Unified Object Bus

Specific Component Managers

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

Table of Contents

Specific component managers are responsible for defining how to create, delete and activate components of a specific type. All specific component managers inherit from Object Life Cycle and must register with the Component Manager, which will be responsible for delegating incoming requests.

1. Base Component Manager

The base component manager implements the methods to create, activate and delete base components. Base Components are the native components of the unified object bus, can be dynamically manipulated and define a minimum interface. These base components are in most of the cases building blocks that will be used to construct more complex component models. Because of this inherent simplicity, base components do not know about interface negotiation and communication with remote components (i.e., components outside their component container). The reason is that not all component models and clients strictly require this functionality. Therefore it is delegated to subclasses of the base component or to specialized services.

```
class BaseComponent
{
        private:

        ComponentConfigurator *localConfigurator_;
        ComponentConfigurator *domainConfigurator_;

        //Unique Component ID.
        char UCR_[MAX_UCR];

        protected:

        BaseComponent();

        public:

        ~BaseComponent();

        virtual int init(int argc, char **argv);
        virtual void finish();
        virtual void main();

        virtual void createLocalConfigurator();
        void setLocalConfigurator(ComponentConfigurator *lc);
        ComponentConfigurator *getLocalConfigurator();

        void setDomainConfigurator(ComponentConfigurator *cc);
        ComponentConfigurator *getDomainConfigurator();

        void setUCR(char *UCR) ;
        void getUCR(char *outUCR);

        void getInfo(char *info);
        int getInfoLength(){return 0;}
};
```

Every base component has an associated component configurator, which strictly manages dependencies with other components. Component configurators provide the unified object bus with reflective behavior. It is possible to introspect the dependencies of every base component and use this knowledge to modify these dependencies. The base component becomes the base level in the reflective system while the component configurator is the meta-level. All the strategies to enforce dependency integrity are encapsulated in the component configurators, which are specialized by means of inheritance according to every particular base component.

Table 1. Base Component Interface

Whenever a base component is created, the base component manager checks whether or not the component created a specialized component configurator. If it did not, the base component manager automatically creates a default one.

The interface of the base component class consists of twelve methods. Current implementation of the component management core is C++ based. However, it is possible to implement it in other languages, such as Java, just by following the interfaces provided. Table 1 contains the interface of the base component.

When a component is created, the first method that is invoked is the *createLocalConfigurator*. This method allows the developer of the specific base component to create a customized component configurator and assign it to the internal *localConfigurator_* pointer by calling *setLocalConfigurator*. If no component configurator is created, the base component manager assigns a default one. Next, the component management core creates a **Unified Component Reference** (UCR, check section 3) and assigns it to the component. Once the component has a component configurator and a reference, the domain configurator is assigned to the component. This domain configurator allows components to access components executing in the same component container. After this internal initialization phase, the *init* method is called and the component can initialize itself. If an error occurs during the initialization, the method must return a –1, which will force the base compent manager to destroy the component. If the initialization is successful, the m*ain* method is associated a thread and therefore the component is activated. Note that some components will not require being activated. For example, a time component will simply inherit from base component and define a method to return the current time. Therefore the main method will be empty. Other components will invoke the method from the time component, but the component in itself will be inactive. The *getInfo* method returns a string with information about the component. Finally, the component management core calls the *finish* method when the component is about to be destroyed.

All the methods have a default implementation. Therefore, if a component does not require neither a customized component configurator nor initialization and finishing mechanisms, then none of those methods have to be defined. Only the new methods introduced by the new component will have to be defined.

1.1 Base Component Creation

When the base component manager is initialized, it obtains a pointer to the component manager, so it can get access to its functionality and especially to the UOBLoader.

When the base component manager receives a request to create a component, it uses the UOBLoader to load the component. The UOBLoader is responsible for: (1) loading the component factory, (2) maintaining a list of loaded factories and the instance number assigned to the last component created by each factory and (3) creating the component by invoking a method on the factory. Once the loader is done, the Base Component Manager requests the new component to create a component configurator. If the component does

not create a component configurator, then the base component manager creates one by default. Next the base component manager creates and assigns a UCR to the component and calls the init method on the component, passing the arguments sent by the client. If the component fails during the initialization, it returns a –1 and the base component manager destroys the component. If no error is detected, the base component manager creates a hook in the "Created Components" configurator (using the UCR for the name), hooks the newly created component to it and returns a pointer to the object to the client. Figure 1 illustrates de creation process of a component C belonging to the component model "Base".

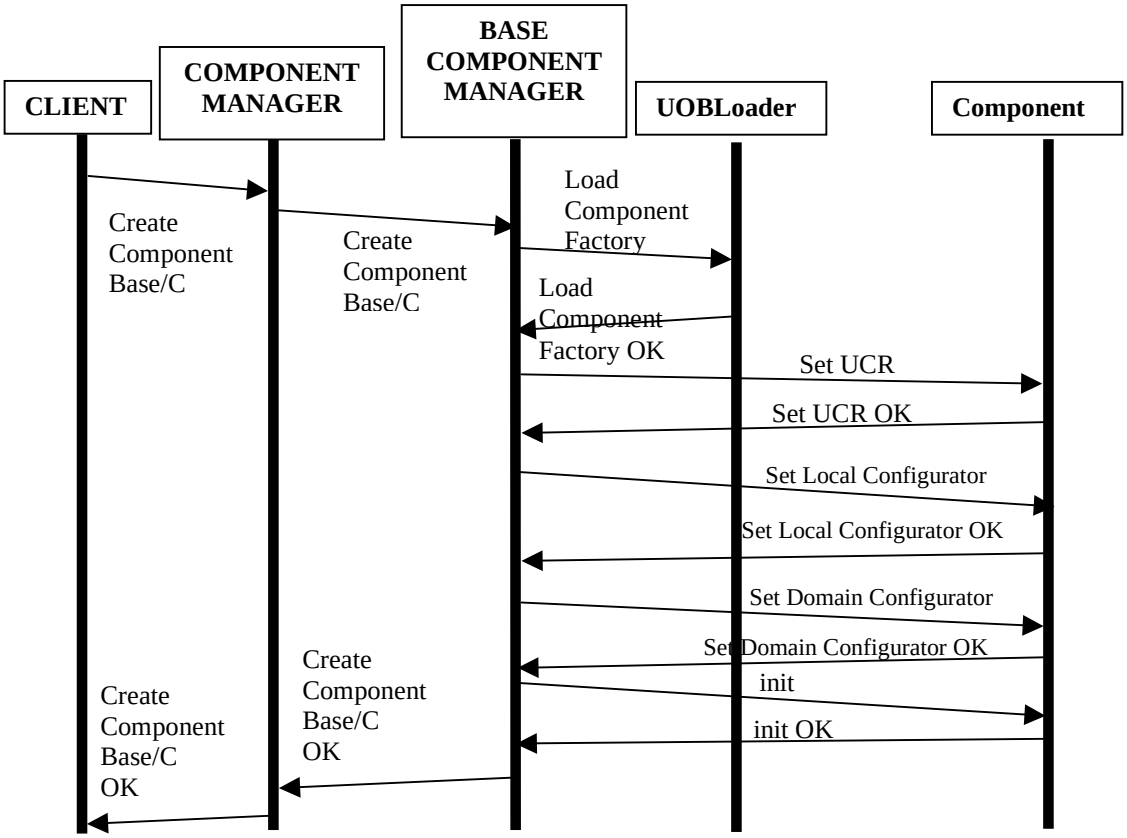

Figure 1. Creation of a Base Component

1.2 Base Component Destruction

Figure 2 contains the destruction diagram for a base component.

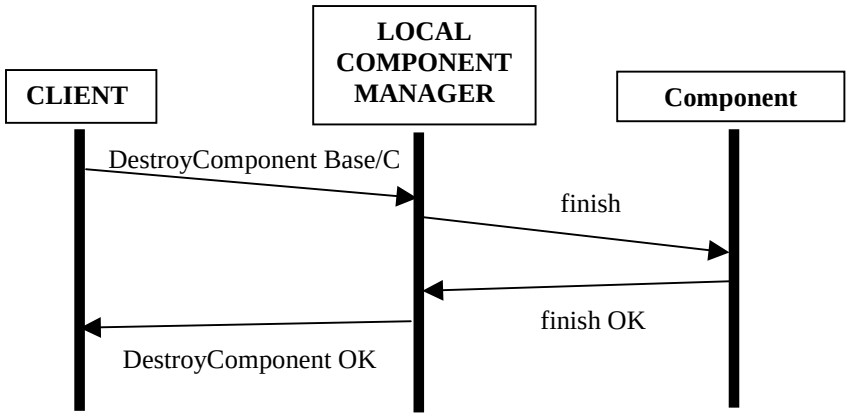

Figure 2. Destruction of a Base Component.

1.3 Base Component Activation

The base component manager creates a thread and associates it to the created base component. This thread executes the code contained in the main method of the base component. It is possible, though, to have inactive base components by simply not defining the Main method of the base component.

2. CORBA Component Manager

The CORBA Component Manager integrates CORBA objects in the Unified Object Bus. The CORBA Component Manager specifies the standard shipping format of CORBA objects, therefore defining CORBA components. By leveraging the functionality exported by the component manager CORBA components can be dynamically manipulated, which contrasts with the static behavior of CORBA objects.

The CORBA Component Manager provides functionality to create, delete and activate CORBA components. It also automates standard tasks such as CORBA object registration with the CORBA ORB and CORBA object registration with the naming service. Therefore developers simply have to implement the methods of the CORBA object defined in the IDL interface and the CORBA Component Manager will take care of the rest. Because of the integration of CORBA components in the Unified Object Bus, these components are assigned a Unified Component Reference; therefore their specific details are hidden from applications.

The CORBA Component Manager and CORBA components use CORBA as the standard ORB. As defined in section 2.3 of the Unified Object Bus document, ORBs are introduced in the system as components; therefore can be instantiated in any component container at run-time. We are currently using TAO [2] as the default CORBA ORB. However, other CORBA ORBs can be used without affecting the implementation of existing CORBA Components. The standard mechanism followed by components to

obtain the ORB is by using the domain configurator. Clients get the pointer to the domain configurator, and then they get the specific type of ORB they require by accessing the right hook.

The CORBA Component Manager defines CORBA Components by specifying their interfaces in CORBA IDL. CORBA Components inherit from BaseComponent so they can be dynamically manipulated. Table 2 shows the IDL interface, which defines an attribute called distributedConfigurator and a method called createDistributedConfigurator. The attribute is used to keep track of dependencies with remote objects and implements the component configurator pattern. This attribute is similar to the local component configurator of base components but is used in distributed environments. The CORBA Component Manager calls the *createDistributedConfigurator* method during the initialization of the CORBA component. The objective is to allow the component to create its own specialized version of this object.

```
interface Gaia::UOB::CORBAComponent
{
        attribute Configuration::ComponentConfigurator distributedConfigurator;
        void createDistributedConfigurator();
};
```

Table 2. The CORBAComponent Interface

2.1 CORBA Component Creation

The CORBA Component Manager leverages the functionality implemented by the Base Component Manager. Therefore, the CORBA Component Manager inherits from the Base Component Manager and specializes it with extra functionality relevant only to CORBA Components.

When the CORBA Component Manager is initialized, it looks for a CORBA ORB registered in the domain configurator. If there is not any available, it automatically creates one. The CORBA Component Manager uses the CORBA ORB to automatically register CORBA objects.

When creating a CORBA component, once the Base Component part of the CORBA Component has been initialized, the CORBA Component Manager follows the next steps: (1) sends a request to the CORBA component to create a distributed configurator. If the component does not create it, the CORBA Component Manager creates a default one, registers it with the CORBA ORB and assigns it to the new CORBA component. Once the CORBA Component is properly initialized, the CORBA Component Manager registers it with the CORBA ORB (2) and also with the CORBA Naming Service (3).

Component Registration
The CORBA Component Manager automatically registers all CORBA Components in the CORBA naming service for administration purposes. This component registration creates a hierarchy of names, which is independent of any other hierarchy that users can create. For example, Active Spaces assume a certain hierarchy for registering objects that belong to a particular space. This hierarchy is completely independent of the administrative one.

When the CORBA Component Manager is created, it must be provided with the IOR of a naming context. All CORBA components will be registered in this naming context under:

system/
machines/
<machine name>/
containers/
<component container name>/
<component UCID>.

If no default IOR is provided when creating the CORBA Component Manager, it will use the root naming context.

The name associated to the component in the binding is the component's UCID. The rest of the UCR is removed since it is redundant. It can be obtained from the structure of the naming hierarchy.

2.2 CORBA Component Destruction

When the CORBA Component Manager receives a request to delete a CORBA component, it automatically unregisters the component from the naming service and calls the *finish* method (which is part of the Base Component Manager destruction).

2.3 CORBA Component Activation

The CORBA Component Manager does not provide additional functionality to the one exported by the Base Component Manager.