Unified Object Bus

Development Guidelines

Manuel Roman (mroman1@cs.uiuc.edu)
Software Research Group
University of Illinois at Urbana-Champaign

Table of Contents

1. Implementation guidelines for Base Components

This section explains the steps required to create a Base component using C++. The explanation uses the HelloWorld base component as an example.

Before starting with the explanation, it is important to note that a base component is the simplest type of component that can be created. It cannot be addressed remotely, though its dependencies can be manipulated from any host in the system by using the standard tools offered by the Unified Object Bus.

BaseComponents are used to build infrastructure required by other components. For example, in the case of a componentized CORBA ORB, the pieces of the ORB would be base components.

Base components must inherit from the "BaseComponent" base class and redefine, if required, any of the virtual methods (*createLocalConfigurator*, *init*, *Main* or *finish*). Some components may not need any special behavior while being created and destroyed. Therefore they will not redefine any method and will simply use the default behavior defined by Base Component, which is an empty implementation. Table 1 shows the C++ interface of the HelloWorld component.

```
class ClassExport HelloWorld: public BaseComponent
{
   public:
            HelloWorld (){};
            ~HelloWorld (){};

            int init(int argc, char **argv);
            void sayHello() {printf("Hello World\n");}
 };
```

Table 1. C++ interface of the HelloWorld Component

The HelloWorld component does not require any special behavior for destruction and it does not require being active either. However, when the component is initialized, it registers itself in the domain configurator. Therefore the component redefines the init method and uses the default implementation for the main, finish and createLocalConfigurator methods.

The *ClassExport* keyword is a macro that exports all the methods of the class. This is required for all components so other components can access their methods.

In the .cpp file that contains the code for the methods of the component, a macro has to be used to automatically generate the factory method for the component. This macro is called COMPONENT_FACTORY_DECLARE(*Component*), where *Component* is the

name of the class of the component. In the case of the HelloWorld component, the macro is used in this way: COMPONENT_FACTORY_DECLARE(HelloWorld).

When compiling the components for Windows environments, it is required to enable RTTI (Run-Time Type Information). This is important for specific component managers, so they can check that the component being created belongs to that component model type. The RTTI option can be found in Visual C++ in: Projects/Settings/ C/C++ / C++ Language / Enable RTTI.

Also, depending on the target platform, different macros have to be declared. Windows environments (except Windows CE) require WINNT. Windows CE requires WINCE and finally, Solaris requires SOLARIS.

The component has to be compiled as a DLL, and once the DLL is obtained it has to be put in a directory contained in the PATH, so the loader can find it. In order to be able to link your project, you need to link the component with the LocalComponentConfigurator.lib, the ComponentManager.lib and OSDependent.lib

Table 2 contains the implementation of the HelloWorld component.

The init method starts retrieving a pointer to the domain configurator by using the method *getDomainConfigurator* (line 4). The BaseComponent class defines this method. Next, the init method checks whether or not the obtained pointer is valid (line 6). If the pointer is not valid, the method prints an error message using the standard printToConsole method (line 8) and returns a –1, which implies that an error occurred. If the pointer to the domain configurator is valid, the init method adds a new hook to the domain configurator (line 12) and attaches the HelloWorld component configurator to the domain configurator (line 14). Once the component configurator has been attached, any other component can access the HelloWorld component by obtaining its pointer from the HelloWorld hook.

After building the DLL, the component is ready to be instantiated in the Unified Object Bus.

```
1.int HelloWorld::init(int argc, char **argv)
2.{
3.   ComponentConfigurator  *cc;
4.   cc=getDomainConfigurator();
5.
6.  if(cc==NULL)
7.  {
8.          OS::printToConsole("Error trying to retrieve the Domain Configurator");
9.          return –1;
10. }
11.
12. cc->addHook("Hello World");
13.
14. cc->hook("Hello World",this->getLocalConfigurator());
15.
16. return 0;
17.}
18.
19.COMPONENT_FACTORY_DECLARE(HelloWorld);
```

Table 2. C++ implementation for the HelloWorld component.

2. Implementation guidelines for CORBA Components

The implementation of a CORBA component starts with the definition of the component's interface in IDL. This IDL interface has to inherit from CORBAComponent. Table 3 contains the IDL interface of the HelloWorld CORBA component.

```
#include "twoKComponent/twoKComponent.idl"

interface HelloWorld: CORBAComponent
{
        void SayHello();
};
```

Table 3. IDL for the HelloWorld CORBA component.

After compiling the IDL interface, the servant class has to be defined. This class follows the standard CORBA rules (inheriting from the POA_HelloWorld class created by the IDL compiler) but it also inherits from the CORBAComponent_i. Table 4 shows how to inherit from both classes. Note that the order of inheritance is important.

```
class HelloWorld_i: public CORBAComponent_i, public virtual POA_HelloWorld
{
        . . .
};
```

Table 4. Definition of the HelloWorld_i C++ class.

Note that virtual inheritance is only used with the POA_xxx base class, but no with the CORBA Componen_i class.

Table 5 shows the implementation for the HelloWorld CORBA Component.

```
void HelloWorld_i::SayHello (CORBA::Environment &ACE_TRY_ENV)
                ACE_THROW_SPEC ((CORBA::SystemException))
{
        OS::printToConsole("Hello World!!");
}

COMPONENT_FACTORY_DECLARE(HelloWorld_i);
```

Table 5. C++ Implementation of the HelloWorld_i class.

Since CORBA Components inherit from Base Component, the rules for init, Main, finish and createLocalConfigurator still apply. CORBA Components define a new method called *createDistributedConfigurator*. This method allows the component to create a customized version of a distributed component configurator. However, if the component does not require any customized configurator, it can simply use the default implementation.

CORBA components also have to use the COMPONENT_FACTORY_DECLARE macro to generate the factory method that is used to instantiate the components.

Finally, once the object methods have been implemented, registering the object with the ORB is not required, since it is done by default by the CORBA Compnent Manager.

3. Starting a component container

Starting component containers requires launching an executable (ComponentContainer) providing a set of parameters in a specific order. The first parameter is the name of the component container. Next comes a list of component managers followed by the object request brokers. Finally comes the list of generic components that we want to start in the component container. After the name of each provided component, it is possible to define parameters, which will be sent to the component. Parameters simply have to be preceded by a hyphen.

When building systems based on the unified object bus, users are not likely to start component containers manually. Component containers will be created automatically by specialized services. In the case of Gaia OS, this task is performed by the resource manager service, which uses information about the status of the hosts in the system to decide where to start new component containers.

Next is an example that starts a component container:

```
componentcontainer   SYSTEM
                     Base/BaseComponentManager
                     Base/CORBAComponentManager
                     Base/TAOExporter
                     Base/Console -debug
                     Base/HTTPBroker –p20000
                     CORBA/CORBAComponentManagerExporter
```

The previous component container uses a Base Component Manager, a CORBA Component Manager, a TAOExporter as the ORB, an HTTPBroker that allows browsing the status of the Object Bus by using a Web browser and finally a CORBA Component Manager Exporter (more details in section 3 in this document).