Unified Object Bus: Providing Support for Dynamic Management of Heterogeneous Components

Manuel Roman, Roy H. Campbell
{mroman1, rhc}@cs.uiuc.edu
Department of Computer Science
University of Illinois at Urbana-Champaign

Report No. UIUCDCS-R-2001-2222, UILU-ENG-2001-1729

---

1 This research is supported by grants from the National Science Foundation, NSF 98-70736 and NSF 99-72884 CISE.
Unified Object Bus: Providing Support for Dynamic Management of Heterogeneous Components

Abstract
Existing middleware implementations successfully address hardware and operating system heterogeneity. However, they are built assuming a single component model type (e.g. CORBA, DCOM or JavaBeans). We claim that such assumption is not necessarily desirable in certain ubiquitous computing scenarios where services implemented as components of different middleware platforms can coexist.

We present in this paper the Unified Object Bus (UOB), a flexible middleware platform for component-based systems that allows integrating heterogeneous component models while preserving a common interface to manipulate the lifecycle of their associated components. The UOB provides functionality to manage components both locally and remotely, and constitutes one of the backbone technologies to enable Active Spaces, which support mobile users in an unobtrusive ubiquitous computing environment that couples models and virtual representations to the physical world.

1. Introduction
In this paper, we present in this paper a middleware platform for the development of component-based software. The proposed platform is customized for scenarios involving mobile users in an unobtrusive ubiquitous computing environment that couples models and virtual representations to the physical world. We will refer to such scenarios as Active Spaces [1].

Active Spaces characteristically: (1) contain a large number of heterogeneous devices (i.e. desktops, workstations, sensors, handheld and embedded devices), (2) present a dynamic behavior where frequent change is the norm, and (3) assume the coexistence of heterogeneous component models. All these requirements have a significant impact on the middleware platform required for these scenarios.

An appropriate middleware platform must provide functionality to manipulate both seamlessly and dynamically a collection of heterogeneous components running on a set of heterogeneous devices that may be either local or remote. They must provide an open architecture that can be configured to accommodate new policies and component models. They must provide an explicit mechanism to manage inter-component dependencies in order to preserve the consistency of the system.

The UOB is designed according to these requirements. It exports functionality to manipulate the lifecycle of heterogeneous software components, manages inter-component dependencies, and presents an open architecture that allows dynamic incorporation of new component models. Note, however, that the UOB does not provide services to allow components from different models to interoperate; these are considered user level services.
The UOB does not address inter-component interaction protocols (e.g. IIOP or SOAP); the UOB does not provide a universal inter-component model bridging mechanism. Bridging functionality can be considered at the user-level and can be implemented taking benefit of the functionality exported by the UOB.

This paper is structured as follows: Section 2 presents an example of an application for an Active Space, and explains why an infrastructure such as the Unified Object Bus is required. Section 3 contains a detailed description of the Unified Object Bus architecture. Section 4 presents the current status of the implementation of the UOB. Section 5 describes our future work, section 6 presents related work, and section 7 concludes the paper.

2. Active Space Application Example. Importance of a Unified Middleware Platform

In this section we consider several issues involved in building an application to control a slide show presentation in the context of an Active Space. The slide show application consists of: a service that controls the slides (Slide Manager), a GUI running on a handheld device to control the slide show, a component that receives and displays slides in a handheld device, and finally, a service that detects the user entering the conference room and automatically starts the presentation. Figure 1 illustrates all the main components of such application.

The scenario depicted in figure 1 presents hardware heterogeneity (Intel based desktops, PalmPilot, and Pocket PC device), operating system heterogeneity (Windows 2000, Windows CE, and PalmOS), and software heterogeneity. The slide manager running on the Pentium IV is implemented as a C++ CORBA [2] component (shipped as a DLL). The control GUI running on the PalmPilot is a component packaged as a PalmOS
executable that implements the GUI and uses IIOP (Internet Inter-ORB Protocol) [2] to communicate with the slide manager. The slide renderer running on the Windows CE device is a component packaged as an executable that receives and renders the slides streamed by the slide manager. Finally, the User Detector is a Lua-based script [3] component (packaged as ASCII text) that receives CORBA events and uses CORBA to interact with the slide manager.

While component models such as CORBA, DCOM [4], JavaBeans [5] and SOAP [6] address the hardware and operating system heterogeneity, they all assume a single component model. It is required, therefore, to provide a middleware platform that allows manipulating components and their dependencies homogeneously, regardless of their specific component model type. This requirement is a necessity in ubiquitous computing scenarios (such as an Active Space), where it is not reasonable to assume the use of a single component model type. The Unified Object Bus provides functionality that allows the designer of the slide show application to use the different components of the application regardless of their specific implementation details.

There are several issues involved in the example presented in this section. First, the example uses several heterogeneous devices and different Operating Systems. Each device may use a different component implementation. Therefore, we must implement a user-level component repository service that uploads the specific component implementation to the different devices on demand despite the heterogeneity of the devices and operating systems. Second, all the components presented in the example, although running on different devices, are parts of the same application and therefore are related. The UOB offers functionality explicitly to define the dependencies among all these components, regardless of their specific component type. Third, because of component model heterogeneity, component references must be independent of any particular component model implementation. The UOB, by manipulating the lifecycle of components, provides functionality to create references that are independent of specific component model types. These references are used to build high-level services that combine different component types. Fourth, remote creation, deletion and configuring of the components on the devices are requirements. A user with a handheld device cannot manipulate the components on his/her device. Instead, a configuration service should remotely control all such tasks. The services provided by the UOB allow remote component manipulation.

While the Unified Object Bus does not define any high level service, it provides the tools to build such services. In the scenario described in this section, we assume the existence of a component repository service, an automatic configuration service, a unified naming service, and a unified trading service. All these user-level services have been created using the low level primitives provided by the UOB.

Using the functionality provided by the UOB we can create the different components required for the application and establish dependencies among them. Once the components are created, they use high-level UOB services to find each other and start exchanging data.
3. Unified Object Bus Architecture

This section describes the overall architecture of the Unified Object Bus, including the abstract component types and the required low-level services.

3.1 Base Component

The Base Component is the basic component managed by the Unified Object Bus. Base Components can be dynamically loaded and unloaded and export an interface consisting on three methods: init, finish and main. Init is used to initialize the state of the component when it is first loaded into memory; finish is called just before the component is deleted, and main contains the code that the component will execute when activated. Every component has an associated thread, which is responsible for executing the code contained in main.

The Unified Object Bus is platform independent and language neutral. Therefore, the Base Component is an abstract class that has to be specialized using a particular programming language. Figure 2 depicts the class hierarchy for base component and some example specializations. The first two subclasses of Base Component are specific implementations of Base Component for C++ and Java. Base Components cannot be accessed remotely and are used as building blocks for constructing more complex components. This is the case of SOAP and CORBA components illustrated in figure 2. C++ CORBA Components implement the Base Component interface (init, main and finish) and export additional functionality to allow remote access.
3.2 Component Configurator

Component Configurators [7] are objects responsible for storing runtime dependencies between components. Every Base Component has an associated Component Configurator known as Local Configurator, which stores references to other local components (same address space). Distributed components use an instance of a Distributed Configurator whose implementation depends on the specific distributed object model details. This distributed configurator manages dependencies with remote components. 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 in component configurators terms, B is hooked to A. 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.

### 3.3 Component Container

A component container is an execution environment that provides the resources required to execute components. By default, every component container has an instance of a component configurator called *domain configurator*. This domain configurator stores the references of the components running in the component container. From the point of view of the domain configurator, the components running in the component container are *hooked components*, while the domain configurator is a client of these hooked components. Unified components use the domain configurator reference to get access to all the components running locally in the container. Therefore, this domain configurator is used also as a directory of local components (same address space). CORBA components, for example, use the domain configurator to access the ORB instance, which is shared among all CORBA components running in the same component container.

In practical terms, component containers are processes with additional software to manipulate their internal components.

![Figure 4. Component Container](image)

Besides the component container and the domain configurator, a set of mechanisms is required to manipulate components and their dependencies as well as to maintain a list of all components running in the container. This is the task of the component managers.

### 3.4 Component Managers

Component Managers encapsulate the functionality required to integrate different component models into the Unified Object Bus. They are responsible for instantiating, activating, assigning an identifier, and deleting components of a specific component model type. Component Managers are normally not accessible remotely, that is, they export an interface that can only be accessed locally by components running in the same component container. Another component called Component Manager Exporter wraps locally accessible component managers and exports their functionality remotely –see section 3.5 for more information.
3.4.1 Component Managers’ Types

There are two different types of component managers: generic component managers, and specialized component managers. Every component container has a single instance of a generic component manager, which is responsible for aggregating different specialized component managers.

a) Generic Component Manager

The number of component managers integrated in the Unified Object Bus is not predefined. The architecture of the Unified Object Bus allows dynamically adding and removing component managers. All component managers are registered with a component manager called Generic Component Manager. This generic component manager defines a default API, which exports functionality to: (1) create components, (2) manage the dependencies between components, (3) browse components and their dependencies, and (4) destroy components. However, the generic component manager is independent of any particular component model. Therefore, when it receives a request, it delegates the request to the appropriate component manager (e.g. Base Component Manager, and CORBA Component Manager), which has to be registered in advance.

Figure 5 illustrates the Unified Object Bus with several components running in their respective component containers and using the generic component manager instance as the standard entry point for component manipulation.

![Figure 5. Unified Object Bus and Component Managers](image)

The generic 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 specialized component manager must provide an implementation for each of these methods as well as an interface definition for the type of object it manipulates. Figure 6 presents the interface of the generic component manager interface.

The createHook method creates a new hook in the domain configurator. 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 specified hook. 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, which stands for Unified Component Reference) 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). 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 NULL in case of error.

DestroyComponent destroys the component specified by UCR.

GetConfiguratorFromUCR returns a pointer to the component configurator associated to the component specified by UCR. If no component with such an identifier exists then NULL 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.

GetUOBSLoader 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 a counter to each component factory that represents the number of components for this component factory created so far. Some specialized component managers require the reference of the UOBSLoader to perform additional tasks when loading and unloading a component.
**RegisterSpecializedComponentManager** is responsible for registering a new specialized component manager with the generic component manager. The method receives two parameters: a component manager name and a pointer to the specialized component manager. The generic component manager has a component configurator attached to a hook named Specialized Component Managers. Whenever it receives a request to register, it creates a new hook in the component configurator with the name of the generic component manager.

**UnregisterSpecializedComponentManager** unregisters existing specialized component managers. It receives as a parameter the name of the specialized component manager to unregister.

```cpp
class ComponentManager: public ObjectLifeCycle
{
  public:
  ComponentManager();
  ~ComponentManager();

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

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

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

  char registerSpecializedComponentManager(const char *compManagerName,
                                          ObjectLifeCycle *specCompManager);
  char unregisterSpecializedComponentManager(const char *compManagerName);

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

**Figure 6. Local Component Manager interface**

**b) Specialized Component Managers**
Specialized Component managers are responsible for creating, deleting and activating components of a specific type. Specialized Component managers are also responsible for keeping track of all the components they have created. By default, every specialized 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 specialized component manager. The name of the hooks is 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 recursively iterating over each component manager. This mechanism helps also to avoid the existence of orphan components running in component containers.

3.4.2 Component Lifecycle

a) Component Creation
In order to create a component, clients must provide the component type and the name of the component factory. The generic component manager present in every component container uses the component type to locate the right specialized component manager and delegate the creation request to it. As a result, each specialized component manager is free to create its own components in the most appropriate way.

Figure 7 illustrates the steps involved in the creation of a component named C of type M. This scenario requires the existence of a specialized component manager that knows how to create components of type M.

Figure 7. Creation of component “C” of type “M”

b) Component Name Creation
At the component creation stage, every Unified Component is assigned a unique self-contained identifier. This unique identifier, known as Unified Component Reference (UCR), contains enough information to locate and interact with the component it refers to. The UCR is composed of six fields preceded by a prefix (“UCR:”) and its format is illustrated in figure 8.

UCR:<UHID>/<CCID>/<CTID>/<CFID>/<CIN>/<UID>

Figure 8. Unified Component Reference
The UOBHost ID (UHID) is the identifier of the Unified Object Bus Host where the component is running. For details about the UOBHost see section 3.6. The Component Container ID (CCID) is the identifier of the Component Container that provides the execution environment to the component. The Component Type ID (CTID) specifies the type of the component model the component belongs to. The Component Factory ID (CFID) is the actual name of the factory used to instantiate the component (i.e. name of a dynamic loadable library). Since it is possible to have more than one instance of a specific component running in a same component container, an additional field is required to distinguish between instances of the same component. The Component Instance Number (CIN) is used to resolve such ambiguity. Finally, the Unique ID (UID) is a unique serial number used to avoid UCR duplication. Figure 9 depicts an example of a UCR.

![UCR:srg192.cs.uiuc.edu/0/CORBA/PPTService/0/USD23dnv2](image)

**Figure 9.** Unified Component Reference example

Unique Component References are “raw” identifiers that are not part of any name hierarchy. However, in a system composed of a large number of components, a flat naming scheme does not scale. The unified object bus allows user defined policies to create such naming hierarchies. Although the UOB does not define how to implement such mechanisms, a possibility would be a service that aggregates different UCRs under a single UCR (similar to the concept of a directory in file systems). Developers can extend the format of the UCR by adding new fields, while maintaining compatibility with standard UCRs.

c) **Component Destruction**
Destroying a component requires a client providing the UCR of the component to be destroyed to the generic component manager. The component manager uses the component type from the UCR (CTID) to locate the right specialized component manager.

Destroying a component 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 10 depicts the steps involved in the destruction a component of type M.
**d) 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.

**3.5 Component Manager Exporter**
The *generic component manager* provides the functionality required to manipulate components. It accomplishes such task by delegating the requests to the appropriate specialized component managers. Therefore, this generic component manager is the one that implements functionality to manipulate the component containers. However, the generic component manager is implemented as a local component, which cannot be accessed from outside the component container where it has been instantiated. One of the reasons for this configuration is to preserve the independence from any specific component model, which is one of the principles of the UOB. The UOB defines an entity called Component Manager Exporter, which is responsible for wrapping the functionality of the generic component manager and exporting it to remote components. These component manager exporters are implemented as components of a particular component model such as CORBA, JavaBeans or SOAP.

**3.6 UOB Host**
The term UOB Host denotes any device capable of hosting the execution of unified components. Any UOB Host must be remotely accessible, must export functionality to manipulate component containers (i.e. create, delete and inspect), and must also export functionality to manipulate components. Examples of UOB Hosts are desktops, workstations, and handheld devices.

Figure 11 presents the interface implemented by UOB Hosts.
interface UOBHost
{
    string createComponentContainer(in string parameters)
        raises (ComponentContainerNotCreated);

    stringList listContainersCCID();

    string getComponentContainer(in string CCID)
        raises (InvalidCCID);

    string createComponent( in string CFID,
        in string params,
        in string CCID)
        raises (ComponentNotCreated, InvalidUCCID);

    void deleteComponent(in string UCR)
        raises (CouldNotDeleteComponent);

    void killContainer(in string CCID)
        raises (CouldNotKillContainer);
}

Figure 11. UOBHost CORBA IDL Interface

The `createComponentContainer` method is used to create a new component container in
the specified UOBHost. The method’s parameter can be used to specify a list of unified
components that will be instantiated in the newly created container.

`ListContainersCCID` returns a list with the Ids of all the component containers currently
active in the specified UOB Host.

`GetComponentContainer` returns a string that represents the UCR of the component
manager exporter associated to the component container. The component manager
exporter allows remote manipulation of the components contained in the container passed
as a parameter. Since the UOB does not assume any particular component model, clients
must use an external service to obtain the component manager exporter implementation
from the returned UCR.

The method `createComponent` is used to instantiate a new component (CFID), with some
specific parameters (params) in the defined component container (CCID). The method
returns the UCR of the newly created component.

`DeleteComponent` allows users to terminate the execution of the specified component
(UCR).

Finally, `killContainer` makes it possible to terminate a component container, and
therefore all its contained components.
4. Current Implementation Status

We are currently using the Unified Object Bus as the foundation of our component-based software infrastructure to enable Active Spaces. The UOB runs on Solaris, Windows 2000 and Windows CE, and we are currently working on the PalmOS version. We have implemented four different specialized component managers: (1) Base Component Manager, (2) CORBA Component Manager, (3) Executable Component Manager, and (4) Lua Component Manager. We have also implemented a CORBA Component Manager Exporter and a CORBA component to implement the functionality of the UOB Hosts (i.e. Solaris, Windows 2000, Windows CE, and PalmOS devices).

The **Base Component Manager** allows dynamically creating, naming, activating, and destroying Base components shipped as DLLs. The current implementation of this component manager can manipulate C++ based Base components. However, it is easy to extend the component manager to allow instantiating Base objects implemented in other programming languages.

The **CORBA Component Manager** provides functionality to instantiate CORBA components shipped as DLLs, and provides additional functionality to automatically register the instantiated CORBA servant with a CORBA ORB. This automatic registration avoids users from manually having to register their objects with an ORB, therefore simplifying the task of creating CORBA objects. The CORBA Component Manager inherits from the Base Component Manager and reuses most of the functionality implemented by the Base Component Manager. It simply adds some extra functionality for the automated registration mechanisms.

The **Executable Component Manager** exports functionality to instantiate components shipped as executables. These components are not loaded in the address space of a component container. They have their own associated process. However, each of these executable components has an associated wrapper that is instantiated in the original component container and manages the dependencies of the executable component. This level of indirection allows managing the lifecycle of executables as any other unified component. The Executable Component Manager allows easily integrating legacy applications in the UOB, as well as GUIs, which are normally shipped as executables. Figure 12 shows an example of an executable component implementing an application GUI, and a CORBA component implementing the application’s backend. The GUI interoperates with the backend using IIOP and takes benefit of the UOB to explicitly define the dependency between both components.

![Diagram of Component Container, CORBA Component, Executable Component, and GUI](image)

*Figure 12. Example of a CORBA Component and an Executable Component*
The third specialized component manager we have implemented is called *Lua Component Manager*. Lua is a scripting language [3] which provides functionality to easily interoperate with CORBA, COM and Java objects. The Lua Component Manager automatically creates a Lua virtual machine, which is used to interpret the Lua components, which are shipped as ASCII text. As defined by the UOB, Lua components’ dependencies can be explicitly manipulated.

In order to simplify the manipulation of unified components in the system, we have created a visual tool called UOB Browser. This tool presents a list of all available UOB Hosts, with their active component containers, as well as a list of components running in each container. Currently the tool allows creating and deleting component containers, as well as creating/deleting components in/from each of these component containers. A screenshot of the tool is depicted in figure 13.

![UOB Browser Screenshot](image)

*Figure 13. UOB Browser*. The left panel lists six different UOB Hosts. The first five are desktop computers while the last one (wbama) is a Windows CE device. The right panel lists all the components running on UOB Host PC2401-2.cs.uiuc.edu.

5. Future Work

We have been extensively taking benefit of the remote component lifecycle manipulation functionality exported by the UOB. However, we have used CORBA as our main component model. As part of our future work we plan to incorporate other component types to exploit the full potential of the UOB. The integration of different component types will include the development of services to automatically create bridges between incompatible component models.
Our future work includes also the development of a unified naming service and a unified trading service. These two services will aggregate heterogeneous naming hierarchies and trading services, using the unified component references as the indirection level. We plan to use a similar approach as the one used to implement component managers. In the case of a unified naming service, for example, a generic naming service manager will aggregate specialized naming service managers, providing a common entry point, and specialized managers customized for particular component models.

Finally, we consider an essential requirement the ability to provide a standard mechanism to describe functionality exported by different unified components. We plan to build user-level semantic description services, which will leverage standards such as WSDL (Web Service Description Language) [8] and RDF (Resource Description Language) [9]. This functionality, will allow the implementation of semantic discovery mechanisms on top of the UOB.

6. Related Work

The Information Bus [10] is a middleware platform for distributed object applications. The Information Bus is built according to four main principles: (1) core communication protocols with minimum semantics, (2) self-describing objects, (3) dynamically defined types, and (4) anonymous communications. The Unified Object Bus shares some design principles such as the requirement for continuous operation, dynamic system evolution, and implementing policies as services outside the core implementation. However, unlike the Information Bus, the UOB does not specify any communication protocol and provides specific support for heterogeneous component models’ lifecycle management.

The CORBA Component Model (CCM) [11] is a server-side component model for building and deploying CORBA applications. The CCM defines a standard packaging mechanism that allows easily shipping, deploying and configuring CORBA services. The Unified Object Bus also provides a standard API to manipulate components packaged in different formats. While the CCM is specific for CORBA components, the UOB can manipulate different types of components.

JavaBeans [5] define a standard format and interfaces for Java components. These components can be easily manipulated by builder tools to create applications, applets, and composite components. Like in the case of the CCM, JavaBeans allows manipulation of a single type of components.

7. Conclusions

Developing component-based software involving heterogeneous devices, operating systems, and component types is not an easy task. This challenge is exacerbated in scenarios such as Active Spaces, where the level of heterogeneity and dynamism are further increased.
The Unified Object Bus is a component-based middleware platform that allows manipulating the lifecycle of heterogeneous component types, and provides functionality to enforce inter-component dependencies. The UOB defines the strictly required low-level mechanisms (lifecycle management and dependency management), and does not impose specific communication protocols, or distributed component models wire protocols, or high level policies. Developers can define these protocols and policies at user level.

It is important to remark that the UOB is not a tangible entity. The UOB is the collection of all unified components that have been created and are managed by the services defined by the UOB (i.e. component containers, component managers and UOB Hosts). While some devices (e.g. desktops and workstations) host essential UOB services, other devices (e.g. specific embedded devices) will not have enough resources to host such services. However, this last type of devices can still access the UOB and take benefit of the functionality of any unified component.

References