Section	VI	Other Author(s)	Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan
Chapter	38	E-mail Address	nanbor@cs.wustl.edu, schmidt@uci.edu, coryan@cs.wustl.edu
Pages		Phone Number	(314) 935-6355
Date Complete		Dates Reviewed	

Overview of the CORBA Component Model

Introduction:

The demand for distributed object computing (DOC) middleware, such as OMG's Common Object Request Broker Architecture (CORBA) (OMG, 1999a), is growing rapidly as deregulation and global competition makes it increasingly hard to develop and maintain complex applications from scratch. CORBA allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, or hardware (Henning and Vinoski, 1999). The use of CORBA as middleware for distributed client/server applications has increased over the past five years, as shown by the OMG CORBA "success stories" web site at `http://www.omg.org/`.

The traditional CORBA object model provides a standard middleware framework that enables CORBA objects to interoperate with each other. CORBA defines a software bus that allows clients to invoke operations on objects that can reside locally or remotely. Moreover, to provide higher-level reusable components, the OMG also specifies a set of CORBA Object Services that define the standard interfaces to access common services, such as naming, trading, and event notification. By using CORBA and its Object Services, programmers can integrate and assemble large, complex distributed applications and systems using features and services from different providers. However, the traditional CORBA object model culminating in CORBA 2.3 (OMG, 1999a) failed to address the following important design forces:

- *No standard way to deploy object implementations*: The CORBA specification does not define a standard way to deploy object implementations in server processes, *i.e.*, there is no standard way to distribute object implementations, install those implementations in their execution context, or activate the implementation in a particular ORB. Thus, system designers must use *ad hoc* strategies to instantiate all objects in a system. Moreover, since objects may depend on one another, instantiating objects in a large-scale distributed system can become complicated.
- *Lack of support for common programming idioms for CORBA servers:* The CORBA family of specifications provides a rich set of features to implement servers. In many application domains, however, only a limited subset of those features are ever used. Consequently, it is desirable to support subsets of necessary features via tools that can generate CORBA code automatically to implement common application use-cases. For example, the CORBA 2.2 specification introduced the *Portable Object Adapter* (POA), which is the ORB mechanism that forwards client requests to concrete object implementations. The POA specification provides standard APIs to register object implementations with the ORB, to deactivate those objects, or to activate object implementations on-demand. The POA is a flexible component of the CORBA programming model and provides numerous policies to configure its behavior. However, an important class of applications only use a subset of the possible configurations of a POA, yet, server developers must learn how to configure many POA *policies* to obtain their desired behavior.
- *Difficulty extending object functionalities:* In the traditional CORBA object model, objects can be extended only via inheritance. To support new interfaces, therefore, application developers must: (1) define a new IDL interface that inherits from all the require interfaces, (2) implement the new interface, (3) deploy the new implementation across all their servers. However, multiple inheritance in CORBA IDL is fragile since overloading is not allowed in IDL due to languages like C that lack overloading. Therefore, the techniques described above have only limited applicability. Moreover, applications may need to expose the same IDL interface multiple times to allow developers to either provide multiple implementations or multiple instances of the service

through a single access point. In contrast, multiple inheritance makes it impossible to expose the same interface more than once, or to determine which is the most derived interface available to clients.

- *Availability of CORBA Object Services is not defined a priori*: The CORBA specification does not mandate what Object Services are available at run-time. As a consequence, object developers must use *ad hoc* strategies to configure and activate these services when planning a system.
- *No standard object lifecycle management*: Although the CORBA Object Service defines a Lifecycle Service, its use is not mandated. Therefore, clients often require explicit knowledge to manage the lifecycle of an object in *ad hoc* ways. Moreover, the developers of CORBA objects controlled through the LifeCycle service must be aware of this fact, and must define auxiliary interfaces to control the object life cycle. Defining these interfaces is a tedious process, and should be automated when possible, but the earlier CORBA specifications lacked the features required to implement such automation.

The inadequacies of the CORBA specification, prior to and including version 2.3, outlined above often lead to tightly coupled, *ad-hoc* implementations of objects that are hard to design, reuse, deploy, maintain and extend.

To address these inadequacies, the OMG has adopted the CORBA Component Model (CCM) (OMG, 1999b) as part of the CORBA 3.0 specification. The CCM extends the traditional CORBA object model by defining features and services that allows application programmers to implement, manage, configure, and deploy components that integrate commonly used CORBA services, such as persistence, security, transaction, and event services, in a standard environment. The CCM standard not only enables greater software reuse for servers, it also provides greater flexibility for dynamic configuration of CORBA applications. With the increasing acceptance of CORBA in a range of application domains (Schmidt, 1995; Levine, 1998; O'Ryan and Schmidt, 1999), the CCM appears well-suited for scalable, mission-critical client/server applications. In this chapter, we describe the major features and services defined by the CORBA Component Model (CCM) architecture and illustrate the benefits of using the CCM architecture.

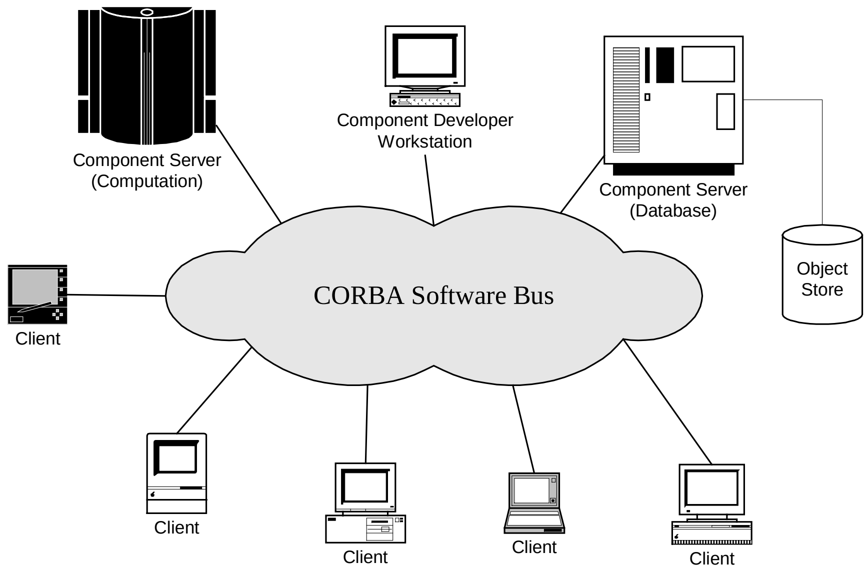

Figure 1: An Example System Developed with the CORBA Component Model (CCM)

`Figure 1` shows a conceptual diagram of a distributed system developed using CCM components. Component developers define the IDL interfaces that component implementations will support. Next, they implement the component using tools provided by their CCM provider. The resulting component implementation then can be packaged in a dynamic-linked library (DLL). Finally, a deployment mechanism provided by the CCM provider can be used to deploy the component in a *component server,* which is a standalone process that hosts component implementations by loading their associated DLLs. Thus, components execute in component servers and are available to process client requests. A major contribution of the CCM is standardizing the development cycle we described above, using CORBA as its middleware infrastructure. Below, we first describe components in the CCM from the viewpoint of a client and then from the viewpoint of a component developer. In addition, we describe the CORBA ORB extensions that are necessary to support the CCM.

Client View

Below, we present what are components and how to use components from a client's point of view.

Components: `Figure 2` shows the client view of a CORBA component that has the following IDL definition:

```
interface A, B;                    // Forward declaration.
component Foo supports A, B        // Definition of equivalent interface
{                                  // and its supported interfaces.
  provides W, X, Y, Z;             // Facets (provided interfaces.)
  . . .                            // Other component definitions.
};
```

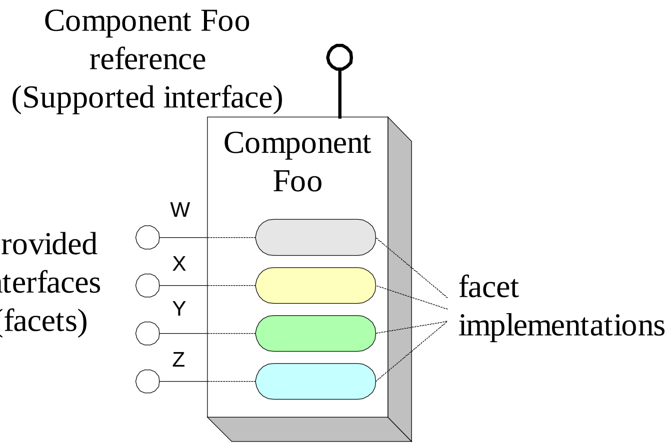

Figure 2: Client View of a CORBA Component

A reference to an instance of component `Foo` appears to its clients just like a regular CORBA object reference to an instance of interface `Foo`. Thus, component-unaware clients can invoke operations via an object reference to a component's *equivalent interface*, which is the interface that identifies the component instance uniquely. As with a regular CORBA object, a component's equivalent interface can inherit from other interfaces, which are called the component's *supported interfaces*.

As we mentioned earlier, however, it is hard to extend CORBA objects solely using inheritance because objects cannot associate multiple interfaces with a single implementation entity. To address this issue, therefore, the CCM adds *facets* to components. Facets, also known as *provided interfaces*, are interfaces that a component provides, yet which are not necessarily related to component's supported interfaces via inheritance. CCM facets are designed in accordance with the Extension Interface pattern (Schmidt, 2000) and are similar to component *interfaces* in Microsoft's Component Object Model (COM) (Box, 1997), in that they allow a component to support *unrelated interfaces*.

A `Navigation` interface is defined in `CORBA::CCMObject` interface, which all equivalent interfaces of components inherit from. Clients use the operations in the `Navigation` interface to enumerate all facets that a component provides. Thus, any client that holds a reference to a facet can use the standard `get_component()` operation to obtain a reference to the component's equivalent interface.

The addition of facets to the CCM object model enhances component reusability significantly. For instance, new interfaces can be provided in a new component without affecting existing clients of the component. Moreover, new clients can check whether a component provides a certain interface by using the standard CCM `Navigation` interface. Since the CCM allows the *binding* of several unrelated interfaces with a component implementation entity, clients need not have explicit knowledge of a component's implementation details in order to use the alternative interfaces that it offers.

Component Lifecycle Management: It is important for a component server to deal with lifecycle management issues properly and for clients to help component servers manage the lifecycles of their component instances. For example, a component can export services to multiple clients simultaneously. Likewise, a component server can host many components. Moreover, a component server must know when to create a component instance and when to remove the instance in order to prevent resource leaks. Although the CORBA Object Services specification defines a Lifecycle Service, its use is not mandated for general CORBA objects. Therefore, CORBA developers often implement their own *ad hoc* lifecycle management strategies. In practice, however, this flexibility in the CORBA specification can tightly couple clients with a specific implementation of an object interface. To standardize the component lifecycle management interface, therefore, the CCM introduces a new keyword called `home` that specifies the lifecycle management strategy of each component.

Clients can use the `home` interface to control the lifecycle of each component instance it uses. Each `home` interface manages exactly one type of component. A `home` can be either *keyed* or *keyless*. Keyless `home`s often support *factory* operations (Gamma, 1994) that create new instances of components. In contrast, keyed `home`s also often support *finder* operations that clients can use to index into persistent component instances using keys provided by clients. To indicate to a component server that a particular component instance is no longer needed, a client can invoke the `remove_component()` operation on the `home` interface to notify the component server. The component server can then determine how to remove the component instance based on strategies that it implements.

Component Usage Scenarios: To use a component, a client must first acquire the `home` interface of the component. However, clients require a standard bootstrapping mechanism to locate the `home` interface of the component. To simplify this bootstrapping process, references to available component `home`s can be stored in a centralized database. Thus, all clients need to bootstrap themselves is to acquire a reference to that centralized database. To access this centralized database, the CCM specification defines a `HomeFinder` interface, which is similar to the CORBA Interoperable Naming Service (OMG, 1998). A client first uses the standard CORBA bootstrapping API `resolve_initial_references("HomeFinder")` to acquire the object reference to a `HomeFinder` interface. The CCM `HomeFinder` implements a directory service for component `home`s that clients use to locate and acquire a reference to the desired `home` interface of the component. Once the client acquires a reference to its `home` interface, it then invokes the proper factory operation to create or find the target component reference.

Component Developer View

As we mentioned in the chapter introduction, a significant deficiency in CORBA specifications prior to 3.0 was the lack of support for common techniques to implement CORBA servers. Below, we describe how the CCM specification behaves from the point of view of a component developer, focusing on how its features can be used to automate the generation of an important class of server applications.

Components: The CCM specification extends the CORBA Interface Definition Language (IDL) to support *components*. Just as object interfaces are defined in the CORBA object model, component developers can use IDL definitions to specify the operations a component supports.

In general, component programming models encourage reuse of components via composition rather than inheritance. To better support this model, therefore, a CCM component can provide unrelated interfaces and support interface navigation, as described in the *Client View* discussion above. These features provide an environment for developers to extend components and for clients to locate and use them. However, since components can be installed in component servers that have no knowledge of how to configure and instantiate their installed components *a priori*, components need other generic interfaces to assist component servers in installing and managing them. In CCM, components can interact with external entities, such as services provided by the ORB, other components, or clients via a set of interfaces called *ports*, which define the standard mechanisms to modify the component configurations.

`Figure 3` shows how the CCM port mechanism helps to compose components together.

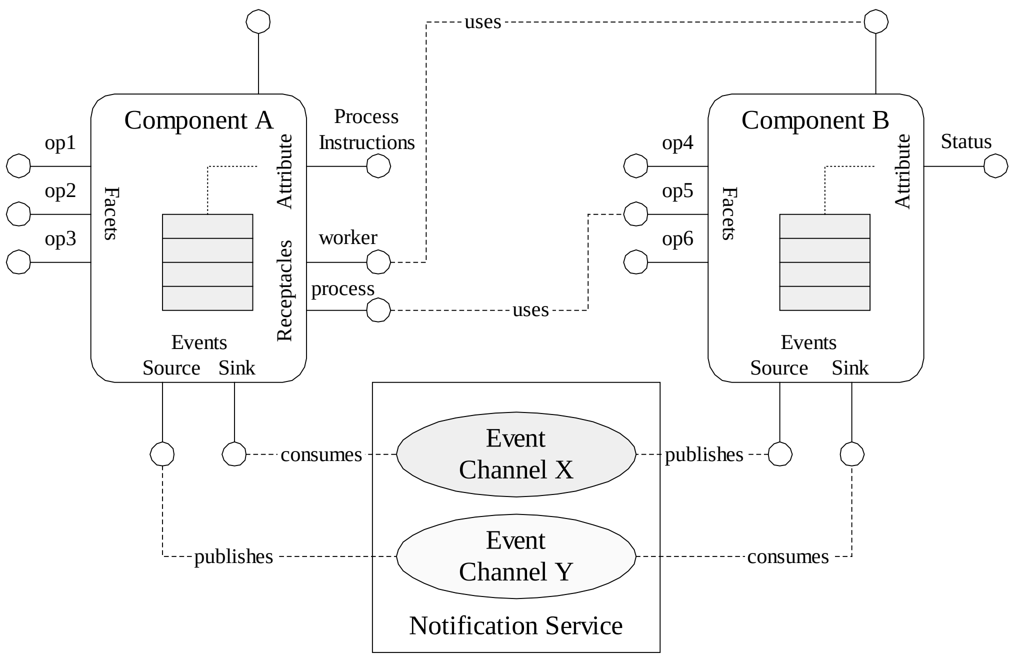

Figure 3: CCM Components Interact with One Another Through Port Mechanisms

We have mentioned one port mechanism called facets in the *Client View* presentation above. Other port mechanisms include the following:

- **Receptacles**: Since component programming models encourage reuse via composition more than inheritance, a component may delegate some of its operations to other components. In this case, a component must obtain the object reference to an instance of the other component. The CCM calls these references “object connections” and calls the port names of these connections “receptacles.” Receptacles provide a generic way to connect certain types of objects to a component. Components may *connect* to other objects, including those of other components, and invoke operations upon those objects. Component developers can indicate the willingness of a component to connect to other objects/components using receptacles.

- **Event sources/sinks**: In addition to interacting with each other via operation invocations, components can also interact by monitoring the state of other components and reacting when certain state change events occur. These loosely-coupled interactions are based on the Observer pattern (Gamma 1994), which is commonly used in distributed application development (Pyarali, 2000). Component developers specify components' interests to publish or subscribe to events by specifying *event sources* and *event sinks* facets in the components' definitions.
- **Attributes**: To facilitate the configuration of components, the CCM specification extends the notion of *attributes* defined in CORBA. Attributes can be used by configuration tools to preset configuration values of a component. Unlike previous versions of CORBA, the CCM specification allows operations that access and modify attribute values to raise exceptions. The component developer can use this feature to raise an exception if an attempt is made to change a configuration attribute after the system configuration has completed. The specification does not disallow the use of component attributes for purposes other than configuration, but their use as configuration parameters is supported directly. As with previous versions of the CORBA specification, the component attributes only define an interface, and may or may not be part of the transient or persistent state of the component.

All four CCM port mechanisms, *i.e.*, *facets*, *receptacles*, *event sources/sinks*, and *attributes*, can be used by component clients. With the exception of facets, however, the other port mechanisms are primarily intended for use by the CCM deployment framework to perform component configuration, which is described shortly below.

Component home: As mentioned in the *Client View* discussion above, component developers use the `home` interface to define component lifecycle management operations. Component developers may choose to provide multiple `home` interfaces that implement different lifecycle management strategies for the managed component. These different lifecycle strategies may be used to apply customized caching policies for component instances or to modify component behaviors at creation time.

Component Configuration: In CCM, components are configured automatically when they are installed on a component server. The port mechanisms described above provide interfaces to configure a component, *i.e.*, to set up the object connections, to subscribe or publish events, and to set up component attributes. However, application developers still need a mechanism to express a concrete configuration for a component, *e.g.*, what component(s) must be connected and which event channel will components use to publish and receive events. A standard component configuration interface, `Components::StandardConfigurator`, is defined in the CCM specification to help component servers perform component configuration. Component developers can extends this configuration interface to specify how their components should be configured in order to improve the flexibility of component implementations. Optionally, the `home` interface can accept a component configuration object reference to be used to perform component configuration on a component instance.

A complex component may require several operations to publish/subscribe events and to establish object connections before it can be configured completely. To prevent clients from accessing an uninitialized component prematurely, the CCM framework divides the component operation status into a *configuration phase* and an *operation phase*. The CCM defines a standard `configuration_complete()` operation in every component interface. This operation is called by the component's configuration interface in order to transition it from its configuration phase to its operation phase. Components developers can restrict access to certain "configuration-only" operations after the configuration phase.

Component Implementation Framework (CIF): As described above, the CCM defines a large number of interfaces to support the structure and functionality of components. Fortunately, the implementations for many of these interfaces can be generated automatically. Moreover, the lifecycle management and the state management of components also can be factored out and reused. The CORBA *Component Implementation Framework* (CIF) is designed to shield component developers from these tedious tasks. Below, we outline how the CIF helps to automate these tasks.

Many business applications use components to model "real world" entities, such as employees, bank accounts, and stock brokers. These entities are often represented as database entries and persist over time. The CIF defines the programming model for managing the persistent state of components and constructing component implementations. Components have persistent state when an instance of these components maps to persistent data that must be recalled whenever the component instance is activated. For example, when a bank account component is instantiated, the CCM needs to recall the previous status of the account from a database.

The CCM specification defines a declarative language, the *Component Implementation Definition Language* (CIDL), to describe implementations and persistent state of components and component homes. As shown in `Figure 4` the CIF uses the CIDL descriptions to generate programming skeletons that automate core component behaviors, such as navigation, identity inquiries, activation, and state management.

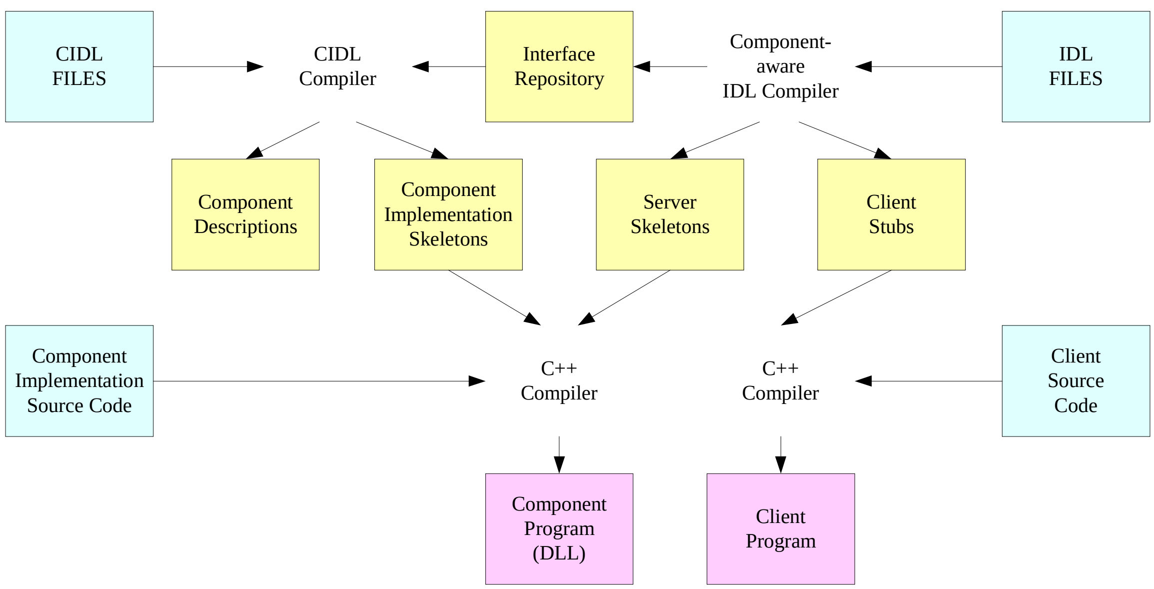

Figure 4: Implementing Components using the Component Implementation Definition Language (CIDL)

Implementations generated by a CIDL compiler are called *executors*. Executors contain the aforementioned auto-generated implementations and provide hook methods (Gamma, 1994) that allow component developers to add custom component-specific logic. Executors can be packaged in dynamically linked libraries (DLLs) and installed in a *component server* that supports a particular target platform and programming language.

In addition, the CIDL is responsible for generating *component descriptors*, which are used to describe component features, along with the type of services required by the component being described.

Containers: As described above, components are packaged in DLLs and run on component servers. Component implementations depends upon the POA to dispatch incoming client requests to their corresponding servants. However, a component itself need not know how to interact with its local run-time environment, such as creating its POA hierarchy and locating the common services defined by the CCM. Moreover, components may require notification via *callbacks* when certain events occur.

To support the functionality outlined above in a reusable manner, the CCM specification defines *containers*, which perform these tasks for components. The CCM container programming model defines a set of APIs that simplify the task of developing and/or configuring CORBA applications. A container encapsulates a component implementation and uses these APIs to provide a run-time environment for the component it manages. This run-time environment typically provides the following features:

- Activating/passivating component implementations to preserve limited system resources, such as main memory.
- Providing an adaptation layer for four commonly used services: Transaction, Persistence, Security, and Notification services. This adaptation layer frees the clients from locating these services by forwarding client requests to the actual services managed by the CCM.
- Providing adaptation layers for callbacks that the container and the ORB use to inform the component about interesting events, such as messages from the Transaction or the Notification Service.
- Managing POA policies to determine how to create component references.

Figure 5 shows the architecture of the container programming model.

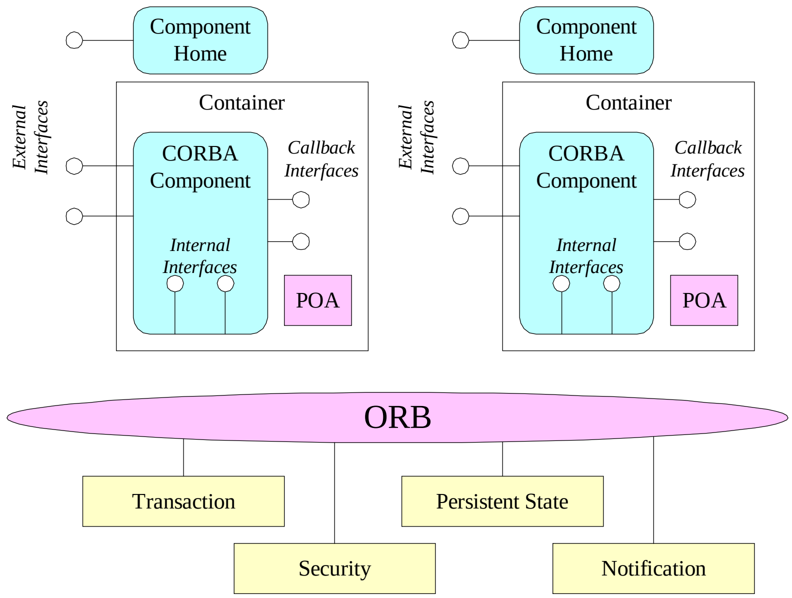

Figure 5 The CORBA Component Model's Container Architecture

As shown in the figure, clients access external interfaces of components directly and components access the ORB functionality through the containers APIs. Each container manages one component implementation defined by the CIF. For all the interfaces it manages, a container creates its own POA. These interfaces can be decomposed as follows:

- **External APIs:** These interfaces are provided by the component and include the *equivalent interface*, *facets*, and the component *home* interface. Clients can access these external APIs directly.
- **Container APIs:** These interfaces include the *internal interfaces* that the component can invoke to access the services provided by the container, as well as the *callback interfaces* that the container can invoke on the component.

Through the collaboration of these interfaces, a container provides the component it manages access to its POA and other services defined by a CCM-enabled ORB.

CCM containers also manage the lifetime of component servants. Four types of servant lifetime policies -- *method*, *session*, *component*, and *container* -- control the timing of activating and deactivating components.

A CCM provider defines a `ServantLocator` that is responsible for supporting these policies. When installed, a POA delegate the responsibility of activating and passivating servant to the `ServantLocator`. *Method* and *session* policies cause `ServantLocator`s to activate and passivate component on every method invocation or session, whereas *component* and *container* policies defer the servant lifetime policies to components and containers, respectively. High-quality CCM container implementations should provide component instance caching and dynamic linking/unlinking of executors to gain a balance between component performance and the resource usage (Wang, 2000a).

Component Categories: The CCM specification expects that different applications will use components in various ways. For example, some applications will always map component instances to a particular persistent data store, whereas components in other applications will possess no persistent state at all. In turn, the CORBA object model defines the *CORBA usage model*, which categorizes object references to be either *transient* or *persistent*. In the former case, the object reference becomes invalid after the POA that generated it is deactivated. In the latter case, the lifetime of the object reference can exceed the lifetime of the POA that created it. The CCM *component category* model extends the original CORBA usage model to define the legal combinations of container API types and the CORBA usage models. Component developers specify the component category of a component in a CIDL file and the CIDL compiler is responsible for generating the appropriate interfaces that match the designated component category. `Table 1` shows the component categories supported by the CCM.

Component Categories	CORBA Usage Model	Object Reference	Keyed Home Interface
Service	Stateless	TRANSIENT	No
Session	Conversational	TRANSIENT	No
Process	Durable	PERSISTENT	No
Entity	Durable	PERSISTENT	Yes

Table 1 Component categories in the CCM

Service components have no state, and only control the behavior of the system, good examples include wrappers for legacy procedural applications. Session components, such as iterator objects, maintain transient states that are often associated with transactions. Process components, such as business process objects, have client invisible persistent state and may be associated with transactions. Entity components, such as inventory components, may have transaction-related operations and have persistent state visible to clients. In addition, entity components must be indexed by keys.

In order to manage component references with and without persistent state, the container programming model defines two types of container interfaces: (1) *session* container interfaces for managing transient components and (2) *entity* container interfaces for managing persistent components. The container programming model uses the CORBA usage model to specify the required interaction pattern between a container, its POA, and CORBA Services, such as Notification or Transaction. Furthermore, the CCM container programming model can exploit advanced features of the POA to implement multiple CORBA objects using a single servant, *i.e.*, the same C++ or Java object can be used to implement several distinct CORBA objects. In addition, a container can determine if the object can be activated on-demand in order to conserve memory resources.

Packaging and Deployment: In large-scale distributed systems, component implementations may be deployed across multiple servers, often using different implementation languages, operating systems, or compilers. Moreover, component implementations often depend on other component implementations. Thus, the packaging and deploying of components can become complicated. Moreover, some groups of components must be deployed in the same process or host.

When two components are located in the same process or host, the CORBA specification refers to them as being *collocated.* In such cases, the ORB can improve performance by avoiding extra communication overhead. The CCM defines standard techniques and patterns that component developers can apply to simplify the packaging and deployment of components. The CCM describes components, and their dependencies using the *Open Software Description* (OSD), which is an XML *Document Type Definition* (DTD) defined by the WWW Consortium. Components are packaged in DLLs. Package descriptors are XML documents conforming to the Open Software Description DTD, they describe the content of DLL and their dependencies. A component may depend on other components, in addition, a component may require components that it depends upon to be deployed in a collocated manner via package descriptors.

In addition to defining component descriptors, which we described earlier in the Component Implementation Framework (CIF) discussion, the CCM OSD also defines *component assembly descriptors*.

These descriptors characterize key component deployment information, such as assembly instructions and interconnection topology. A standard *assembly interface* is defined to support automatic deployment. The assembly mechanism supported by a CCM-enabled ORB consumes the information in component descriptors and other assembling instructions specified by developers. The deployment mechanism allows remote installation and activation of new or modified components.

ORB Extensions

The CCM specification adds many ORB extensions to simplify the implementation of CCM framework. Most of the extensions do not affect directly how component developers implement components. Some of the extensions, however, help component developers to improve the performance of components or simplify their work. Below, we describe these ORB extensions and outline how they can help developers.

Locality-constrained interfaces: Many interfaces for communicating between a component and its container are only used locally. Prior to the CORBA 3.0 specification, CORBA also required certain interfaces to be "locality constrained," *i.e.*, these IDL interfaces are only valid in the process where they are instantiated and cannot be exported to remote clients. For example, a POA is accessed through an IDL interface, but it cannot be used remotely. The CCM introduces a new keyword, `local`, to define locality constrained interfaces in IDL, which eliminates several ambiguities with respect to the semantics of these interfaces.

Adding standard support for locality-constrained interfaces to CORBA is particularly important for server-centric components because it helps improve performance and minimizes memory footprint. Nevertheless, for components that can be accessed both remotely and locally, a quality ORB should be able to optimize the operation invocation performance while maintaining the object transparency property of these objects.[1]

Extensions to the Interface Repository: The Interface Repository is a standard mechanism in CORBA that allows applications to discover, at run-time, the definition of IDL interfaces and data types. In essence, it provides instrospection capabilities for CORBA. The CCM specification extends the Interface

---

[1] Unlike EJB, the CORBA component model uses CORBA as its wiring mechanism. CCM's does not mandate the use of remote interfaces to maintain object location transparency.

Repository to support components and provides operations to traverse all the interfaces of components. Interface introspection plays a major role in the CCM programming model because it enables component to adapt to an unfamiliar component at run-time. Although the Interface Repository also defines a component introspection mechanism, the component's introspection mechanisms inherited from `CCMObject` are preferred for efficiency because operation invocations on them are collocated.

Extension to IDL Language: In previous CORBA specifications, the IDL compiler relied in the textual inclusion capabilities of the C preprocessor to import external IDL constructs. Likewise `#pragma` directives where used to define interface repository identifiers. Unfortunately, these features resulted in extraneous constructs for languages different from C or C++. Moreover, the C preprocessor performs textual substitution without consideration for the scoping rules of IDL. Therefore, the CCM specification extends the IDL language with constructs to import declarations of IDL types defined in separate files, as well as defining standard mechanisms to control Interface Repository identifiers.

Related Technologies

Enterprise Java Beans (EJB): The CCM specification is modeled closely on the Enterprise Java Beans (EJB) specification (Thomas, 1998). Unlike EJB, however, the CCM uses the CORBA object model as its underlying object interoperability architecture and thus is not bound to a particular programming language. Since the two technologies are similar, the CCM specification also defines the standard mappings between the two standards. Therefore, a CCM component can appear as an EJB "bean" to EJB clients and an EJB bean can appear as a CCM component by using appropriate bridging techniques. EJB also support CORBA IIOP as its communication framework. We believe the CCM and EJB are mutually complement to each other.

Microsoft COM+: Unlike CORBA, Microsoft's COM+ was designed to support collocated component programming model initially and later added the ability to distribute COM objects. In addition, COM+ includes commonly used business services, such as the Microsoft Transaction Service (MTS). The CORBA specification defines a bridging mechanism between CORBA objects and DCOM components. However, unlike CORBA and EJB, COM+ is limited mostly on Microsoft platforms.

Concluding Remarks

The CORBA object model is increasingly accepted as a standard, cross-platform, cross-language distributed object computing framework. The recent addition of the CORBA Component Model (CCM) integrates a successful component programming model from EJB, while maintaining the interoperability and language-neutrality of CORBA. Therefore, we believe the CCM programming model is suitable for leveraging proven technologies and existing services in order to develop the next-generation of highly scalable distributed applications. As shown in the chapter above, however, the CCM specification is large and complex. Therefore, ORB providers have only started implementing the specification recently. Just as first-generation CORBA implementations several years ago, the quality and performance of CCM implementations remains an open issue. Moreover, the interoperability of components and containers from different providers is not well understood at this time.

Over time, however, we expect that CCM providers will implement the complete specification, as well as support value-added enhancements to their implementations, just as operating system providers have done historically. In particular, containers provide an ideal run-time environment for imbuing quality of service (QoS) capabilities into CCM components, and can be extended to provide more services to components to relieve components from implementing these functionalities in an *ad-hoc* way (Wang, 2000b). These container QoS extensions provide services that can monitor and control certain aspects of components behaviors that cross-cut different programming layers or require close interaction among components, containers, and operating systems. As CORBA and the CCM evolve, we expect some of these enhancements will be incorporated into the CCM specification.

BIBLIOGRAPHY

- Object Management Group (1999a), *The Common Object Request Broker: Architecture and Specification*, version 2.3.
- Object Management Group (1998), *Interoperable Naming Service Specification*.
- Henning, M. and Vinoski, S (1999), *Advance CORBA Programming with C++*, Addison-Wesley Longman, Reading, MA.

- Schmidt, D. et. al. (1995), *Experience Developing an Object-Oriented Framework for High-Performance Electronic Medical Imaging using CORBA and C++*, Proceedings of the ‘‘Software Technology Applied to Imaging and Multimedia Applications mini-conference” at the Symposium on Electronic Imaging in the International Symposia Photonics West, San Jose, CA.
- Levine, D. et. al. (1998), *Dynamic Scheduling Strategies for Avionics Mission Computing*, Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC), Seattle, WA.
- O’Ryan, C and Schmidt, D. (1999), *Applying a Real-time CORBA Event Service to Large-scale Distributed Interactive Simulation*, 5th International Workshop on Object-oriented Real-Time Dependable Systems, Monterey, CA.
- Irfan Pyarali, Carlos O’Ryan, and Douglas C. Schmidt, *A Pattern Language for Efficient, Predictable, Scalable, and Flexible Dispatching Mechanisms for Distributed Object Computing Middleware,* Proceeding of the IEEE/IFIP “International Symposium on Object-Oriented Real-time Distributed Computing”, March 15-17, 2000, Newport Beach, California.
- Box, D. (1997), *Essential COM,* Addison-Wesley, Reading, MA.
- Gamma, E. et. al. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, MA.
- Wang, N. et. al. (2000a), *Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation*, 24th Computer Software and Applications Conference, Taipei, Taiwan.
- Wang, N. et. al. (2000b), *Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications*, Reflective Middleware Workshop, ACM/IFIP, Pallisades, NY.
- Object Management Group (1999b), *CORBA Component Model Joint Revised Submission*.
- Thomas, A. et. al. (1998), *Enterprise JavaBeans Technology*,
  URL: `http://java.sun.com/products/ejb/white_paper.html`