Nuptse tutorial

Matthieu ANNE

November 23, 2007

Abstract

1 Introduction

By following this tutorial, you will go through the basic concepts that you will inevitably need while developing your own Nuptse application. All these concepts are illustrated with concrete examples which can be found on ObjectWeb svn[1].

This document contains:

- basic example;
- presenting the use of fractal controllers
- Attribute controller
- LCC
- Binding controller
- component identity
- Abstract component
- composite component
- content controller

1.1 Configuration

To build the examples, you need to install the Nuptse environment and get a copy of the Kortex component library. The latest *ant* version[2] (at least the 1.7.0), *Java 5* and a *C* compiler are also required. You can get a copy of Nuptse and Kortex from the ObjectWeb svn[3]. `$THINK_PATH` and `$KORTEX_PATH` must

[1] To get a copy of examples used in this tutorial:

```
# svn checkout svn://svn.forge.objectweb.org/svnroot/think/experiments/mattBasis/nuptseBasis
```

[2] http://ant.apache.org/

[3] To get the Nuptse and Kortex trunk:

```
# svn checkout svn://svn.forge.objectweb.org/svnroot/think/trunk
# svn checkout svn://svn.forge.objectweb.org/svnroot/think/kortex/trunk/src
```

be defined according to the place you "checkout-ed" each project. Finally, you need to build a Nuptse distribution[4].

Once you have done or checked all these configuration aspects, you are ready to build and run each example described below. To do so, get into one of the examples directory, build the example using the default *ant* target (`# ant`), and then look in the `build/obj_unix` to run the example.

2 Basic example *(00helloWorld)*

The helloWorld example described here is the simplest example which serves as an example of reference. All the following examples are based on the same structure (cf. Figure 1) with different variations: a component called ***wtsh*** ("Wish To Say Hello") wants to say "Hello World" in several languages, others know how to do this. Therefore, components need to be bound to one another to accomplish this fundamental mission. You might find the description of this example a bit long, and getting into too much detail, but this is needed to make it possible to get straight into the interesting points in the next examples.

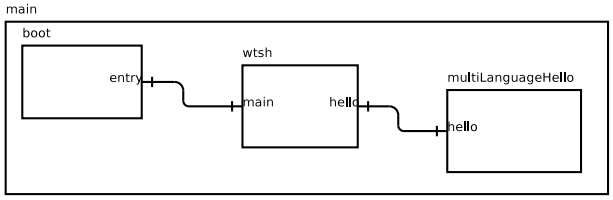

Figure 1: Architecture of 00helloWorld example.

The aim of this example is to point out the roles of the three types of files you will usually find in a Nuptse application. Those files correspond to *.adl*, *.idl* and *.c* files:

- *.adl* are used to specify components architecture, establishing the relationship between the components used to build the application;
- *.idl* gives a description of a required or provided interface, specifying the methods which are accessible through, or defined in, the interface;
- *.c* is the implementation associate to a component notably invoking and defining interface methods.

Three different programming languages are used for those files, which are respectively the **Architecture Description Language**, the **Interface Description Language** and **NuptC**. Through the description of the following examples some elements of those languages will be introduced. To have more information about these programming languages, please refer to the Think User Manual (TODO thinkUserMan Ref).

[4]To build a Nuptse distribution:

```
# cd $THINK_PATH/thinkadl
# build dist
```

2.1 Architecture description

The architecture is described using *.adl* files. One of this type of file is needed for each component. Therefore, one *adl* file will give a general description of the application architecture, and other *adl* files will give a description of components needed by the application.

TODO this is approximative...

2.1.1 Main architecture description

The general architecture of this first example (corresponding to the diagram shown in Figure 1) is described in the *main.adl* file (cf. Figure 2). This file specifies that the *main* component is composed of three components, namely `boot`, `wtsh` (wish to say hello) and `multiLanguageHello` components. Those components are respectively described in `boot.lib.boot`, `wishToSayHello` and `multiLanguageHello` *adl*'s. The first one comes from the Kortex library and will not be described here[5]. The two other components will be described below.

```
1   component main {
        contains boot = boot.lib.boot
        contains wtsh = wishToSayHello
        contains multiLanguageHello = multiLanguageHello
5
        binds boot.entry to wtsh.main
        binds wtsh.hello to multiLanguageHello.hello
    }
```

Figure 2: *main.adl*: description of the 00helloWorld example architecture.

The *main* architecture description also contains two binding descriptions. The client interface *entry* of the *boot* component is bound to the server interface *main* of the *wtsh* component, and the client interface *hello* of the *wtsh* component is bound to the server interface *hello* of the *multiLanguageHello* component. To be bound together, a client and a server interface must have the same type, that is to say they must share the same interface description (`.idl`). Client and server interface description of a given component is addressed in the next two sections.

2.1.2 *wishToSayHello* description

The *wishToSayHello* component is what we call a primitive component; this component doesn't contain any sub-components. The description of this component is given in *wishToSayHello.adl* (cf. Figure 3). This description file specifies that this component has one server interface called `main` and one client interface called `hello`. Those interfaces are respectively of `activity.api.Main` and `api.multiLanguageHello` type. As the `activity.api.Main` interface is

[5] If the application you've build never ends, it might be because the boot component you are using has a `while(1)` loop. This is because the Kortex library is meant to be use to build os's. You can still use your own boot component if you wish your application to end.

part of the Kortex library[6], it will not be discussed here. However, interface description will be discussed in section 2.2 based on `api.MultiLanguageHello` interface. Finally, `content wishToSayHello` specifies that the implementation of the wishToSayHello component can be found in *wishToSayHello.c*.

```
1  component wishToSayHello {
       provides activity.api.Main as main
       requires api.MultiLanguageHello as hello
       content wishToSayHello
5  }
```

Figure 3: *wishToSayHello.adl*: description.

2.1.3 *multiLanguageHello* description

As we can see in Figure 4, the *multiLanguageHello* component is even simpler. This component only has one server interface called `hello`. We can notice that this provided interface corresponds to the interface required by the *wishToSayHello* component (cf. Figure 3), both of them being of type `api.MultiLanguageHello`. The implementation of this component can be found in *multiLanguageHello.c* as specified by `content multiLanguageHello`.

```
1  component multiLanguageHello {
       provides api.MultiLanguageHello as hello
       content multiLanguageHello
   }
```

Figure 4: *multiLanguageHello.adl*: description.

2.2 Interface description

In this first example, two different types of interface are used. The first one, `activity.api.Main`, is defined in the Kortex library. The second one, `api.MultiLanguageHello`, is describe in *MultiLanguageHello.idl* (cf. Figure 5) and can be found in the *src/api/* directory. This *MultiLanguageHello* interface is very simple, providing one method for each required languages. The purpose of those methods is to print this universal greeting message, "Hello World", in different languages. Nevertheless, a `moreInfo` argument has been added, for example to be able to print out who wanted to say "Hello World", to increase the comprehension of the background mechanism at runtime.

2.3 Component implementation

Once we have done all the description part, describing architectures, describing interfaces; it is time for us to really get doing something. As there are three

[6]cf. `$KORTEX_PATH/generic/activity/api/` to get more details

```
package api;

public interface MultiLanguageHello {
    void printEnglishHello(string moreInfo);
    void printFrenchHello(string moreInfo);
    void printGermanHello(string moreInfo);
}
```

Figure 5: *MultiLanguageHello.idl*: description of the MultiLanguageHello interface.

primitive components, at least three implementations need to be defined. For the same reason as previously mention, the *boot* component implementation is not detailed here. The *multiLanguageHello* isn't detailed either for being too simple, but you can still have a look the *multiLanguageHello.c* file. So let's have a look at the implementation of the *wtsh* component (Figure 6).

```
/**
 * @@ DefaultClientMethods @@
 * @@ DefaultServerMethods @@
 */

/**
 * @@ PrivateData @@
 */
struct {
    unsigned int repeatNb;
} privInfo;

/**
 * @@ PrivateMethod @@
 */
void myPrivMeth(char *moreInfo) {
    printEnglishHello(moreInfo);
    printFrenchHello(moreInfo);
    printGermanHello(moreInfo);
}

void main(jint argc, jstring* argv) {
    int i;
    privInfo.repeatNb = 2;

    for (i = 0; i<privInfo.repeatNb; i++)
        myPrivMeth("Using private method");

    printEnglishHello("Direct call to client method");
    printFrenchHello("Direct call to client method");
    printGermanHello("Direct call to client method");
}
```

Figure 6: *wishToSayHello.c*: implementation.

After a quick look, any readers should notice some strange notations located

in commented regions of code. Actually, these are annotations that are used for example to determine to which interface a specific method belongs, or to rename a client method to disambiguate special situation. These annotations are specific to *NuptC* (TODO Ref Think User Man) and there are more of those annotations to come in the following examples. However, annotations are defined in comment sections between `@@` flags.

In the implementation of this example, a private data `privInfo` and a private method `myPrivMethod` are defined respectively using `PrivateData` and `PrivateMethod` annotation. Thus, these data and method can only be used locally. The three other annotations, that is to say `DefaultServerMethods` and `DefaultClientMethods`, define that server methods, and clients methods use default names. Default server or client methods names comes from the *idl* description. In this example, there aren't any ambiguous situations between client methods or server methods, so default names can be used. We will introduce later some cases where default names can't be used. Going further, if you need to define more than one private data, it must be done in `struct` structure. You can also notice that private methods can call methods defined in a client interface.

3 Attribute controller

Attribute manipulation and multiple instance.

- declaration dans l'adl
- declaration (annotation) dans .c
- initialisation dans adl
- initialisation dans .c
- attribute controller (modification des attribut d'un composant a partir d'un autre)

The description of this component also defines a component attribute called `repeatNbAtt` to which a specific value is assigned. This attribute is only used to repeat "Hello World" as many times it is defined (in case you did not understand).

Annotation `DefaultAttributes` defines that attributes use default names. Default attribute names comes from the *adl* description.

ambiguous situations between attributes names so default names can be used

4 Life cycle controller

Unlike other controllers, lcc bindings do not necessarily need to be explicitly defined. These bindings are manage at compilation time, and the `start` and `stop` methods it proposes are called only once respectively at the start and end of the runtime.

5 Binding controller

Binding description is one of the foundamental concept of the Fractal model[7]. Indeed, to enable a component to use the functionalities offered by a second, the two components need to be linked. As it has been previously described, this can be done through the architecture description (using *adl* files), doing it only this way defines statical bindings. However, some application might need dynamic bindings, ie the bindings between components might be defined and changed at runtime.

The following examples are based on the same structure shown in Figure 7. Here the application is composed of two instances of the *multiLanguageHello* component (ie *hello1* and *hello2*) to which a `name` attribute is added. A *wtsh* component requires two *hello* interfaces offered by *multiLanguageHello* components. The main architecture description defines the initial links (cf. Figure 8) which will dynamically change at runtime, swapping one *multiLanguageHello* component with the other one.

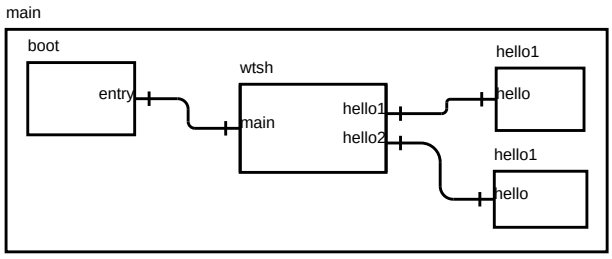

Figure 7: Representation of the helloworld example ADL.

```
1   component main {
        contains boot = boot.lib.boot
        contains wtsh = wishToSayHello
        contains hello1 = multiLanguageHello
5       contains hello2 = multiLanguageHello

        assigns hello1.name = "hello 1"
        assigns hello2.name = "hello 2"

10      binds boot.entry to wtsh.main
        binds wtsh.hello1 to hello1.hello
        binds wtsh.hello2 to hello2.hello
    }
```

Figure 8: *main.adl* description.

They are mainly two ways for dealing with dynamic binding. The first way is doing it locally, considering bindings methods, such as `bind` or `unbind`, as private methods. The second way is to require a binding controller interface and to bind to a component which provides this interface.

[7]TODO insert Fractal REF

5.1 Local binding management *02helloWorld*

In this example we decided to manage the dynamic bingings of `hello` interfaces required by the *wtsh* component (cf. Figure 9) directly inside the component itself. To do so, we use a set of keywords provided by *NuptC* to help implementing Fractal controller interfaces. These keywords corresponds to macros enabling to initialize implementation code at compile time considering the architecture, and to access or modify meta-data at runtime. All these keywords are prefixed with `META_`. To get more information on these keywords please refer to the Think User's Manual [8].

```
1   component wishToSayHello {
        provides activity.api.Main as main

        requires api.MultiLanguageHello as hello1
5       requires api.MultiLanguageHello as hello2

        content wishToSayHello
    }
```

Figure 9: *wishToSayHello.adl* description.

Let's have a look at the *wishToSayHello* implementation shown in Figure 10, starting with the annotations[9]. Here, the *wishToSayHello* component requires two interfaces of the same type (ie `api.MultiLanguageHello`), this result in an ambiguous situation. To determine which interface the implementation refers to, the `print`*Language*`Hello` methods are renamed as `hello1_print`*Language*`Hello` and `hello2_print`*Language*`Hello` using the `@@ ClientMethod(...) @@` annotation. Then the `bind` and `lookup` methods are defined as private methods using the `@@ PrivateMethod @@` annotation. These methods uses `META_CLT_ITF_SET` and `META_CLT_ITF_GET` macros to repectively set the server interface identifier to a given interface identifier, and get the server interface identifier of a given client interface identifier. Finally, invocations to those methods are done in the provide `main` function, implementing the functional code of the component *wtsh*.

Here the choice has been made to implements binding methods as private methods. However, as the `wtsh` implements binding methods, we could have decided that this component also provides a `BindingController` interface that could be offered to other components. This can be done by specifying the provided interface (`fractal.api.BindingController`) in the *wishToSayHello.adl* description, adding the `unbind` method in the implementation of this component, and deleting the `@@ PrivateMethod @@` annotations[10]. Doing it this way, bindings method can either be use locally or invoked by other components. If those methods are called locally, they will be assimilated to private methods, ——- A REVOIR —- so there is no need for this component to require a `BindingController` interface. This is not true anymore when the

[8] TODO give the REF of the Think user's manual

[9] cf fn8

[10] As the `@@ DefaultServerMethods @@` is used, there is no need to add more annotations.

```
/**
 * @@ ClientMethod(hello1, printEnglishHello, hello1_printEnglishHello) @@
 * @@ ClientMethod(hello1, printFrenchHello, hello1_printFrenchHello) @@
 * @@ ClientMethod(hello1, printGermanHello, hello1_printGermanHello) @@
 * @@ ClientMethod(hello2, printEnglishHello, hello2_printEnglishHello) @@
 * @@ ClientMethod(hello2, printFrenchHello, hello2_printFrenchHello) @@
 * @@ ClientMethod(hello2, printGermanHello, hello2_printGermanHello) @@
 * @@ DefaultServerMethods @@
 */

/**
 * @@ PrivateMethod @@
 */
jvoid bind(any clientItfId, any serverItfId) {
    META_CLT_ITF_SET(clientItfId, serverItfId);
}

/**
 * @@ PrivateMethod @@
 */
void* lookup(any clientItfId) {
    return META_CLT_ITF_GET(clientItfId);
}

void main(jint argc, jstring* argv) {
    any srvItfId1 = lookup(CLTID_hello1);
    any srvItfId2 = lookup(CLTID_hello2);

    hello1_printEnglishHello("using hello1 interface");
    hello2_printEnglishHello("using hello2 interface");

    // swapping interfaces
    bind(CLTID_hello1, srvItfId2);
    bind(CLTID_hello2, srvItfId1);

    hello1_printFrenchHello("using hello1 interface");
    hello2_printFrenchHello("using hello2 interface");

    // swapping interfaces
    bind(CLTID_hello1, srvItfId1);
    bind(CLTID_hello2, srvItfId2);

    hello1_printGermanHello("using hello1 interface");
    hello2_printGermanHello("using hello2 interface");
}
```

Figure 10: *wishToSayHello* implementation. TODO `any`, `jvoid` and `CLTID_` keyword

`BindingController` interface is implemented in another `content`, as it will be done in section 5.3 and which impose an *auto-binding* structure.

5.2 Using the BindingController interface *03helloWorld*

In this section we introduce how to re-use existing interface implementations and how binding methods offered by another component are used. In the example

described here, the *wtsh* component is splitted in two separate components (cf. Figure 11) to distinguish two functional aspect. The first one, *stammer*, having to manage the number of *"Hello World"* double repetitions and to initiate the dynamic binding changes of the second one, *wtsh*. Therefore, the *wishToSay-Hello* component provides two interfaces, one to say double hello's and another one to swap its bindings with *multiLanguageHello* components. Offering a binding controller interface the *wishToSayHello* component doesn't have to deal with binding anymore, regaining its original function which is only to ask to say Hello to the World which is already a hard job.

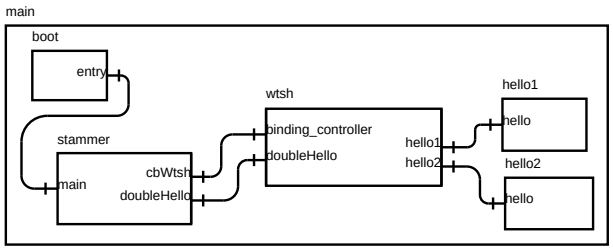

Figure 11: Representation of the helloworld example ADL.

You can have a quick look to the *main.adl* file corresponding to this example, but nothing is really new here. However, going through the *wishToSayHello.adl* description (cf Figure 12), you will find a new structure to describe provided interface and associate implementation. This corresponds to the `provides ... as ... in ...` and `content ... for ...` structure[11]. This structure enable to define a compoment from multiple implementations. In this case, the *wishToSayHello* component is build from its default implementation (`content wishToSayHello`) to which the implementation of the `BindingController` interface is added. This implementation can be found in `fractal.lib.bcstring` and is renamed as `BindingController`.[12]

5.3 Local use of the BindingController interface *04helloWorld*

We illustrate here an evolution of the *02helloWord* discuss in section 5.1. In this example we introduce a commonly used and useful structure which is the auto-binding structure. This structure enables a component to locally use a required interface implemented in the same component but in a different content.

[11] TODO Liste the limitation of the multiple content approach

[12] dire qu'on peut jeter un oeil a bcstring et quon y retrouve les `META_` macros

```
component wishToSayHello {
    provides fractal.api.BindingController as binding_controller
                                              in BindingController
    provides api.MultiLanguageHello as doubleHello

    requires api.MultiLanguageHello as hello1
    requires api.MultiLanguageHello as hello2

    content wishToSayHello
    content fractal.lib.bcstring for BindingController
}
```

Figure 12: *wishToSayHello* architecture description

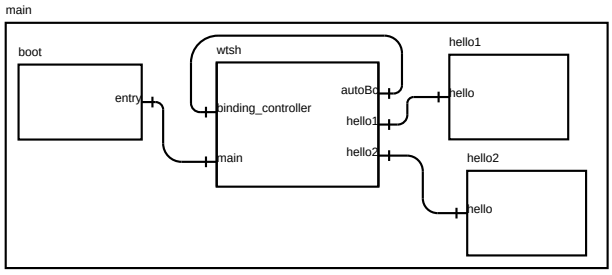

Figure 13: Representation of the helloworld example ADL.

6 Component identity *05HelloWorld*

In the previous example discussed in section 5 the *wishToSayHello* component had two `api.MultiLanguageHello` interfaces, one for each *multiLanguageHello* component. The objective is how to enable the *wishToSayHello* component to alternatively share one client interface with several components. Even if it was quite amusing (at least I hope that it wasn't too boring) to play with multiple instances of identical components. We are now going to build a more relevant multi-language HelloWorld application architecture.

Instead of having several printing methods in one *multiLanguageHello* component, the idea is to have several components (*prefixHello*, eg. *englishHello*, *frenchHello* ...), each of them knowing how to say Hello World in different languages. Each of those *prefixHello* components provides an identical *hello* interface (`api.SayHello`) with one `printHello` method. The *wtsh* component can then be bound indifferently to one or another (cf. Figure 14). The *wtsh* component doesn't have to worry about the language used, it just has to ask to say Hello. Actually, here the *wtsh* asks to say hello but also to change the current language, however the language change could be manage by other components.

In the previous section we used the `lookup` method provided by the *bindingController* interface to get all the client interfaces required by a component. We now need to retrieve server interfaces provided by a component so that a client interface could bind to them. This can be done using the component identity

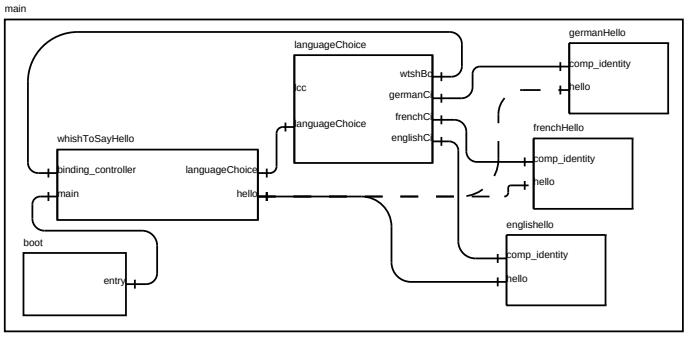

Figure 14: Representation of the helloworld example ADL.

interface specified in the Fractal model (TODO not sure...).

6.1 *wtsh*

TODO A revoir depuis multiLanguageHello

There aren't any foundamental changes in the architecture description of the *wtsh* component (cf. wishToSayHello.adl file). This component has now only one `hello` interface of type `api.SayHello`, and a required interface `languageChoice` of type `api.LanguageChoice` has been added. As it is specified in figure 15, the interface description provides only one method called `changeLanguageTo` with one argument to specify which is the targeted language.

```
1  package api;

   public interface LanguageChoice {
       void changeLanguageTo(string language);
5  }
```

Figure 15: *LanguageChoice* interface description.

6.2 *languageChoice*

The *languageChoice* component is central to this architecture. Its role is to manage the bindings between the *wtsh* and *prefixHello* components. To do this, *languageChoice* needs to be aware of the `hello` interfaces provided by *prefixHello* components (`BindingController` interfaces). The *languageChoice* component also has to know when *wtsh* requires to change the current language (`api.LanguageChoice` interface), and finally it needs to invoke the binding methods of the *wtsh* component to dynamicaly swap from one *prefixHello* to another (`fractal.api.BindingController`).

Here a life cycle controller (lcc) server interface is added. The `start` method is used to get the i*hello* interfaces' ids when the application startes. In the `start` method, invocation to the `getInterface` methods provided by the `comp_identity`

```
component languageChoice {
        provides api.LanguageChoice as languageChoice
        provides fractal.api.LifeCycleController as lcc

        requires fractal.api.BindingController as wtshBc
        requires fractal.api.ComponentIdentity as englishCi
        requires fractal.api.ComponentIdentity as frenchCi
        requires fractal.api.ComponentIdentity as germanCi

        content languageChoice
}
```

Figure 16: *languageChoice.adl* description.

interfaces are made. As nothing has to be done while the application stopes, the `stop` method is an empty method.

Then, the `changeLanguageTo` implementation is quite simple. Considering the *wtsh* language choice, the *languageChoice* component binds the *wtsh hello* interface to the corresponding component.

Even if this approach seems to be more flexible, the *languageChoice* component still needs to have one *comp_identity* interface for each *prefixHello* component. The probleme has only been shifted from the *wtsh* to the *languageChoice* component. In section 8 we will see how to avoid this in order to become even more flexible.

6.3 *prefixHello*

In this example, *prefixHello* components description are modified to add a *comp_identity* interface. As shown in figure 18, the *prefixHello* architecture description specifies that the implementation of this interface can be found in `fractal.lib.ci`. Nothing else needs to be done here.

The *05HelloWorldEBis* example proposes a variation on how to build *prefixHello* components introducing the `abstract component` concept. In this example, all *englishHello*, *frenchHello* components etc... extends an abstract component description named *hello*. The *hello* adl (cf. Figure 19) contains all previously provided interfaces, and where to find the *ComponentIdentity* implementation. The only part missing is which implementation to use for the `api.SayHello` interface. This implementation is then specified in each particular *prefixHello* adl description (see the *englishHello.adl* description in Figure 20).

7 Composite components

To get keep on going with controllers, we need to introduce here the composite component concept. The Fractal model specifies two types of components, namely *primitive* and *composite* components. So far, we have only used primitive components, but the possibility of using composite components drastically increases the structuring capabilities of the Fractal model, and thus the Think framework.

```
#include <libc/string.h>
/**
 * @@ PrivateData @@
 **/
struct {
    any english_helloItfId;
    any french_helloItfId;
    any german_helloItfId;
} info;

/**
 * @@ DefaultServerMethods @@
 * @@ ClientMethod(englishCi, getInterface, english_getInterface) @@
 * @@ ClientMethod(frenchCi, getInterface, french_getInterface) @@
 * @@ ClientMethod(germanCi, getInterface, german_getInterface) @@
 */
void start() {
    info.english_helloItfId = english_getInterface("hello");
    info.french_helloItfId = french_getInterface("hello");
    info.german_helloItfId = german_getInterface("hello");
}

void stop() {}

/**
 * @@ DefaultClientMethods(wtshBc) @@
 **/
void changeLanguageTo(char *language) {
    if (!strcmp(language,"English")){
        bind("hello", info.english_helloItfId);}
    else if (!strcmp(language,"French")){
        bind("hello", info.french_helloItfId);}
    else if (!strcmp(language,"German")){
        bind("hello", info.german_helloItfId);}
    else printf ("unknown language\n");
}
```

Figure 17: *languageChoice.c* implementation TODO REf a l'implem de l'anguageChoice.c.

```
component englishHello {
    provides api.SayHello as hello
    provides fractal.api.ComponentIdentity as comp_identity
                                        in ComponentIdentity

    content englishHello
    content fractal.lib.ci for ComponentIdentity
}
```

Figure 18: *prefixHello* architecture description

Composite components are formed out of two parts: a membrane and a content. The content corresponds to a set of other components called *sub-components*, which are under control of the enclosing component. The mem-

```
1   abstract component hello {
        provides api.SayHello as hello
        provides fractal.api.ComponentIdentity as comp_identity
                                            in ComponentIdentity
5
        content fractal.lib.ci for ComponentIdentity
    }
```

Figure 19: *hello* abstract architecture description.

```
1   component englishHello extends hello {
        content englishHello
    }
```

Figure 20: *prefixHello* architecture description extending *hello* component.

brane can have external and internal interfaces. External interfaces are accessible from outside the component, while internal interfaces are accessible only from the component's sub components.

The use of composite is shown in the next two examples. The first one is a basic example where a composite component contains one primitive component. In this example we will see how interface methods invocation goes through the composite membrane (TODO Definition of the membrane). The second example is mostly based on the same architecture, but showing how to put fonctional code in the membrane (TODO I'm not sure that it is in adequation with the Fractal model).

7.1 Basic composite example *06helloWorld*

The general aspect of this example architecture is shown in figure 21. The only change with the referent example is that now the *multiLanguageHello* component is included in a *helloComposite* component. So we are only going to describe how this component is build.

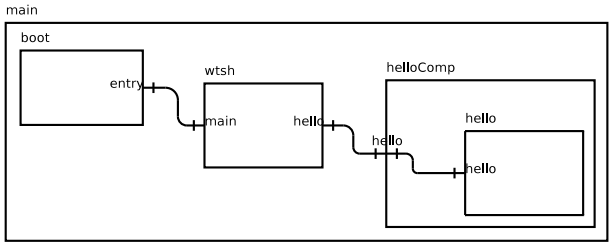

Figure 21: Representation of the helloworld example ADL.

As shown in figure 22 this component has the same interfaces as the *multiLanguageHello* component, that is to say it only provides an `api.MultiLanguageHello`

interface. But this component differs from the *multiLanguageHello* component by containing the *multiLanguageHello* component and by not specifying any implementation associated to this composite component. To enable it to accomplish its Hello World mission through the use of the external `hello` server interface, we need to specify in its adl that the corresponding internal interface (which is a client interface) needs to be bound to the `hello` interface of the *multiLanguageHello* component. This is done by binding the internal `hello` client interface, associated to the external `hello` server interface of the component *helloComposite*, to the `hello` server interface of component *multiLanguageHello*. This is specified in the *helloComposite* adl with `binds this.hello to hello.hello` (`this.hello` being the internal `hello` client interface of component *helloComposite*). From now, when the `hello` interface of the *helloComposite* composite is invoked, this invocation is transmitted through the internal `hello` interface to the `hello` interface of the *multiLanguageHello* component.

```
1   component helloComposite {
        provides api.MultiLanguageHello as hello
        contains hello = multiLanguageHello
        binds this.hello to hello.hello
5   }
```

Figure 22: *helloComposite* architecture description

7.2 Membrane implementation *07helloWorld*

TODO The composite membrane says hello too ....

In the *Nupse* implementation of the Fractal component model, composite components can still contain functional code. This code is then contained in the component membrane. As shown in figure 23, in this example the *helloComposite* provides an interface which is not transmited to a sub component. All of these interface methods are implemented in the component membrane.

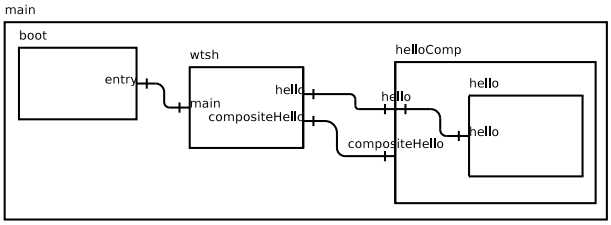

Figure 23: Representation of the helloworld example ADL.

As shown in figure 24 the *helloComposite* component provides two `api.MultiLanguageHello` interfaces. This first one has its corresponding internal interface bound to the *multiLanguageHello* sub component. The second one has his interface implementation in the *helloComposite.c* file as specified by `content helloComposite`. The implementation of a membrane is identical to the implementation of a primitive component.

```
component helloComposite {
    provides api.MultiLanguageHello as hello
    provides api.MultiLanguageHello as compositeHello

    contains hello = multiLanguageHello

    binds this.hello to hello.hello

    content helloComposite
}
```

Figure 24: *helloComposite* architecture description

8 Content controller *11helloWorld*

Now that we've presented the composite component concept, we introduce here the content controller which makes it possible to manage sub components. The architecture represented in figure 25 has several similitudes with the architecture of the example *05helloWorld*, presented in section 6. *wtsh*, *languageChoice* and *prefixHello* can be retreived here, but this time they are all sub components af a component called *helloWorld*. However, interfaces and bindings of those components are quite the same as in *05helloWorld*.

The major difference is located in the *languageChoice* component. Instead of having one `fractal.api.ComponentIdentity` interface for each *prefixHello* component, there is only one of this type of interface. Moreover, an interface of `fractal.api.ContentController` has been added. We also need to introduce here the *dummy* component which is a component similar to *prefixHello* components (extending the *hello* asbtract component). This component has two different roles, the first is to be used as default component when there isn't any component satifying the required language. The second role is justified by compilation reasons, if the client `hello` interface from the *wtsh* component is not bound to something at compilation time, this interface can not be used (TODO optimisation reasons, is the keyword `optional` is to be used here).

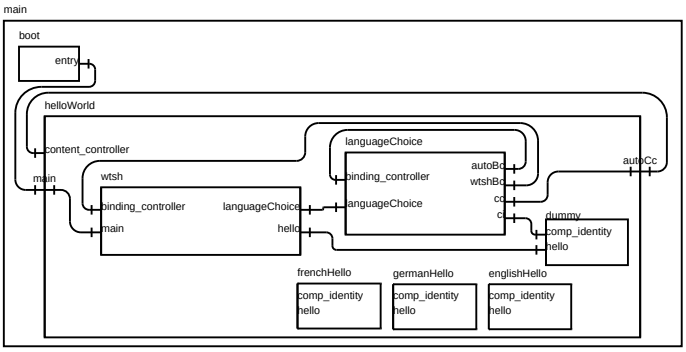

Figure 25: Representation of the helloworld example ADL.

We are now focusing on the content introspection problem. There is no

way of directly getting the *prefixHello* interfaces. To discover *prefixHello* components and their interfaces, the *languageChoice* component needs to use the `content_controller` interface of the *helloWorld* component to get the sub component identity interface of the component satisfying the required language (cf. Figure 26)(TODO need some info explaining the reason of: `x_ci = x_ci + sizeof(void *);`). Once this interface is found, the `getInterface` method is used to get the coresponding *hello* interface. Finally the `hello` server interface is bound to *wtsh* coresponding client interface using the `wtshBc` interface.

```
#include <libc/string.h>
/**
 * @@ ClientMethod(autoBc, bind, autoBind) @@
 * @@ ClientMethod(wtshBc, bind, wtshBind) @@
 * @@ DefaultClientMethods(cc, ci) @@
 * @@ DefaultServerMethods @@
 */

void changeLanguageTo(char *language) {
    any x_ci;
    any srv_itf;

    if (!strcmp(language,"English")){
        x_ci = getSubComponent("main.helloWorld.englishHello");}
    else if (!strcmp(language,"French")){
        x_ci = getSubComponent("main.helloWorld.frenchHello");}
    else if (!strcmp(language,"German")){
        x_ci = getSubComponent("main.helloWorld.germanHello");}
    else {
        printf ("unknown language\n");
        x_ci = getSubComponent("main.helloWorld.dummy");}

    x_ci = x_ci + sizeof(void *);
    autoBind("ci", x_ci);
    srv_itf = getInterface("hello");
    wtshBind("hello",srv_itf);
}
```

Figure 26: *languageChoice* implementation

This approach has several avantages primarily based on the flexibility it provides. Actually the flexibility comes from the fact that in this architecture very little previous knowledge is needed on the number and type of *prefixHello* components. This results in the possibility of easely adding more components able to say hello in even more languages. Going further, as it will be presented in a future tutorial, more *prefixHello* components can be added at runtime.

A Overview of examples

- 01helloWorld [**basic example**]: one component asks a second one to say hello;
- 02helloWorld [**binding controller use**]: two components know how to

say hello and a third one asks them to say hello sequentially through two separated interfaces, then the bindings are swapped;

- 03helloWorld [**sharing a binding controller locally and remotely**]: similar to the previous example but now the binding controller is used either locally, either called by another component through the binding controller interface;
- 04helloWorld [**component made of several contents**]: this example is similar to the previous one but this time we use one content for the hello interface and a second one for the binding controller interface;
- 05helloWorld [**dynamic binding**]: three components know how to say hello in three different languages and a fourth component dynamically binds to one or another depending on the language in which he wants to say hello. The bindings are controlled by a fifth component.
- 06helloWorld [**composite structure**]: similar to the previous example, but this time the component which knows how to say hello is included in a composite component;
- 07helloWorld [**membrane implementation**]: similar to the previous example, but now both the composite and the included component know how to say hello;
- 08helloWorld [**membrane implementation**]: similar to the previous example using several contents for the composite and an auto binding structure;
- 09helloWorld [**content controller**]: ;
- 10helloWorld [**content controller**]: ;
- 11helloWorld [**content controller**]: ;