iCOCOA
Software
Developer’s Guide

Graham Thomson, Sebastien Bianco

Abstract

This software developer's guide presents in detail, the architecture of the iCOCOA middleware, how to use each of the methods of the iCOCOA public API, as well as detailing of advanced iCOCOA middleware configuration.

Keyword list

HOWTO, ambient intelligence, networked home system, semantic concept, ontology, service description vocabulary, service description language, semantic reasoning, service matching, service composition, service adaptation, service execution, middleware.

Table of Contents

1 iCOCOA Overview

Provider
INRIA Paris-Rocquencourt

Introduction
Short description of the service/module

Development status
Prototype complete.

Intended audience
Users of the iCOCOA who wish to create semantic service descriptions for services, and use the semantic composition, adaptation and execution middleware.

License
The GNU Lesser General Public License v3.0 (http://www.gnu.org/licenses/lgpl.html).

Language
Java, WSDL, BPEL.

Platform
JDK 5 (Note that the ActiveBPEL library, upon which iCOCOA depends, is not supported at JDK 6).

Tools
The eclipse IDE, available at www.eclipse.org, is recommended to build the iCOCOA software and examples.

Files
All source code files are available via anonymous checkout from the Amigo SVN repository at: https://gforge.inria.fr/scm/?group_id=160.

Documents
In addition to this User's Guide, an iCOCOA Developer's Guide, an integrated Howto for all Amigo middleware components, iCOCOA examples with accompanying tutorials, and iCOCOA API documentation, are available for download at https://gforge.inria.fr/frs/?group_id=160.

Bugs
The following exception will be encountered when attempting to run iCOCOA using a Java 6 JVM:
```
java.lang.ClassCastException:
com.sun.xml.internal.messaging.saaj.soap.ver1_1.Envelope1_1Impl cannot be cast to
java.lang.String org.apache.axis.SOAPPart.getAsString(SOAPPart.java:554)
org.apache.axis.SOAPPart.writeTo(SOAPPart.java:322)
org.apache.axis.SOAPPart.writeTo(SOAPPart.java:269)
org.apache.axis.Message.writeTo(Message.java:539)
org.apache.axis.transport.http.AxisServlet.sendResponse(AxisServlet.java:902)
org.apache.axis.transport.http.AxisServlet.doPost(AxisServlet.java:777)
javax.servlet.http.HttpServlet.service(HttpServlet.java:710)
org.apache.axis.transport.http.AxisServletBase.service(AxisServletBase.java:327)
javax.servlet.http.HttpServlet.service(HttpServlet.java:803)
```
This is due the ActiveBPEL library upon which iCOCOA depends, which is not supported at JDK 6. Please use a Java 5 SDK / JVM.

2 iCOCOA Deployment

2.1 System requirements

In order to run iCOCOA, you must have the following software installed on your machine:
Apache Axis 1.4 Final
Apache Tomcat 5.5
ActiveBPEL 4.1

2.2 Download

The required software can be downloaded from the following locations:
Apache Axis 1.4 Final - http://ws.apache.org/axis/java/releases.html
Apache Tomcat 5.5 - http://tomcat.apache.org/download-55.cgi
ActiveBPEL 4.1 - http://www.active-endpoints.com/active-bpel-engine-overview.htm

2.3 Install

Installation instructions for the required software can be found at the following locations:
Apache Axis 1.4 Final - http://ws.apache.org/axis/java/install.html
Apache Tomcat 5.5 - http://tomcat.apache.org/tomcat-5.5-doc/index.html
ActiveBPEL 4.1 - http://www.active-endpoints.com/active-bpel-engine-overview.htm

3 iCOCOA Architecture

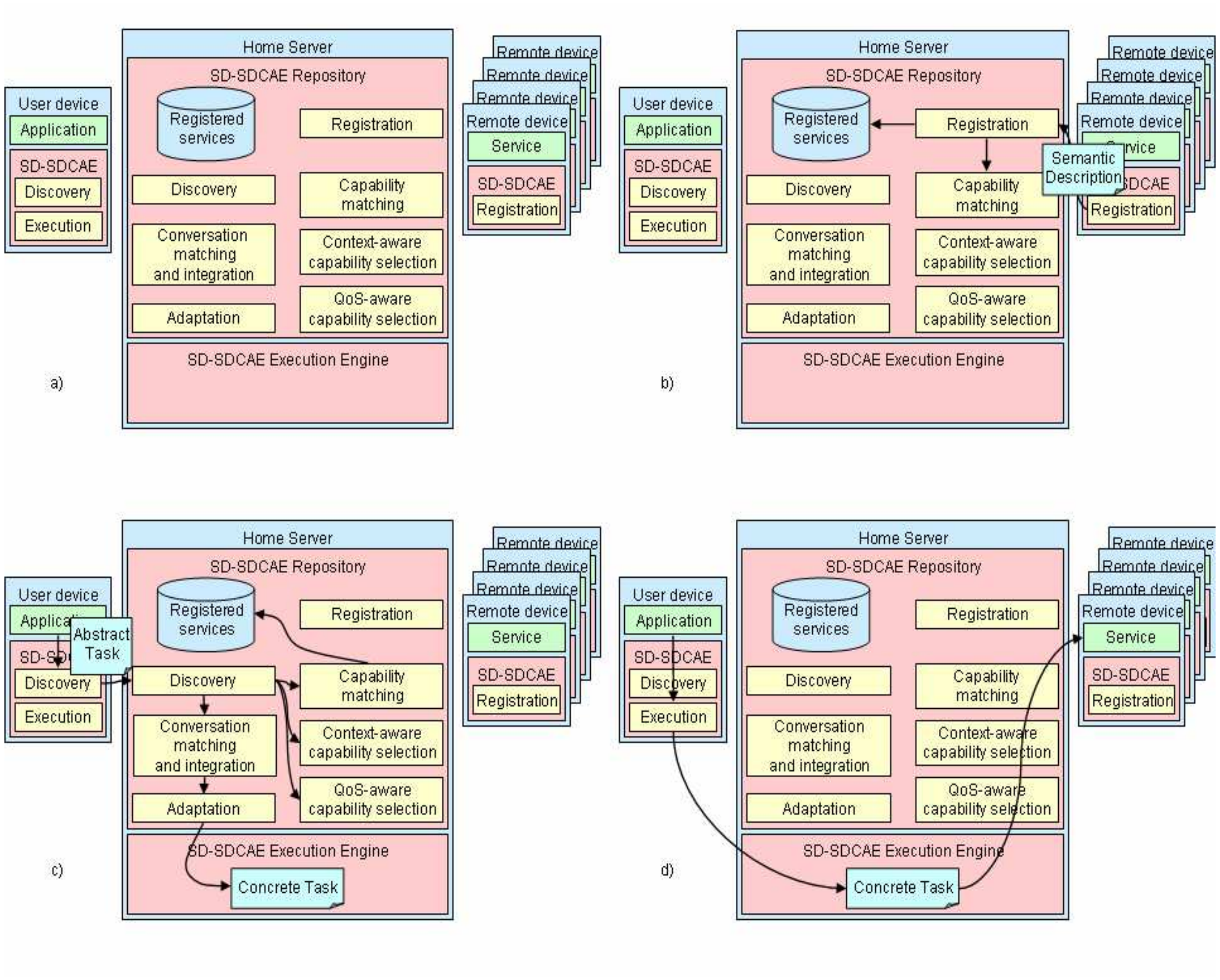

Figure 3-1 : The iCOCOA high-level architecture

Figure 3-1 shows the high-level architecture of iCOCOA. Figure 3-1 a) shows the main components within iCOCOA. These include a user's device, which hosts the application which exploits iCOCOA, other remote devices, which host the services that the application wishes to use, and the Amigo home server, which host the iCOCOA Semantic Service Repository and Execution Engine. This configuration is flexible however, and can be reconfigured to suit the need of a deployment. The repository and execution can be hosted on independent machines, for example, should a particular deployment require it. Similarly, common services and/or application could be hosted on the Amigo home server.
Figure 3-1 b) to d) show the basic steps involved in registering services, dynamically composing and application using the available services, and executing the application, respectively.
Figure 3-1 b) shows a service registering itself with the Amigo Semantic Service Repository. The first step in this phase is that the service discovers the repository. The repository appears in the Amigo home network as a basic Amigo service, and can be discover using the standard Amigo basic service discovery mechanisms [Amigo-D3.5]. Next the service registers itself by passing its Amigo-S semantic service description to the repository.
Providing an Amigo-S semantic service description for a service provides several features and advantages to the Amigo application developer:

It allows semantic service descriptions to be created for basic services, where the service provides a collection of atomic capabilities. This allows the service to be discovered via semantic matching at both the service and capability levels, increasing the service's availability and promoting its interoperability.
Semantic service descriptions can provide semantic concepts for a whole service, for each capability and its inputs and outputs, and optionally its pre-conditions and effects.
It allows semantic descriptions of conversation-based services, where the capabilities the service provides are described as a workflow of sub-capabilities. This allows the expression of data and control dependencies between the service's capabilities.
Complex workflows can be created by composing capabilities using a variety of control constructs, e.g. Sequence, Choice.
Context parameters can be described for a service, both at the level of individual capabilities, and for the service as a whole. Then, the service can be incorporated by the context-aware discovery mechanisms~\cite{AmigoD34}, which enables an application to optimise service selection based on services' current context, and frees the application developer from the need to actively search for the optimal service.
The quality of service parameters a service supports can be described both at the individual capability level, and for the service as a whole. This makes the service available to applications which employ the priority-based quality of service selection model provided by the Quality of Service Aware Service Selection Tool (QASST)~\cite{AmigoD34}, which allows the Amigo middleware to serve as many requests as possible while satisfying the majority of Amigo users.
Multiple groundings can be supported, enabling interoperability between different service technologies.

Figure 3-1 c) shows an application being dynamically composed, based on the services currently available in the repository. Again, the first step in this phase is that the application discovers the repository using standard Amigo basic service discovery. Then the application passes an abstract task, which describes the parts of the application that rely on external services available in the environment to run in an abstract way, to the repository. The repository then attempts to select suitable services based on matching the capabilities, context-aware parameters, and QoS parameters declared for the provided services, against those requested in the abstract task. If suitable services are found, the repository than attempts to compose and if necessary, adapt the available services to fit the workflow of capabilities required by the abstract task. Finally, if a successful composition is found, the concrete task -the version of the abstract task now instantiated with real service capabilities, is deployed on the home server.
Figure 3-1 d) shows an application executing the dynamically composed service, the concrete task, that realises the abstract task in the previous phase. The concrete task appears as a regular Amigo service, and be discovery and executed as such. In fact, other external applications could use the newly-composed service, should it match their task description. When a capability of the task is executed, the iCOCOA Execution Engine automatically orchestrates the execution of each of the composite service capabilities that feature in the task capability's workflow, one or more remote devices.
Describing the service-based parts of application as a task offers the following features and advantages to the user:
A task provides an abstract description of the required capabilities of an application. In doing so, we are not bound to any particular remote service in terms of the capabilities provided or the specific orchestration of these capabilities, increasing the availability and promoting interoperability of the potentially matching services.
The required capabilities of a task can be identified by the semantic of the capability and of its inputs and outputs. This allows concrete details of the services' provided capabilities used in a composition to be reliably and automatically adapted to the needs of the required capabilities in the absence of an exact match.
An application may invoke capabilities of varying complexity, from lightweight atomic calls to complex, interleaved conversations (i.e. service workflows), for one or many tasks from within the same application.

The conversation (workflow) of a required capability of a task is reconstituted by weaving together the conversations (workflows) of the provided capabilities of the available services. This offers automatic and reliable service composition, while offering fine-grained control over the placement of capabilities in the task, and guaranteeing that the data and control dependencies of each of the provided capabilities are preserved.
The resulting (composed) service is generated as an executable ActiveBPEL bundle and automatically deployed. Orchestration of the execution of the composed service is handled automatically by the ActiveBPEL execution engine.

4 iCOCOA Public API Tutorial

4.1 *iCOCOA API*

In this section, we shall look at each of the available methods in iCOCOA public API in turn, giving examples of use.

4.1.1 Adding and removing a service

To add a service to the repository, call the following method:

```
/**
 * Register a service in the repository. Adds the semantic description of the service
 * together with its associated WSDL document. <p>
 * Returns the URI of the service, as declared in the SDL description. This URI can
 * further be used to remove the service using the removeService method.
 * @param providedService
 *                        SDL description of the service
 * @param WSDLDocument
 *                        WSDL Document of the service
 * @return URI of the service (unique identifier to be used with removeService)
 * @throws SemanticRepositoryException
 *                         if an error occurs during the parsing of the service, if an
 * error occurs with a file, or if a similar service as ever been registered.
 */
public String addService(String providedService, String WSDLDocument) throws
SemanticRepositoryException;
```

This method registers a service in the repository. It adds the semantic description of the service together with its associated WSDL document.

An example call to this method is:

```
String id = semrep.addService( "file://local/path/to/service/description/MyService.owl",
"file://local/path/to/wsdl/description/MyService.wsdl");
```

To remove a service from the repository, call the following method:

```
/**
 * Unregister a service in the repository. Removes the semantic description of the
 * service. The service is identified by its URI, which is the identifier that is returned by
 * the addService(java.lang.String, java.lang.String) method.
 * @param serviceURI
 *                    URI of the service to unregister.
 * @throws SemanticRepositoryException
 *                     an exception if the removal of the service fails. E.g, if there is no
 * registered service referenced by the given URI
 */
public void removeService(String serviceURI) throws SemanticRepositoryException;
```

This method un-registers a service in the repository and removes its semantic description. The service is identified by its URI, which is the identifier that is returned by the add service method.

An example call to this method is:

```
semrep.removeService(id);
```

4.1.2 Adding an ontology

To enable working with ontologies that available locally, but not on the network, ontologies can be added directly into the repository. This is useful if the ontology is still under development, and not yet published on the network, or if the application is running behind a firewall.

To add an ontology, call the following method:

```
/**
 * Calling this method will add the specified local ontology to the repository. This method
 * allows local copies of ontologies to be used, when required ontologies are not available
 * over the network, e.g. when the application is run behind a firewall, or the ontologies
 * are still in development and not yet published.
 * @param filename
 *             the path to the local ontology file
 * @throws SemanticRepositoryException
 *             if an exception occurred processing the ontology file
 */
public void addOntology(String filename) throws SemanticRepositoryException;
```

4.1.3 Retrieving a service by name or type

To retrieve a service by name, we can use the following method:

```
/**
 * Retrieve a service by its name.
 * @param serviceName
 *               the name of the required service
 * @return the Service object corresponding to the given name
 */
public String getServiceWithName(String serviceName);
```

For example:

```
semrep.getServiceWithName("http://my/ontology/MyService.owl#MyService");
```

To retrieve a service by name, we can use the following method:

```
/**
 * Retrieve the set of services that match a semantic type.
 * @param serviceType
 *               the semantic type of the required services
 * @return a list of Service objects corresponding to the given semantic type
 */
public String[] getServicesWithType(String serviceType);
```

For example:

```
semrep.getServicesWithType("http://my/onotology/MyTypes.owl#MyType");
```

4.1.4 Retrieving information related to a registered service

Several methods exist that allow information about a registered service to retrieved, such as a service's WSDL description, its QoS parameters, and its type.

To retrieve a WSDL description for a registered service, call this method:

```
/**
 * This method returns the associated WSDL description for the specified service.
 * @param serviceName
 *                  the service
 * @return The WSDL description of the specified service, or null if the service is not
 * registered with the repository
 */
public String getWsdlForService(String serviceName);
```

To retrieve the QoS properties for a registered service, call this method:

```
/**
 * Get the XML Fragment of a service description corresponding to its QoS properties.
 * @param serviceName
 *                  the name of the service we required the QoS properties
 * @return an XML fragment containing QoS properties associated with the service.
 */
public String getQoSProperties(String serviceName);
```

And to retrieve the type information for a registered service, call this method:

```
/**
 * Get the semantic type of a service from its name.
 * @param serviceName
 *                  the name of the service that we want to know the name
 * @return the semantic type of the service, <i>null</i>
 */
public String getServiceTypeFromServiceName(String serviceName);
```

Each of these methods takes simply the name of the service as a parameter, for example `"http://my/ontology/MyService.owl#MyService".`

4.1.5 Retrieving (composed) services that match a task

To retrieve the services that match a task description, call the method below. If no services match the task exactly, the repository will attempt to automatically compose a compatible service from those that are available in the repository.

```
/**
 * Retrieve a set of services that match the task description. If no service match exactly,
 * repository will attempt to compose and deploy a match service from the services
 * currently registered with the repository.
 * @param taskURI
 *                  the URI of the file containing the description of the required task
 * @return a list of matching services if a match has been found, an empty list if no
 * match is possible, or <i>null</i> if it was not possible to process the request
 * @throws SemanticRepositoryException
 *                   if an error occurs during the parsing of the service, or if the task
 * description contains no required capability
 */
public String[] getServicesForTask(String taskURI) throws SemanticRepositoryException;
```

4.1.6 Retrieving all services

All services that are registered with the repository can be retrieved by calling the follwing method:

```
/**
 * Get a list of all service object managed by the repository.
 * @return the list of all services, empty list if no services are manages by the repository.
 */
public String[] getAllServices();
```

4.1.7 Clearing the repository

During development, it can be useful to clear out the content of the repository. To do so, call the method below. However, it is not recommend to call this method on a live repository.

```
/**
 * Remove all services and service descriptions from the repository. <p>
 * Note: This method is intended for developers to provide a convenient means to clear
 * the repository during development and testing. It is not recommend to call this
 * method on a live repository.
 */
public void clearRepository();
```

4.1.8 Setting a new matcher to be used by the repository

The repository can be configured to match services to tasks in different ways. To set a new matcher, call the following method:

```
/**
 * Set the type of matcher to be used by the repository.
 * @param matcher
 *                the name of the matcher to be used.
 * @throws SemanticRepositoryException
 *                if the name of the matcher is not recognised.
 */
public void setMatcher(String matcher) throws SemanticRepositoryException;
```

By default, the repository will matches only identical concepts. That is, the capability semantic for a service’s provided capability must be identical to the capability semantic of the task’s required capability, for there to be a match between them. This is termed “syntactic” matching.

The repository can be set to use this type of matching as follows:
`semrep.setMatcher("syntactic");`

In addition, the repository offers two other looser modes of matching. First, is “exact” matching – this will match concepts that are identical, but also concepts that are either explicitly equivalent (tagged with the `owl:equivalentClass` relation), or implicitly equivalent (it has similar properties of another class tagged with the `owlowl:equivalentProperty` relation). The repository can be set to use this type of matching as follows:
`semrep.setMatcher("exact");`

Second, is “weak” matching – this will match all concepts that exact matching does, but also this will match concepts that are identical, but also concepts that are within a stated radius of the to be matched concept. For example, if we have a radius of 1, the concept would be matched with either an immediate super-class or an immediate sub-class. The repository can be set to use this type of matching as follows:
`semrep.setMatcher("weak 1");`

4.2 Advanced iCOCOA Configuration

The level of logging used throughout the iCOCOA middleware can be controlled through a configuration file.
You can configure the path of your file that contains the log4J properties by using the environment variable "log4j.configfile". To set it, run the JVM with the option:
-Dlog4j.configfile = "YourPath"
If a path does not exist, then the default property file will be used for the log4j logger. The default file is "./log4j.prop". If no file is found, a default Appender will be set (System.err stream).
By setting the environment property "log4j.debuglevel", the depth of logging information reported can be controlled (-1 -no message, 0 -debug, 1 -info, 2 -warning, 3 -error, 4 fatal). Only messages with a level greater than or equal to the current logging level will be displayed. The default value is "0".

5 References

[Amigo-D3.5] Amigo Consortium. Deliverable D3.5: Amigo overall middleware: Final prototype implementation & documentation. November 2007. Available at: http://www.hitech-projects.com/euprojects/amigo/deliverables.htm.