detail

Context	Refers to	Description
UserLocation	User, IsLocatedIn, Space (Room, Building)	User is located in a certain Space (Room, Building)
ObjectLocation	Object, IsLocatedIn, Space (Room, Building)	Same for objects
BlueToothScan		
UserPositionEstimation	User, hasPosition, Position	Relative position in x,y co-ordinates (or other system)
RoomContent	Room, holds, Object	... this list needs to be extended in future versions of this document

All ontologies mentioned in this table have been defined in the Amigo vocabulary, which means that all have prefix <http://amigo.gforge.inria.fr/owl/ContextTransport.owl#> which has been left out to make the table smaller. Next to their *type*, capabilities can also differ in other

parameters such as *timeliness*, *accuracy* and *precision*. This will be explained further on in this document.

3.1.3 Discovering Context Sources

Context aware applications (or context clients) discover context sources by interacting with the context broker. During this interaction they submit their needs to the context broker, and in return the context broker provides the list of context sources that are able to satisfy these needs.

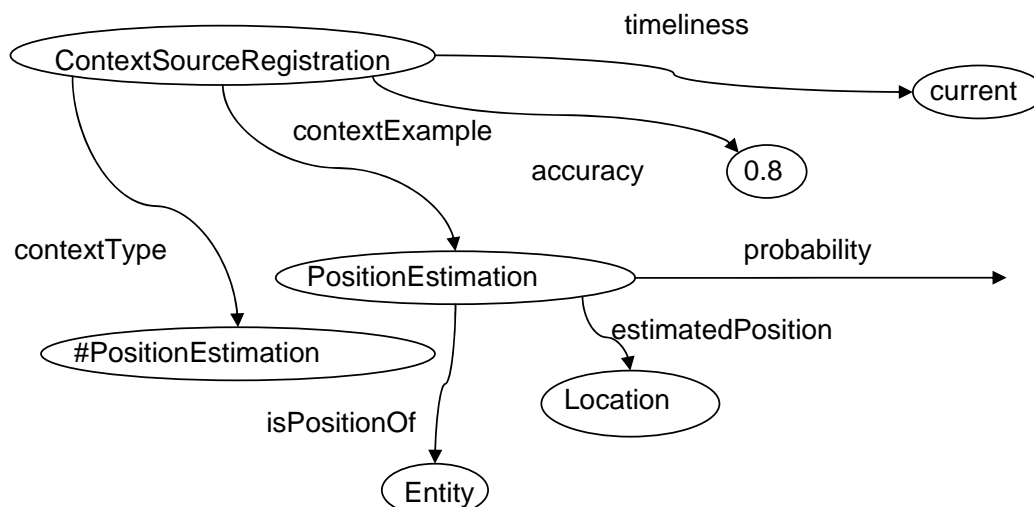
Needs are modeled using RDF descriptions by the client and passed to the context broker. The context broker will match this needs description to the Context Source capabilities descriptions it received from Context Sources at the time they registered with the broker. If a match is found the context source is considered as able to fulfill the context client needs. For those familiar to logic programming, this process is similar to unification.

We will describe the process of modeling needs and matching needs against capabilities using the example of the Accoustic based Positioning Estimator context sensor.

In order to express the needs of a context aware application for context information, the application developer has to navigate through the Amigo context source description ontology in order to locate the concepts that capture at best the nature of context he/she is interested in. If the context source description ontology is considered as a vocabulary for speaking about context source, this amounts to finding the words from this vocabulary that will be used to talk about the context source of interest. Failing to find such words suggests that the context source description ontology should be extended. This, of course, has been the philosophy of the Amigo context ontology elicitation task throughout the Project lifetime.

Back to the location context example, suppose that our location aware application requires the positioning of the user, we will naturally search the context source description ontology for concepts and relations related to positioning. We will find concepts such as "Relative2DLocation", "User" and relations such as "X", "Y" for the coordinates.

Extracting these concepts from the entire ontology and figuring out how practical context information can be derived by instantiating those concepts, can be sketched to produce the following diagram:



Where the concepts Location and Entity refer to the context ontology and capture the context information the context aware application is interested in.

Once such diagram has been drawn, it is easier to design the filter that will "match" with capabilities of context sources that will produce such information. The example below shows what description should be passed to the `discoverContextSource` method of the context broker to return the context sources that provide acoustic positioning information, without specifying further restrictions on precision etc:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:context="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#">
  <context:ContextSourceRegistration>
    <context:timeliness>current</context:timeliness>
    <context:contextType>AcousticPositionEstimation</context:contextType>
  </context:ContextSourceRegistration>
</rdf:RDF>
```

3.1.4 Querying context information

The main query language used for Context Sources is the SPARQL[<http://www.w3.org/TR/rdf-sparql-query/>] query language.

It is similar in structure to the well-known SQL language for relational databases.

The query specifies a series of triples, known as a graph pattern that is used for matching against the model.

A nice introduction and tutorial for SPARQL can be found at the following URL:

<http://www-128.ibm.com/developerworks/xml/library/j-sparql/>

Additionally, the Context Source Tester bundle from OBR contains an example SPARQL query, which can be used as a basis for defining ones own queries.

3.2 Context Interpreter interface

Context Interpreter does not support SPARQL querying because expressing desired rules in SPARQL is not easy; instead, CI provides web interface *IContextInterpreter*, similar to other Amigo services: a set of functions which applications can call.

The functions are:

1. *string AndOrRule(string [] leftHandSide, string leftHandSideOperator, string leftHandSideOperatorLast, string rightHandSide, string whatToReturn)*
2. *string findFact (string factToSearch, int minNumberOfFactsToExist, string rightHandSide, string whatToReturn).*
3. *string findBestFact (string factToSearch, string comparisonGoal, string comparisonCriteria, string comparisonOperator, string notValidResult, string KnowledgeBase, string rightHandSide, string whatToReturn).*
4. *string getUsersAvailableTime (string[] userIDs, string whatToReturn).*

The detailed description of how these functions work is provided in Appendix 6.2

Generally, the CI considers each context as a set of type (context descriptor) – value (corresponding value) pairs, where each set contains all relevant information about one fact. For example, the fact that Jerry is in the kitchen is described by CI as a set of context descriptors:

{personid: Jerry@JGAV3006.amigo.net, roomid: Kitchen@JGAV3006.amigo.net, TimeStamp: 2007/12/4 16:00}

All CI functions take as an input and provide as output similar sets of type-value pairs and type-only context descriptors.

3.2.1 AND, OR and NotAND rules

string AndOrRule(string [] leftHandSide, string leftHandSideOperator, string leftHandSideOperatorLast, string rightHandSide, string whatToReturn)

This function implements such traditional rules as AND, NotAND and OR rules. AND rule works as follows: “*IF (**fact1** exists AND **fact2** exists... AND **factN** exists) THEN **conclusion***”, OR rule checks whether one of the listed facts exists.

NotAND rule works as follows: “*IF (**fact1** exists AND **fact2** does not exist... AND **factN** does not exist) THEN **conclusion***”,

Function parameter *leftHandSideOperator* specifies which of these rules to apply via one of three predefined terms: AND, OR or NotAND

Parameter *whatToReturn* allows to restrict return string only to context types application is interested in (if empty, the function will return the whole **fact1** which produced **true** result).

Parameter *string [] leftHandSide* is traditional left-hand-side part of rule, that is, list of facts which should be **true** in order to fire the rule.

Parameter *rightHandSide* is high-level conclusion of the rule, e.g., “user activity: sleeping”

Parameter *leftHandSideOperatorLast* specifies whether the function should return true if all possible **facts1** in the data result in **true** conclusion of the rule, or just one. This parameter can take one of two predefined values: “AND” or “OR”. In case of “user is sleeping” rule parameter *leftHandSideOperatorLast* = “AND” would mean that application is only interested in a situation when all family members are sleeping, while “OR” would mean that application is interested in a situation when any of family members is sleeping.

Rules can be specific or generic; this depends on whether *leftHandSide* facts contain type-value or type-only context descriptors. The exact syntax is xml-based and will be described in the appendix 6.2, here we will describe rules in a common language.

leftHandSide strings determine which parts of stored facts will be compared and how. First, applications can specify all details of **facts** to be compared, e.g., to create a rule “*IF (**personid: Jerry@JGAV3006.amigo.net, roomid: bedroom AND lightLevel: low, roomid: bedroom AND soundLevel: low, roomid: bedroom**) THEN **personid: Jerry@JGAV3006.amigo.net, activity: sleeping***”

However, if application is sure that for all family members such sensor data in a bedroom means “sleeping” activity (not e.g. “playing hide-and-seek” activity), then application can create more generic rule “*IF (**personid:, roomid: bedroom AND lightLevel: low, roomid: bedroom AND soundLevel: low, roomid: bedroom**) THEN **activity: sleeping***”

This rule will check “sleeping” activity for any personID. Since CI returns not only “true” or “false” result, but also the whole **fact1** which produced “true” result, application will also know who is sleeping. (Don’t forget to add “type-only” field “personid” to the rule, without it CI will infer “sleeping” activity also for empty rooms).

If application is also sure that such sensor data for any room implies “sleeping”, it can create a rule “*IF (**personid:, roomid: AND lightLevel: low, AND soundLevel: low**) THEN **activity: sleeping***”

Type-only fields should be always included into the first fact of the rule. In other facts such fields are ignored even if they are different from the first fact. When CI performs AND/ OR logic with rules containing “type-only” descriptors, it makes “true” conclusion only in two cases: first, if **fact1** and **fact2** both contain all “type-only” descriptors listed in the rule, values of “type-only” descriptors in **fact1** should be equal to values in “type-only” descriptors in **fact2**. Second, if **fact1** and **fact2** both contain only part of “type-only” descriptors listed in the rule, values should be equal in those descriptors which are present in both facts, while values of non-overlapping types can differ. This is needed because physical sensors provide only values of a certain type, e.g., light sensor only knows the room where “light level is low”, but it does not know who is in this room. Thus, fact “light level is low” does not initially contain “personid” type. Similarly, positioning sensor does not provide light information, but they both contain “roomid”. However, it makes impossible to create rules of the kind “if light level one room is low AND light level in another room is high” with type-only fields; instead, applications have to include also “roomid” into such rules.

3.2.2 FindFact function

string findFact (string factToSearch, int minNumberOfFactsToExist, string rightHandSide, string whatToReturn).

This function simply searches for facts which match to *factToSearch* set of “types and values” pairs, and collects all facts which contain required set (other components of a set can differ). For example, application might be interested whether light level in any room is high, and it will get back all rooms where light level is high. If application is interested only in cases when light level in not less than two rooms is high, it can set *minNumberOfFactsToExist* = 2. If application wants to know whether light level in all rooms is high, it should set this parameter to be equal to number of rooms in the house.

String *rightHandSide* allows to attach any additional information to returned facts if needed.

3.2.3 FindBestFact function

string findBestFact (string factToSearch, string comparisonGoal, string comparisonCriteria, string comparisonOperator, string notValidResult, string KnowledgeBase, string rightHandSide, string whatToReturn).

This function searches for the fact which satisfies application needs best of all, but this is not a multi-criteria comparison, although parameter *notValidResult* allows to exclude facts absolutely unsuitable by some other criteria. This function finds “best” value of one context type only. For example, application can find nearest device with this function, and it can find device with a largest screen size, but application needs two separate queries for this.

Parameter *factToSearch* specifies which context descriptor (type) to look or, e.g., “personid” type will mean that application needs to find a “best” person, while “deviceid” will mean that application needs to find a “best” device.

Parameter *comparisonCriteria* specifies which context descriptor (type) has to be best, e.g., “location” type will mean that application needs to find nearest *factToSearch*, while “StartTime” will mean that application needs to compare start times of events.

Parameter *comparisonOperator* specifies whether the smallest or the largest value is “best”, e.g., is a “best” parent the nearest one or the one who is far away.

Parameter *comparisonGoal* is value of *comparisonCriteria* type. This parameter can be empty. For example, if application searches for TV with the largest screen size, it does not need any *comparisonGoal* because it needs to find a maximum of values. However, if application needs to find a TV with almost same screen size as somebody’s favourite size, it can set favourite

size as a goal. Then *factToSearch* will be “best” if its distance to *comparisonGoal* is smaller than of any other fact. Similarly, if application needs to find device nearest to the user, it needs to set user location as *comparisonGoal*. With numerical values finding “best” value simply implies finding largest or smallest value. With non-numerical values CI needs additional information, and for this we have *knowledgeBase* parameter.

knowledgeBase parameter can be empty if it is needed to compare numerical values. However, if we want to find a TV nearest to a user, calculating distance between user's coordinates and TV coordinates might not be the best approach because user location is often known only at a room granularity, and also not more than one TV is usually present in one room. Thus, we need only to find a nearest room to a room where the user is located. The fastest way to do it is to list rooms according to how far they are located from each other. Thus, such model will look like this:

room1: room2, room3, room4

room2: room3, room1, room4

.....

Which means that if the user is in room1, closest room is room2 (and room3 is a bit further away), while if the user is in room2, room3 is closer than room1 and room4.

This approach can also deal with cases when smallest Euclidian distance between rooms is not “best”. For example, for a person with disabilities longer way to go may be easier if there are fewer obstacles. Since knowledge base can be application-dependent and user-dependent, applications can either use different knowledge bases for different queries, or the only one created when model of the house was built.

notValidResult parameter can be empty. All facts which are listed in this parameter will be excluded from “best or not best” reasoning. For example, if application needs to find a person nearest to user1, *findBestFact* function should not return user1 as a nearest person. Thus, application should inform the function that it is interested only in other persons. If application knows that even if user2 is nearest to user1, user2 will be of no help for him, application can exclude user2 as well.

3.2.4 getUsersAvailableTime function

string getUsersAvailableTime (string[] userIDs, string whatToReturn).

This function queries HomeAgenda CS and calculates time to the first appointment of group members. This functionality was requested by recommender applications, e.g., a movie recommender application or a game recommender application would recommend only short movies or games for the situation when users have to leave soon.

3.3 Mechanisms of interaction

Interaction with Context Sources is done using the standard Amigo protocols; typically Web Services. The interaction can be either synchronous, using the query method, or asynchronous using the subscription methods.

In the synchronous case the Context Source will use the specified SPARQL query to match against its current internal context model and return the answers immediately.

In the asynchronous case the Context Source will store the SPARQL query and match it against its current internal context model and inform the client, using WS-Eventing, whenever the answer to the query changes. This approach allows for a more event-based style of programming; reacting to context changes if and when they occur.

The asynchronous subscription approach does place a higher burden on a Context Source in terms of resources, since it has to keep track of all subscriptions and queries, and has to reevaluate all the queries whenever its context model changes.

Interaction with Context Interpreter is done using standard Amigo Web Service protocol, so that applications can directly call functions of Context Interpreter.

3.4 Overview and reference to internals

Helper bundles are available for OSGi/Java and .NET that simplify the task of writing Context Sources and Context Consumers.

These helper bundles shield most of the interactions and specifics of the exporting and communications needed for queries and subscriptions from the Context Source and Consumer. For example: The Context Sources and Consumers do not need to know the Amigo specific interaction protocols, they only communicate with the helper bundles using standard Java or .NET calls.

The following sections describe the functionality of the different helper bundles in more detail.

3.4.1 Context Source helper bundle

3.4.1.1 OSGi

The Context Source helper bundle assists in creating and managing Context Sources. The two central components in this bundle are the Context Source Manager (CSM) and Context Source Front(s) (CSF). The Context Source Front is a 'front-end' to a Context Source that takes care of all the interactions with Amigo interaction protocols; including queries and subscriptions. If needed however, a Context Source can opt to handle queries itself.

In order to use the Context Source helper bundle, the Context Source needs to get a handle on the Context Source manager. This can be achieved using the normal OSGi methods for specifying dependencies, so that a reference to a CSM is available at the startup of the Context Source; in essence the Context Source needs to specify a dependency on the *nl.telin.amigo.contextmanagement.csmw.ContextSourceManager* service.

Using this reference the Context Source requests the CSM to register a new Context Source, specifying the capabilities using an RDF document. The CSM will instantiate a new Context Source Front (CSF) on behalf of the Context Source and export and register it with the Context Broker. The reference to the CSF is then returned to the Context Source for later use. [Figure 3-1](#) shows the sequence diagram for the CS creation.

Deleted: Figure 3-1

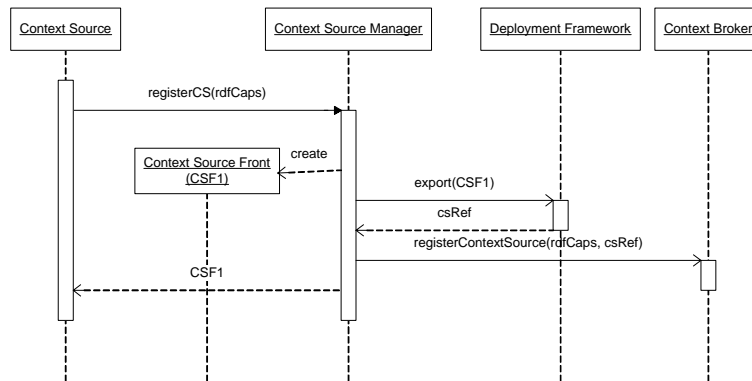


Figure 3-1: Creating a Context Source.

At deactivation of the Context Source; the Context Source instructs the CSF to deregister itself. The CSF and the CSM will then take care of the deregistration of the Context Source at the Context Broker (Figure 3-2).

Deleted: Figure 3-2

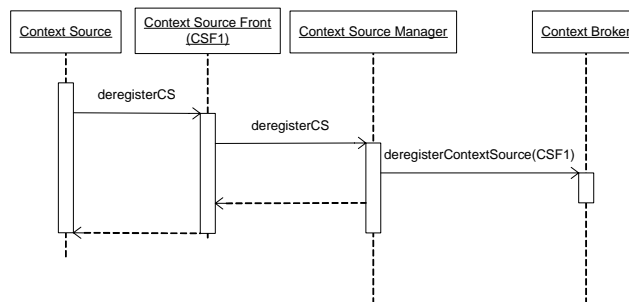


Figure 3-2: Removing a Context Source.

The main responsibility of the Context Source is to keep the context model up to date. A clean model can be requested from the CSF, usually at startup, which the Context Source can then fill with its context knowledge and return back to the CSF. The clean model can be totally empty, or 'preset' with references to the Amigo Context Transport ontology. Every time the context model is updated the Context Source informs the CSF using the `updateModel`; which will trigger the CSF to reevaluate any subscriptions it may have (Figure 3-3). The model can be reused by calling the `model.removeAll()` method.

Deleted: Figure 3-3

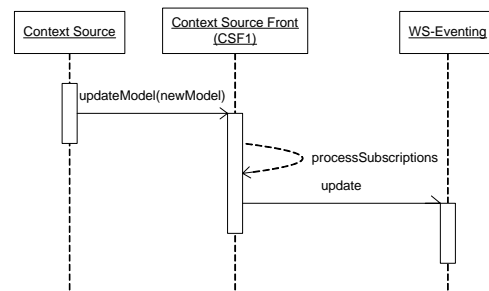


Figure 3-3: Updating a context model.

Keeping the model updated allows the CSF to handle queries and subscriptions from clients; as is shown for a query in [Figure 3-4](#).

Deleted: Figure 3-4

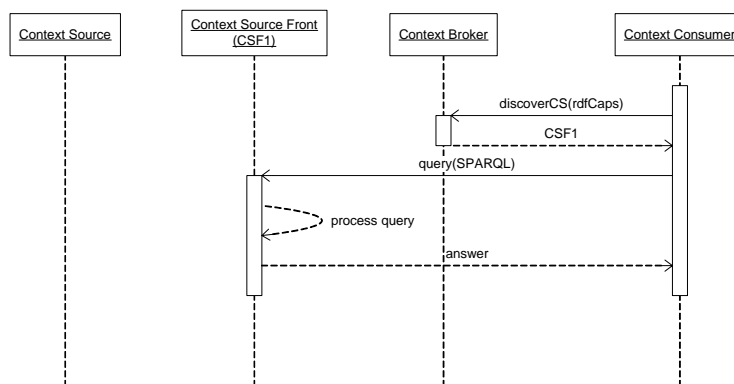


Figure 3-4: Querying a Context Source.

Since the CSF handles all the interactions with clients, the responsibilities of the Context Source are limited to registering with the CSM, keeping the CSF model up-to-date and deregistering before it terminates. [Figure 3-5](#) shows an overview of this from the point of view of the Context Source.

Deleted: Figure 3-5

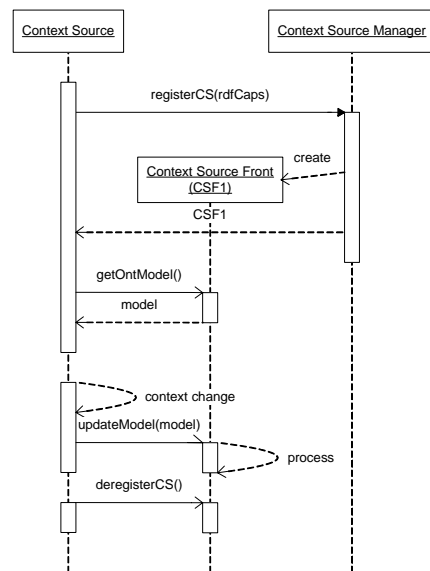


Figure 3-5: A day in the life of a CS.

3.4.1.2 .NET

The .NET Context Source helper takes a slightly different approach than the OSGi version. While the OSGi version uses bundles that run independently and are used by multiple Context Sources, the .NET version provides base classes that the Context Source developer can use as a starting point for implementing a Context Source. This base class provides a default set of methods that are similar to all Context Sources, regardless of the type of context they provide, like registering with a Context Broker, running queries on the context model, etc.

In order to use the Context Source Helper, the developer only has to subclass the `nl.telin.amigo.ContextSourceHelper.ContextSourceHelper` class, override the `myServiceName` property and call the helper's constructor supplying the port to listen on and a service description. An example on how to do this is provided in the Amigo subversion repository.

Upon construction the `ContextSourceHelper` will publish its service, find a context broker and register its service with it using the given service name and description.

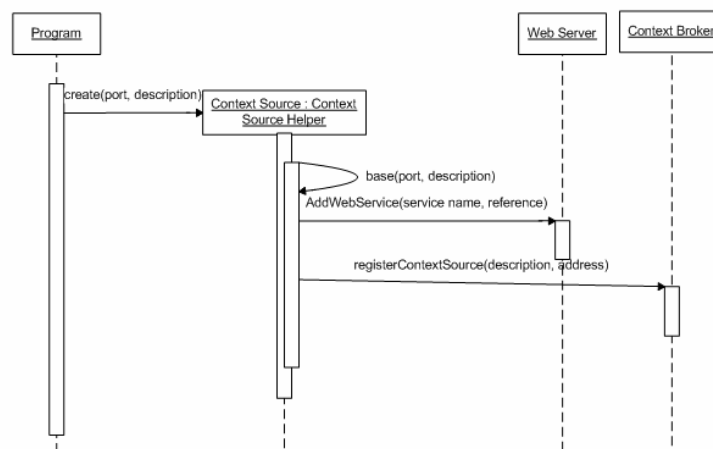


Figure 3-6: Creating a Context Source in .NET.

When the Context Source is disposed it will automatically deregister with the context broker and unpublish its service.

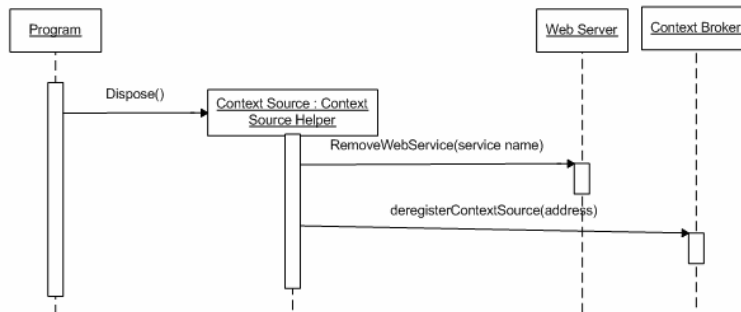


Figure 3-7: Removing a Context Source in .NET.

In order to create a 'pull' or query based ContextSource (only updating the model after a query is received); the 'GetRdfModel' method has to be overridden from the base class. This method should return the current RDF model.

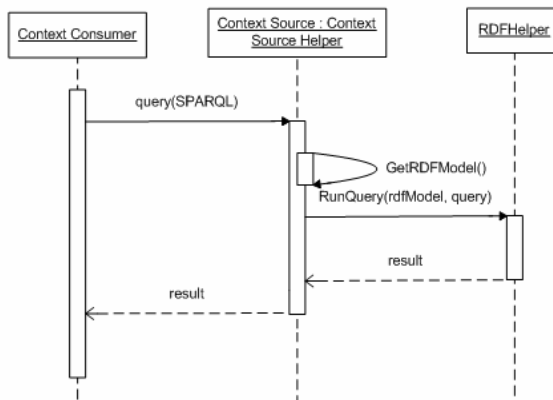


Figure 3-8: Querying a .NET Context Source.

A 'push' style Context Source can be created by forcing an update every time the underlying (sensor) data changes; this can be achieved by calling the 'ProcessRequests' method to force the processing of any subscriptions that may be present. (Note that the 'GetRdfModel' should still be overridden to supply the RDF model for the 'push'.)

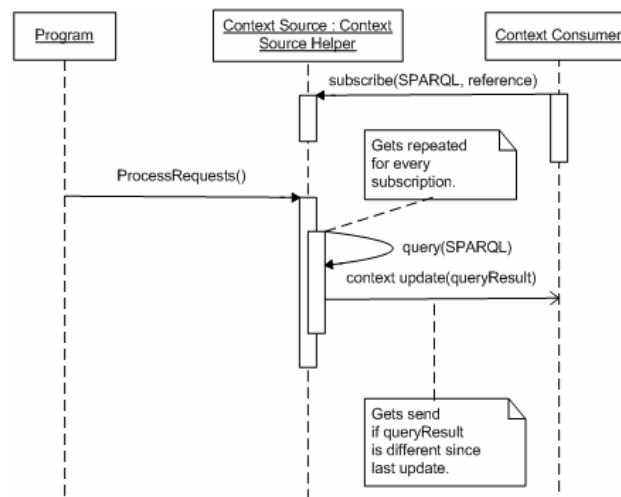


Figure 3-9: A 'pushing' .NET Context Source.

3.4.2 Context Consumer helper bundle

3.4.2.1 OSGi

The Context Consumer helper bundle assists context consumers in finding and interacting with Context Sources and handling subscriptions. The same OSGi approach as for Context Source Manager is used for getting a reference; in this case by specifying a dependency on the `nl.telin.amigo.contextmanagement.contexthelper.ContextBrokerHelper` service.

The first point of contact for a Context Consumer (CC) is the Context Broker helper (CBH). The CC asks the CBH to find context sources that meet the demands as specified by the CC. The CBH will interact with the Context Broker and will instantiate a Context Source Helper (CSH) for every reference to a Context Source returned by the Context Broker. The CC can then interact with these Context Source Helpers, which are local Java objects, instead of communicating with the Context Sources directly. [Figure 3-10](#) shows the discovery process of Context Sources.

Deleted: Figure 3-10

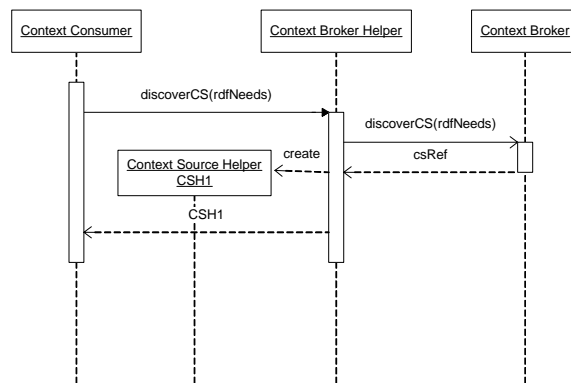


Figure 3-10: Discovering a Context Source.

After discovery the resulting Context Source Helpers (CSH) can perform queries on their associated Context Source and subscribe to them on behalf of the Context Consumer. Especially subscriptions are useful since the CSH takes care of all the subscription specific interactions; the only thing a Context Consumer needs to do is implement a Java interface with the `contextChanged` and `subscriptionEnded` methods. The `contextChanged` method will be called whenever the CSH receives notification from the CS about the context results for the subscription query. The `subscriptionEnded` method will be called if the subscription has ended for a reason other than the cancellation by the Context Consumer. [Figure 3-11](#), shows the interactions for subscribing to a Context Source, while [Figure 3-12](#), shows a call to the `contextChanged` method of the Context Consumer after the CSH receives a notification from the Context Source.

Deleted: Figure 3-11

Deleted: Figure 3-12

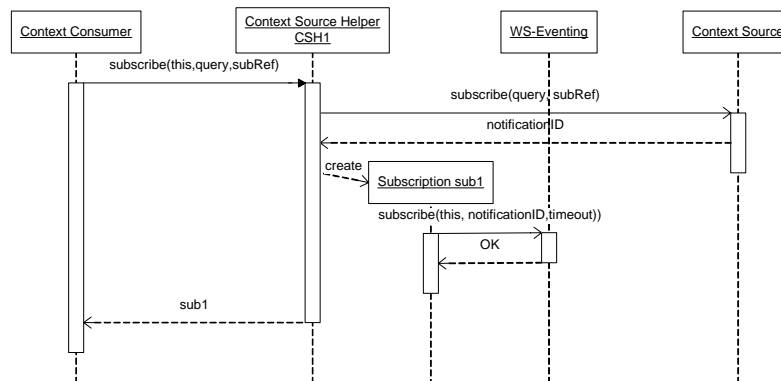


Figure 3-11: Subscribing to a Context Source.

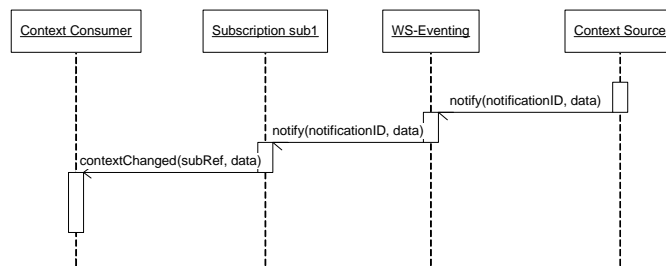


Figure 3-12: Context changed notification.

To unsubscribe the Context Source can call the unsubscribe method on the CSH; which will then take care of the rest.

3.4.2.2 .NET

The ContextHelper.NET helper classes have a similar approach to the OSGi ContextHelper bundle in that they provide classes that facilitate the interaction between a Context Consumer (CC), the Context Broker (CB) and Context Sources (CS). In order to find a CS, the CC can create a ContextBroker object, and call the discoverContextSource or discoverContextSourceSimple method with a description of the type of CS that is needed. The result of this method will be zero or more ContextSource objects that can be queried using a SPARQL query. The ContextSource class has two query methods: a queryRaw method that delivers the result of the SPARQL query verbatim from the real ContextSource, and a query method that returns the results in the form of objects (Results, Bindings) that represent the results of the SPARQL query.

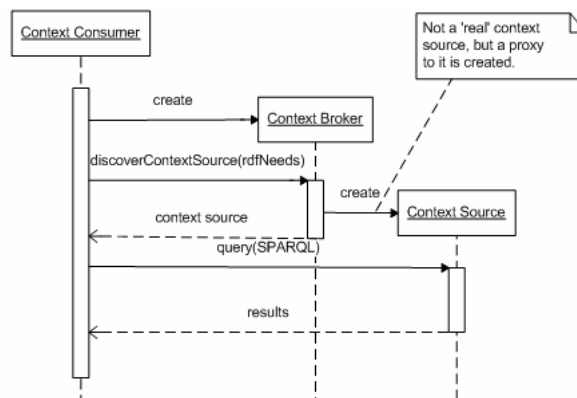


Figure 3-13: Discovering a context source in .NET.

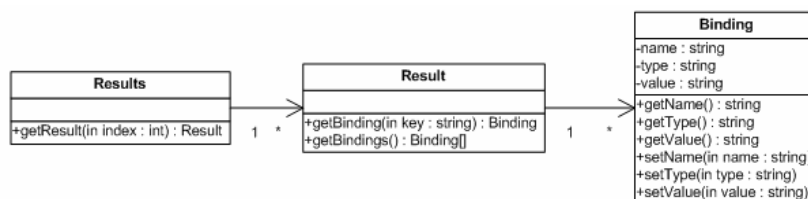


Figure 3-14: Results class diagram.

A consumer can call the subscribe method from the context source in order to subscribe to context updates. This method will return a subscription object which has a ContextUpdate event to which the consumer can add a handler. The event will be fired when the context source indicates that there's a context update.

3.5 Detailed documentation

The components of the Context Management Service have all been created with inline comments for all classes, properties and attributes. In this way, detailed documentation has been made available that will not be repeated in this document. In stead, we have copied the main overview here and refer the user to the inline documentation for the rest. For the java components this documentation has been created using javadoc.

4 Tutorial

4.1 Introduction

This section is a tutorial for the context management service, containing screenshots and installation guidelines for an "instructor-less" mode. The tutorial is designed to be used in combination with java for the most part, but services can also be implemented in C#. All required source codes for this tutorial can be accessed from the Amigo challenge web site (see <http://challenge.amigo-project.org/tutorials.htm>).

The first part of the tutorial shows an example of how to implement a context source for the Amigo context management system without the use of helper components (which are available for OSGi and .NET). It uses the Amigo components to publish itself for other context clients.

The second part of the tutorial describes how to implement a context client using the Amigo middleware and services, but again without the use of helper components.

The third part is the "active" part of the tutorial. You will be guided through the process of writing your own context source and an according context client.

These first three sections of the tutorial together will provide you with a good insight into the inner workings of context sources and clients.

Since most context sources are similar to a large extent, differing only in the information model and the specifics for e.g. reading sensors, the common parts can be provided by helper components, which take care of common administrative functions, such as finding and registering with a broker (for context sources), or handling WS-Eventing for context updates (for context clients). To handle these common functions, helper components are provided for OSGi and .NET, which make writing context sources or clients for these platforms much easier. The use of these helper components is explained in the final section (4.9) of this tutorial. It is possible to skip the first section of the tutorials and start right at section 4.9, but it is advisable to at least read the preceding sections to give some insight into the inner workings of context sources and clients.

Sources for the more complex example from part one and two as well as the "active" part of the tutorial can be found on the Amigo challenge web site (<http://challenge.amigo-project.org/tutorials.htm>).

A preconfigured Oscar platform with an offline repository is also available - go to the <http://amigo.gforge.inria.fr/obr/tools> web site.

4.2 Software requirements

For the tutorial you need an installed version of the Java Development Kit, including the javac compiler and the jar tools (Java Version 1.5.x). Furthermore ant from <http://ant.apache.org> and an integrated development environment (IDE) like eclipse from <http://www.eclipse.org> is recommended.

Also you need a pre-configured Oscar platform (<http://amigo.gforge.inria.fr/obr/tools/>).

The source code of the context broker, the example/tutorial context source and the example/tutorial context client are accessible via the Amigo challenge web site (<http://challenge.amigo-project.org/tutorials.htm>). The zip-file of the sources accompanying this tutorial includes the following components:

- ContextBroker: the context broker described in section 4.4
- ExampleAmigoCSCC: the context client and the context source for the example shown in section 4.5 and discussed in detail in section 4.6 and 4.7
- SkeletonAmigoCSCS: the context client and the context source for the tutorial in section 4.8 – the source code is designed as a cloze to be completed by you
- TutorialAmigoCSCC: the solution to the tutorial

It is recommended to get familiar with the IDE and JAVA before starting the tutorial, since IDE related problems can not be solved during the training sessions.

Software list:

- Java Software Development Kit (<http://java.sun.com>)
- Java IDE (<http://www.eclipse.org>)
- Ant (<http://ant.apache.org>)
- Oscar (<http://amigo.gforge.inria.fr/obr/tools/>)

Development of context aware services is also possible using C# in the .NET Framework. The solution to the tutorial in C# is included in the zip-file, too:

TutorialAmigoCSCC.NET: the tutorial context source and context client for C# in the .NET Framework

Given the solution, the tutorial should be easily accessible to C# programmers as well.

4.3 Oscar

Oscar is an open source implementation of the Open Services Gateway Initiative ([OSGi](#)) framework specification, which is used by many Amigo services. The main components of the CMS system are based on Oscar bundles, which are available through the Online Bundle Repository (OBR) of Amigo.

4.3.1 Installation

Oscar does not have a special installation routine, so the full software package is available as a zip-file (e.g. http://amigo.gforge.inria.fr/obr/tools/oscar_j2se.zip) and has to be extracted to a directory of your choice. If you are connected to the Internet through a proxy server, you'll have to add the appropriate proxy settings to the "system.properties" file located in the "lib" sub-directory. You'll find the start scripts "oscar.bat" (Windows) and "oscar.sh" (Linux1) in the main directory. After calling the start script you are asked for a profile name (enter CMSTutorial, for example), which defines the name of the current Oscar profile. You can use profile names for separating different configurations.

4.3.2 GUI

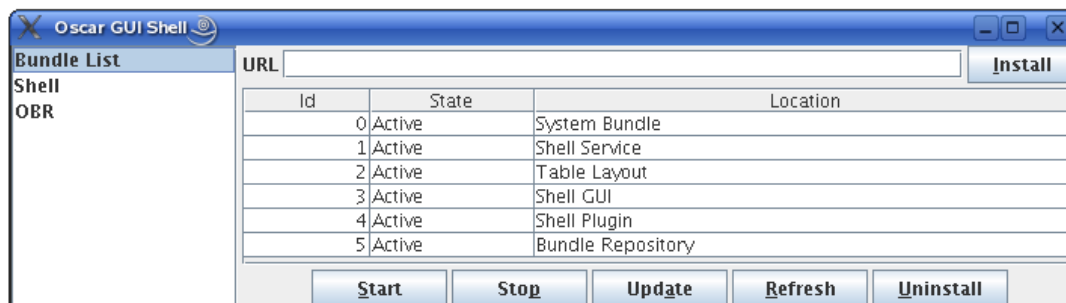
Below you see a screenshot of a running version of Oscar. On the left side you find the three items:

Bundle List: A list of all installed bundles, showing Id , State and Location; additionally, local URLs can be entered for installing bundles (e.g.: file:///home/user/amigo_bundle.jar)

Shell: Command shell

OBR: A list of all bundles available from the repositories, this can be online or offline repositories as defined in /lib/bundle.properties.

Use the OBR item to install bundles from the repository and the Bundle List item for starting and stopping installed bundles.



4.3.3 Installing bundles

Now start installing bundles to the Oscar platform:

1. Click on the item "OBR"
 2. Select "amigo_core" and press "Start All"
- Installed dependencies:
- log4j

¹If "oscar.sh" is not executable use "chmod u+x oscar.sh" to change the file attribute.

- Service Binder
3. Select “amigo_ksoap_binding” and press “Start All”
 4. Select “amigo_ksoap_export” and press “Start All”
- Installed dependencies:
- Servlet
 - HTTP Service (+ Amigo mods)
5. Select “amigo_wsdiscovery” and press “Start All”

Now click on the “Bundle list” to check if all bundles are installed. You should get the following list:

Id	State	Location
0	Active	System Bundle
1	Active	Shell Service
2	Active	Table Layout
3	Active	Shell GUI
4	Active	Shell Plugin
5	Active	Bundle Repository
6	Active	log4j
7	Active	Service Binder
8	Active	amigo_core
9	Active	amigo_ksoap_binding
10	Active	Servlet
11	Active	HTTP Service (+ Amigo mods)
12	Active	amigo_ksoap_export
13	Active	amigo_wsdiscovery

If during the bundle installation the above listed dependencies are not installed (an error will occur and pop up), please install the dependencies prior to the bundles by hand.

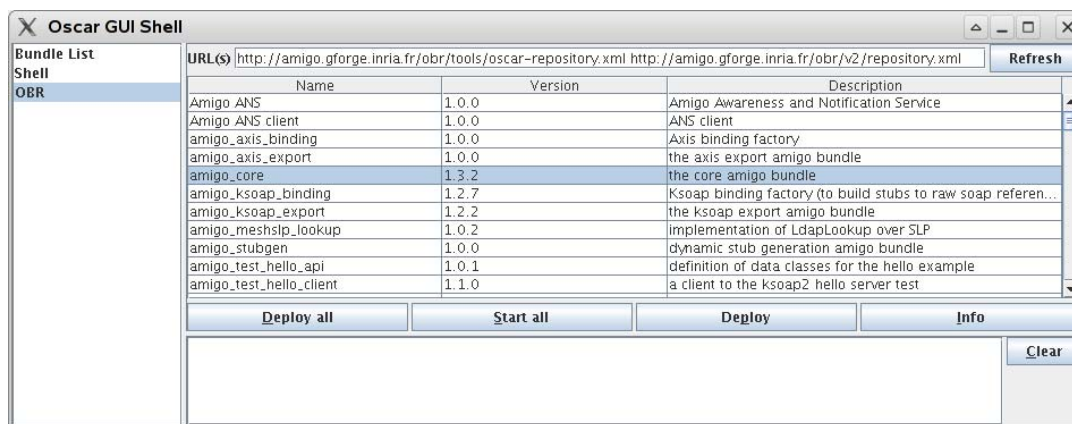
In the “OBR” list you also find several other software bundles, which you can use for your services. Now we should start with compiling bundles and installing them.

4.3.4 Repositories

The URL of the repositories is defined in “lib/bundle.properties”. The standard Amigo repositories are defined by the following copy of the “bundle.properties” file. You also find the address of the actual repositories on the item OBR.

```
# all known repositories
oscar.repository.url=\
http://amigo.gforge.inria.fr/obr/tools/oscar-repository.xml \
http://amigo.gforge.inria.fr/obr/v2/repository.xml
```

```
# the port on which the http server will listen
# if you have several OSGi platforms on the same machines change the port number
org.osgi.service.http.port=8080
```



4.3.5 Stopping Oscar

You can stop Oscar by typing “shutdown” in the Oscar shell. If nothing happens you can press “Ctrl+c” for killing Oscar, but try to shutdown Oscar gently through the “shutdown”-command beforehand.

4.4 CMS Context Broker

The CMS Broker is the main component for the context management service, as it provides all required functionality for discovering and using context sources. For software developers interested in building their own context sources or context clients it is sufficient to install the context broker via the Oscar OBR (section 4.4.1). Software developers interested in CMS technologies may take a look at the source code and build the broker on their own (section 4.4.2). Please remember: do not start more than one instance of the context broker in a network.

During a guided tutorial the presenter will host a context broker for the group. If you do this tutorial on your own, you have to start a broker in order to test your software. If you try to start a broker while another one is running, your broker will realize this and stop.

4.4.1 Installation by Oscar

Install the context broker by starting the Oscar platform with the same profile as you used before (just in case you closed the window) and perform the following steps:

6. Click on the item "OBR"
 7. Select "context-broker-service" and press "Start All"
- Installed dependencies:

- jena-2.4

In the shell you can observe some logging information about the context broker.

4.4.2 Building the CMS broker

For this part we assume that you have recent versions of eclipse and ant installed on your computer. First of all, copy the context broker sources (included in the source zip-file for this tutorial which is available in the tutorial section of the Amigo challenge web site) to your favourite directory.

For building the context broker with eclipse perform the following steps:

1. Click "File" -> "New" -> "Project"
2. Select in Wizard: "Java Project" and click "Next"
3. Enter "ContextBroker" as the Project Name
4. Select "Create Project from Existing source" and click "Browse" to select the directory where you put the context broker sources
5. Click "Finish"

Now on the left panel the project "Context Broker" should be available. Select the project and search the "build.xml" file

6. Right-Click on "build.xml" -> Select "Run As" -> Select "Ant Build"

You should see some compiler messages and perhaps some warnings in the lower window. Two versions of the context broker are build:

- dist/broker.jar: Small version of broker containing just the required libraries (use this version, as it is smaller and needs less memory!)
- dist/broker-src.jar: Version of context broker containing all libraries (like jena, etc.)

You can install this version on the Oscar platform by selecting the "Bundle List" and specifying the location of the "broker.jar" file in the URL field (e.g.

"file://c:\path_to_my_project\dist\broker.jar") followed by a click on "install". But remember to start only one broker per network.

Alternatively you can also build the context broker with ant from the command shell by typing "ant" in the project directory.

4.4.3 Building the Amigo Context Client / Context Source example

The client example and the source example are in the "ExampleAmigoCSCC"-folder of the source bundle accompanying this tutorial. Copy them to your hard disc and afterwards import them like the broker.

For building the example with eclipse perform the following steps:

7. Click "File" -> "New" -> "Project"
8. Select in Wizard: "Java Project" and click "Next"
9. Enter "AmigoCCExample" (for the context client) or "AmigoCSExample" (for the context source) as the Project Name
10. Select "Create Project from Existing source" and click "Browse" to select the directory where you copied the files to
11. Click "Finish"

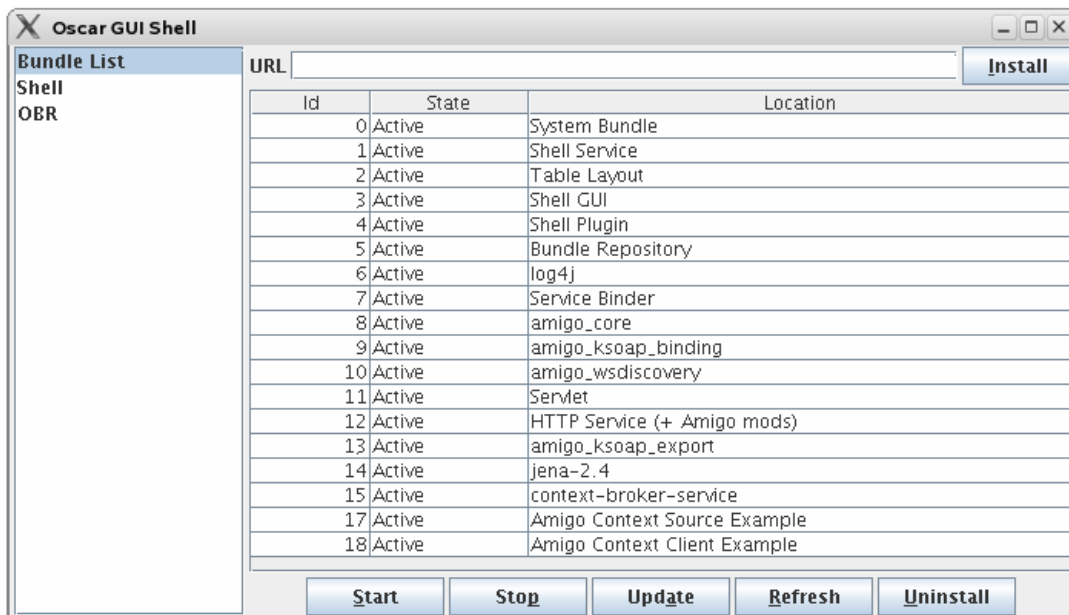
Now on the left panel the project should be available. Select the project and search the "build.xml" file

12. Right-Click on "build.xml" -> Select "Run As" -> Select "Ant Build"

4.5 Example Context Source and Client

Before actually writing a context source and a context client, this chapter shows an example (to be found in the "ExampleAmigoCSCC"-folder of this tutorial) illustrating the principles of context source and context client interaction.

The following screenshot displays the Oscar platform as introduced earlier. The bundles required by the tutorial's context source and context client are installed and already active.



Starting the *Amigo Context Source Example* will implicitly call the context source's *activate()*-method, which starts the following user interface:

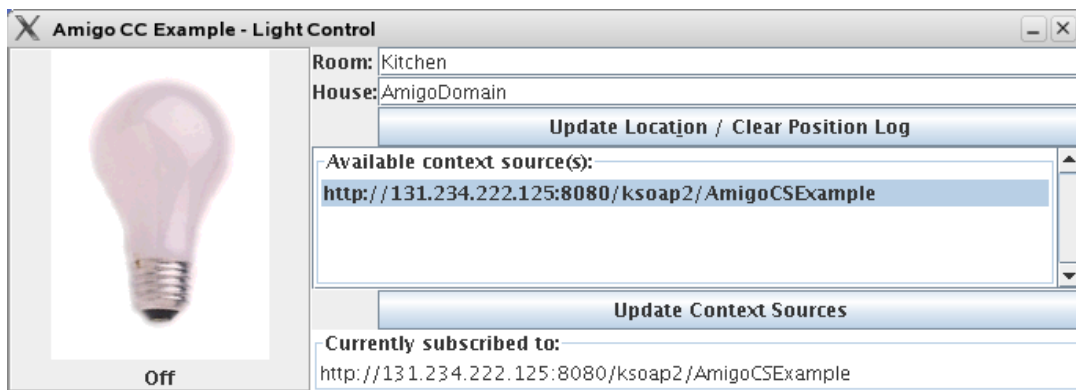


The user interface allows for manually providing the context source with simulated *sensor information*. The information passed to the context source by pressing the 'Send Update'-button is the user location of the specified 'Person', characterized by the 'Room' and the 'House' the person is in.

Alternatively, a simulation can be started by check marking the 'Simulation' check box. The simulation creates a set of random user locations in the 'AmigoDomain' (e.g.: Peter is in Bathroom@AmigoDomain) and passes the information to the context source. The simulation can be stopped by either pressing the 'Send Update' button, which will pass the current text entries to the context source, or by un-checking the check box.

Pushing the *sensor information* to the context source is done by calling the *contextChanged()*-method, which will update the ontology model and *notify()* all subscribed context clients.

Starting the *Amigo Context Client Example* calls the *activate()*-method of the context client, which also starts the client application displayed below:



The context client will automatically put all context sources registered at the context broker and matching its *rdfCapability* into the list of 'Available context source(s)'.

If a context source is started while the context client is already running, the 'Update Context Sources' button can be used to reset the context client and ask the context broker for available context sources, again. This is done by calling *findContextSources()*, which is invoking the context broker's *discoverContextSource()*-method. The list of context sources will refresh.

By double-clicking on a list entry, the context client tries to *subscribe()* to the chosen context source. If subscription succeeded, the context source subscribed to will be displayed in the 'Currently connected to' field. In the current example, the context client can only subscribe to one context source at a time.

Now the context client is ready to get notified by the context source of any changes in the context information.

The light bulb on the left will light up if a person is located in the specified 'Room' and 'House'. By default, the application is sensitive to users in the 'Kitchen' of the 'local domain'.

If the light control should be set for other locations, the location itself can be re-specified by changing the entries in the 'Room' and 'House' field and pressing the 'Update Location/Clear Position Log' button. Since the context client keeps track of all different users whose locations were once pushed by the context source, the location log is cleared, too.

The context client and the context source should be stopped by marking the according bundle in the Oscar GUI Shell and pressing the stop button.

Lets take a look at implementation details now.

4.6 Writing a Context Source

You have two options for writing a context source for the Amigo system. It is possible to use the helper component Context Source Manager, which will be responsible for the interfaces and the registration process at the broker, or you may implement the full complexity of the *IContextSource* interface and the registration process.

This tutorial is focusing on the latter case, since writing the full functionality by yourselves allows for a deeper insight into the communication process between context source, context

client and the context broker. Thus, facilitating not only the understanding of the current example, but also the development of your own context applications (i.e. context source and a matching context client).

4.6.1 Exporting the AmigoService

To allow the information of the context source to be accessed by client applications, the way the context client communicates with the context source has to be specified in beforehand.

In general, the process of exchanging information can be split up into 3 parts:

1. The context client, interested in getting information from the context source, needs to make itself known to the context source. The method provided by the context source is called *subscribe()*.
2. The context source posts information onto the network. Basically, there are two ways the context source can provide information to the context client – ‘query driven’ or ‘data driven’. In the ‘query driven’ approach, the context source only replies to a specific query of the context client. The method provide by the context source for this case is *query()*. In the ‘data driven’ approach, the context source pushes data to the context client each time it's sensor information changes. The context client is notified of this change by the *notify()*-method it provides.
3. The context client, no longer interested in getting information from the context source, needs to unsubscribe from the context source. The method provided by the context source is *unsubscribe()*.

As mentioned above, the example's context source provides 3 methods accessible by the context client: *subscribe()*, *query()* and *unsubscribe()*.

These 3 methods are exported as the AmigoService by the *activate()*-method of the “ContextSourceComponent.java”-file in the “AmigoCSEExample/src/contextsource”-folder.

The first part of the *activate()*-method exposes the ContextSourceImpl as an AmigoExportedService:

```
public void activate(){
    ...
    server = new ContextSourceImpl();
    myService = serviceExporter.createService(server);
    ...
}
```

The description of the methods to be exposed onto the network is the following:

```
...
ExportedMethod[] methods = new ExportedMethod[3] // there are 3 methods to be exported
Methods[0] = new ExportMethod("query", new String[]{"contextQueryExpression"});
Methods[1] = new ExportMethod("subscribe",
    new String[]{"contextSubscriptionCharacterisation",
        "contextSubscriptionReference"});
Methods[2] = new ExportMethod("unsubscribe", new String[]{"contextSubscriptionID"});
myService.exportMethods(AmigoReference.DEFAULT, methods);
...
```

After setting up the event sender, the *activate()*-method looks up the context broker:

```

...
final EventSender sender = myService.getEventSender();
server.setSender(sender)
/*now try to find a context broker to register this context source at */
AmigoService[] services = new AmigoService[0];
...
try{
    services = lookup.lookup("urn:amigo","ContextBroker",3);
    ...
}catch(Exception e1){
    ...
}
...

```

To allow the context source to be discovered by context clients, it has to be registered at the context broker by invoking the *registerContextSource()*-method. The specified string argument *rdfCapability* describes the capability of the context source and is expressed as RDF. A context client has to use the identical *rdfCapability* string to discover the context source.

```

...
context_broker = services[0];
/* specify the rdf capability of the context source */
String rdfcapability = "<?xml version='1.0'>" +
    "<rdf:RDF"+
    ...
    "<\rdf.RDF>";
/* and register the Context Source */
String csref = myService.getReference().getURL();
...
...
try{
    context_broker.getGenericStub().invoke("registerContextSource",
                                           new String[]{"contextInfoDesc",csref },
                                           new Object[]{rdfCapability,csref});
}catch(Exception e2){
    ...
}
}

```

The exported methods themselves are specified in the "ContextSourceImpl.java"-file, located in the "AmigoCSEExample/src/contextsource"-folder.

4.6.1.1 Query()

The *ContextSourceImpl* -class provides a method called *query()* that takes a String as argument and returns a String. The String argument will contain a contextQueryExpression expressed in SPARQL², the standard RDF³ query language. The string returned will contain the reply to this query.

Depending on how the context source handles the query, the implementation of this method can be simple or complex. In the AmigoCSEExample the Jena SPARQL interpreter is used. The context information is stored in a jena RDF model accessible through a data member "model":

```
public String query(String contextQueryExpression) {
    String returnStr = "";
    Logger.info("Example Context Source query() method called.");
    synchronized (model) {
        Query query = QueryFactory.create(contextQueryExpression) ;
        ...
        QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
        try {
            ResultSet results = qexec.execSelect() ;
            ...
            returnStr = ResultSetFormatter.asXMLString(results);
            ...
        } finally { qexec.close() ; }
    }
    return returnStr;
}
```

4.6.1.2 Subscribe()

The *ContextSourceImpl*-class provides a method called *subscribe()*. This method takes two string arguments. The first argument 'contextSubscriptionCharacterisation' specifies the context information the client is interested in.

This argument is expressed using the same language already used for the contextQueryExpression in the *query()* method.

The second argument, the 'contextSubscriptionReference', is used by the context source to hold the reference to the client. The method returns a string which holds an identifier of the subscription. This identifier will be sent each time an event is posted by the context source. It is used by the client to identify the context source that has posted the event, as a client might have subscribed to several context sources. This identifier is also used by the context source to index this subscription in a hashtable structure that stores all subscriptions registered so far.

The implementation of this method is the following:

```
public String subscribe(String contextSubscriptionCharacterisation,
                       String contextSubscriptionReference) {
    logger.info("IContextSubscription.subscribe("+contextSubscriptionCharacterisation+",
```

²see <http://jena.sourceforge.net/ARQ/Tutorial/index.html> for further information on SPARQL

³see <http://www.w3schools.com/rdf/default.asp> for further information on RDF

```
        "+contextSubscriptionReference+") called");  
  
    /*  
     * add the subscription to the table;  
     * the event is the 'own' name + some increasing number  
     */  
    eventNr++;  
    String eventID = eventName+eventNr;  
    /* Ignore possible double subscriptions for the moment ... */  
    Subscription sub = new Subscription(contextSubscriptionCharacterisation,  
                                       contextSubscriptionReference,eventID);  
  
    /* The subscriber should listen to this event name */  
    subscriptions.put(eventID, sub);  
    lastResults.put(eventID, "");  
    return eventID;  
}
```

4.6.1.3 Unsubscribe()

The *ContextSourceImpl*-class provides a method called *unsubscribe()* that is invoked by the context client to disable one of its previous subscription. This subscription is identified by the 'contextSubscriptionID' argument, which corresponds to the event identifier assigned to it during subscription.

```
public boolean unsubscribe(String contextSubscriptionID) {  
    logger.info("IContextSubscription.unsubscribe("+contextSubscriptionID+") called");  
    /* We don't check for double subscriptions yet */  
    subscriptions.remove(contextSubscriptionID);  
    lastResults.remove(contextSubscriptionID);  
    return true;  
}
```

4.6.2 Context Source Control behaviour

At this point we come back to the two approaches for designing the context source control behaviour – the 'query driven' and the 'data drive' approach. This tutorial focuses on the 'data driven' approach. Thus, the 'query driven' way of designing a context source is just described briefly.

4.6.2.1 Query Driven Context Source

Within the first approach the sensor data is retrieved and the corresponding model created only after the query of a context client has been received.

Here, the query is passed on to the 'underlying' context source, which will retrieve the data, create a context model and process the query. Processing the query is done by the *query()*-method introduced earlier. This is the 'query driven' approach.

How the context client can make use of the *query()*-method is described in the *doQuery()*-method of the "ContextClientImpl.java"-file, located in the "AmigoCCExample/src/contextclient"-folder.

4.6.2.2 Data driven context source

However, parallel to this 'query driven' approach of designing a context source there is a 'data driven' alternative. Sensor data is pushed to the context source, which updates its internal model according to the information the sensor passes to the *contextChanged()*-method. This approach is the appropriate one for asynchronous access to context information by context clients.

Real context is stored in a hashtable structure. This structure will be used to build a new context model and synchronizing the old model and the new one afterwards.

Thus far, general structures common to all context sources have been described. But when it comes to deal with data to be exchanged, a strict separation between general (*subscribe()*, *query()*, *unsubscribe()*) and specific structures is not possible any more.

The following section therefore utilizes the example context source introduced in chapter 4.5 to show how information is stored in an ontology model.

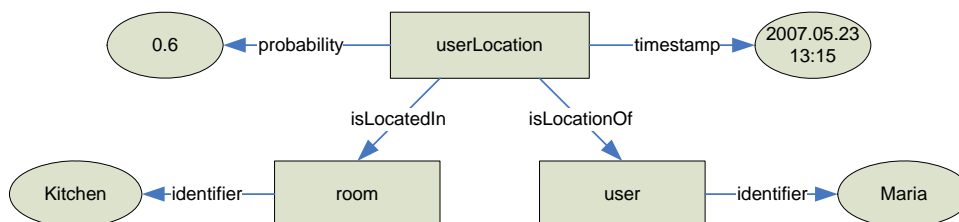
4.6.3 The Ontology Model

As already mentioned, all context information of the 'data driven' context source is stored in an ontology model. Building the ontology model is a two step process. First we have to create the basic elements of our model. Afterwards we can put data into it.

4.6.3.1 Creating the Ontology Model

The ontology model with the default settings is created through the Jena ModelFactory by simply calling *ModelFactory.createOntologyModel()*.

The created model and the methods provided with it are now used to create the desired topology. At this point, it is necessary to think about the information the model should keep. For this tutorial the information about a 'user location' is chosen, characterized by a 'user', the 'room' she/he is in and some properties of the current information (such as a timestamp and the probability/confidence).



The preceding example shows a model already filled with some information:

The given information 'userLocation' is the location of a 'user' with identifier Maria. The 'userLocation' is located in a 'room' with identifier Kitchen. The information about the 'userLocation' is tagged with a timestamp from the 5th May 2007 at 13:15. The confidence in the current information about the 'userLocation', expressed in terms of probability, is 60%.

The 'userLocation' is the top-level class of the model. The 'user' and 'room' can be considered lower-level classes. All classes are created using the *createClass()*-method of the ontology model.

The properties 'identifier', 'timestamp' and 'probability' are directly connecting classes with data of different kind. Thus, they are called data type properties and are created by calling the *createDatatypeProperty()*-method. The properties 'isLocationOf' and 'isLocatedIn' describe the interconnection of different classes. They are called object properties and are created by calling the *createObjectProperty()*-method. Note that the 'identifier' will be used twice later on - for the 'room' and the 'user' class.

The classes and different properties are created before actually assigning data to the different classes. For illustrative purposes, the ontology model is created each time the *contextChanged()*-method is called, though this may not be the most efficient way. This is done in the *contextChanged()*-Method in the file 'ContextSourceImpl.java':

```
public void contextChanged(Hashtable ht) {
    Enumeration en = ht.keys();
    /* build a new model based on this context */
    OntModel newModel = ModelFactory.createOntologyModel();
    /* create the class nodes of the model */
    OntClass userLocation = newModel.createClass(contextURI+ '#UserLocation');
    OntClass user = newModel.createClass(amigoURI+'#User');
    OntClass room = newModel.createClass(amigoURI+'#Room');
    /* create data type properties for this example */
    DatatypeProperty userName = newModel.createDatatypeProperty(contextURI #timestamp') ;
    DatatypeProperty roomName = newModel.createDatatypeProperty(contextURI #probability') ;
    DatatypeProperty probability = newModel.createDatatypeProperty(contextURI #identifier') ;
    /* create the object properties for this example
    ObjectProperty isLocationOf = newModel.createObjectProperty(contextURI +'#isLocationof') ;
    ObjectProperty isLocatedIn = newModel.createObjectProperty(contextURI +'#isLocatedIn') ;
    /* last but not least create the instances ... */
    ...
}
```

The created properties still have to be associated with the classes. This is done while creating the instances.

4.6.3.2 Creating Data Instances

The *createIndividual()*-method is used to create anonymous nodes for each class. The nodes are connected to the classes via adding a created property to it calling *addProperty()*. Since the sensor information is passed in a hashtable, each entry of the hashtable is processed.

```
...
while(en.hasMoreElements){
    /* get the information out of the hashtable */
    String newUser = (String)en.nextElement();
    String newRoom = (String)ht.get(newUser);
    /* the new user */
}
```

```

Individual newUser = user.createIndividual() ;
newUser.addProperty(identifier, newUserName) ;
/* the new room */
Individual newRoom = room.createIndividual() ;
newRoom.addProperty(identifier, newRoomName) ;
/* the new userLocation */
Individual newUserLocation = userLocation.createIndividual() ;
/* the timestamp of the current information */
...
newUserLocation.addProperty(timestamp, df.format(now.getTime())) ;
newUserLocation.addProperty(probability, '0.6')
newUserLocation.addProperty(isLocationOf, newUser)
newUserLocation.addProperty(isLocatedIn, newRoom)
}
...

```

For each entry in the hashtable, nodes are created and new data is associated with the classes by adding properties respectively.

Note that this example is only one way of implementing a 'data driven' context source, the precise implementation depends on the data being produced.

4.6.3.3 Updating the Ontology Model and Notifying all Subscribed Context Clients

The context source will change the context model accordingly and update the ontology model within the *contextChanged()*-method by simply overwriting the 'real' model, created in the *ContextSourceImpl* constructor, with the recently created one:

```

...
synchronized(model){
    /* copy the created model to the real one */
    model = newModel ;
}
...

```

The model created for the example user location of Maria is shown below.

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:j.0="http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:j.1="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#">
  <owl:Class rdf:about="http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#Room"/>
  <owl:Class rdf:about="http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#User"/>
  <owl:Class rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#UserLocation"/>
  <owl:ObjectProperty rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#isLocationOf"/>
  <owl:ObjectProperty rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#isLocatedIn"/>
  <owl:DatatypeProperty rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#probability"/>
  <owl:DatatypeProperty rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#timestamp"/>

```

```

<owl:DatatypeProperty rdf:about="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#identifier"/>
<j.1:UserLocation>
  <j.1:isLocatedIn>
    <j.0:Room>
      <j.1:identifier>Kitchen</j.0:identifier>
    </j.0:Room>
  </j.1:isLocatedIn>

  <j.1:isLocationOf>
    <j.0:User>
      <j.1:identifier>Maria</j.0:identifier>
    </j.0:user>
  </j.1:isLocationOf>
  <j.1:probability>0.6</j.0:probability>
  <j.1:timestamp>2007-05-23T13:15:06.672+0200</j.0:timestamp>
</j.1:UserLocation>
</rdf:RDF>

```

The first part is just defining some abbreviations for longer strings used repeatedly. Afterwards, the classes the model consists of are described, followed by object properties and data type properties. For each user in the model (here only one), a user location is specified - starting with <j1:UserLocation> and ending with </j1:UserLocation> respectively. Each property and each class is encapsulated by such start/end marks.

The query the context client has to pose to the context source thus looks the following⁴:

```

PREFIX amigo: <http://amigo.gforge.inria.fr/owl/AmigoCCS.owl#>
PREFIX context: <http://amigo.gforge.inria.fr/owl/ContextTransport.owl>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?name ?room ?time WHERE {
  ?id rdf:type context:UserLocation .
  ?id context:isLocationOf ?p .
  ?p context:identifier ?name .
  ?id context:isLocatedIn ?r .
  ?r context:identifier ?room .
  ?id context:timestamp ?time .}

```

The query asks for the 'name', the 'room' and the 'time' in the current model by looking for the occurrence of the specified RDF-triples.

The context source is now processing all context client subscriptions. Recall that each context client is given a subscription ID while invoking the context source's *subscribe()*-method.

Each subscription ID, and thus each context client, is associated with its query, too.

```

...
for (int i =0; i<SubIDs.length()){
  Subscription sub = (Subscription) subscription.get(subID[i]) ;
  String subQuery = (String) sub.getQuery() ;
  ...
}

```

⁴ <http://jena.sourceforge.net/ARQ/Tutorial/index.html> provides a tutorial for SPARQL

The context source uses this query in the call to its *query()*-method (also designed to be used in the 'query-driven' approach) to process the created ontology model.

```
...  
String result = this.query(subQuery) ;  
...
```

If the query returns a valid result (a result that is not empty and different from the result last pushed), the subscribed context clients are notified by using the *notify()*-method with the according eventID as the first argument.

```
...  
if (result !=null && (result.compareTo((String)lastResults.get(SubIDs[i])) != 0)){  
    ...  
    NotificationData data = new NotificationData() ;  
    /* use the notificationKey as requested by the subscriber */  
    data.setProperty(sub.getNotificationKey(), result);  
    ...  
    /* notify the subscriber using the eventID and the query result */  
    sender.notify(sub.getEventID, data) ;  
}  
}
```

Thus, if the answer to the subscription query has changed, the subscribed context client(s) will receive a notification.

4.7 Writing a Context Client

The last section already indicated that writing a context client is directly connected to writing a context source. The context client has to be aware of the context source, the kind of information it provides and especially the ontology model it is using to represent the context information. Knowledge of the latter is extremely important, since the query of the context client has to be adapted to the model created by the context source.

The context client is activated by calling the *activate()*-method in the "ContextClientComponent.java"-file (to be found in the "AmigoCCExample/ src/contextclient"-folder) initializing both, the implementation of the context client itself and the GUI-interface.

To get information from a context source, the context client has to perform the following steps :

1. Find a context broker (remember that only one context broker should be running in your network).
2. Ask the context broker for available context sources matching the context client's needs (specified in the RDF capability of the context client).
3. *subscribe()* to a context source returned by the context broker and start a thread refreshing the subscription from time to time (necessary, since subscriptions may expire).
4. The context source is controlled 'data driven', thus wait for the sensor information to change and the context source to call the context client's *notify()*-method .
5. Unsubscribe from the context source when the context client should be stopped by calling the context source's *unsubscribe()*-method.

Note: If the context source is to be controlled 'query driven', there's no need to subscribe to it. Instead use the context source's *query()*-method directly.

The implementation of the methods performing the described steps can be found in the "ContextClientImpl.java"-file, which is located in the "AmigoCCExample/src/contextclient"-folder

4.7.1 Finding a Context Broker

The first step in the initialization of the context client is to find a context broker in the network. This is done by calling the *findContextBroker()*-method once during *init()*, which is in turn called by the *activate()* method:

```
private AmigoService findContextBroker(){
    AmigoService broker = null ;
    try{
        broker = lookup.lookupFirstService('urn:amigo','ContextBroker') ;
        ...
    }catch(AmigoException e){
        ...
    }
    return broker ;
}
```

The *findContextBroker()*-method utilizes the middleware's lookup service to get a reference to the current context broker.

4.7.2 Finding the Context Sources

With the reference to the current context broker, the *findContextSource()*-method can be called from within the *getContextSources()*-method invoking the context brokers *discoverContextSource()*-method.

```
private boolean findContextSources(AmigoService broker, String rdfCapability){
    ...
    try{
        ...
        String result = (String) broker.getGenericStub().invoke('discoverContextSource',
                                                                new String[]{contextInfoDesc}, new Object[]{rdfCapability});
        ...
    }
```

The context client is only interested in information from context sources matching its 'rdfCapability' (specified in the first part of the *ContextClientImpl*-class). The 'rdfCapability' on the context client's side are sometimes called 'rdfNeeds', too.

The returned string contains a list of all context sources matching the RDF capability of the context client:

```
< ?xml version='1.0' encoding='UTF-8' ?>
<listref>
  <ref> http://131.234.222.123:8080/ksoap2/AmigoCSEExample </ref>
```

<\listref>

In the preceding example, only one reference to a context source is returned.

4.7.3 Subscribing to a Context Source

Using the context source reference(s) from the listing above, the context client can subscribe to a context source by invoking the context source's *subscribe()*-method from within the *subscribeCS()*-method. The reference to the context source is used to create/import the *AmigoService* (take a look at the *connectCS()*-method):

```
...
source = AmigoImportedService.createService(
    new AmigoReference(AmigoReference.SOAP, csref )
...

```

The reference to the *AmigoImportedService* can now be used to subscribe to the context source.

Recall: the context source's *subscribe()*-method requires two arguments to be passed to it. The one is the subscription reference or notificationKey (here the string 'result'), the other is the query expression, which is expressed in SPARQL, the standard RDF query language.

```
private subscribeCS(AmigoService source){
    boolean retval = true ;
    try{
        ...
        ...
        String eventID = (String) source.getGenericStub().invoke('subscribe',
            new String[]{"contextSubscriptionCharacterisation",contextSubscriptionReference'},
            new Object[]{queryString,notificationKey}) ;
        /* the subscribe method returns the event to subscribe to/listen for */
        ...
        /* subscribe to the event */
        Source.getSubscriptionManager.subscribe(this,eventID,-1);
        /* also start up a thread for refreshing the subscription ... */
        subscriptionRefresh = new SubscriptionRefresh(this, eventID,
                                                    source.getSubscriptionManager, 60000) ;
        subscriptionRefresh.start()
        ...
    }
}
```

The context client has now successfully subscribed to the context source.

Since subscriptions can expire (time-out is set to one hour) a *SubscriptionRefresh()*-thread is started, too.

4.7.4 Receiving and Processing Context Information

The implemented context client is now able to receive context information from the context source. Since the context source is designed to be 'data driven', the context client has to specify the *notify()*-method mentioned in chapter 4.6.

Note: The example code also contains a *doQuery()*-method, which shows the way the 'query driven' approach can be implemented on the context client's side. There is no need to subscribe to the context source. The reference to the context source will be used to do the query. On the context source's side, only slight modification should be necessary.

The *contextChanged()*-method of the context source is used to notify the context client. The arguments passed to the *notify()*-method are the subscription's 'eventID' and the 'notificationData'. Recall that the 'notificationData' contains the result of the query the context source poses on its ontology model. The 'eventID' is just a tag indicating the origin of the context information.

The notificationKey (here the string 'result') used for subscription is used to retrieve the answer from the 'notificationData'.

```
public void notify(String tag, NotificationData data){  
    /* use the notificationKey, which we used earlier for the subscription, to retrieve the answer */  
    String result = (String)data.get(notificationKey) ;  
    ...  
}
```

The SPARQL result is now processed using the *SPARQLResultHelper()*.

```
...  
/* now create a SparqlResultHelper to process the SPARQL result */  
String shr = new SparqlResultHelper() ;  
String rsIts = shr.process(result) ;  
...  
}
```

Finally, the returned data is examined for the desired entries by searching for the variables already defined in the SPARQL query. Each found context information is put into a hashtable.

4.7.5 Unsubscribing from a Context Source

If the context client is no longer interested in getting information from the context source, it should invoke the context source's *unsubscribe()*-method. The started subscriptionRefresh thread should be stopped, too.

```
private void unsubscribeCS(){  
    subscriptionRefresh.stop();  
    try{  
        source.getGenericStub().invoke('unsubscribe',  
                                       new String[]{contextSubscriptionID},  
                                       new Object[]{eventid}) ;  
        ...  
        Source.getSubscriptionManager().unsubscribe(this) ;  
    }catch(Exception e){  
        ...  
    }  
}
```


4.8 Tutorial Context Source and Client

For the following part, skeletons for a context source and a context client are provided. Both can be found in the tutorial's "SkeletonAmigoCSCC"-folder (AmigoCSSkeleton and AmigoCCSkeleton).

The Java sources are design as a cloze. Thus, missing code fragments have to be completed/inserted by you. Each time you have to put in your implementation, you'll see a structure like this in your code:

```
/*
 * *****
 * ***** some instructions *****
 * *****
 */
// your implementation goes in here
/*
 * *****
 */
*/
```

The tutorial concentrates on the application specific parts of the context source and context client only. The steps all context source/client implementations have in common are mentioned but not covered in detail anymore (e.g.: registering a context source/client at the context broker, subscribing a context client to a context source). For a more detailed description we'd like to refer to the previous chapters.

The skeletons for the context source and context client can be used to implement any kind of context source/client application.

We'll illustrate the process of completing a context source and an according context client on a simple example:

The context source will get information about the temperature in a specified room from a (virtual) temperature sensor, which will periodically push information to it by calling it's *contextChanged()*-method.

The context source will then process the information and *notify()* the context client of any context changes. The context client will parse the received information and display the result.

4.8.1 Preparing the Projects for being imported into Eclipse

First of all, you should copy the "AmigoCSSkeleton" (for the context source) and the "AmigoCCSkeleton" (for the context client) folders to your favourite directory. If you feel like renaming the folders, choose a name appropriate for the context source and context client you're about to implement. Since we are implementing a temperature application, we'll rename the "AmigoCSSkeleton" to "AmigoTemperatureCS" and the "AmigoCCSkeleton" to "AmigoTemperatureCC" respectively.

You should also change the project names of the context source and context client to match your needs, since this will be the names visible in eclipse. This could be done either prior to importing by editing the ".project"-file in the project folders (possibly hidden!) or after importing by simply renaming the projects, which is what we do.

4.8.2 Importing the Projects into the Eclipse Workspace

To import the projects to the eclipse workspace perform the following steps twice - for the context source (AmigoCSSkeleton/AmigoTemperatureCS) and the context client (AmigoCCSkeleton/AmigoTemperatureCC):

1. Click "File"->"Import" or right-click in the Navigator/Package Explorer and choose "Import" from the menu popping up
2. Select "General"-> "Existing Project into Workspace"
3. Click "Next"
4. Enter *PathToYourProject* (the full path to where the ".project"-file is located) in the "Select Root Directory" field – ok, browsing to the location of the .project-file is possible, too
5. Click "OK"
6. Click "Finish"

After performing the mentioned steps for the context source and the context client, the two projects should be visible in your Navigator/Package Explorer and already open. Now we can rename the project:

7. Right-Click on the project folder (still AmigoCSSkeleton/AmigoCCSkeleton)
8. Choose "Refactor->Rename" from the popped-up menu
9. Enter the new project name: AmigoTemperatureCS/AmigoTemperatureCC
10. Click "OK"

4.8.3 Setting Project Variables for Oscar

After importing the context source and the context client projects into the eclipse workspace, some variables have to be set/changed.

The first file to edit for the context source (and the context client respectively) is the "build.xml"-file in the according project folder. Perform:

11. Double-Click on the "build.xml"-file to open it for editing
12. Set the project name in "<project name=" to the project name chosen earlier (i.e. AmigoTemperatureCS and AmigoTemperatureCC)
13. Set the bundle name (the name of the .jar-file to be generated) in <property name="bundle.name" value=" to an appropriate value (typically the project name again)

Changing the bundle name in step 13 is just giving the .jar-file a new name, which is only important for the installation of the bundle(s) in Oscar (since we need to know the name under which we can find the bundle's .jar-file afterwards).

What will be displayed in the Oscar GUI after installation has to be set in the "manifest.mf"-file in the "res"-folder of your project(s).

14. Double-Click on the "manifest.mf"-file in the "res"-folder to open it for editing

An overview over the main bundle properties will appear. At the bottom there'll also be some tabs. One of them is the "manifest.mf" file, another the "build.properties"-file. These are the files we want to edit. Click on the "manifest.mf"-file first:

15. Set the "Bundle-Description"; the bundle description will be used only if the bundle will be included in the OBR and should give a brief description of the bundle

16. Set the “Bundle-Name”; the bundle name will appear in the Bundle List as the Location of the bundle and in the OBR as its Name; we’ll choose AmigoTemperatureCS/AmigoTemperatureCC, again
17. Set other optional properties (Created-By, Bundle-Vendor, Bundle-Version, ...) if desired, too

Next, click on the “build.properties”-tag and adapt the “Bundle-Name” and the “Bundle-Version” to the once chosen/specified in the “manifest.mf”-file.

Now we are set up to complete the source code of our context source and context client.

4.8.4 Completing the Context Source

Completing the context source (AmigoTemperatureCS) is the first we need to do. The two files to be edited are located in the “src/contextsource”-folder in the project path.

The first file to be modified is the “ContextSourceComponent.java”-file. We jump directly to the *activate()*-method. As the AmigoService is exported here, we need to change some properties concerning the communication between context source and context client. Since communication is based on the context broker, we’ll first give our context source an appropriate name. This is done by setting the “oid” (object identifier) of the context source, which specifies the name under which the context source will be registered at the context broker. This is done by calling the service’ method *addproperty()* with the appropriate arguments:

```
/*
 * *****
 * ***** specify object (the CS) identifier (oid) here! *****
 * *****
 */
myService.addProperty("oid", "TemperatureSensor");
/*
 * *****
 */
```

We choose *TemperatureSensor*, since that’s what our context source represents.

The next step is to give the event posted to the context client(s) a name. If a context client has subscribed to the context source, a number is added to this name. The pair “event name + number” is unique for each context client and is used by the context source to index each subscription (i.e. each context client) and by the context client to identify the context source and the events posted by it respectively.

This identifier will be sent each time an event is posted. The event number will automatically be added by the context source at subscription, the event name has to be set as follows:

```
/*
 * *****
 * ***** set the event name here! *****
 * *****
 */
server.setEventName("TemperatureSensor");
/*
 * *****
 */
```

Not only for convenience, we choose “TemperatureSensor”, again. Thus each event is directly associated with our context source.

The last thing to set in the “ContextSourceComponent.java”-file is the capability of our context source. The capability is expressed in RDF.

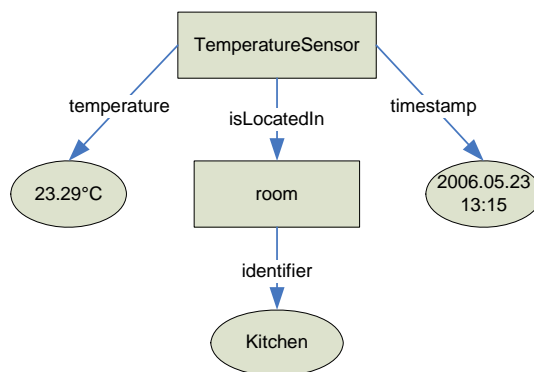
To specify the RDF capability, knowledge about the information (structure of the ontology model) is necessary. That is: we need to know the URIs (uniform resource identifiers) used for our ontology model and a name for our context.

Our Model should convey information from a "TemperatureSensor". This is the first class of our model, too. The "TemperatureSensor" posts a "temperature", tagged with a "timestamp". The "TemperatureSensor" "isLocatedIn" a specific "Room" (the second class in our model), which has an "identifier".

The classes and properties of our context source are already defined and their URIs should be used⁵. The URIs are:

<http://amigo.gforge.inria.fr/owl/Domotics.owl>, for "TemperatureSensor"
<http://amigo.gforge.inria.fr/owl/AmigoICCS.owl>, for "Room" and "temperature"
<http://amigo.gforge.inria.fr/owl/ContextTransport.owl>, for "timestamp", "identifier" and "isLocatedIn"

Thus, our ontology model (already filled with information) looks the following:



Therefore, our RDF capability can be specified with:

```

/*
 * ***** specify the rdfcapability of the context source *****
 * *****
 */
String rdfcapability = ""+
"<?xml version='1.0'?>" +
"<rdf:RDF"+
"  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'"+
"  xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'"+
"  xmlns:owl='http://www.w3.org/2002/07/owl#'"+
"  xmlns:j.0='http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#'"+
"  xmlns:j.1='http://amigo.gforge.inria.fr/owl/Domotics.owl#'"+
"  xmlns:daml='http://www.daml.org/2001/03/daml+oil#'"+
"  xmlns:j.2='http://amigo.gforge.inria.fr/owl/ContextTransport.owl#'>" +
"  <j.2:ContextSourceRegistration>"+
"    <j.2:timeliness>current</j.2:timeliness>"+
"    <j.2:contextType>TemperatureSensor</j.2:contextType>"+

```

⁵ If you write your own context source, you'll probably use your own URIs. No problems here, just keep in mind that the URIs are used by the context client later on, too. For further information on RDF and URIs please refer to one of the numerous tutorials like the one on <http://www710.univ-lyon1.fr/~champin/rdf-tutorial/rdf-tutorial.html>.

```
" </j.2:ContextSourceRegistration>" +
"</rdf:RDF>";
/*
 * *****
 */
```

That's it for the "ContextSourceComponent.java"-file. The next file to edit is the "ContextSourceImpl.java"-file in the same folder.

Since we are using the mentioned URIs for our ontology model, which will be created in the *contextChanged()*-method of this file, we need to specify them here, again:

```
/*
 * *****
 * ***** specify the URIs used for the Ontology Model *****
 * *****
 */
private final String domoticURI = "http://amigo.gforge.inria.fr/owl/Domotics.owl";
private final String amigoURI = "http://amigo.gforge.inria.fr/owl/AmigoICCS.owl";
private final String contextURI = "http://amigo.gforge.inria.fr/owl/ContextTransport.owl";
/*
 * *****
 */
```

Since we are not working with real sensors, we need to start a thread that is generating our sensor information, too. The thread's source code can be found in the "ContextGenerator.java"-file. All it does is generating random temperatures in the range from 18°C to 36°C for a specified room. The information is pushed to the context source in a hashtable (pairs are: room, temperature) via the *contextChanged()*-method. Therefore, we need to pass a reference to the calling class (this-pointer) to the thread. Starting the thread is done in two steps, creating an instance of it and *start()* it:

```
/*
 * *****
 * ** start up a new thread that is generating sensor information **
 * *****
 */
tempSensor = new ContextGenerator(this);
tempSensor.start();
/*
 * *****
 */
```

Once started, the thread can be stopped by calling *stopGen()* from within the *stopCtxtSource()*-method:

```
/*
 * *****
 * ** stop the thread that is generating sensor information **
 * *****
 */
if (tempSensor != null) {
    /* stopping simulation */
    tempSensor.stopGen();
    tempSensor = null;
}
/*
 * *****
 */
```

In a real application using real sensors, these are the parts where a hardware interfaces should be started/stopped. Now we can proceed to the main issue of creating the ontology model of our context source.

The ontology model of our context source is created in the constructor. It is updated by the one created in the *contextChanged()*-method. Thus we take look at the *contextChanged()*-method.

The model with its classes, object and data type properties has already been introduced. Thus the first step is to create a default ontology model (i.e. an ontology model with default properties). Next, we create the two classes "TemperatureSensor" and "Room" preceded by the URIs specified in the RDF capability of our source (the pair separated by a '#' is now called a *qualified URI*). In a similar fashion, we create the data type properties "timestamp", "temperature" and "identifier" and the object property "isLocatedIn", each with its corresponding URI.

This *empty* model has now to be filled with data. The information about the temperature of a specific room is passed to the *contextChanged()*-method in a hashtable. This hashtable is used to get the "newRoomName" and the "newTemp". The ontology class' method *addproperty()* is used to map the information (room, time, temperature) to the according classes and to draw the "isLocatedIn"-interconnection between the two classes "TemperatureSensor" and "Room".

The whole process of building the ontology model is intelligible from the following code fragment, which has to be inserted at the according position in the source code:

```
/*
 * ****
 * ** create the ontology model based on the information to be passed to the CC **
 * ****
 */
/* create a default ontology model*/
OntModel newModel = ModelFactory.createOntologyModel();
/* create the class nodes for this example */
OntClass temperatureSensor = newModel.createClass(domoticURI+"#TemperatureSensor");
OntClass room = newModel.createClass(amigoURI+"#Room");
/* create the datatype properties for this example */
DatatypeProperty timestamp = newModel.createDatatypeProperty(contextURI+"#timestamp");
DatatypeProperty temperature = newModel.createDatatypeProperty(amigoURI+"#temperature");
DatatypeProperty identifier = newModel.createDatatypeProperty(contextURI+"#identifier");

/* create the object properties for this example */
ObjectProperty isLocatedIn = newModel.createObjectProperty(contextURI+"#isLocatedIn");

/* create the instances: 'OntClass.createIndividual' is used to create anonymous nodes */
while (en.hasMoreElements()) {
    /* get the information out of the hashtable */
    String newRoomName = (String)en.nextElement();
    String newTemp = (String)ht.get(newRoomName);
    /* the new room */
    Individual newRoom = room.createIndividual();
    newRoom.addProperty(identifier,newRoomName);
    /* the new sensor location */
    Individual newTemperatureSensor = temperatureSensor.createIndividual();
    /* the timestamp of the current information */
    Calendar now = Calendar.getInstance();
    SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
    newTemperatureSensor.addProperty(timestamp, df.format(now.getTime()));
    /* and the remaining properties */
    newTemperatureSensor.addProperty(temperature, newTemp);
}
```

```

newTemperatureSensor.addProperty(isLocatedIn, newRoom);
}
/*
 * *****
 */

```

Ok, we're done with the context source and can compile it now.

Right-Click on the "build.xml"-file and chose "run->Ant Build" from the popped-up menu. If you did all the above steps, you shouldn't get any errors and the "AmigoTemperatureCS.jar"-file should have been created in the "dist"-folder of your project.

4.8.5 Completing the Context Client

The file to be modified for the context client is the "ContextClientImpl.java"-file in the project folder of the context client (AmigoTemperatureCC). If you haven't done the steps to import the context client project into the eclipse workspace, you should go to section 4.8.1 and repeat all steps up to section 4.8.3 for the context client.

The context client is interested in the information the context source posts. To discover the context source, the context broker compares the RDF needs of the context client (the RDF capability the context client is looking for) with the RDF capabilities of all registered context sources. Only if there's a match, the context client will be provided with references to those context sources and can subscribe to one (or even more) of them.

Thus, the first thing to do is to specify the RDF capability the context client is looking for. Again, it is important that context client and context source specify the same RDF capabilities in order for the context source to be discoverable by the context client:

```

/*
 * *****
 * ** specify the rdf capability of the Context Client ****
 * ***** remember that it needs to match the rdf *****
 * ***** capability of the Context Source *****
 * *****
 */
private static String rdfCaps = ""+
"<?xml version='1.0'?>" +
"<rdf:RDF"+
"  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'"+
"  xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'"+
"  xmlns:owl='http://www.w3.org/2002/07/owl#'"+
"  xmlns:j.0='http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#'"+
"  xmlns:j.1='http://amigo.gforge.inria.fr/owl/Domotics.owl#'"+
"  xmlns:daml='http://www.daml.org/2001/03/daml+oil#'"+
"  xmlns:j.2='http://amigo.gforge.inria.fr/owl/ContextTransport.owl#'"+
"  <j.2:ContextSourceRegistration>"+
"    <j.2:timeliness>current</j.2:timeliness>"+
"    <j.2:contextType>TemperatureSensor</j.2:contextType>"+
"  </j.2:ContextSourceRegistration>"+
"</rdf:RDF>";
/*
 * *****
 */

```

Since we know how the ontology model should look like and also should have the RDF capability right, we can define the query. Our query asks for a "room", it's temperature "temp" and the corresponding timestamp "time":

```

/*
 * *****
 * ** specify the query; keep the created ontology model in mind **
 * *****
 */
final private String queryString = ""+
"PREFIX domotic: <http://amigo.gforge.inria.fr/owl/Domotics.owl#> "+
"PREFIX amigo: <http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#> "+
"PREFIX context: <http://amigo.gforge.inria.fr/owl/ContextTransport.owl#> "+
"PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> "+
"SELECT ?room ?temp ?time WHERE { "+
"?id rdf:type domotic:TemperatureSensor . "+
"?id context:isLocatedIn ?r . "+
"?r context:identifieur ?room . "+
"?id amigo:temperature ?temp . "+
"?id context:timestamp ?time . "+
"}",
/*
 * *****
 */
*/

```

Since information without using it is not useful at all (at least for the Context Management System), we need an application using it.

The application of choice is just a simple gauge displaying the current temperature and the room the temperature is measured in (specified in the file 'Gauge.java' in the src/contextclient directory).

The application will be started in the constructor:

```

/*
 * *****
 * **** start up an application using the received information ****
 * *****
 */
temperatureGauge = new Gauge();
/*
 * *****
 */
*/

```

The application should stop when the context client will be stopped. Therefore, calling the *stop()*-method from within the "ContextSourceComponent.java"s *deactivate()*-method should also quit the application:

```

/*
 * *****
 * ***** remember to stop the client application *****
 * *****
 */
temperatureGauge.hide();
temperatureGauge.deactivate();
temperatureGauge = null;
/*
 * *****
 */
*/

```

Each time the context information changes and the context source *notify()*s the context client of this change, the gauge will update the displayed information. The context client is notified with the eventID (eventID=event name+number) given by the context source during the subscription process and the result of *query()*ing the ontology model with the specified query (passed to the context source during subscription as well).

As a first step in the *notify()*-method, the data is processed using the SPARQL helper. The result can now easily be parsed for the desired information:

```

/*
 * *****
 * ** Examine the SPARQL Results for the information interested in **
 * *****
 */
/*
 * Examine the SPARQL Results for new temperature information by searching for
 * "room", "temp" and "time" in the result (these where the variables we defined in
 * the SPARQL Query)
 */
String RoomName = null;
String Temperature = null;
String TimeStamp = null;
for (int i=0;i<rsIts.size();i++) {
    logger.debug(" Result #" + i + " " + ((Result)rsIts.get(i)).size() + " bindings");
    for (int j=0;j<((Result)rsIts.get(i)).size();j++) {
        Binding b = (Binding) ((Result)rsIts.get(i)).get(j);
        if (b.getName().equals("room")){
            RoomName = b.getValue();
        }else{
            if (b.getName().equals("temp")){
                Temperature = b.getValue();
            }else{
                if (b.getName().equals("time")){
                    TimeStamp = b.getValue();
                    logger.debug(TimeStamp + ": Temperature in " + RoomName + " is " + Temperature );
                    /* update the hashtable by setting the value "Temperature" for the hashtable key "RoomName"
                     * if the key "RoomName" doesn't exist, a new key is added to the hashtable;
                     */
                    temperatureLog.put(RoomName, Temperature);
                    /* update the display information*/
                    temperatureGauge.update(RoomName, Temperature);
                }
            }
        }
    }
}
}
}
/*
 * *****
 */

```

This is also the location to call the gauge's *update()*-method, updating the displayed temperature and room by the recently received information.

That's it. We're done with the context client, too. Right-Click on "build.xml" and choose "run->Ant Build" from the popped-up menu. If you performed all the steps, you shouldn't get any errors, the "AmigoTemperatureCC.jar"-file should have been created in the "dist"-folder of your project and we're ready to install the two created bundles (AmigoTemperatureCS and AmigoTemperatureCC) in Oscar.

4.8.6 Installing our Bundles

To install the bundles, start (if not already done) an Oscar platform and switch to the Bundle List view. There should already be a heap of bundles installed. Take a look at the following snapshot to find out, if you are missing one of them:



If you are missing one or more of the displayed bundles, mark the according bundle(s) in the OBR and click “Start All”.

Now you can proceed with installing the context source and the context client.

In the Bundle List URL field on top of the Oscar GUI, you can specify the location of the bundles’ .jar-files (to be found in the “dist”-folder of your projects) preceded by “file:” (Windows) or “file://” (Linux).

Thus, a valid URL for Windows would look like this:

<file:\\MyData\\Amigo\\Tutorial\\AmigoTemperatureCS\\dist\\AmigoTemperatureCS.jar>,

where “MyData” is a folder in the root directory (e.g.: C:) of your System.

A valid URL for Linux would look like this:

<file:///home/user/MyData/Amigo/Tutorial/AmigoTemperatureCS/dist/AmigoTemperatureCS.jar>,

where “home” is a folder in the root directory of your System.

Clicking the “Install”-button will add the bundle to the list.

After you installed both, the context source and the context client, you can activate them. Mark the corresponding bundle in the bundle list by clicking on it and simply press the “start”-button.

Since your context client is asking the context broker for available context sources at start-up only, the context source should be started first. This ensures that the context source has already registered at the context broker when the context client asks the context broker for available context sources.

That’s it! You just wrote and started your own Amigo middleware compatible context application – a context source posing context information onto the network and a context client processing this context information.

Given the provided skeletons for context source and context client, you should now be able to implement your own context sources and applications.

In the next section we will show you how to use provided helper components for either .NET or OSGi to make developing context sources and context clients even easier.

4.9 Writing Context Sources and Client using Helper components

There are several API packaged as helper bundles that can be used for easing the development of context sources. Example code can be seen in the source bundles of CMS, downloadable from the CMS OBR at <http://core.lab.telin.nl/~amigo/obr/repository.xml>.

4.9.1 OSGi helper component for Context Sources

A collection of helper or utility components makes it easier for developers to implement context sources or applications. This is a recommended option, not a mandatory way to go. When ignoring these components, the developer has to do everything by himself, including working with ontologies, exposing the CMS interface, registering with the Context Broker, handling subscriptions and so on.

In this section, we introduce the helper components and explain how to use them.

ContextSourceManager provides functionality that is common to all Context Sources, such as (de)registering with the Context Broker. Likewise, SimpleContextSourceFront provides functionality for answering queries from Consumers, and handling subscriptions. They also provide Java classes that aid in creating ontology models.

To illustrate the use of this component, we give below an excerpt of the source code of a context source that uses its functions.

```
...
import nl.telin.amigo.contextmanagement.contextsource.IContextSource;
import nl.telin.amigo.contextmanagement.csmw.ContextSourceManager;
import nl.telin.amigo.contextmanagement.csmw.SimpleContextSourceFront;
...
```

These three lines are necessary for the context source to implement the IContextSource interface and for using the ContextSourceManager and SimpleContextSourceFront classes.

The context source should bind to the ContextSourceManager component through two methods that we call for convenience bindContextSourceManager and unbindContextSourceManager.

The implementation of the two methods should be as simple as:

```
...
private ContextSourceManager manager = null;
...
public void bindContextSourceManager(ContextSourceManager manager) {
    logger.debug("bindContextSourceManager()");
    this.manager = manager;
}

public void unbindContextSourceManager(ContextSourceManager manager) {
    logger.debug("unbindContextSourceManager()");
    if (this.manager == manager) {
        this.manager = null;
    }
}
```

```
}

```

Where manager is a private member used to access the ContextSourceManager functionality.

When while activating the context source, it should register to the context broker. This is done within the activate method as:

```
...
private SimpleContextSourceFront csf = null;
private OntModel model = null;
...
public void activate() {
    logger.debug("activate");
    csf = manager.registerContextSource(rdfCaps, this, 5);
    model = csf.getOntModel();
}

public void deactivate() {
    logger.debug("deactivate");
    if (csf != null) {
        csf.deregisterContextSource();
    }
}
...
```

The method registerContextSource is called for registering the context source to the Context Broker. It also returns a handle to an instance of ContextSourceFront object which can be used to support queries and subscriptions. The third argument of the method specifies a timeout in second between re-evaluations of subscriptions. Whenever this timeout elapses, the current subscriptions will be re-evaluated by calling the context source query method. If the answer is different from the previous one the subscriber will get a notification.

The method of the ContextSourceFront object returns a preloaded model of the Amigo ontology. At deactivation, the context source should call the deregisterContextSource method. This method will properly remove the registration of the context source to the broker and remove the model associated with this context source if one exists.

At this point, there are two approaches for designing the context source control behaviour. Within the first approach the sensor data is retrieved and the corresponding model created only after the query of a context consumer has been received. Here, the query is passed on to the 'underlying' Context Source, which will retrieve the data, create a context model and process the query. This is a query driven approach. For this approach the method should be implemented this way:

```
public String query(String contextQueryExpression) {
    String returnStr = "";
    /* This is a 'pull' model.
     * In other words: we only build the model once someone is asking us something
     * Note that we re-use the model by emptying it beforehand
     */
}
```

...

The model member should be updated here, by retrieving and interpreting the sensors rough information in terms of the Amigo Ontology

```

...
returnStr = doQuery(model, contextQueryExpression);

return returnStr;
}

```

In a second stage the doQuery method is called to complete the "real" query. This method could be implemented this way:

```

public String doQuery(OntModel model, String queryString) {
    if (model == null) return "";
    String returnStr = "";
    if (model != null) {
        Query query = null;
        model.enterCriticalSection(LockMRSW.READ);
        QueryExecution qexec = null;
        try {
            query = QueryFactory.create(queryString);
            qexec = QueryExecutionFactory.create(query, model);
        } catch (RuntimeException e) {
            System.out.println("Exception : "+e.getMessage());
            e.printStackTrace();
        }
        if (qexec != null) {
            try {
                ResultSet results = null;
                if (query.isSelectType()) {
                    results = qexec.execSelect();
                    returnStr = ResultSetFormatter.asXMLString(results);
                } else if (query.isConstructType()) {
                    Model m = qexec.execConstruct();
                    RDFWriter rdfW = m.getWriter();
                    ByteArrayOutputStream bos = new
ByteArrayOutputStream();

                    rdfW.write(m, bos, null);
                    returnStr = bos.toString();
                } else if (query.isDescribeType()) {
                    Model m = qexec.execDescribe();
                    RDFWriter rdfW = m.getWriter();

```

```

        ByteArrayOutputStream      bos      =      new
ByteArrayOutputStream();

        rdfW.write(m, bos, null);
        returnStr = bos.toString();

    }
    } finally { qexec.close() ; }
}
model.leaveCriticalSection();
}
return returnStr;
}

```

The implementation of the doQuery method is a straightforward processing of a SPARQL query over an OWL/RDF model using the jena functionality. Interested reader could refer to the Jena documentation for a detailed explanation of this code.

Special care should be given to the time-out argument to the in this query driven approach. As introduced earlier, a time-out can be specified by the Context Source at startup when invoking the registerContextSource method. If the time-out is 0, no timer is started and therefore, subscriptions will not function.

Parallel to this query driven approach of designing a Context Source there is a data driven alternative. To illustrate this approach, we give below an excerpt of the source code of a Context Source where the sensor data is pushed to the Context Source. This approach is the appropriate one for asynchronous access to context information by context consumers.

```

public void contextChanged(Hashtable ht) {
    Enumeration en = ht.keys();
    // Build a new model based on this context
    // We will exchange the model in one go after building,
    // since the model can be accessed by different threads
    // Note that we re-use the model by emptying it just before rebuilding.

    model.removeAll();

    model.enterCriticalSection(LockMRSW.WRITE);
    while (en.hasMoreElements()) {
        ...
    }
}

```

Real context is stored in an ht hashtable structure. This structure will be explored to update the new context model model.

```

        ...
    }
    model.leaveCriticalSection();

    /*
    * (Re)setting the model will trigger processing of subscriptions
    */
}

```

```
*/  
    csf.setModel(model);  
}
```

The Context Source will change the context model accordingly and update the model within the `contextChanged` method towards the Context Source Manager. In this case the middleware will take care of almost everything on behalf of the Context Source. Queries will be processed by the middleware and subscriptions are processed every time the Context Source pushes a new model.

To prepare the context source bundle for deployment jump directly to the "Edit the `res/manifest.mf` file" section. The following section addresses the development of a context source when helper components are not used.

4.9.2 .NET helper component for Context Sources

In a few steps we will explain how to create a basic context source using the available helper classes. We start by creating a new project in Visual Studio and choosing the *Console Application* template.

First add the needed references to the project which are:

- - ContextSourceHelper.dll
- - EMIC.WebServerComponent.dll
- - EMIC.WSDiscovery.dll

These can be found in the gforge repository at `\ius\context_mgmt\common\trunk\ContextSourceHelper.NET\bin\Debug` and are the result of building the ContextSourceHelper solution.

We need to let the default *Program* class inherit the *ContextSourceHelper* class:

```
public class Program : nl.telin.amigo.ContextSourceHelper.ContextSourceHelper
```

The constructor of the *Program* class should use its base class constructor in order to initialize the helper class properly:

```
public Program() : base(0, "UserLocation") { }
```

The first argument specifies port on which the service should be published; '0' means to automatically assign the service to an arbitrary free port number. The second argument describes what kind of context this context source will provide.

Besides a description the context source should also be given a name, which can be done by overriding the *myServiceName* property:

```
public override string myServiceName { get { return "SimpleContextSource"; } }
```

Probably the most important method of a context source is the *GetRdfModel* method. This function is called whenever a query should be handled and is responsible for providing the current context information in RDF format. Our example context source will be very static and will keep providing the same RDF model every time:

```
public override string GetRdfModel()
{
    return ""+
        "<?xml version='1.0'?">" +
        "<rdf:RDF xmlns:iccs='http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#" +
        "xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#" +
        "xmlns:amigo='http://amigo.gforge.inria.fr/owl/Amigo.owl#" +
        "xmlns:daml='http://www.daml.org/2001/03/daml+oil#" +
        "xmlns:context='http://amigo.gforge.inria.fr/owl/ContextTransport.owl#" +
        "xmlns:owl='http://www.w3.org/2002/07/owl#" +
        "xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#" +
        "xml:base='http://amigo.gforge.inria.fr/owl/RFLocation#" +
        "<context:UserLocation>" +
        "    <context:timestamp>1-2-2007 16:37:11</context:timestamp>" +
        "    <context:probability>0.9</context:probability>" +
        "    <context:isLocatedIn>" +
        "        <iccs:Room>" +
        "            <context:identifier>garage</context:identifier>" +
        "        </iccs:Room>" +
        "    </context:isLocatedIn>" +
        "    <context:isLocationOf>" +
        "        <amigo:Person>" +
        "            <context:identifier>John</context:identifier>" +
        "        </amigo:Person>" +
        "    </context:isLocationOf>" +
        "</context:UserLocation>" +
        "</rdf:RDF>";
}
```

Next to manually specifying the RDF model, it is also possible to use the *SemWeb* library together with the *RDFHelper* class as illustrated below:

```
public override string GetRdfModel()
{
    // Create a new store for the triplets
    MemoryStore store = new MemoryStore();
}
```



```

// We want to add a new context item, user and location to the store
Entity context = new BNode();
Entity user = new BNode();
Entity location = new BNode();

// The context is of type UserLocation with a given TimeStamp and Probability
store.Add(new Statement(context, RDFHelper.RDF + "type", ContextTransport.UserLocation));
store.Add(new Statement(context, ContextTransport.TimeStamp, (Literal)DateTime.Now.ToString()));
store.Add(new Statement(context, ContextTransport.Probability, (Literal)"0.9"));

// The user is a person as defined in the Amigo ontology, he's identified as john
// and this context items says something about the location of this user.
store.Add(new Statement(user, RDFHelper.RDF + "type", Amigo.Person));
store.Add(new Statement(user, ContextTransport.identifier, (Literal)"John"));
store.Add(new Statement(context, ContextTransport.isLocationOf, user));

// The location of a room as defined in the AmigoICCS onotology, is is identified as garage
// and this context item has information about something inside this location/room.
store.Add(new Statement(location, RDFHelper.RDF + "type", AmigoICCS.Room));
store.Add(new Statement(location, ContextTransport.identifier, (Literal) "garage"));
store.Add(new Statement(context, ContextTransport.isLocatedIn, location));

return RDFHelper.GetAsString(store);
}

```

If we want to use this second method for creating the RDF model you should also be sure to add the SemWeb.dll as a reference. Our context source is now ready to be instantiated, which we will do in the program's main method. Once instantiated it will automatically publish its service, find a context broker, and register its service there. In order to properly shutdown the context source the *Dispose* method should be called. One could use the *using* statement to let the disposal be done automatically:

```

static void Main(string[] args)
{
    using (Program p = new Program())
    {
        Console.WriteLine("Running on " + p.myAddress);
        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    } // 'p' will be disposed here
}

```

We should now have a fully working Context Source. You can test it by starting up Oscar, building your solution in Visual Studio and executing the resulting executable. Using the Context Source Tester bundle in Oscar you should be able to find the context source using the 'Lookup CS' command and by executing the default query. If you want to shutdown the context source you should bring up the console window of the context source and press any key. The context source will now be properly closed and you should notice that you won't find it anymore if you execute the 'Lookup CS' command again.

4.9.3 OSGi Helper component for Context clients

ContextHelper is a bundle that provides Java Classes abstractions of the context broker, context sources, and facilities for subscribing to Context Sources. Those components make it easier for developers to implement context aware applications. This is a recommended option as these components will support many low level tasks.

To benefit from this component, the application should declare its dependency towards the ContextBrokerHelper. This should be done in the res/metadata.xml file, which content should be:

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component
class="nl.telin.contextmanagement.impl.contextclient.example.ContextClientComponent">
    <requires service="nl.telin.amigo.contextmanagement.contexthelper.ContextBrokerHelper"
      filter=""
      cardinality="1..n"
      policy="dynamic"
      bind-method="bindContextBrokerHelper"
      unbind-method="unbindContextBrokerHelper"
    />
  </component>
</bundle>
```

The application code should then implement the two methods "bindContextBrokerHelper" and "unbindContextBrokerHelper" both of them should return null;

The following line, give an example of how to implement bindContextBrokerHelper this method should be that of the application class that implements the Lifecycle interface.

```
...
private ContextBrokerHelper cbh = null;
...
public void bindContextBrokerHelper(ContextBrokerHelper cbh) {
    logger.debug("bindContextBrokerHelper");
    this.cbh = cbh;
}
...
```

The ContextBrokerHelper class provides functionality for interacting with a context broker and the ContextSourceHelper class provides functionality for interacting with a context source.

To illustrate the use of these components, we give below excerpts of the source code of a context client application that uses their functions.

```
...
import nl.telin.amigo.contextmanagement.contexthelper.*;
...
public class ContextClientImpl implements IContextChanged {
...

```

The two main helpers classes the ContextBrokerHelper class and the ContextSourceFront class are defined in the

nl.telin.amigo.contextmanagement.contexthelper package.

To access context sources asynchronously (to be notified of context change events), the application which main component has been named ContextClientImpl should implement the interface.

The context client application should bind to a ContextBrokerHelper instance through the method setContextBrokerHelper.

```
private ContextBrokerHelper cbh = null;
...
public void setContextBrokerHelper(ContextBrokerHelper cbh) {
    this.cbh = cbh;
}

```

Where cbh is a private member used as the first point of contact for context consumers. This object acts as an abstraction of interaction with the Context Broker, allowing for easier interaction. For instance, for discovering the Context Sources that are able to fulfill the needs of the application in terms of context information the instructions are as follow:

```
...
/*
 * Ask the context broker for a reference to a context source
 */
ContextSourceHelper[] csh = cbh.discoverContextSource(rdfNeeds);
...

```

Where discoverContextSource is the method used for finding Context Sources that match the needs specified by the application in the form of an RDF document rdfNeeds.

As briefly mentioned previously, elements of the list csh are not Context Sources but handles or abstraction of context sources.

Once the list of context sources is returned, several methods are available to interact with the context source.

For subscribing to a context change events, such instructions could be used:

```

...
    private Subscription subscriptionOn = null;
...
        subscriptionOn = csh[0].subscribe(this, queryStringOn, notificationKeyOn);
        if (subscriptionOn == null) {
            logger.debug("subscription ON failed");
            return false;
        }
...

```

Where is a object used to interact with subscription. At this time, the only action on subscription needed is to unsubscribe. In the lines of code above, the first Context Source of the list `csh[0]` is selected and a subscription to this Context Source is posed by invoking the `subscribe` method. The parameters of this method are the object that will be notified, in our case the application itself (`this`). This object should be an instance of a class that implements the `IContextChanged` interface. Such class is expected to implement the method `contextChanged` that will be triggered whenever the object receives an event notifying a context change. The second parameter is the SPARQL query `queryStringOn` to be used for this subscription. The third parameter `notificationKeyOn` is a string that is a reference internal to this application. It will be passed back to the object to be notified when the method `contextChanged` will be called. This reference makes it possible for an application to subscribe to more than one event, and to use this reference in the `contextChanged` method, to find out which event triggered the `contextChanged` method.

We show how such mechanism can be used in the following implementation of the `contextChanged` method:

```

...
import nl.telin.amigo.sparqlhelper.Binding;
import nl.telin.amigo.sparq

Ihelper.Result;
import nl.telin.amigo.sparqlhelper.Results;
import nl.telin.amigo.sparqlhelper.SparqlResultHelper;
...
    public void contextChanged(String subscriptionReference, String newAnswer) {
        ...
        /*
         * Now create a SparqlResultHelper to help process the SPARQL result
         */
        SparqlResultHelper srh = new SparqlResultHelper();
        Results rslts = srh.process(newAnswer);
        ...
    }

```

In addition to the argument `subscriptionReference`, a second argument `newAnswer` is used to convey more specific information. This argument is a `String`. The CMS convention is that this string is an XML document that encodes the reply of the SPARQL query that was used by the application as the context of interest while subscribing to the Context Source. To make the parsing of this additional information easier, several classes from the `nl.telin.amigo.sparqlhelper` package are available. Here we create an instance of the `SparqlResultHelper` class, which can be processed to generate a list of `Result` objects. Each basically corresponds to a RDF triple that form the reply.

The rest of the `contextChanged` method dispatches the processing of the event on the basis of the `subscriptionReference` key, and looks like:

```
...
    if (rsIts.size()>0) {           // We have someone entering or leaving!
        if (subscriptionReference.equals(notificationKeyOn)) {
            logger.debug("Switch on light!");
        }
        if (subscriptionReference.equals(notificationKeyOff)) {
            logger.debug("Switch off light!");
        }
    }
...

```

The following code illustrates how the `Result` objects can be analyzed

```
...
    for (int i=0;i<rsIts.size();i++) {
        logger.debug(" Result #" +i);
        logger.debug(" "+((Result)rsIts.get(i)).size()+" bindings");
        Result rslt = (Result)rsIts.get(i);
        Enumeration en = rslt.elements();
        while (en.hasMoreElements()) {
            Binding b = (Binding)en.nextElement();
            logger.debug(" "+b.getName()+" = (" +b.getType()+") "+b.getValue());
        }
    }
}

```

4.9.4 .NET helper component for Context clients

A Context Client helper library solution is made available in the CMS distribution. It provides abstractions for Context Broker and Context Sources that shield the client from having to

perform the low-level interaction with these components directly. The ContextHelperTest project within the solution provides an example of how to use the ContextHelper library.

The ContextClient.NET library can be used by adding a reference to ContextHelper.dll in \ius\context_mgmt\common\trunk\ContextHelper.NET\ContextHelper\bin\Debug.

The first thing a client must do is to get a reference to a Context Broker, this is done by simply instantiation a ContextBroker object:

```
ContextBroker cb = new ContextBroker();
```

The default constructor of the ContextBroker class will try to discover a Context Broker using WS-Discovery. Context Source(s) can now be found by calling the discoverContextSourceSimple or discoverContextSource method of the ContextBroker object. With the former method a simple type name is enough as a parameter; the ContextBroker object will expand it to a full RDF description for the ContextBroker component, the latter method can be used if the client wants to specify a full rdfNeeds description by itself. As an example, to find all Context Sources that provide a UserLocation, the client has to do the following:

```
ContextSource[] css = cb.discoverContextSourceSimple("UserLocation");
```

Where the length of the array that is returned is bigger than or equal to zero.

The resulting ContextSource objects are the abstractions of the real Context Sources returned by the query to the ContextBroker. These Context Sources can be queried and subscribed to.

For querying 2 methods are available, query and queryRaw, both taking a single argument specifying the (SPARQL) query. The queryRaw method directly returns the answer for the Context Source without interpreting it. The query method returns the answer from the Context Source as a Results; which is basically an ArrayList of type type Result. Each Result is a single answer to the SPARQL query, containing every variable Binding for that Result in its Dictionary.

The following example shows how to query a Context Source and process the results for simple display at the Console.

```
ContextSource cs = css[0];
Results results = cs.query(ulQuery);
Console.WriteLine("Got "+results.Count+" results");
for (int i = 0; i < results.Count; i++)
{
    Console.WriteLine(" result : " + i);
    Result res = (Result)results[i];
    Console.WriteLine(" # of bindings:" + res.Count);
    Binding[] bindings = res.getBindings();
    for (int j=0;j<bindings.Length;j++)
    {
        Binding b = bindings[j];
        Console.WriteLine(" " + b.getName() + " = (" + b.getType() + ")"+b.getValue());
    }
}
```

Please note that the query method assumes that the result of the query will be a SPARQL result set. So it is only suitable for SPARQL SELECT queries. For CONSTRUCT or DESCRIBE queries, the queryRaw method should be used!

For subscriptions a subscribe method is provided by the ContextSource class.

```
Subscription sub = cs.subscribe(ulQuery, "thisSubscriptionTag");
```

The client has to provide a delegate method with the following signature:

```
public void ContextUpdate(Subscription s, ContextEvent e)
```

Note that the method does not have to be called ContextUpdate per se, it only needs the specified arguments. The client itself has to add this method to the resulting Subscription object:

```
sub.ContextUpdate += this.ContextUpdate;
```

The ContextUpdate method will now be called with every update coming from the Context Source. The ContextEvent will contain the subscription tag ("thisSubscriptionTag" in the example) and, the raw result data in the raw property, and the possible Results in the results property.

The following code shows the ContextUpdate method taken from the ContextHelperTest example project in the ContextHelper.NET solution:

```
public void ContextUpdate(Subscription s, ContextEvent e)
{
    Console.WriteLine("ContextUpdate received! (tag="+e.tag+"");
    Results results = e.results;
    Console.WriteLine("Got " + results.Count + " results");
    for (int i = 0; i < results.Count; i++)
    {
        Console.WriteLine(" result : " + i);
        Result res = (Result)results[i];
        Console.WriteLine(" # of bindings:" + res.Count);
        Binding[] bindings = res.getBindings();
        for (int j = 0; j < bindings.Length; j++)
        {
            Binding b = bindings[j];
            Console.WriteLine(" " + b.getName() + " = (" + b.getType() + ") " + b.getValue());
        }
    }
    /* We also have the raw info from the CS */
    Console.WriteLine(e.raw);
}
```

It is the responsibility of the client to clean up a Subscription by unsubscribing it. This can be done by calling the unsubscribe method on the Subscription object:

```
sub.unsubscribe();
```

The following bit of code shows the combined steps the client takes for a subscription.

```
Subscription sub = cs.subscribe(ulQuery, "thisSubscriptionTag");  
sub.ContextUpdate += this.ContextUpdate;  
Console.WriteLine("Subscribed to CS; press ENTER to exit");  
Console.ReadLine();  
sub.unsubscribe();
```


5 Deploying and using T4.1 Context Sources

Although the CMS has been designed to be open enough for any body to write and integrate his/her own context source as described in part 3 of the tutorial, a collection of ready to use context sources have been developed within the T4.1. In this section we describe the steps for discovering and interacting with them. To install and deploy these context sources you could refer to the D4.5 document. In this tutorial we detail the way context consumers should formulate their needs to ensure that the context source is discovered, and explain how to interact with them once discovered.

5.1.1 Context History

The context history component contributes to the context management intelligent user service. Based on context histories, the context management service will provide data that helps AMIGO applications to get to know users over time and if possible do predictions based on patterns found. Reasoning based on context histories, as opposed to ontology-based reasoning, will especially take into account an enhanced understanding of the users' interactions over time and be embedded into and related to their other interactions happening in parallel, before, or afterwards (time-based reasoning).

5.1.1.1 Context needs formulation

Since the context history component stores different types of the historical context data, it should make a separate registration at the context broker for every type of context data it provides. Therefore, the context history description is very similar to the description of the capabilities of any other context source. The only difference is that it includes the formulation that the context history component takes into account the history of stored context data.

For example, in case the user location is stored in the context histories database, we would use the following description for the registration to the context broker:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  <ContextSourceRegistration>
    <timeliness>history</timeliness>
    <accuracy>0.8</accuracy>
    <contextType>UserLocation</contextType>
  </ContextSourceRegistration>
</rdf:RDF>
```

5.1.1.2 How to query/subscribe to context information

Since the context history component uses MySQL database for storing the historical context data, the most natural way of interaction is through the SQL queries.

5.1.2 Acoustic Position Estimator

The acoustic position estimation sensor (APE) is part of the acoustic scene analysis, which intends to collect all kinds of available information from acoustic signals. The APE sensor is based on microphones placed in one room and the retrieved position information is given in Cartesian coordinates relative to the room.

Acoustic position estimation must be seen as a building block for a location management system, offering position information of different accuracy's by integrating different sensing techniques. The accuracy of the APE sensor is approximately below 0.5m with an average adaptation time of 0.4s, depending on the reverberation of the room, the number of arrays and the amount of microphones per array. As a context source for the Amigo system, the APE sensor uses a Java based context wrapper to offer its contextual information to consumers. After registering at the APE wrapper the context consumers are asynchronously notified about context changes. The APE context source has a standard IContextSource interface with subscribe and unsubscribe methods.

5.1.2.1 Context needs formulation

The RDF description of the context source is as follows:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://amigo.gforge.inria.fr/owl/ContextTransport.owl">
  <ContextSourceRegistration>
    <timeliness>current</timeliness>
    <contextType>
      AcousticPositionEstimation
    </contextType>
  </ContextSourceRegistration>
</rdf:RDF>
```

5.1.2.2 How to query/subscribe to context information

The APE context source collects all gained information in RDF descriptions based on the amigo ontology. Here an example for the detection of the user "Roberto" in the room "kitchen" at the Cartesian coordinates (x,y)=(3.4,1.2) at timestamp "2006-08-30T09:17:29.452+0200" is given.

RDF – metadata example

This is the RDF representation of the metadata according to the amigo ontology.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:j.0="http://amigo.gforge.inria.fr/owl/AmigoIACS.owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2000/01/rdf-schema#
  http://www.w3.org/2000/01/rdf-schema#
  http://www.w3.org/2002/07/owl#
  http://amigo.gforge.inria.fr/owl/AmigoIACS.owl#
  http://www.daml.org/2001/03/daml+oil#"/>
```

```

xmlns:j.1="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#">
<j.1:AcousticPositionEstimation>
  <j.1:timestamp>2006-08-30T09:17:29.452+0200</j.1:timestamp>
<j.1:estimatedPosition>
  <j.1:Relative2DLocation>
    <j.1:relativeToSpace>
      <j.0:Room>
        <j.1:identifier>Kitchen</j.1:identifier>
      </j.0:Room>
    </j.1:relativeToSpace>
    <j.1:Y>3.4</j.1:Y>
    <j.1:X>1.2</j.1:X>
  </j.1:Relative2DLocation>
</j.1:estimatedPosition>
<j.1:isPositionOf>
  <j.0:User>
    <j.1:identifier>Roberto</j.1:identifier>
  </j.0:User>
</j.1:isPositionOf>
</j.1:AcousticPositionEstimation>
</rdf:RDF>

```

SPARQL – Query

The context consumer is interested in position information from the kitchen, so he subscribes to the context source “AcousticPositionEstimation”. The SPARQL query can be expressed in plain words with: “Send me information about the x and y position and the name of a user, if someone is detected in the kitchen”. Every time the context source has information which matches this query, the context consumer will get a notification with the results of the query.

Example Query

```

PREFIX j.1: <http://amigo.gforge.inria.fr/owl/ContextTransport.owl#>
SELECT ?User ?XPos ?YPos
WHERE {
  ?APE j.1:isPositionOf ?US .
    ?US j.1:identifier ?User .
    ?APE j.1:estimatedPosition ?RelLoc .
    ?RelLoc j.1:relativeToSpace ?ROOM .
    ?RelLoc j.1:X ?XPos .
    ?RelLoc j.1:Y ?YPos .
    ?ROOM j.1:identifier "Kitchen" .
}

```

SPARQL response

The SPARQL response to the above query is:

```

<?xml version="1.0"?>
<sparql
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xs="http://www.w3.org/2001/XMLSchema#"
  xmlns="http://www.w3.org/2005/sparql-results#" >
<head>
  <variable name="User"/>
  <variable name="XPos"/>
  <variable name="YPos"/>
</head>
<results ordered="false" distinct="false">
  <result>
    <binding name="User">
      <literal>Roberto</literal>
    </binding>
    <binding name="XPos">

```

```

    <literal>1.2</literal>
  </binding>
  <binding name="YPos">
    <literal>3.4</literal>
  </binding>
</result>
</results>
</sparql>

```

We get the information that Roberto is in the kitchen. The room kitchen is not explicitly mentioned in the answer, as we only get notifications if someone is in the kitchen. Additionally the x and y coordinates of Roberto are contained in the message.

5.1.3 RF Positioning

Provides location information for persons and object that are linked to active tags, based on signal strength information from several active RF readers.

5.1.3.1 Context needs formulation

Each device has a specific RDF file, but all these RDF files are based on the following generic pattern:

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns:AmigoDevices="http://amigo.gforge.inria.fr/owl/Devices.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns="http://amigo.gforge.inria.fr/owl/Domotics.owl#" >
  <rdf:Description rdf:nodeID="A0">
    <rdf:type
rdf:resource="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#ContextSourceRegistration"/>
    <timeliness>current</timeliness>
    <accuracy>0.8</accuracy>
    <contextType>appliance1</contextType>
  </rdf:Description>
</rdf:RDF>

```

RF Positioning service supports the following context parameters:

- UserLocation (e.g. related to triples (User, isLocatedIn, Room), (User, isLocatedIn, Building), ...)
- ObjectLocation (e.g. (Object isLocatedIn Room))
- RoomContent (any tagged person or object in a room)
- RFReading (raw data (Sensor, hasScanned „Tag) with certain value)

Context parameters can be found in (or will be added to)
<http://amigo.inria.fr/owl/ContextParameter.owl>

5.1.3.2 How to query/subscribe to context information

In order to get the Object Location or User Location, a SPARQL query must be sent. An example to get any location of any person is shown below:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX amigo: <http://amigo.gforge.inria.fr/owl/amigo#>
SELECT ?personid ?roomid
WHERE { ?context amigo:isLocatedIn ?room .
        ?context amigo:isLocationOf ?person .
        ?person rdf:type amigo:User .
        ?room rdf:type amigo:Room .
        ?room amigo:identifier ?roomid .
        ?person amigo:identifier ?personid }
```

The subscription needs the same SPARQL query (described above) and returns a unique identifier (Guid) that is used to unsubscribe or event receiving.

The subscription mechanism is actually developed using WSDDiscovery, where the client must publish an object of type ContextClient defined in ContextClient.cs. The CS will call this interface wherever the subscribed context data changes. The CS sends a SPARQL result to the subscribed context client.

5.1.4 Topic Recognition

The topic recognizer is a fully functional Amigo OSGI service. However, RDF and SPARQL support is not yet available: this shall be done very soon.

5.1.4.1 Context needs formulation

Registration to the context broker is not yet implemented, as just explained.

5.1.4.2 How to query/subscribe to context information

For the time being the Topic Recognizer can be accessed in JAVA via the exported CanRecognizeTopic interface. The calling class shall implement the TopicInterpreter interface.

Its usage is then extremely simple: whenever a new topic is detected, the topic recognizer calls the TopicInterpreter.newTopic (Topic) method.

The topic recognizer is shipped with a default graphical application (class LiveApp), which uses it to show different slides depending on the recognized topics.

The topic recognizer makes use of a definition file with all the possible topics, along with their keywords. This file is a text file, by default

/src/default.topics

You can edit this file for your own application.

The topic recognizer should receive its input (sentences) either from an OSGI-compliant web service amigo speech recognizer, or from any JAVA class that implements the TextStream interface. It is shipped with a default keyword recognizer, that is available under GForge under the sub-repository named "implicitSpeech". This keyword recognizer is implemented with a default small vocabulary, grammar and speaker-dependent acoustic models. If you want to

use it for your own application, you'll probably have to edit the vocabulary and grammar files, and adapt the acoustic models to your voice and microphone. All the required tools and scripts to achieve this are given, but this process is not fully automated, because of its inherent complexity (and because it is only a default fallback solution). Please contact cerisara@loria.fr in case you would like to adapt this keyword recognizer to your needs.

5.1.5 Domotic Sensors

Domotic sensors provide information from domotic appliances and sensors. A wrapper (Gateway) sends/receives all the domotic commands from physical devices using different communication interfaces (RS-232, Bluetooth,...). A specific Context Source is generated for each detected device.

5.1.5.1 Context needs formulation

Each device has a specific RDF file, but all these RDF files are based on the following generic pattern:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:AmigoDevices="http://amigo.gforge.inria.fr/owl/Devices.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns="http://amigo.gforge.inria.fr/owl/Domotics.owl#" >
  <rdf:Description rdf:nodeID="A0">
    <rdf:type rdf:resource="http://amigo.gforge.inria.fr/owl/Domotics.owl#ContextSourceRegistration"/>
    <timeliness>current</timeliness>
    <accuracy>0.8</accuracy>
    <contextType>appliance1</contextType>
  </rdf:Description>
</rdf:RDF>
```

Each sensor or appliance will add its features into the generic RDF, generating the own one, and updates the contextType property with the appliance or sensor code. The following example shows the RFD description file of a Fridge (with code 3), offering State info, Content info and inner temperature info as Context information.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:AmigoDevices="http://amigo.gforge.inria.fr/owl/Devices.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns="http://amigo.gforge.inria.fr/owl/Domotics.owl#" >
  <rdf:Description rdf:nodeID="A0">
```

```

<rdf:type rdf:resource="http://amigo.gforge.inria.fr/owl/Domotics.owl#ContextSourceRegistration"/>
<timeliness>current</timeliness>
<accuracy>0.8</accuracy>
<contextType>appliance3</contextType>
<hasState>ON</hasState>
<hasContent></hasContent>
<hasTemp>17</hasTemp>
</rdf:Description>
</rdf:RDF>

```

5.1.5.2 How to query/subscribe to context information

In order to get the context info of a concrete CS, an SPARQL query must be sent. An example to get any context information of the Fridge, where **strQueriedContextData** is the wished feature:

```

string sparqlQueryState = "PREFIX ex: <http://amigo.gforge.inria.fr/owl/Domotics.owl#>\n"
    + "SELECT ?" + strQueriedContextData + "\n"
    + " WHERE { ?kk ex:" + strQueriedContextData + " ?"
    + strQueriedContextData + " }";

```

The query will return the SPARQL result.

The subscription needs the same SPARQL query (described above) and returns a unique identifier (Guid) that is used to unsubscribe or event receiving.

The subscription mechanism is actually developed using WSDiscovery, where the client must publish an object of type ContextClient defined in ContextClient.cs. The CS will call this interface wherever the subscribed context data changes. The CS sends a SPARQL result to the subscribed context client.

5.2 Debugging with eclipse

You may use eclipse to debug your bundle. In brief, to do that, you have to launch oscar from within eclipse, and attach the source files to your bundle. Then you can put breakpoints, watch variables and so on.

5.3 Troubleshooting

If you encountered problems running the context broker:

1- Oscar Shell Gui did not show.

Maybe Oscar is not configured to show a shell gui. Edit the lib/system.properties file. Fill in the oscar.auto.start.1 property as follows :

```

oscar.auto.start.1=file:bundle/shell.jar file:bundle/tablelayout.jar \
    file:bundle/shellgui.jar file:bundle/shellplugin.jar \
    file:bundle/bundlerepository.jar file:bundle/shelltui.jar

```

2- The shell gui shows, but when I click on OBR nothing happens.

Have a look to the console. If a « timeout » exception is displayed, your proxy parameters must be wrong.

3- When I click on OBR I do not see the Amigo bundles.

Check the contents of oscar.repository.url in lib/bundle.properties

6 Appendix

6.1 Description of context capabilities

This document is intended to give an example of how a concept of ContextParameter can be used to specify context by linking different entities. The examples here do not use the full Amigo ontology; since they are intended to be illustrative of the core assumptions for specifying context information.

It is the intention that this document will evolve, over the course of a number of iterations, into a normative reference on how to specify context produced by Context Sources within the Amigo framework and how to use the Amigo ontology for that purpose.

6.1.1 Identification of entities

How is a user (or any other entity) identified in context? Two possibilities exist:

- using some URI:
`<user rdf:id="http://server.domain.co/...#someid">`
...
- using properties:
`<user>`
 `<identifier>someone@somedomain</identifier>`
 `<identifier>someone@someotherdomain</identifier>`
...

The first method has the advantage that all properties assigned to a user with the specified id will be merged “automatically”, since all properties relate to the same URI with which the user is identified, it does however assume an agreed global naming scheme. Automatic merging can be a disadvantage in cases where it is specifically unwanted, since it cannot be ‘disabled’; it is a result of assigning a meaning to the URI of an element.

The second case has the advantage that multiple instances pertaining to the same user can be created, which is useful for e.g. properties that change over time or contain conflicting information. It also enables a user to have multiple identities.

Since there are cases where automatic merging is unwanted (like history tracking etc) we propose to use the second approach. This means that context messages **MUST NOT** contain URIs for rdf:IDs; they should be anonymous nodes with respect to RDF. It also means that any entity which has to play a part in a context (as context or as subject) must have an Identifier property, this property can contain URIs.

6.1.2 Modeling of context

6.1.2.1 Usages of the context ontology

- Modeling of the world
 - Last known state
 - State including history
- Distributing context

A world model (and even context messages) may contain history, or conflicting information. In order to deal with this, meta-data should be available for context; such as timestamp or Quality of Context. A direct property relationship between entities does not allow meta-data to be

attached to it; which implies that in order to be able to assign meta-data to context all relations between entities should be indirect.

So a property 'isLocatedIn' directly between a User and a Space is not possible: there should be some indirection so a model can for example contain a history of locations for a user, or conflicting statements (for example: a user is at Space A with 80% probability, and in Space B with 15% probability).

6.1.3 ContextParameter (Generic)

The ContextParameter should be that indirect link between entities. In the general case a piece of context information may look like this:

```
<ContextParameter>
  <timestamp>2006-05-04T13:39:20</timestamp>
  <probability>0.2</probability>
  <isContextOf>
    <entity>
      <identifier>...</identifier>
    </entity>
  </isContextOf>
  <isContextOf> ...
  ...
</ContextParameter>
```

With the following assumptions:

- A ContextParameter MAY have a unique ID, which can be used for tracing purposes.
- It is open for discussion whether Quality of Context (QoC) properties refer directly to the ContextParameter, or that there should be some QoC RDF resource for that. Either approach is valid.
- isContextOf is the most generic relationship between a ContextParameter and an Entity and does not imply a 'direction' or a meaning. If the generic example would be used to specify that a user is in a certain room then there would be two 'isContextOf' properties: one for the room, and one for the user. To add meaning to the relation ContextParameter can be sub classed. Subclasses may add more meaningful relationships that should be sub-properties of isContextOf. These sub-properties for those subclasses can then also be restricted to the Range and Domain that are meaningful for that relation (see the following section for an example).

Sub classing ContextParameter also aids in specifying what type of context information a Context Source provides, since the Context Source can state the specific sub classes of ContextParameter it supports.

6.1.4 ContextParameter (Specific)

The (incomplete) hierarchy of ContextParameters is shown in the figure below:

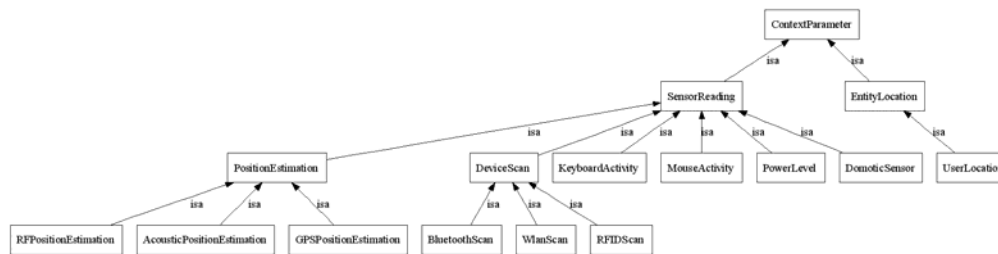


Figure 6-1: ContextParameter hierarchy.

This hierarchy can be extended further as more context sources of different types become available. In the following sections some examples will be given for the different types of ContextParameters. Other types not mentioned here use the same principles.

6.1.4.1 UserLocation

A specific example of a sub class of ContextParameter instance can look like this:

```

<UserLocation>
  <timestamp>2006-05-04T13:39:20</timestamp>
  <probability>0.8</probability>
  <isLocatedIn>
    <Room>
      <identifier>B3.08B@telin.nl</identifier>
    </Room>
  </isLocatedIn>
  <isLocationOf>
    <User>
      <identifier>remco.poortinga@telin.nl</identifier>
    </User>
  </isLocationOf>
</UserLocation>
  
```

- Here UserLocation is a subclass of ContextParameter.
- isLocatedIn and isLocationOf are sub-properties of isContextOf.
- isLocatedIn is restricted in range to Room (although realistically it should be restricted to a more generic location entity, like 'Place').

6.1.4.2 BluetoothScan

A BluetoothScan is a specific type of DeviceScan; which in itself is a subclass of a SensorReading. It specifies the AmigoDevices:Bluetooth that performed the scan as well as the AmigoDevices:Bluetooth entities that were seen by that Bluetooth device. For every AmigoDevices:Bluetooth a name and/or address is specified.

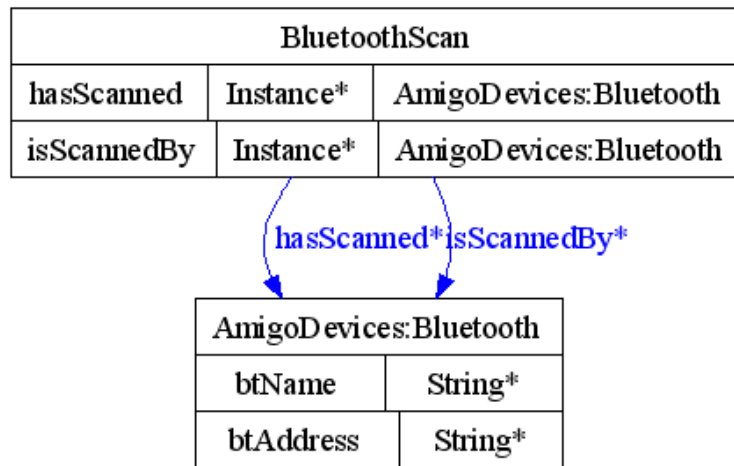


Figure 6-2: BluetoothScan ContextParameter.

6.1.4.3 PositionEstimation

The PositionEstimation is a generic class that is sub classed to more specific types of position estimations, such as AcousticPositionEstimation.

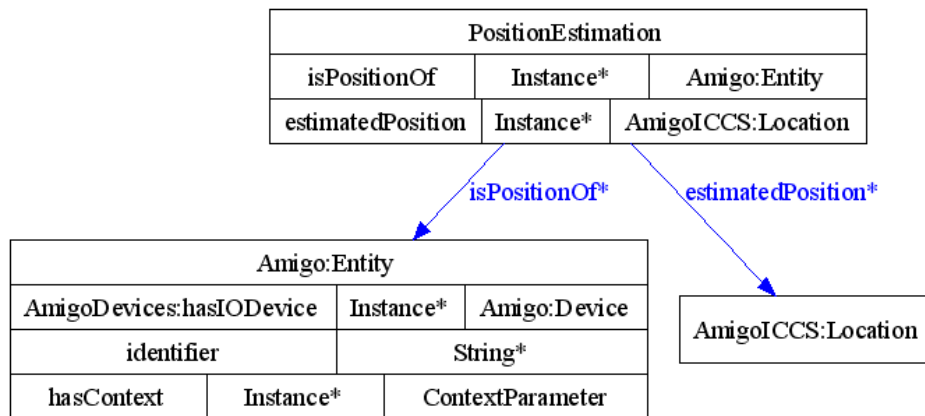


Figure 6-3: PositionEstimation.

Every PositionEstimation can link to an entity for which an estimation is specified as well as a Location that is the estimation itself. An example of an AcousticPositionEstimation is given below.

6.1.4.4 AcousticPositionEstimation

```
<AcousticPositionEstimation>
  <probability>0.9</probability>
  <timestamp>2006-07-03T12:30:25</timestamp>
```

```
<estimatedPosition>
  <Relative2DLocation>
    <Y>1.2</Y>
    <X>3.5</X>
  </Relative2DLocation>
</estimatedPosition>
</AcousticPositionEstimation>
```

6.1.5 Summary

- At least for transmission of context an indirect link between entities should be used in the form of a (sub-class of) ContextParameter.
- ContextParameter is made more specific by sub-classing and using sub-properties of isContextOf.
- Meta-data such as timestamp or Quality of Context properties can be assigned to ContextParameter instances, allowing changes over time and supporting conflicting context information.
- To avoid 'automatic merging' every type of entity to be used in a ContextParameter (or sub-classes thereof) must have an Identifier property with which it can be addressed.
- Any type potentially used in a ContextParameter (sub-property of) isContextOf property must be a (sub-class of) an entity.

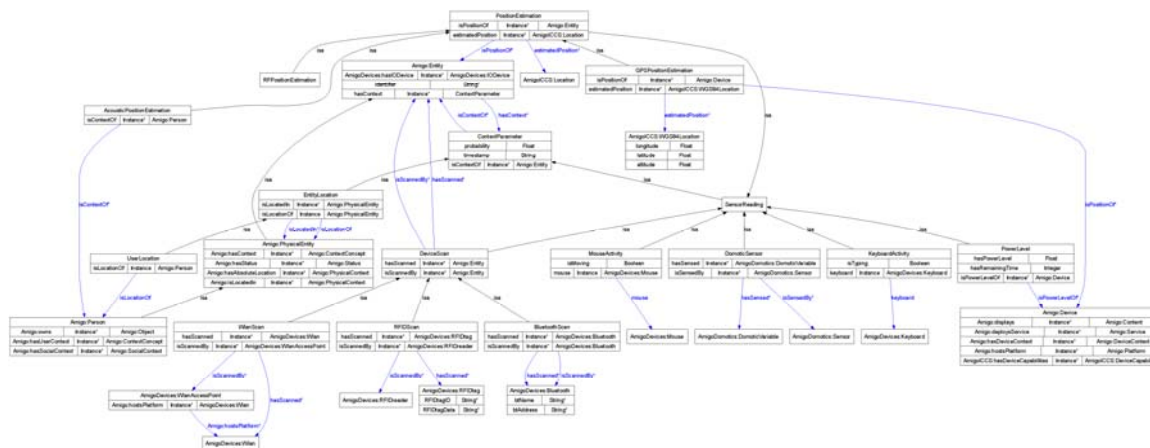


Figure 6-4: A complete overview of Context Parameters and their relations.

6.2 Context Interpreter data and syntax of function calls

Figure 1 shows, as an example, relations of the classes and properties used in the reasoning of the social context. The figure describes ontology which captures the semantic information provided by the other Amigo context sources.

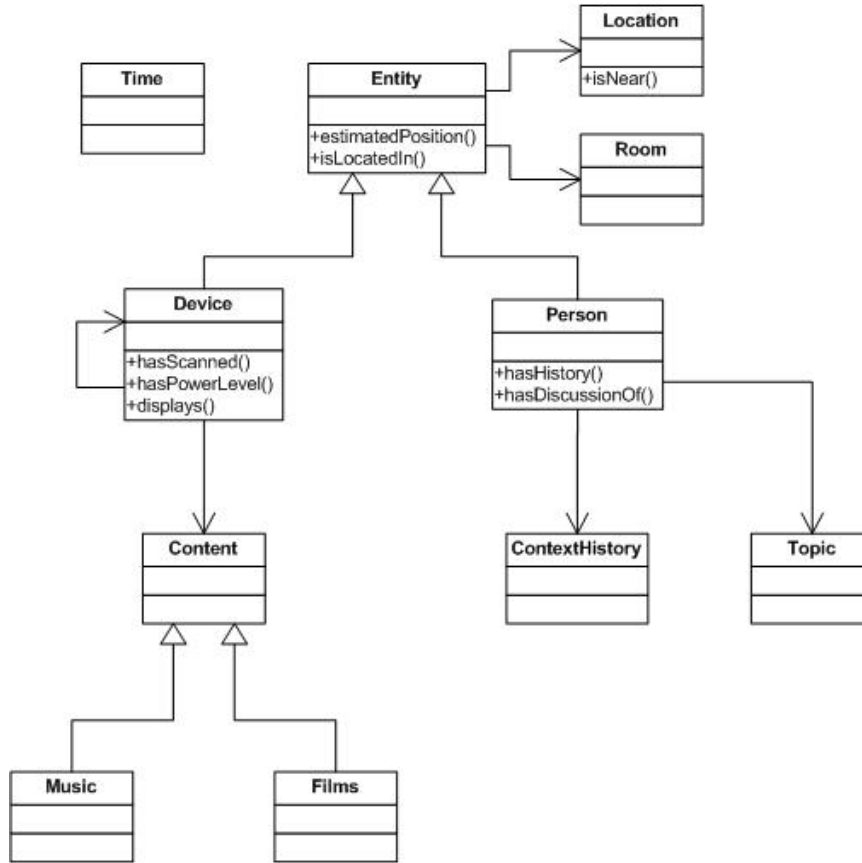


Figure 1. Partial definition of the ontology for social context reasoning.

The reasoning module of the Context Interpreter uses context information provided by context sources to create corresponding high-level context information. This high-level information is sent back to the applications when the rule, created by the application, holds true (see Figure 2), so that applications can act upon reasoned information.

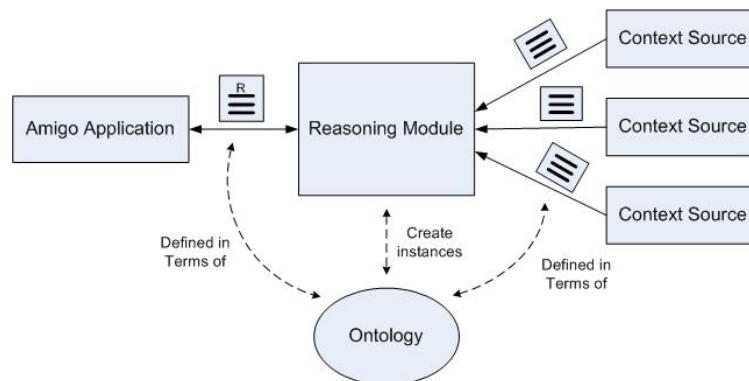


Figure 2. Reasoning high-level contexts.

Context Interpreter subscribes to Context Sources, and uses such context data for making high-level conclusions about context. Context Interpreter differs from traditional rule engines because rule engines use each fact only once in firing rules and doing actions (otherwise they would do same action endlessly), while CI can use same fact in reasoning many times (until corresponding Context Source will send new context information).

CI considers each context as a set of type (context descriptor) – value (corresponding value) pairs, where each set contains all relevant information about one fact. For example, the fact that Jerry is in the kitchen is described by CI as a set of context descriptors:

```
{personid: Jerry@JGAV3006.amigo.net, roomid: Kitchen@JGAV3006.amigo.net, TimeStamp: 2007/12/4 16:00}
```

Context types here are taken from Amigo ontologies because Context Sources provide information in such terms, e.g., the set of context descriptors described above is derived from location CS query/ subscribe result

```
<result>
  <binding name="personid">
    <literal>Jerry@JGAV3006.amigo.net</literal>
  </binding>
  <binding name="roomid">
    <literal>Kitchen@JGAV3006.amigo.net</literal>
  </binding>
</result>
```

However, CI can use also any other application-specific context types (e.g., high-level descriptions “user attitude towards movie: likes very much”), as long as these application-specific types do not overlap with a few predefined terms, described below. (One example of such predefined terms is “Time” term: all time-related descriptors should contain “Time” substring, while other descriptors should not).

Context sets can be of any size, but they can contain each context descriptor only once. For example, the fact that both Jerry and Maria are in the kitchen is represented by the following set of context descriptors:

```
{personid: Jerry@JGAV3006.amigo.net; Maria@JGAV3006.amigo.net, roomid: Kitchen@JGAV3006.amigo.net, TimeStamp: 2007/12/4 16:30}
```

None of context descriptors is obligatory to use, although normally each fact has its source and timestamp (time when this fact was created). Other time-related information can be represented by other context descriptors, containing “Time” substring. For example, the fact that Jerry has 30 minutes dentist appointment tomorrow morning in Medivire clinic can be represented by the following set of context descriptors:

```
{personid: Jerry@JGAV3006.amigo.net, Appointment: dentist,
AppointmentStartTime:2007/12/3 10:00, AppointmentEndTime: 2007/12/3 10:30,
AppointmentLocation: Medivire Clinic, AppointmentAddress: Kaitovaula 1}
```

6.2.1 CI function input/ output format

Format of strings is XML. For example, for a rule described above is “*IF (**personid:**, **roomid:** **bedroom** AND **lightLevel: low**, AND **soundLevel: low**) THEN **activity: sleeping***”

parameter *string [] leftHandSide* is:

```
leftHandSide [0] = "<typesOnly>
    <type>personid</type>
</typesOnly>
    <contextSource name='Positioning'>
        <roomid>bedroom</roomid>
    </contextSource>";

leftHandSide [1] = "<contextSource name='LightSensor'>
    <lightLevel>Low</lightLevel>
</contextSource>";

leftHandSide [2] = "<contextSource name='AudioSensor'>
    <soundLevel>Low</soundLevel>
</contextSource>";
```

And the result can be:

```
<queryResults>
    <personid>Jerry@JGAV3006.amigo.net</personid>
    <Activity>Sleeping</Activity>
</queryResults>
```

6.2.2 List of predefined terms and symbols:

Time

AND

NotAND

OR

& (reserved for parsing function parameters)

⌘ (reserved for parsing function parameters)

TypeOnly

6.3 Description of tools/languages provided by the CMS

6.3.1 Test tool for testing newly developed context sources and SPARQL queries

For testing SPARQL queries against existing Context Sources (when creating context clients), or for testing new Context Sources, the Context Source Tester tool is provided; it is available from OBR.

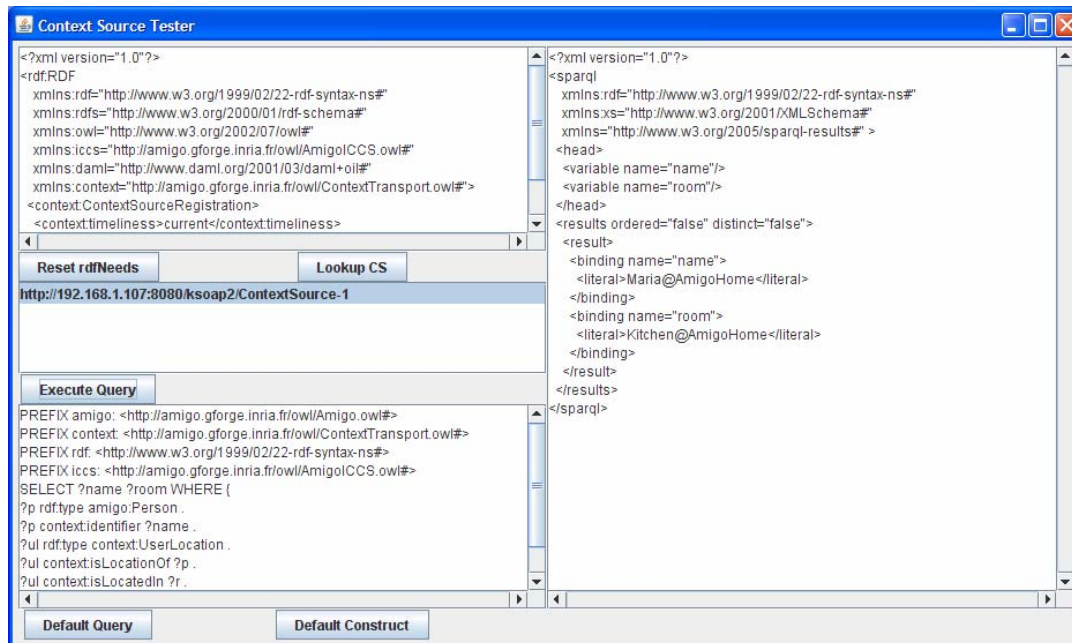


Figure 6-5: Context Source Tester tool: screenshot

The default rdfNeeds and SPARQL examples, which are present when the Context Source Tester is started up, work with the Context Source Push Example also available from OBR, so that new SPARQL queries can be tested based on a working starting point.

6.3.2 Test tool for the RF positioning monitor

This tool can be found in

[amigo]ius/cntxt_mgmt/rf_positioning/test/trunk

Its use is quite simple, in the GUI you can

- look for a context source either by name or by address,
- load a sparql query (from a file)
- send the query to the context source (ie invoke the query function)
- watch the result

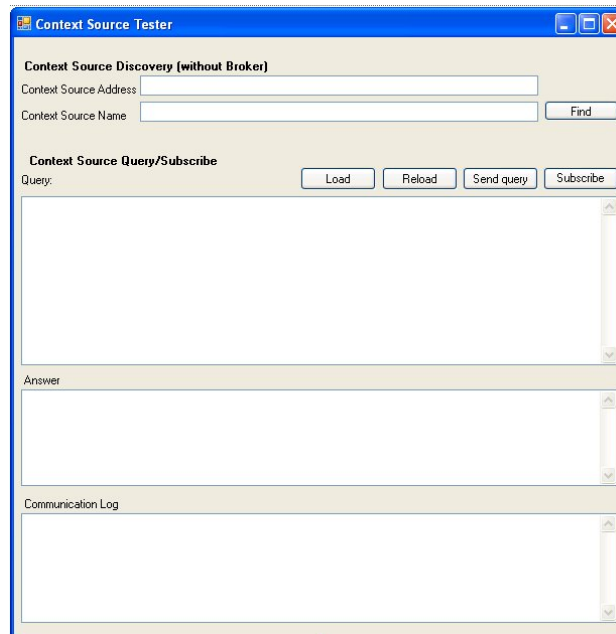


Figure 6-6: RF Positioning (Context Client) Test Tool: Screenshot

Figure 6-6 Shows a screenshot of the Application, A user can contact a context source directly by typing its address in textbox with label 'Context Source Address' or by typing its name in the textbox 'Context Source Name' and pressing the 'Find' button. In the second case, the tool will look for 5 seconds and update the textbox 'Context Source Address' when a context source with this name is found (first occurrence). After this the User can load a sparql query into the textbox labeled 'Query' by using the button 'Load' an example query is provided in the file sparqlexample.txt. After the query has been loaded it can be edited, 'Reload' button will revert the box back to the original situation (save functionality was not added on purpose). After the query is as needed, it can be send to the Context Source by pressing the button 'Send query', the answer will be shown in the textbox labeled 'Answer'.

Deleted: Figure 6-6

Note that Location service sends back exception information in the situation where the query was found to be illegal.

6.3.3 Configuration tool for the RF Position monitor

It is possible to configure the positioning tool via (parts of) the Amigo ontology. , though the provided tool is not yet complete, a description is provided below on how the configuration is currently handled.

Spaces

Spaces are parsed from a file with the name spaces.xml, or directly from the amigo ontology files. Since it is the intention that these files are compatible to the Amigo OWL ontology, we describe both the current version (based on straightforward xml and the future version).

The current structure is as shown below

```
<SpacesInformation>
  <Space>
    <SpaceName>Kitchen</SpaceName>
```

```

    <SpaceID>Kitchen@JGAV300600.amigo.net</SpaceID>
    <Precision>4</Precision>
    <Type>Room</Type>
    <isLocatedIn>TheHome</isLocatedIn>
  </Space>
  <Space>
    <SpaceName>TheHome</SpaceName>
    <SpaceID>Home@JGAV300600.amigo.net</SpaceID>
    <Precision>3</Precision>
    <Type>Building</Type>
    <isLocatedIn>TheTown</isLocatedIn>
  </Space>
  ... etc
</SpacesInformation>

```

In the ontology, this is represented as:

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:amigo="http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#"
  xmlns:contexttransport="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#" >

  <amigo:Room rdf:about="http://amigo.gforge.inria.fr/owl/RFLocation#Kitchen">
    <contexttransport:identifier
rdf:datatype="xsd:string">Kitchen@JGAV300600.amigo.net</contexttransport:identifier>
    <amigo:contextPrecision
rdf:datatype="xsd:integer">4</amigo:contextPrecision>

  <amigo:isLocatedIn rdf:resource="#TheHome" />
    </amigo:Room>
    <amigo:Building rdf:about="http://amigo.gforge.inria.fr/owl/RFLocation#TheHome">
    <contexttransport:identifier
rdf:datatype="xsd:string">TheHome@JGAV3006.amigo.net</contexttransport:identifier>
    </amigo:Building>

  etc...

```

Note that 'contextPrecision' has not yet been added to the amigo *Space* ontology, but this value is used in the 'precision' fields of the *ContextParameter* ontology, which does exist already (see the section on PrivacyLevels).

Tags (tagged Persons and or objects)

Tags are assigned to persons and objects by parsing from a file with the name tagdata.xml, or directly from the amigo ontology files. Since it is the intention that these files are compatible to

the Amigo OWL ontology, we describe both the current version (based on straightforward xml) and the future version).

```
<TagDatabase>
<Tag>
  <TagID>00443A</TagID>
  <UserID>Maria@JGAV300600.amigo.net</UserID>
  <UserName>Maria</UserName>
  <Interval>5000</Interval>
  <Offset>0</Offset>
  <History>0</History>
  <Types>
    <Type>User</Type>
    <Type>Person</Type>
  </Types>
</Tag>
</TagDatabase>
```

In the amigo ontology, the tag data can be assigned by adding the tagID as parameter to the corresponding objects/persons. This can then be stored (for example in the UMPS service). At this moment the TagData is local to the RF location source only and the Tag Database is used to create contextdata with types corresponding to the ones mentioned in tagdata file. The fields have the following meaning:

TagID: Hexadecimal value that is the Tag ID (ie the ID reported by the Sensite hardware)

UserID: (or ObjectID): the field that is used in the identifier of amigo concepts

UserName: the field that is used as reference name (and in logging)

Interval: The interval period of the Tag (ie Tag sends signal every xx microseconds)

Offset: value to be added to measured data (ie to assure that tags give similar readings)

History: Set the buffersize of the filter: filter will look at the last xx values measured for this Tag and perform a median filter

Types: List of the types (from the amigo ontology) that should be reported (Note that these types should exist in the Amigo .owl files)

Implementing Precision for Perceived Privacy task

The perceived privacy task of the Amigo project has defined the concept of Context Precision and its use to enforce privacy policies. For the RF Positioning this has resulted in the corresponding precision field in the spaces ontology. This precision Field is used when the Positioning context source creates its ContextParameter (UserLocation or ObjectLocation), the context parameter is assigned the precision that is connected to the space in the spaces configuration file

ReaderLocation

The readerlocation file contains the actual mapping of RFID (signal strength) readings onto spaces. For each reader, this file indicates which values can be expected in which room. In

order to facilitate filling in of these values a second tool has been created called HBLConfigurator. This tool can show the typical reader values (assuming tag is in a certain room) and determines the boundaries. The values provided can be copied into the readerlocation file and changed while the Positioning service is running.

Configuration info

The Configuration tool uses the Configuration context source (see 1.9) to obtain data about the Users in the home and the Spaces (e.g. Rooms) in the home. It then uses this information when creating the taginfo and spacesinfo files. As mentioned before, In the future the user is allowed to edit his tag info in his UMPS profile and RF Tool will update the taginfo file accordingly (only the tagID field of the User with corresponding identifier).

6.4 Domotic / Sensor information Demonstrator

The general purpose of this demo is to show how household appliances and sensors can interact with amigo middleware as Context Source devices.

To do so, a base application (Gateway) has been developed. This is the responsible to process all the incoming messages from the physical devices and sensors and create a wrapper with CS capabilities (a CS for each detected device or sensor). These specific Context Sources will offer appliance/sensor related context information.

6.4.1 Context Sources

The gateway application will receive all the messages from the physical devices, and will create a CS that will implement the Query, Subscribe and Unsubscribe methods. It will also register each CS on the Broker.

The gateway is designed using a dynamic connector infrastructure, in order to communicate with any device or sensor using any available technology (Power Line, Bluetooth,...). These connectors are configured in a file that is loaded when application starts.

The Context Clients will locate the desired CS asking the broker about the capabilities that are required. To do so, each device/sensor, will be described with a RDF file.

6.4.2 Context Capabilities

As mentioned above, each CS will be described using a RDF file. The following file is the description of a Washing Machine (*Appliance1*) that offers one context info capability (*hastate*):

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:AmigoDevices="http://amigo.gforge.inria.fr/owl/Devices.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns="http://amigo.gforge.inria.fr/owl/Domotics.owl#" >
  <rdf:Description rdf:nodeID="A0">
    <rdf:type rdf:resource="http://amigo.gforge.inria.fr/owl/Domotics.owl#ContextSourceRegistration"/>
    <timeliness>current</timeliness>
```

```

    <accuracy>0.8</accuracy>
    <contextType>appliance1</contextType>
    <hasState>ON</hasState>
  </rdf:Description>
</rdf:RDF>

```

This description will be used to find the required CS in the broker.

6.4.3 Context Queries and Events

The queries and events will be expressed using SPARQL sentences. An example of a SPARQL to get context info (*hasState*) will be the next one:

```

SPARQL = "PREFIX ex: <http://amigo.gforge.inria.fr/owl/Domotics.owl#>\n"
+ "SELECT ?hasState"
+ " WHERE { ?kk ex:hasState ?hasState }";

```

For the previous description file the returning info would be:

```

binding name hasState
literal ON literal
binding

```

All the queries and events will have this result format. The Query method of a CS will take the SPARQL sentence as input parameter and will return the desired context info.

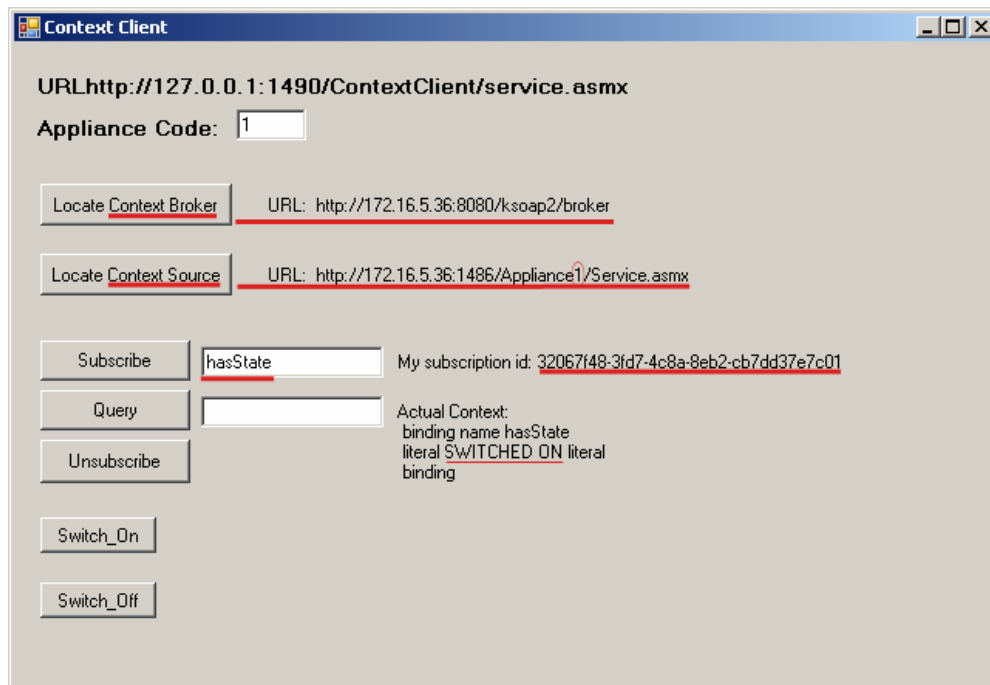
If a Context Client is Subscribed to a Context Info, a SPARQL result will be sent to the CC each time the info changes its value.

6.4.4 Domotic execution process

The Gateway will receive all the messages from the devices. When a new device is detected, a new CS will be created and registered in the CB. This CS will update its context info for each event received by the gateway from the device/sensor.

A context client will discover the CS in the broker specifying the capabilities that the CS must have. The broker will return the URL.

The CC now can subscribe or query to the CS in order to obtain the current context info and subscribe to future events.



6.5 Topic recognizer

6.5.1 Quick guide to test the topic recognizer

The topic recognizer is an Amigo service. It is developed as an OSGI service and it is a CMS compliant context source. If you just downloaded the *topic_recognition.jar* and simply want to test it, you can follow the steps:

1. Requirements: you need a Windows XP computer with a microphone, OSCAR and the amigo/CMS bundles
2. Start OSGI with the amigo configuration, including the CMS bundles.
3. Extract from *topic_recognition.jar* the zip file *microphone.zip*
4. Unzip *microphone.zip* into an empty directory: run the *microphone.bat* script: it shall wait for the bundle.
5. Install and start the topic recognition module within OSGI: by default in this version, it launches a demonstration application that pops up a GUI
6. The bundle shall connect to the *microphone* script: wait about 10 seconds for the script to complete initialization
7. Press the "push-to-talk" button on the GUI (let it on until the end of your test): you can now talk, the system shall recognize some keywords of the application (see examples next), infer the corresponding topics, and display some slides that correspond to the currently recognized topic of your talk.

6.5.1.1 Examples of topics from the default application

The default application supports 4 topics:

Weather / Sports / Amigo / Computer

A few keywords are supported. Just try out some of them:

Weather, cloudy, temperature, baseball, basketball, football, amigo, ambient_intelligence, computer, science, engineering, studies

When the speech recognizer recognizes a keyword, it prints it on the text line just below the VU-meter. Otherwise, it just prints a sequence of dots. You can also check the “winning topic” at any time by looking at the topic's probabilities that are shown on the left.

At start-up, the initial slide simply shows a diagram of the architecture of the topic recognizer. Whenever a new winning topic has been detected, this slide should change and rather show one of the four slides saved with the application.

Troubleshooting:

- You should first check the input level of the microphone: on the top of the GUI, there is a VU-meter that displays the energy of the speech signal recorded: the first slider just below controls the display scaling factor (you can ignore it). **But the second slider is important:** it controls the sensitivity of the voice activity detector. You should tune it so that when you don't talk, the colour of the VU-meter is blue, and when you talk, the colour must become red, and stay red as long as you are talking !
- The connection between the GUI and the *microphone* script is realized via some socket: check that your firewall authorizes this connection.
- The speech recognizer may require the *cygwin1.dll* file on the path to run: just download it from www.cygwin.org if required.
- **It is very likely that the keyword spotting accuracy will be quite low with your voice and your microphone.** This is because the current module is shipped with **native English speaker** models trained on the Wall Street Journal: if your microphone is not good enough, or if your English pronunciation is not so standard, then the recognition results might be very bad. You definitely should train and adapt acoustic models to your voice: see section 4 for that. Another option is to wait for next upgrades of the topic recognition module (see section 5).

6.5.2 How to adapt the system to your own topics and needs

The list of topics and the keywords associated with each of them are defined in the file *src/default.topics* in the jar file. The format is the following:

```
-- topic
keyword1
keyword2
...
--++
initial probas of every topic
transition probabilities out of topic1
transition probabilities out of topic2
...
```

Once you have updated this file to your needs, you need to check that every keyword you have defined is associated with a phonetic transcription in the file *dico* that was zipped in *microphone.zip*. If it is not the case (a free 5000-words version of the Wall Street Journal

lexicon is shipped with the topic recognizer), you need to create it, taking similar words as example. Finally, you need to replace the previous list of keywords in the file *gram* that was zipped in *microphone.zip* with your personal ones. Don't forget to also indicate at the beginning of the *gram* file the new number of keywords.

In the next versions of the software, this adaptation procedure should be greatly simplified, and made partly automatic.

6.5.3 Guide to create an Amigo service that uses the topic recognizer

6.5.3.1 Place of topic in the ontology

The main concept in the ontology is the **UserTopicActivity**. This concept has several properties:

- **isTopicOf : Person**
- **hasTopic : Topic**
- timestamp : date and time at which this topic has been discovered
- probability : probability (from 0 to 1) with which this topic has been recognized

Person is a well-known concept of the Amigo ontology.

Topic has one property: **topicName**, which is a string containing the name of the topic.

6.5.3.2 How to discover a topic recognizer

The same process as for any other context source shall be used to discovery / subscribe and query the topic recognizer. Hence, the following RDF description shall be used to discover a running topic recognizer:

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:iccs="http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:context="http://amigo.gforge.inria.fr/owl/ContextTransport.owl#">
  <context:ContextSourceRegistration>
    <context:timeliness>current</context:timeliness>
    <context:contextType>UserTopicActivity</context:contextType>
  </context:ContextSourceRegistration>
</rdf:RDF>
```

6.5.3.3 How to query a topic recognizer

Once the topic recognizer has been discovered, the client can query some information from it: the name of the person (or unknown when this name has not been resolved) and the name of current topic (or unknown when the topic recognition has not completed successfully – may be

because the user has not talked so far, or because the user has talked about some topic that does not belong to the list of supported topics).

For example, the client can send the following query to the topic recognizer:

```
PREFIX amigo: <http://amigo.gforge.inria.fr/owl/Amigo.owl#>
PREFIX context: <http://amigo.gforge.inria.fr/owl/ContextTransport.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX iccs: <http://amigo.gforge.inria.fr/owl/AmigoICCS.owl#>
SELECT ?name ?top WHERE {
  ?p rdf:type amigo:Person .
  ?p context:identifier ?name .
  ?ul rdf:type context:UserTopicActivity .
  ?ul context:isTopicOf ?p .
  ?ul context:hasTopic ?t .
  ?t rdf:type context:Topic .
  ?t context:topicName ?top .
}
```

The answer will map the variable ?name to the user name, and the variable ?top to the topic name.

6.5.4 How to adapt the system to your voice and microphone (Implicit speech part)

Good recognition results can only be obtained with acoustic models adapted to your voice and microphone. Another option is to replace the default keyword recognition component shipped with the topic recognizer by any other speech recognizer. We have not put many efforts in developing techniques to facilitate automatic adaptation of the acoustic models, because this is clearly not the focus of the project. This is why it may be a good solution to connect the topic recognizer to another (better) speech recognizer : all this requires is a JAVA class that wraps your speech recognizer and calls the *TopicReco.pushText(String)* method whenever a new sentence has been uttered by the speaker. On the other hand, the shipped keyword recognizer is a light-weight component with few hardware requirements that runs very fast and does not consume much resources.

As a consequence, the acoustic model adaptation procedure is not straightforward, and it is recommended to first get more acquainted with this topic:

1. you should create about 20 to 50 sentences, each of them containing between 1 to 6 keywords used in the application (other words may also work, but it is best to use these ones to avoid lexicon issues)
2. you should run the script (requires cygwin) *res/record.sh* <sentence file> available from the *implicitSpeech* jar file that you can download on the Amigo Gforge. This script displays the sentence to be said. Press "REC" before starting and "REC" again after the end of the sentence.
3. the recorded files are saved in *tmp/*. Rename this directory and removes *tmp/phrases.mlf*,
4. edit the script *res/adapt.sh* by adding the new directory to the list of training directories

5. run the script *res/adapt.sh* and gets the new models in *adaptedModels/mods*
6. Replace the topic recognizer *si.mods* file by these models. That's it !

6.5.5 Workplan for the next versions

We are currently working on improving the topic recognition module in several aspects related to the deployment of implicit speech interactions in Ambient Intelligent platforms:

- Automatic unsupervised adaptation to new voices / microphones
- Automatic discovery and recognition of new keywords associated to a given topic (research activity)
- Automatic discovery and recognition of new topics, which are frequently used by a specific user (research activity)

References

- Ami05a Amigo Deliverable D1.2: "Report on User Requirements", M. D. Janse (ed.), IST-004182 Amigo, April 2005
- Ami05b Amigo Deliverable D2.1: "Specification of the abstract architecture of the Amigo System", IST-004182 Amigo, 2005
- Ami05c Amigo Deliverable D2.2: "State of the Art", Julia Kantorowitch (ed.), IST-004182 Amigo, 2005
- Ami05d Amigo Deliverable D2.3: "Specification of the Amigo Abstract System Architecture", M.D. Janse (ed.), IST-004182 Amigo, July 2005.
- Ami05e Amigo Deliverable D3.1a: "Detailed Design of the Amigo Middleware Core; Service Modelling for Composability", J. Kalaoja (ed.), IST-004182 Amigo, September 2005.
- Ami05f Amigo Deliverable D3.1b: "Detailed Design of the Amigo Middleware Core; Service Specification, Interoperable Middleware Core", N. Georgantas (ed.), IST-004182 Amigo, September 2005.
- Ami05g Amigo Deliverable D4.1: "Report on Specification and Description of Interfaces and Services", M.D. Janse (ed.), IST-004182 Amigo, October 2005.
- Ami05h Amigo Annex I – "Description of Work", update T0+12-T0+30, September 2005
- Ami06 Amigo Deliverable D3.2: Prototype implementation. N. Georgantas (ed.), IST-004182 Amigo, in progress.
- Ami06b Amigo Deliverable D4.2: Report on detailed Intelligent User Interface design, B Kladis (ed)
- Ami06c Amigo Deliverable D3.3
- Amiloc04 Thibaud Flury, Gilles Privat, Fano Ramparany, "OWL-based location ontology for context-aware services", Proceedings of the UbiComp'2004: AIMS workshop, Nottingham, UK, September 2004.
- Men03 Diego Rios Mendoza, "Using Ontologies in Context-Aware Services Platforms", Master Thesis, University of Twente, Enschede, The Netherlands, 2003.