Unified Object Bus

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

Table of Contents

1. Introduction and goals

Ubiquitous computing promotes collections of heterogeneous devices embedded in everyday scenarios. These devices are specialized in particular tasks and export their functionality unobtrusively therefore becoming "invisible". Users do not perceive the devices but instead the functionality they export. The bottom line is that information and services become more accessible.

Due to the heterogeneous nature of the previous devices, hardware and software resources will differ greatly. Different OSes and middleware platforms will coexist; therefore no standard interaction protocols can be assumed. However, the real benefit of ubiquitous computing is the coordination and management of all these devices to obtain a single computational force focused towards a common goal. The key in achieving this single computational force is seamless interoperability.

Object orientation has proven to be a powerful paradigm to manage large complex systems. Every entity in the system is abstracted as an object, which has a state, exports a well-defined interface and can be unequivocally identified. Component models define the shipping format of the objects, which become components that can be dynamically introduced in an application. The component model is also responsible for defining a set of services such as for example services to query the interfaces supported by components and services to manage the lifecycle of the components.

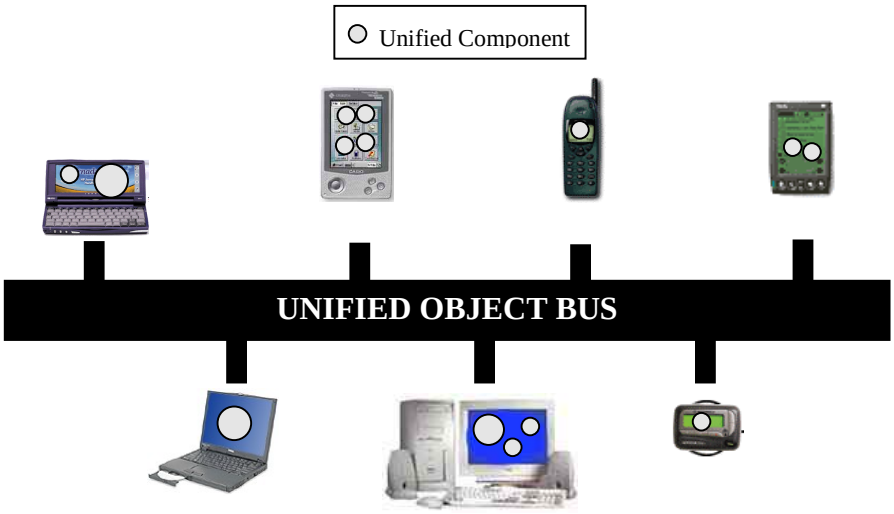

Figure 1. The Unified Object Bus

COM, DCOM, CORBA and Java Beans are examples of component models. However all these component models are incompatible (though it is possible to create bridges) and in most of the cases are not suitable to fit devices with limited resources. Ubiquitous computing scenarios are likely to contain devices using different component models, which difficults interoperability. The unified object bus defines a minimalist component

model on top of which existing component models can be integrated. The unified object bus follows the approach of OS microkernels. It defines a small core and leaves the implementation of the policies to the user.

The goals of the unified object bus are: provide a minimum component model kernel that can fit devices with limited resources, define a reflective architecture that allows introspection of its state as well as dynamic modifications, and finally define a standard mechanism to transparently incorporate new existing component models. Dynamic behavior is essential since it allows expanding and shrinking the bus according to the execution environment properties. The unified object bus, as depicted in figure 1, provides a common underlying infrastructure that allows interoperation of heterogeneous devices running different OSes and middleware implementations.

2. Architecture of the Unified Object Bus

The unified object bus can be divided into three parts: the component management core, the component managers and the object request brokers. The component management core defines the minimum functionality required to bootstrap the unified object bus and manipulate components. The component managers encapsulate the functionality required to create and destroy components of a specific type. Whenever the component management core receives a request to create or destroy a component, it delegates the request to the appropriate component manager. Finally, the object request brokers are the components in the system responsible for sending and receiving requests (e.g. CORBA ORB and SOAP ORB).

Figure 2 depicts the unified object bus architecture.

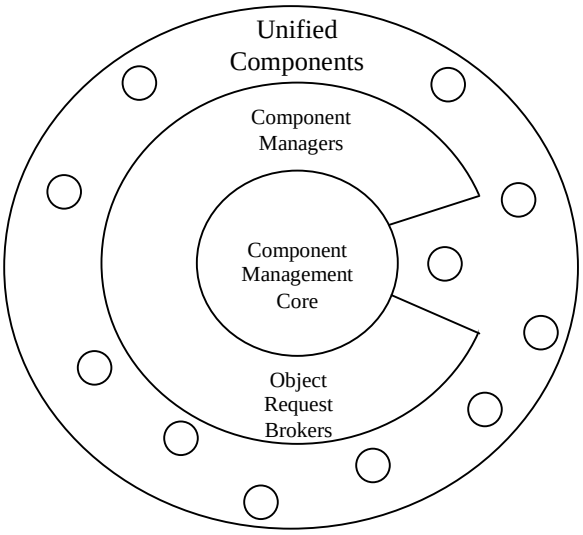

Figure 2. Unified Object Bus Architecture

2.1 Component Management Core

The component management core defines the execution environment for components and exports the minimum required functionality to manipulate components. This manipulation includes loading and unloading components, managing component dependencies, downloading and uploading components and inspecting the status of the components. In systems composed of large numbers of objects, inter-component dependencies become critical. To manage this issue, the component management core uses the component configurator pattern [1], which is applied to every single component.

2.1.1 Local Component Configurator

The Local Component Configurator is the default component configurator associated to components that do not create their own configurator when they are instantiated. The local component configurator is responsible for managing the local dependencies of its associated component. Figure 3 presents a usage example of the local component configurator (or local configurator for short) with two components involved: A and B. Component A depends on component B or by using component configurators' terms, B is hooked to A. In terms of clients and servers, A is a client of B and B is a server of A. This relationship is translated into a hook created in the local configurator of A (hookB) where component B is attached. Likewise, the reference to component A (the client) and the name "hookB" is stored in the list of clients of the local configurator of B.

2.1.2 Component Container

A component container is an execution environment that provides the resources required to execute components. By default, every component container contains a *domain configurator* that keeps references to the components running in the component container. The reference of this domain configurator is assigned to every component created in the component container. These components use the domain configurator to get access to all the components running locally.

From an abstract point of view, component containers can be considered components whose dependencies with their contained components are managed by the domain configurator. Figure 3 illustrates a component container.

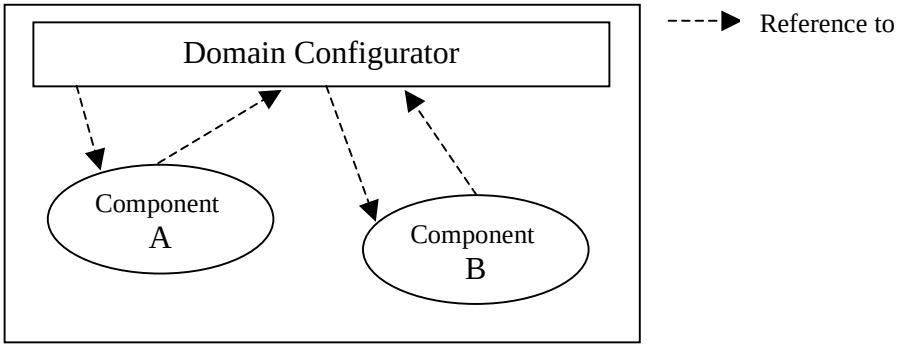

Figure 3. Component Container

However, providing an execution environment and a domain configurator is not enough. A set of tools is required to manipulate components and their dependencies as well as maintain a table with all components running in the container. For this reason, when a component container is created, it instantiates a default component: the Component Manager.

From an implementation perspective, component containers are processes.

2.1.3 Component Manager

The component manager is a default entry point to the Unified Object Bus. Every component container instantiates by default a component manager. This component manager exports the functionality required to manipulate components running in the component container. The component manager inherits from the abstract class *Object Life Cycle* and exports functionality to create and destroy components, hook and unhook components and provide information about the components running in the component container or the hooks exported by specific components.

Figure 4 illustrates the Unified Object Bus with several components running in their respective component containers and using the component manager instance for run-time requirements.

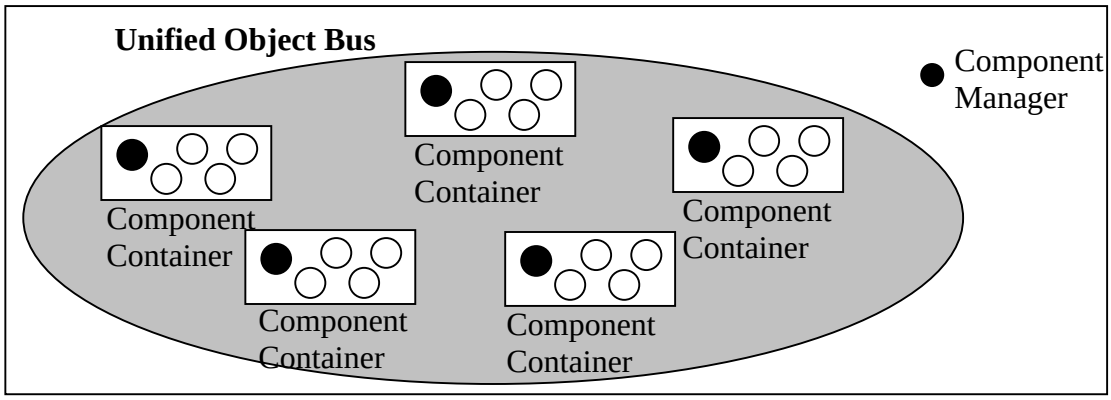

Figure 4. Unified Object Bus and Component Managers

The component manager is independent of any particular component model. Therefore, when it receives a request to create or delete a component, it delegates the request to the appropriate specialized component manager (e.g., CORBA Componet Manager and Base Component Manager), which has to be registered in advance.

Table 1 presents the interface of the component manager. All methods return –1 in case of error, and something different of –1 otherwise.

```
class ComponentManager: public ObjectLifeCycle
{
        public:

        ComponentManager();
        ~ComponentManager();

        int createHook(ComponentConfigurator *cc, const char *hookName);
        int deleteHook(ComponentConfigurator *cc, const char *hookName);
        int hookComponent(const char *UCR, ComponentConfigurator *cc,
                const char *hookName);

        //Next three methods are defined by the ObjectLifeCycle class.
        BaseComponent *createComponent(char *compFactName, int argc, char **argv);
        char destroyComponent(char *UCR);
        int activateComponent(BaseComponent *theComponent);

        ComponentConfigurator *getConfiguratorFromUCR(const char* UCR);
        ComponentConfigurator *getConfigurator (ComponentConfigurator *cc,
                const char *componentPathName, char **finalHook);

        UOBLoader *getUOBLoader(){return UOBLoader_;};
        void createLocalConfigurator();

        char registerSpecificComponentManager(const char *compModelName,
                                              ObjectLifeCycle *specCompManager);
        char unregisterSpecificComponentManager(const char *compModelName);

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

Table 1. Local Component Manager interface

The **createHook** method creates a new hook. The hook name can be compound (separating the different names by '/'). The hook name is relative to the component configurator provided as the first parameter. If the component configurator is NULL, then the hook name is assumed to start at the domain configurator. All the names in the hook name must exist, except the last one, which is the one to be created.

The **deleteHook** method deletes the hook specified. The rules applied to the component configurator and the hook name parameters are the same as the ones described for the createHook**.**

HookComponent attaches the component identified by the first parameter (UCR) to the hook (hookName) of the provided component configurator (cc). The hook name can be

compound and is relative to the provided component configurator. If the component configurator is null, then the domain configurator is assumed.

CreateComponent creates a new component of the type specified by "compFactName" (component factory name, check section 3). The last two arguments are the number of parameters and the parameters for the component. The method returns a pointer to the base component and 0 in case of error.

DestroyComponent destroys the component specified by UCR (see section 3).

GetConfiguratorFromUCR returns a pointer to the component configurator associated to the component specified by UCR (Unified Component Reference, see section XXX). If no component with such an identifier exists then 0 is returned.

GetConfigurator returns a pointer to the configurator attached to the *componentPathName*, relative to the component configurator provided (cc). If this configurator is null, then the DomainConfigurator is assumed as the starting point. If the whole component path name exists, then *final hook* is null. Otherwise, final hook contains the portion of the path that could not be resolved (normally because those hooks do not exist) and the method returns a pointer to the last valid component configurator found while traversing the path.

GetUOBLoader returns a reference to the loader object used by the component manager. This loader object exports functionality to load and unload component factories (normally dynamically loadable libraries) and keeps a list with all the component factories that have been loaded. It also associates to each component factory a counter, which represents the number of components for this component factory created so far. Some specialized component managers require the reference of the UOBLoader to perform additional tasks when loading and unloading a component.

RegisterSpecificComponentManager is responsible for registering a new specific component manager with the component manager. The method receives two parameters: the name of the component model that will be integrated and a pointer to the specific component manager that will be responsible for integrating those types of components. The component manager has a component configurator attached to a hook named SpecificComponentManagers. Whenever it receives a request to register, it creates a new hook in the component configurator with the name of the component model.

UnregisterSpecificComponentModel unregisters existing component models. It receives as a parameter the nameof the component mode to be unregister.

2.2 Specialized Component Managers

Component managers are responsible for creating, deleting and activating components of a specific type. Component managers are also responsible for keeping track of all the components they have created. By default, every component manager has a hook called

CreatedComponents, where a local component configurator is attached. This component configurator must have a hook for each component that has been created by the specific component manager. The name of the hooks should be the UCR of the created components. By following this convention it is possible to easily obtain a list of all the unified components created in a component container by simplly iterating over each component manager.

As stated in section 2.1.3, the Component Manager is not tied to any particular component model. Instead, it delegates creation, deletion and activation requests to the proper specialized component manager. Every component manager inherits from the abstract class *ObjectLifeCycle*, which defines a method to create components, a method to delete components and a method to activate components. Every specific component manager must provide an implementation for each of these methods as well as an interface definition for the type of object it manipulates.

Component Creation

In order to create a component, clients must provide the component model name and the name of the component factory. The component manager present in every component container uses the component model name to locate the right specific component manager and delegate the creation request to it. As a result, each specific component manager has complete freedom to create its own components in the most appropriate way.

Figure 5 illustrates the steps involved in the creation of a component C that in this case belongs to a generic component model M. This scenario requires the existence of a specific component manager that knows how to create components from the component model M.

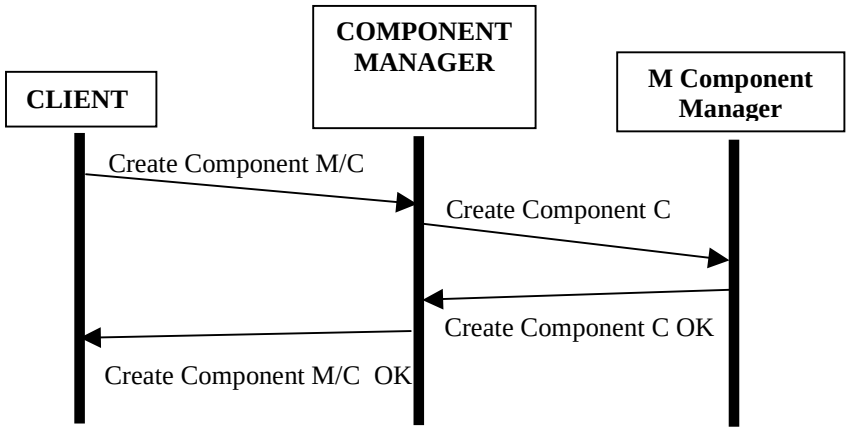

Figure 5. Creation of an "M" component.

Component Destruction
Destroying a component requires a client providing the UCR of the component to be destroyed to the component manager. The component manager uses the component model from the UCR to locate the right specific component manager.

Destroying a component normally involves locating the object, invoking its finish method, unregistering it from the CreatedComponents component configurator, updating the hooks of the clients of that component and finally deleting the component.

Figure 6 depicts the steps involved in the destruction a component that belongs to the M's component model.

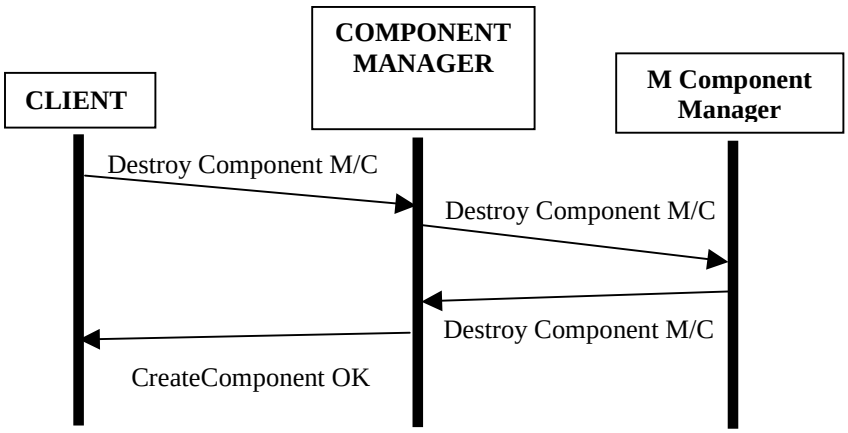

Figure 6. Destruction of an "M" component.

Component Activation
Activating a component is in most of the cases a simple process. It associates a thread to a recently created component. The details heavily depend on the type of object being created.

2.3 Object Request Brokers

Object Request Brokers encapsulate the functionality required to issue method calls and dispatch requests to the right objects. In most of the cases, every specific component manager has its own associated ORB. For example, the CORBA Component Model uses a CORBA ORB that is registered in the domain configurator. Object Request Brokers are also introduced in the system as components and therefore can be dynamically created and deleted.

3. Naming Scheme

Every component is assigned a unique reference that consists of several fields. The references are called Unified Component Reference (UCR) and contain enough information to locate the components from any place in the system.

UCRs are composed of the *UCR:* prefix, the host name, the component container ID and the Unique Component ID (UCID). The UCID is composed of the component model name, the component factory name (DLL) and finally the instance number to distinguish different components instantiated by the same factory. The symbol used to separate different fields is the foreslash (/). Table 2 shows the format of a UCR.

Unified Component Reference UCR : *host name* / *component container id* / UCID **Unique Component ID** *Component model / Component Factory Name / Instance Number*

Table 2. Format of the Unified Component Reference

Next comes an example of a UCR reference:

UCR:barna.cs.uiuc.edu/SYSTEM/Base/HTTPBroker/0;

The information contained in the UCR allows locating any component running in the system.

References

1. Kon, F. and R.H. Campbell, *Dependence Management in Component-Based Distributed Systems.* IEEE Concurrency, 2000. **8**(1): p. 26-36.