Building Services using O^{3}MiSCID , Oscar and Castor

Prima, Gravir

December 20, 2006

Contents

1 Introduction

The goal of this document is to introduce service development using O^{3}MiSCID . O^{3}MiSCID is a multi-plateform, multi language middleware. It currently works under Linux, Windows and Mac OSX and has a C++ and Java implementation. The Java implementation is available as a jar for standalone applications and as an OSGi bundle for OSGi applications. This tutorial will focus on the OSGi version of O^{3}MiSCID .

2 Software Installation

We will need several software to develop services:

- Eclipse + Webtools: the Java IDE.
- Bonjour: the dns-sd daemon for service discovery.
- O^{3}MiSCID : the middleware bundle.
- The O^{3}MiSCID wizard: an eclipse plugin to easily setup a new O^{3}MiSCID project.
- The O^{3}MiSCID GUI ; a service browser.
- The Castor plugin: an XML tool for marshalling and unmarshalling XML messages (for services communication)
- Oscar: the OSGi plateform

The software installation is based on Linux.

2.1 Eclipse + WebTools

Eclipse is a multiplatform IDE. It can be extended using a plugin mechanism (based on OSGi). In this tutorial, we will need several plugins from various sources (see next sections). From the Eclipse main site, we will need the WebTools plugins to facilitate the XML Schema creation. The WebTools are now part of the Callisto project and can be directly installed from the Callisto update site. They can be also downloaded separately and installed in an already existing Eclipse. The WebTools URL is http://www.eclipse.org/webtools/.

2.2 Bonjour

Bonjour is the dns-sd implementation proposed by Apple. It runs under MacOSX, Windows and Linux. It is included in almost all the existing Linux distribution. But you might need to recompile it if the version included in your distribution is too old. In particular, versions prior to `mDNSResponder-107.5` are unstable with Java.
After registering, the source code can be downloaded at
http://developer.apple.com/opensource/internet/bonjour.html.

To install it:

```
tar xvfz mDNSResponder-107.5.tar.gz
cd mDNSResponder-107.5/mDNSPosix/
make os=linux
make os=linux JDK=$JAVA_HOME Java
# As root (dirty but needed)
make os=linux install
cp build/prod/libdns_sd.so $JAVA_HOME/jre/lib/i386/
cp build/prod/libjdns_sd.so $JAVA_HOME/jre/lib/i386/
cp build/prod/dns_sd.jar $JAVA_HOME/jre/lib/ext/
```

To launch the daemon:

```
/etc/init.d/mdns start
```

See you distribution documentation to automatically start the daemon at launch time.

2.3 Castor

In O^{3}MiSCID , messages between services are very often textual (even if they can be binarie). But it is often a good choice to use XML messages, specified by XML schema. Castor (www.castor.org) is a mapping framework between XML, object and relational databases. In particular, it is able to automatically build the Java object corresponding to data types described in an XML Schema. Corresponding XML messages can then be automatically unmarshalled into Java objects and Java objects can be marshalled into XML messages.

The Castor code generation is integrated into the Eclipse IDE through the `Casto Plugin`. It is a jar file that can be downloaded at :
http://prdownloads.sourceforge.net/xdoclipse/install-com.pnehrer.castor_2.0.3.jar?download

To install it, simply run

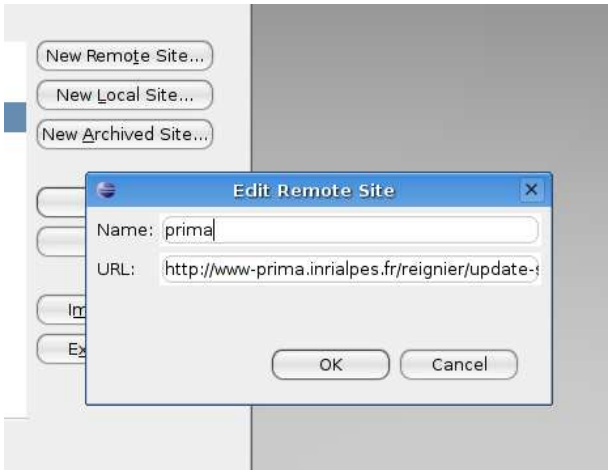
```
$ java -jar install-com.pnehrer.castor_2.0.3.jar
```

and follow the instructions

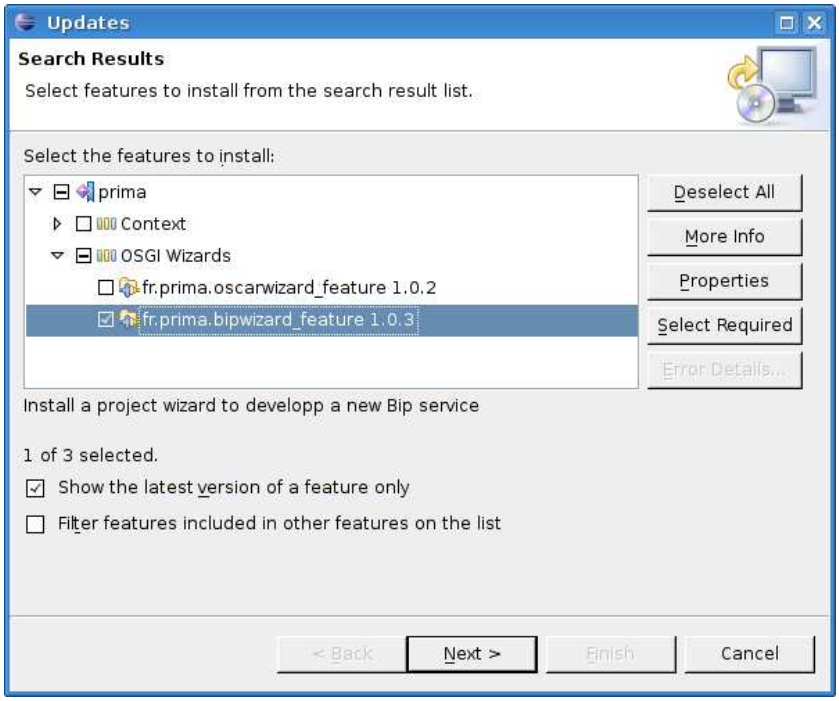
2.4 O^{3}MiSCID Wizard

The O^{3}MiSCID wizard is available as an eclipse update site:

1. In eclipse, open the software update menu: **Help ⇒ Software Update ⇒ Find and installs ...**
2. select **Search for new features to install**
3. select **New remote site**. The URL is HTTP://www-prima.inrialpes.fr/reignier/update-site/



4. It will propose you to install the O^{3}MiSCID wizard.



2.5 O^{3}MiSCID GUI

The O^{3}MiSCID GUI is a graphical service browser. It displays all the available services on the local network. You can inspect the services connectors and variable and send them messages. The GUI is part of the Java version of O^{3}MiSCID and can be downloaded from the forge site.

More details will be given later.

- Uncompress the `omiscid-lib-and-gui` archive
- `java -jar omiscid-gui.jar -anim`

2.6 Oscar

Oscar (http://oscar.objectweb.org/) is an OSGi plateform. It can be downloaded at http://download.forge.objectweb.org/oscar/oscar-1.0.5.jar

To install it, run:

```
$ java -jar oscar-1.0.5.jar
```

and follows the instructions. We note `<oscar>` the Oscar installation path.

2.6.1 Setting the bundle repository

An OSGi plateform[1] is a "plugin environment". It allows to dynamically load (from a repository), unload, update, start or stop plugins. Plugins can dynamically recruit each other. In OSGi, a plugin is called a bundle. They can be downloaded from a local file system or from internet. When developping, it is convenient to have a local repository in its file system. This repository is a folder where the bundles will be copied (published) after compiling, ready to be loaded in the plateform.

- create an `OscarBundles` folder in your file system that will be your bundle repository. We note `<OscarBundles>` this path.

[1] http;//www.osgi.org, http://www-adele.imag.fr/ donsez

- every new bundle project has to know where is your local repository, so that it can publish the bundle after compiling it. To avoid duplicating this information in every new project, we centralise it in a property file located at the root of your homedir : `build.properties`. This file will be used latter by the eclipse omiscid wizard. Create this file and put the following property :

```
# Oscar installation
osgi.publish.dir=<path to your <OscarBundles>/ directory with an endind slash>
```

2.6.2 Oscar shell

Oscar is now ready to run. Go into the Oscar installation directory. Launch `oscar.sh` and enter a profile name (you may need to make it executable to run it). You are now inside oscar shell. You can display the command list using the `help` command. The main commands are :

- `ps`: displays the list of bundle associated to you profile. Each bundle has a unique number (id), and a state:
  - `Active` if running
  - `Resolved` or `Installed` if not running. For more details on the difference between both, see http://www-adele.imag.fr/~donsez/cours/#osgi
- `start <id>`: starts a bundle (specifying its id, see `ps`)
- `start <url>`: loads a bundle and starts it. The URL can be for instance `http://...` to download a bundle from internet, or `file:/....` to load it from a local file system.
- `install <url>`: loads a bundle (without starting it)
- `uninstall <id>`: removes a bundle from the current profile
- `update <id>`: reloads a bundle. The `update` command must be used each time you recompile a bundle to reload the new version in the profile.
- `shutdown`: exit Oscar.

Note: if you are using a bundle repository, your URLs will always be a common root (to the root of the repository) plus the bundle reference. To avoid always retyping the common root, you can specify it in the `bundle.properties` file in the `<oscar>/lib` folder. For instance:

```
oscar.shell.baseurl=file:/home/oscar/
```

You can check this base URL specification with the `cd` command in the oscar shell. Once this base URL is specified, you only have to specify the final part of the URL in the `start <url>` command. It is automatically appended to the base URL.

2.7 O^{3}MiSCID , Castor and the Service Binder bundles

We now have to install the three bundles that will be used by our O^{3}MiSCID services : O^{3}MiSCID , castor (built from the Castor jar files) and the service binder (proposed by Humberto Cervantes). O^{3}MiSCID and Castor's bundles can be downloaded from the forge site :
http://gforge.inria.fr/frs/?group_id=363

To install them, simply unzip both files at the root of your bundle repository. The service binder can be downloaded (from the Oscar Bundle Repository: a web server containing a collection of oscar bundles) and installed directly from the Oscar shell using the `obr` command (see next paragraph).

Caution: each time you start a new Oscar profile (or if you clean you profile by removing the corresponding folder), you will have to reload both bundles and to reinstall the service binder.

```
-> obr start "service binder"
-> start omiscid/omiscid.jar
-> start castor/castor.jar
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Starting   ] [    0] System Bundle (1.0.5)
[   1] [Active     ] [    1] Shell Service (1.0.2)
[   2] [Active     ] [    1] Shell TUI (1.0.0)
[   3] [Active     ] [    1] Bundle Repository (1.1.2)
[   4] [Active     ] [    1] Service Binder (1.1.2)
[   5] [Active     ] [    1] omiscid (1.1)
[   6] [Active     ] [    1] castor (0.1)
->
```

Note: it is possible to configure `oscar` to load those three bundles automatically each time the plateform is started (so they do not have to be manually reloaded when creating a new profile).

- Download the service binder bundle[2] and copy it at the root of your bundle repository.
- Go in the `<oscar>/lib` folder. Edit `system.properties` and add the three jars. For instance :

```
oscar.auto.start.1=file:bundle/shell.jar file:bundle/shelltui.jar \
    file:bundle/bundlerepository.jar file:bundle/servicebinder.jar \
        file:<OscarBundles>/omiscid/omiscid.jar \
        file:<OscarBundles>/castor/castor.jar \
        file:bundle/servicebinder.jar
```

3 Main Concepts

A O^{3}MiSCID service is an OSGi bundle (component). It is defined by a unique name. It is composed of connectors and variables. Service name and connectors and variables list are specified through an XML file.

Connectors are the access points of the component. They can be used to send/receive data (events, stream etc). They can be mono or bi-directional (input, output, or inoutput channels).

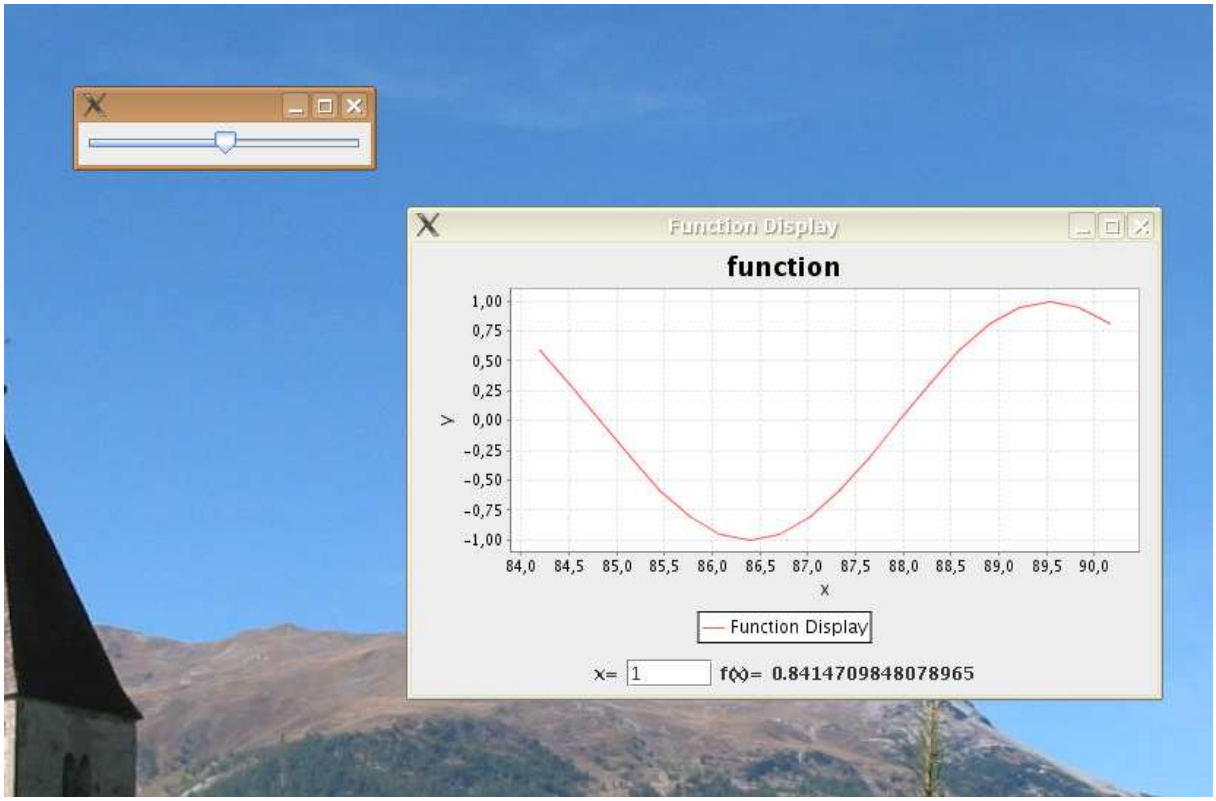
A component can also expose state parameters using variables. Variables can be read and written depending on their access right. Another component can subscribe to variable modifications and receive notifications each time the variable value is changed.

4 The application

In this tutorial, we will transform a "monolithic" application (`curves.zip`) in a distributed version using 3 services. This application displays a sinus function. It is possible to change the sinus period using a slider. The user can also ask the value of the function for a particular abscisse.

- Unzip the archive `curves.zip`.
- Go in the `curves/bin` directory.
- `java -cp .:../lib/jcommon-1.0.1.jar:../lib/jfreechart-1.0.1.jar prima.main.Main`

[2] http://gravity.sf.net/servicebinder/jar/servicebinder.jar



The application has 4 packages:

generator: the function generator. A function generator generates a new point each time the method `generateNext()` is called.

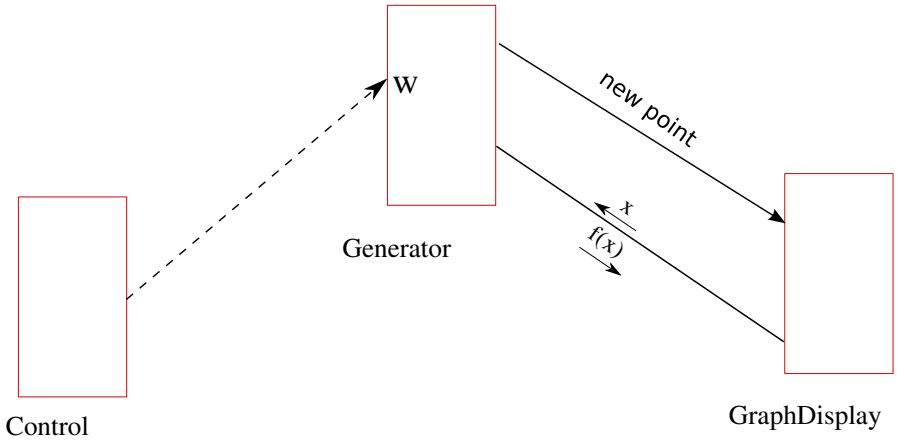
control: the slider that changes the sinus period.

GraphDisplay: the graphical display of the function. The display is updated each time a new point is added using the `addPoint()` method. The graphical displays has also an input field. The user can enter an abscisse `x`. It will compute `f(x)` using the `wantCompute()` method.

Main: the main package. It creates the control, the sinus generator, the graphical display and enters an infinite loop: getting the next point from the generator and feeding this point to the graphical display.

This application will be splitted in 3 services (corresponding to the first three packages):

- A `generator` service. It has an `output` connector, sending periodically new points, an `in/output` connector, returning `f(x)` when receiving `x`. It exposes a variable `w` (the sinus period) that can be changed by other services.
- A `graphical display`. It has an `input` connector, waiting for new points to draw, an `in/output` connector waiting for `f(x)` when sending `x`.
- A `control` service. This service changes the variable value of another service, based on its slider position.



5 First service: the generator

We will first implement a simplified version of the generator, exposing only the `output` connector.

5.1 An empty registered service

1. Create a new Omiscid project (File → New → Other ... + OSGi→ Omiscid). The project name is `generator`. The package name is `prima.generator`. The main class name is `Main`.

The wizard setups a new project and creates all the necessary files to generate a bundle :

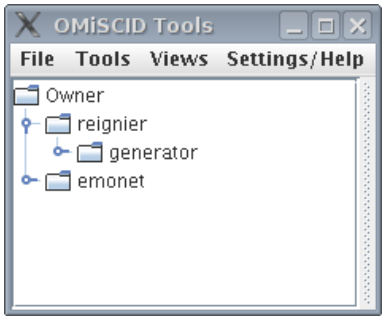
- that will use Omiscid and Castor bundles.
- that will register the service in DNS-SD (using the project name as the service name).

In particular, if we call `<package>` and `<Main>` the name of the main package and the main classname entered in the wizard, the project will have two packages :

package: It contains the interface `<Main>.java`

package.impl: it contains the class `<Main>Impl.java`, implementing the `<Main>` interface. It is this class that must be completed to implement our service. The other generated files are for OSGi.

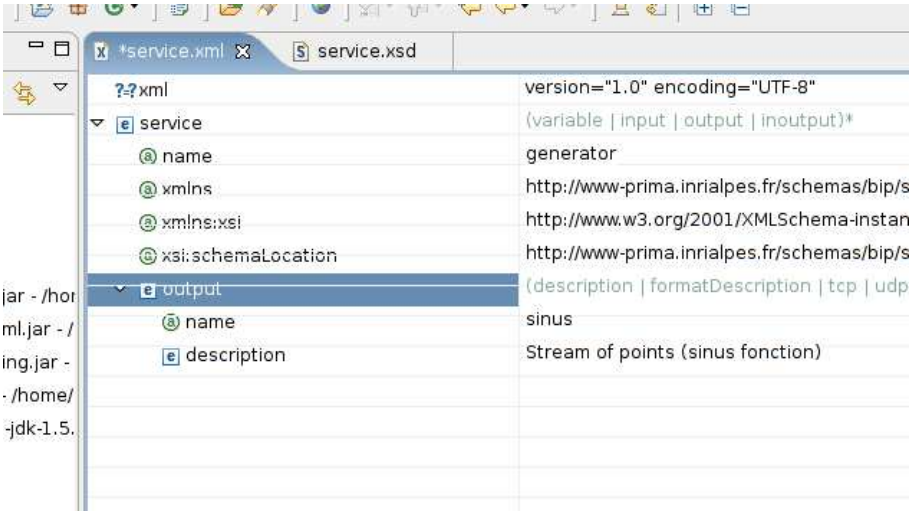
2. Open the contextual menu on the `build.publish.xml` file and select `Run as` → `Ant build` menu. It will build and deploy the bundle.
3. The bundle can now be started from the oscar shell: `start generator/generator.jar`. It does not do much by now ! but you must see it in the O^{3}MiSCID GUI.
4. you can display services by names in the O^{3}MiSCID GUI. You can also display them sorted by owners (convenient if several persons are doing the same tutorial). Menu: Tools → OMiSCID service browser → by owner



5.2 Adding a connector

We are now going to add an output connector named `sinus`. The connector declaration is done in the `service.xml` file located in the `resource` folder. The XML Schema for the service specification file is given in appendix A

- Open the `service.xml` file in Eclipse
- The file can be edited directly in the XML source editor. You can also use the `Design` tab that proposes contextual menu based on the XML Schema.
- In the `Design` tab, opens the contextual menu on the `service` tag and select `Add child` → output.
- Add a `description` field (`Add child` contextual menu on the `output` tag).



- Copy the `Generator`, `Point` and `SinusGenerator` classes from the `curves` project in the `generator` source tree.

When a service is started or stopped (using for instance the `start` and `stop` command of the oscar shell), the `start()` and `stop()` method of the Main class are invoked. The `start()` method is invoked in a new Thread.

- Add a boolean `finished` to the `<Main>Impl` class
- Complete the `start()` and `stop()` methods to print the generated points on the console.

```
public void start()
{
   Generator generator = new SinusGenerator();
   service.start() ; // the service is now published on DNS SD
   while (!finished)
   {
       Point p = generator.generateNext();
       System.out.println(p);
    }
}

public void stop()
{
   finished = true ;
}
```

- Redeploy the bundle (ant file) and update it from the shell:

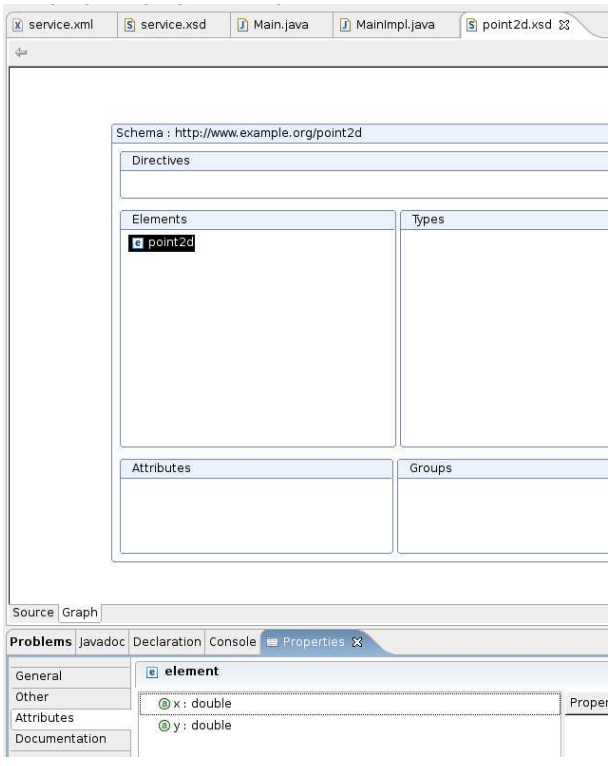
```
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Starting   ] [    0] System Bundle (1.0.5)
[   1] [Active     ] [    1] Shell Service (1.0.2)
[   2] [Active     ] [    1] Shell TUI (1.0.0)
[   3] [Active     ] [    1] Bundle Repository (1.1.2)
[   4] [Active     ] [    1] Service Binder (1.1.2)
[   5] [Active     ] [    1] omiscid (1.1)
```

```
[   6] [Active     ] [    1] castor (0.1)
[   7] [Active   ] [    1] generator (0.1)
-> update 7
```

In the next part, we are now going to generate an XML message (that we still print on the console).

- Create a new schema `point2d.xsd` in the ressource folder (menu `New` → `XML` → `XML Schema`).
- Enter the following schema (the Eclipse Schema graphical interface is very convenient to specify schema)

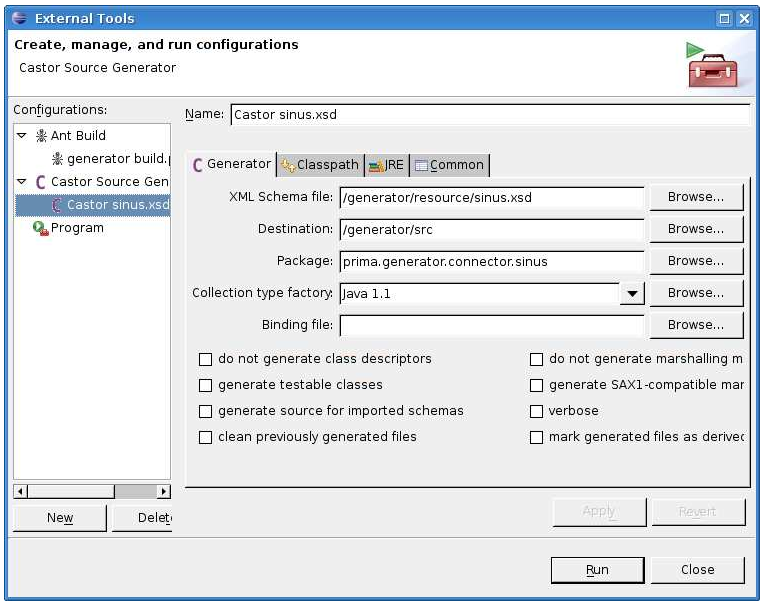
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.example.org/point2d"
      xmlns:tns="http://www.example.org/point2d">
<element name="point2d">
<complexType>
  <attribute name="x" type="double"></attribute>
  <attribute name="y" type="double"></attribute>
  </complexType>
</element>

</schema>
```



- Configure the project so that we can use Castor to generate the Java object described in the Schema. Right click on the project name and select *Castor* → *Import Castor libraries* .... The libraries must be imported in the *generator/externallibs* folder. Importing the Castor jars instead of just adding them to the build path allows to recompile the project[3] outside the Eclipse environment.

[3] using ant

- Using `Castor`, create the corresponding `Point2D` object: contextual menu on the `point2d.xsd` file, `Castor` → `Generate Java source code ...` Fill in the package field and let all the default choices.



- An XML message can now be constructed in the main loop using the generated class:

```
Point2d p2d = new Point2d() ;
p2d.setX(p.getX());
p2d.setY(p.getY());
StringWriter out = new StringWriter();
try {
   p2d.marshal(out);
} catch (Exception e) {
   e.printStackTrace();
}
System.out.println(out.toString());
```

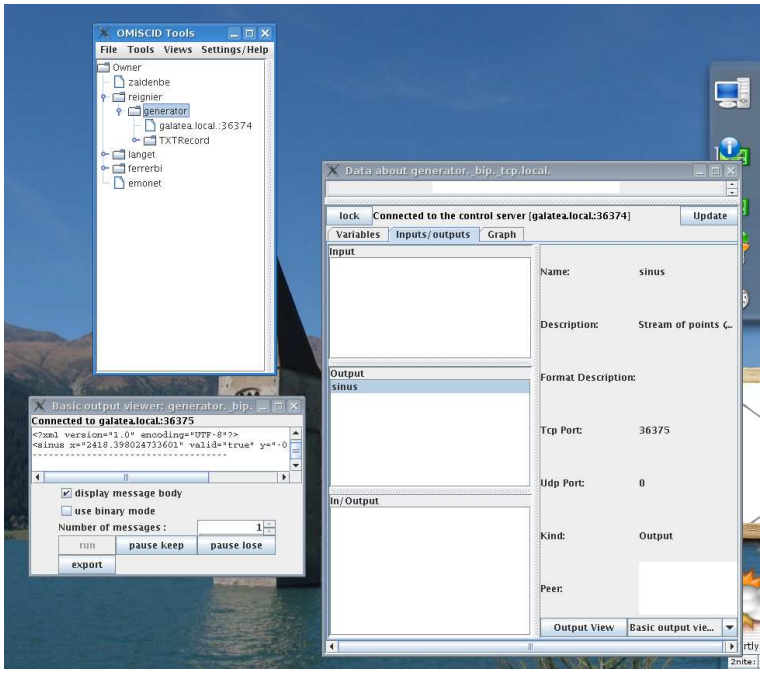
  Note: if we had build the `generator` classes directly for this service, we would have used internally the `Point2d` class instead of using both `Point` and `Point2d`.

Final step: we send the XML message on the `sinus` connector. The `MainImpl.java` class has an attribute: `service`. This object encapsulates all the O^{3}MiSCID methods to manipulate connectors, variables, to send and receive messages.

- send the XML message to all the services that might be connected to `sinus` :

```
service.sendToAllClients("sinus",out.toString().getBytes());
```

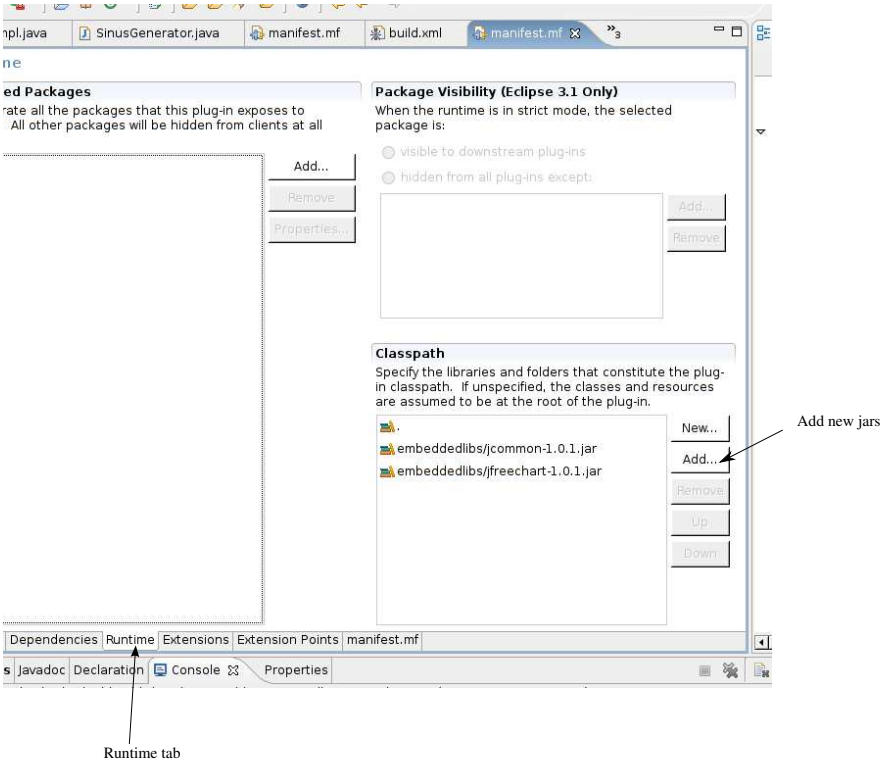
- redeploy and update your service
- on the O^{3}MiSCID GUI, double click the service name. It will open a service description window. Select the `inputs/outputs` tab, click on `sinus` and press the `Output View` button to see the messages



6 The graphical display service: input connector

We are now going to build a simplified version of the graph service. It proposes an input connector to receive new points to display.

- create a new omiscid project: `display`.
- add an input connector: `point`.
- We will need to decode XML messages coming from this connector: copy the `point2D.xsd` schema from `generate` service to the ressource folder of the `display` service. Use castor to generate the corresponding sources.
- Copy the jar files `jcommon` and `jfreechart` from the lib folder of the `curves` project to the `embeddedlibs` of the display `project`. This folder contains additional jars that can be used by the service. Note :
  - the two jars must be also added to your eclipse project build path (so that eclipse can use them for code auto-completion and code checking)
  - the two jars must be added to th runtime classpath of the bundle. This is done in the `manifest.mf` file located in the `manifest` folder. Open the file from Eclipse. Select the `runtime` tab. and add the two jars in the `classpath` section.



- Copy the `prima.display.GraphDisplay` class from the `curves` project to the `display` project. Because of the new service architecture, this class must be modified:
  - it does not need any more a `generator` attribute (and the corresponding getter and setter)
  - the `Point` class can be replaced by `Point2d` generated from the XML Schema.
  - the line `double y = generator.f(x);` of the method `compute()` must be removed. The `y` value will be computed by the generator, using messages (see next section)
  - The line `frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);` (end of class constructor) must be remove. Otherwise, clicking on the close button of the window will kill not only the bundle but the oscar plateform also ! Instead, add the following `stop()` method. We will call this method later.

```
public void stop()
{
    frame.setVisible(false);
    frame.dispose() ;
}
```

To receive a message on an input port, a class must implement the `ConnectorListener` interface and provide the `messageReceived()` and `disconnected()` method.

```
package fr.prima.omiscid.user.connector;

import fr.prima.omiscid.com.interf.Message;
import fr.prima.omiscid.service.Service;

public interface ConnectorListener {
```

```
  /**
     * Processes a received BIP message. As a given message could be processed
     * by several others listeners, the message must not be modified by its
     * processing.
     * @param service the service receiving the message
     * @param localConnectorName the name of the connector that has received the message
     * @param message
     *            the BIP message to process
     */
    public void messageReceived(Service service, String localConnectorName , Message message);

    /**
     * Called when the connexion between the local service and the remote
     * service is broken.
     * @param localConnectorName the name of the connector handling the broken link
     * @param peerId the disconnected remote service
     */
    public void disconnected(Service service, String localConnectorName, int peerId);

   /**
     * Called when a remote service connects to the local connector.
     * The peerId is a unique pair (service + remote connector). The
     * corresponding service filter can be found using a <CODE>PeerIdIs</CODE> filter.
     * The connector name can be found using the <CODE>findConnector</CODE> method
     *  on the <CODE>ServiceProxy</CODE>
     * @param service the service receiving the message
     * @param localConnectorName the name of the connector handling the broken link
     * @param peerId the peerId (a unique Id corresponding to a connector
     *               and a remote service)
     * @see ServiceProxy#findConnector(int)
     */
    public void connected(Service service, String localConnectorName, int peerId);
}
```

- Modify the `GraphDisplay` class to implement the `ConnectorListener` interface. The `disconnected` and `connected` method will remain empty. The `messageReceived()` will just print on the console the message content :

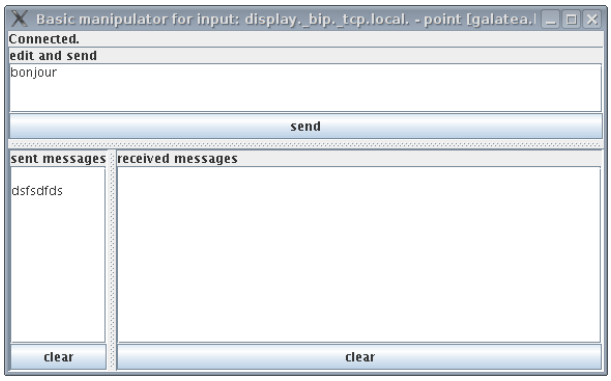
  ```
  System.out.println(message.getBufferAsStringUnchecked());
  ```

- In the MainImpl `start()` method, create a `GraphDisplay` and add it as a listener of the `point` connector.

  ```
  protected GraphDisplay graph=null;

  public void start()
  {
      graph = new GraphDisplay() ;
      try {
          service.addConnectorListener("point", graph);
      } catch (UnknownConnector e) {
          e.printStackTrace();
      }
  ```

```
    service.start();
}
```

- To destroy the graphical widget when stopping (or updating) the bundle, call the graph `stop()` in the MainImpl `stop()` method.
- Deploy the new service and start it under Oscar
- Open service description window under O^{3}MiSCID GUI. Select the `point` connector. Click on the `Manipulate input with` button.
- Check that the text typed in the message window are received by `graph` and printed in the Oscar console.



We are now going to plug the generator `sinus` output to the graph `point` input. To do that, we first need to find the `generator` service. A service is first searched by its name. We can then add other criteria (Filters) like the service owner, the computer running the service etc. All those criteria can be combined (using an and operator). New criteria can be programmed.
Some examples:

- Finding a service by its name (no other criteria):

  ```
  findService(ServiceFilters.nameIs("generator") ;
  ```

  Note: when several instances of a same service are launched, dnssd creates different names automatically by adding a number at the end the service name (`generator`, `generator (1)`, `generator (2)` ...). All those new service names will match a service request using the name `generator` The `findService()` method returns a class implementing `ServiceProxy`. This class is your local view of the remote service. It can be used to interrogate the remote service list of variables, connectors. It is also used to connect a local connector to a connector on the remote service.

- Filtering a service by the computer hosting it:

  ```
  ServiceProxy generator =
        service.findService(ServiceFilters.and(ServiceFilters.nameIs("generator"),
                                               ServiceFilters.hostPrefixIs("prometheus")));
  ```

- Filtering a service by the service owner:

  ```
  ServiceProxy generator =
         service.findService(ServiceFilters.and(ServiceFilters.nameIs("generator"),
                                                ServiceFilters.ownerIs("prometheus")));
  ```

- Filtering a service by its connector's name:

```
ServiceProxy generator = service.findService(ServiceFilters.hasConnector("sinus"));
```

Note: if you need several services, you can ask for them at once, using the `findServices(ServiceFilter[] filters)` method. It is now possible to connect the `sinus` connector. Caution: the connector's connexion must be done **BEFORE** setting the connector's callback (listener).

```
ServiceProxy generator = service.findService(ServiceFilters.nameIs("generator"));
service.connectTo("point",  generator, "sinus");
service.addConnectorListener("point", graph);
service.start();
```

Finally, we need to analyse the incoming XML message (using the castor generated class: `point2d`) and to add the corresponding point to the graph display.

```
public void messageReceived(Service service, String connector, Message mesg) {
    try {
        Point2d point = Point2d.unmarshal(new StringReader(mesg.getBufferAsStringUnchecked()));
        series.add(point.getX(), point.getY()) ;
    } catch (MarshalException e) {
        e.printStackTrace();
    } catch (ValidationException e) {
        e.printStackTrace();
    }
}
```

7 Exposing the service status as a variable : the generator frequency

We are now going to expose the generator frequency with an O[3]MiSCID variable. This variable can be read. It can also be written to change the generator's frequency.

- Create a variable `w` with a `readWrite` access :

```
 ......
<variable name="w">
<access>readWrite</access>
</variable>
....
```

- An object can detect the value changes of this variable by implementing the `LocalVariableListener` interface.

```
public interface LocalVariableListener {

/**
 * This method is called when the value of a variable
 * changes
 * @param service the service owning the variable
 * @param var the information about the variable which value has changed
 */
  public void variableChanged(Service service, Variable var);

  /**
```

```
     * This method is called when a new value is request on a variable. This method must
     * check that this new value is a valid value.
     * @param service the service owning the variable
     * @param currentValue the current value of the variable
     * @param newValue the new requested value
     * @return true if the new value is accepted, false if rejected.
     */
    public boolean isValid(Service service, String currentValue, String newValue);
}
```

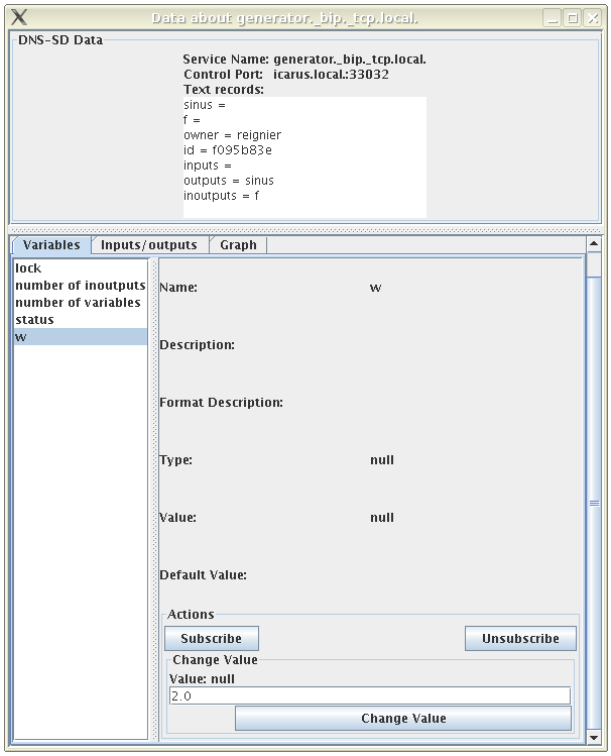
- Add `LocalVariableListener` to the interfaces implemented by the `SinusGenerator`
- Add `service.addLocalVariableListener("w", generator);` in the `start` method of the generator.
- `isValid()` must check if the new proposed value is correct (returns true if correct, false else). In this example, we will only check that the new value is a real : we will not check the range for instance.

```
public boolean isValid(Service service, String currentValue, String newValue) {
     try {
          Double.parseDouble(newValue) ;
     } catch (NumberFormatException e) {
          return false ;
     }
     return true ;
}
```

- Extract the new variable value and set the sinus frequency to this new value :

```
public void variableChanged(Service service, String currentValue, String newValue) {
   double w = Double.parseDouble(arg0.getValue()) ;
   setW(w);
}
```

- Republish the bundle. Opens the O[3]MiSCID GUI, select the variable `w` of the `generator` service. Modify its value. If the graph service is running, you should see the frequency modification.



8 The control

We are now going to implement the third service : the control slider

- Create a new Omiscid Project : `control`
- Copy the `WeightControl` class from the `curve` project to the `control` project.
- Modify the `WeightControl` class so that it receives a `ServiceProxy` in its constructor.
- In the `start()` method of `MainImpl` : search for a `generator` service.
- Construct a `WeightControl` with the found `generator`
- In the `stop()` method of `MainImpl` : hide and dispose the `WeightControl`.
- In the `stateChanged()` method (`WeightControl` constructor), get the new slider position and modify the `w` variable of the `generator` service :

```
public void stateChanged(ChangeEvent e) {
    double value = (double) slider.getValue()/100.0 ;
    generator.setVariableValue("w", new Double(value).toString());
}
```

9 The generator : answering requests

We are now going to complexify the `generator`. The generator must now offer a new *inoutput connector* : `f`. This connector can be used by other services to ask `generator` to compute the value of the function at a particular point.

- Add an inoutput connector `f` to the service definition (`service.xml`).

- Modify `SinusGenerator` to receive messages from the connector (do not forget to add the `SinusGenerator` instance as a listener for the connector).
- Create a new schema to handle XML messages like `<function x="0.0"/>`:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.example.org/function" xmlns:tns="http://www.examp
    <element name="function">
        <complexType>
            <attribute name="x" type="double"></attribute>
        </complexType>
    </element>
</schema>
```

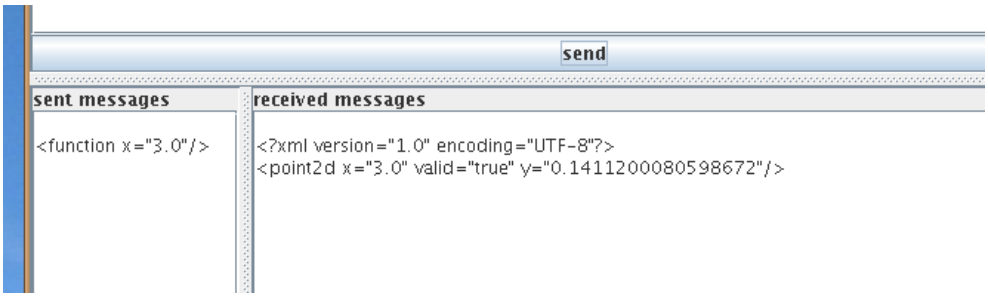
- Generate the corresponding Java code, extract the x parameter in the `messageReceived()` method and compute the fonction value at this point.
- Construct the `Point2d` answer.

We now need to send back the computed value to the service asking for it (the service which sent the message). The `Service` class proposed three `send` methods :

1. `sendToAllClients(String connectorName, byte[] msg)` : sends the message to all the clients listening to this connector (broadcast).
2. `sendToOneClient(String connectorName, byte[] msg, ServiceProxy serviceProxy)` sends to a particular client identified by its local representation (`ServiceProxy`).
3. `sendToOneClient(String connectorName, byte[] msg, int pid)` : sends the message to a particular client identified by its id (`pid`). When a service sends a message, its *id* is attached to the message, so that the recipient can send it back the answer using this method. This is the method that we will use.here

To be able to send a message from the `SinusGenerator` class, we will need the `service` instance (from the `MainImpl` class) :

- Add a `Service` attribute to the `SinusGenerator` class. Modify the `SinusGenerator` constructor to have an instance of `Service` as a parameter.
- Complete the `messageReceived()` method to send the `Point2d` object, answer of the request :
- Open the O[3]MiSCID GUI, select the `f` connector and open the manipulation window. Send an XML message and check the answer.



```
StringWriter out = new StringWriter() ;
Point2d p2d = new Point2d() ;
p2d.setX(function.getX());
p2d.setY(y);

p2d.marshal(out);
service.sendToOneClient("f", out.toString().getBytes(), mesg.getPeerId());
```

A The service specification XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www-prima.inrialpes.fr/s

    <element name="service">
        <complexType>
            <choice maxOccurs="unbounded" minOccurs="0">
                <element ref="omiscid:variable"></element>
                <element ref="omiscid:input"></element>
                <element ref="omiscid:output"></element>
                <element ref="omiscid:inoutput"></element>
            </choice>
            <attribute name="name" type="string" use="required"></attribute>
            <attribute name="docURL" type="string" use="optional"></attribute>
            <attribute name="class" type="string" use="optional"></attribute>
        </complexType>
    </element>

    <element name="variable">
        <complexType>
            <choice  maxOccurs="unbounded" minOccurs="0">
                <element name="description" type="string"></element>
                <element name="formatDescription" type="string"></element>
                <element name="value" type="string"></element>
                <element name="default" type="string"></element>
                <element name="type" type="string"></element>
                <element name="access" type="omiscid:accessType"></element>
            </choice>
            <attribute name="name" type="string" use="required"></attribute>
        </complexType>
    </element>

    <simpleType name="accessType">
        <restriction base="string">
            <enumeration value="read"></enumeration>
            <enumeration value="write"></enumeration>
            <enumeration value="readWrite"></enumeration>
        </restriction>
    </simpleType>

    <complexType name="connectorType">
        <choice maxOccurs="unbounded" minOccurs="0">
            <element name="description" type="string"></element>
            <element name="formatDescription" type="string"></element>
```

```
            <element name="tcp" type="int"></element>
            <element name="udp" type="int"></element>
            <element name="peers">
                <complexType>
                    <sequence maxOccurs="unbounded" minOccurs="0">
                        <element name="peer" type="hexBinary"></element>
                    </sequence>
                </complexType>
            </element>
            <element name="schemaURL" type="string"></element>
            <element name="messageExample" type="string"></element>
            <element name="peerId" type="hexBinary"></element>
        </choice>
        <attribute name="name" type="string" use="required"></attribute>
    </complexType>

    <element name="input" type="omiscid:connectorType"></element>

    <element name="output" type="omiscid:connectorType"></element>

    <element name="inoutput" type="omiscid:connectorType"></element>
</schema>
```