Application Servers
G22.3033-003

Session 8 – Main Theme
CORBA 3
(http://www.ditec.um.es/~dsevilla/ccm/)

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Contents

- Introduction
- CORBA
- CORBA Component Model (CCM)
  - Component Model
  - Container Model
  - Packaging & Deployment
  - ORB extensions
Introduction

- CORBA Success:
  - Distributed **OBJECT** middleware
  - Platform transparency (HW + OS)
  - Language independence
  - Location transparency

Introduction (ii)

- CORBA failed (by now):
  - No standard way of deploying object implementations (non-standard IMR, etc.)
  - Lack of support for common programming idioms for CORBA servers
  - Difficulty extending object functionalities (inheritance vs. composition)
  - Lack of “mandatory” Object Services
  - No standard or automatic Life Cycle management
Introduction (iii)

- Proposed solution: CORBA Component Model (CCM)
  - Will be part of the new CORBA 3.0 specification
  - Component model extends current CORBA object model
  - Standardizes development and deployment cycle, run-time services and interfaces offered to components
  - Based on EJB and COM, but limited to neither Java nor Windows → Conforms to CORBA philosophy
  - Allows non-component aware clients → “Soft migration”

CORBA

- Distributed object middleware:
  - It is object oriented (interface-based): client objects request services to server objects via method invocation
  - It is platform-independent
  - It is language independent: clients and servers can be implemented in any language
  - It makes location transparent to either clients and servers: the ORB acts as a “mediator” abstracting:
    - Object location (object invocations are always local)
    - Networking issues
    - Activation issues (whether server objects are active or not)
    - Persistent state (whether server objects are persistent or not)
CORBA (ii)

- How gets all of this achieved?
  - Object-based: by definition
    - Traditional object model
  - Platform-independent
    - Each ORB vendor implements the standard in the desired platform
    - Common on-the-wire protocol: GIOP and IIOP
      - CDR: Overcomes endianness, data types, etc.

CORBA (iii)

- Language-independent:
  - Standardized HIERARCHY and TYPE SET through a common interface definition language (IDL):
    ```
    interface Something : Entity {
      long someOperation(in string value);
    }
    ```
  - Standard mappings from IDL to languages such as Java, C++, COBOL, Smalltalk, etc.
  - Clients written in any language can interact with server written in any other
CORBA (iv)

**Java Client**

```java
// Obtain reference
Something som = // Obtain reference
java.lang.String myString = "Hello";
int theValue = som.someOperation( myString );
System.out.println( theValue + "" );
```

**C++ Client (even with polymorphism)**

```cpp
// Obtain reference to the most derived type
Entity *myEntity = ... // Obtain reference
const char* myString = "Some test string";
// "someEntityOp" is a method of "Entity" that returns a reference to
// an object of type "SomeOtherInterface"
SomeOtherInterface_var sov = myEntity->someEntityOp( myString );
```

CORBA (v)

**Location transparency**

- The IDL compiler generates “proxys” or “mediators”
- Proxy acts as the representative of the object, and hide the (possibly) remote location of the object
- Calls are always local: the proxy, using the ORB, “forwards” the request to the correct server object.
CORBA (vi)

Server side:
- The IDL compiler generates "skeletons"
- Hooks where the programmer inserts each method implementation
- The compiler offers:
  
  ```cpp
  CORBA::Long Something::someOperation(const char* str) {
    // TODO: Implement here
  }
  ```
- The programmer (assume string length):
  
  ```cpp
  for (CORBA::Long i = 0; *str++; i++) {
    return i;
  }
  ```

CORBA (vii)

Server-side management of implementation objects (called servants):
- It uses POAs (Portable Object Adapter)
- POAs mediate between servants and ORB
- A POA manages a group of related objects
  - Manages object activation and deactivation
  - It offers policies to:
    - Ensure persistent references
    - Assign application-specific object identifiers
    - Assign multiple references to one servant
    - Threading issues, etc.
CORBA (viii)

- Problems (recall introduction):
  - Which services are available to objects?
  - How do I deploy my servers? (ORB-specific)
  - Where do I publish “bootstrap” references? Naming, Trading? Which structure?
  - Who activates my objects? IMR? But... NO STANDARD INTERFACE TO IMR!!!
  - Which POA policies should I use?
  - How do I manage my objects’ life cycle? Do I have to write ad-hoc factories?
  - I’m sure there are COMMON PATTERNS!!

CORBA Component Model (CCM)

- Extends the CORBA object model
- Components enforce composition rather than inheritance
- Based on EJB: standardization of
  - Events, Concurrency, Transactions
  - Managed Persistence
  - Deployment (installed in a component server)
  - All offered through a simplified interface
CCM (ii)

- Modifications to CORBA:
  - Component model
  - Container model
  - Component Implementation Framework (CIF)
  - New Component Implementation Description Language (CIDL)
  - Packaging and deployment
  - Interface Repository extensions
  - ORB extensions

CCM (iii) - Deployment
CCM (iv) - Component Model

- IDL extensions to define components
- Components have "ports" that
  - Define its possible connection graph
  - Define its requirements and offerings
  - Allow component configuration
- Ports:
  - Facets
  - Receptacles
  - Event Sources/Sinks
  - Attributes

CCM (v) - Component Model

- Component declaration & facets
  - We specify a name for the component
  - "Supports" establishes the component's supported interfaces:
    - for non-component-aware clients
    - equivalent to interface inheritance supported in IDL
  - Facets are aggregated interfaces also supported by a component
    - Can be obtained through a navigation interface
  - The Navigation interface is inherited by all components, as all components inherit from CCMObject that inherits from Navigation
  - Navigation is similar to COM's IUnknown interface
CCM (vi) - Component Model

Example (extended IDL):

```java
component Button : GraphicControl supports Embeddable {
    provides Embeddable embed;
    provides Printable print;
    provides Externalizable extern;
};

Button inherits from GraphicControl (properties, not implementation!!!)

Supports the Embeddable interface (can be narrowed to it)

Provides three interfaces
```

CCM (vii) - Component Model

Example (contd.)

- Equivalent IDL:
  ```java
  interface Button: GraphicControl, Embeddable {
      Embeddable provide_embed();
      Printable provide_print();
      Externalizable provideExtern();
  };
  ```
- Eventually GraphicControl inherits from CCMObject (and Navigation)
- Navigation offers: provide_facet, get_component, same_component, …
CCM (viii) - Component Model

- Receptacles
  - An reference of the specified interface may be connected to the component
  - Declaration:
    ```
    component blahblah ...
    {
    uses Embeddable dependant;
    }
    ```
  - The IDL compiler generates operations to connect to the receptacle
    ```
    void connect_dependant(In Embeddable emb) raises ...
    Embeddable disconnect_dependant() raises ...
    ```
  - Receptacles can also be “multiple”
  - Exists also a Receptacles interface of which CCMObject inherits (connect, disconnect, etc.)

CCM (ix) - Component Model

- Events:
  - Allow decoupled communication between components
  - The CCM offers a subset of the Notification Service (push model for both suppliers and consumers)
  - Components can declare that
    - they produce a kind of event (evt. sources)
    - they can accept some kind of event (evt. sinks)
  - Similar to receptacles, but bi-directional
CCM (x) - Component Model

- Attributes: Component Configuration
  - Similar to interface attrs. but raise excepts.
  - Allow component configuration on an instance-basis
  - Component configuration can set QoS parameters for each component instance
  - Programmers can define “configurators” that
    - configure the component
    - call the component’s configuration_complete (inherited from CCMObject), that may raise an “Invalid Configuration” exception.

CCM (xi) - Component Model

- Component Homes & Finders
  - Encapsulate factory behaviour
    - Saves the programmer of witting it
  - Two types: keyed & keyless
  - Keyed homes also include finder operations
  - Example: keyless: (extended IDL)

```idl
home ButtonHome manages Button [ <operations> ];
```

- Equivalent IDL (simplified for clarity):

```idl
interface ButtonHome {
    // Other explicit <operations>
    Button create();
}:
```
CCM (xii) - Component Model

- Component Homes & Finders (contd.)
  - Example: keyed:
    ```c++
    valuetype ButtonName : Components::PrimaryKeyBase {
        public string name; 
    };
    home ButtonHome manages Button primaryKey ButtonName { ... };
    - Equivalent IDL (simplified for clarity):
      ```c++
      interface ButtonHome {
        // Automatically generated factory operations
        Button create(in ButtonName key) raises DuplicateKey, ...;
        // Automatically generated finder operations
        Button find_by_primary_key(in ButtonName key) raises ...;
        void remove(in ButtonName key);
        ButtonName get_primary_key(in Button comp); }
    ```

CCM (xiii) - Container Model

- The Container:
  - Manages the component depending on its category
  - Offers a simplified API to all services
  - Component’s view of the world (run-time environment)
    - Internal interfaces
    - Callbacks
CCM (xiv) - Container Model

- **External APIs**
  - Interfaces available to a component client
    - `home` interface
    - `application` interface

- **Internal API (container API)**
  - Uses the new "local" IDL interfaces
  - Internal & Callback interfaces (all local)
  - Depend on component category
    - Session API for components with transient refs.
    - Entity API for component with persistent refs.

---

CCM (xv) - Container Model

- **CORBA Usage Model**
  - Describes the interaction between the container, the POA and the CORBA services, i.e. reference persistence and servant to ObjectID mapping
  - Types: stateless, conversational, durable

- **Component Categories**
  - Combination of internal and external APIs

<table>
<thead>
<tr>
<th>Usage Model</th>
<th>Container API</th>
<th>Comp. Category</th>
<th>Object Ref.</th>
<th>Servant/VOID</th>
</tr>
</thead>
<tbody>
<tr>
<td>stateless</td>
<td>session</td>
<td>Service</td>
<td>TRANSIENT</td>
<td>1:N</td>
</tr>
<tr>
<td>conversational</td>
<td>session</td>
<td>Session</td>
<td>TRANSIENT</td>
<td>1:1</td>
</tr>
<tr>
<td>durable</td>
<td>entity</td>
<td>Process</td>
<td>PERSISTENT</td>
<td>1:1</td>
</tr>
<tr>
<td>durable</td>
<td>entity</td>
<td>Entity</td>
<td>PERSISTENT</td>
<td>1:1</td>
</tr>
</tbody>
</table>
CCM (xvi) - Container Model

- Component Activation & Servant Lifetime Management
  - These are container concerns
  - Each container has a POA and a ServantLocator
  - Four types of activation/lifetime mgmt.
    - Method: Activate/passivate on a method-basis
    - Transaction: Lifetime tied to a transaction
    - Component: The component decides when to deactivate itself
    - Container: Lifetime tied to container’s lifetime

- Container also controls, on component’s behalf:
  - Transactions, Security, Events, Persistence

CCM (xvii) - Implementation Framework

- CCM Implementation Framework (CIF)
  - Generates "executors": implementation of behavioral elements (homes, containers,...)
  - Based on CIDL definitions for components

- CIDL define the Units of implementation:
  - composition keyword in CIDL. Includes:
    - Component category, component home, home executor, component executor, etc.
  - Component Implementation Definition Language is a superset of PSDL (from PSS2)
**Example: Given the following user’s IDL**

```java
module LooneyToons {
    interface Bird {
        void fly(in long howLong);
    };
    interface Cat {
        void eat(in Bird lunch);
    };
    component Toon {
        provides Bird tweety;
        provides Cat silvester;
    };
    home ToonTown manages Toon {};
};
```

**User can specify the following CIDL**

```java
import ::LooneyToons;
module MerryMelodies {
    composition session ToonImpl {
        home executor ToonTownImpl {
            implements LooneyToons::ToonTown;
            manages ToonSessionImpl;
        };
    };
};
```
CCM (xx) - CIF

Process:

CCM (xxi) - Packaging & Deployment

Developer (or RAD-tools) must generate p&d descriptors:

- XML files conforming to WWW Consortium’s Open Software Descriptor DTD
- Describes the contents of DLLs (files, IDL, interfaces, components), requirements (other DLLs required, ORBs), assembly instructions, properties & values, etc.
- Helps the component server to maintain, install and automatically load/unload DLLs
- This information is stored in ZIP files
CCM (xxii) - ORB Extensions

- Locality-constrained interfaces
- Extensions to Interface Repository
  - New type ComponentDef
  - New get_component_def of CCMObject returning CORBA::IRObject that narrows to:
  - New IR::ComponentDef type
- Extensions to IDL: component, etc.

CCM (xxiii) - References & OMG docs

- OMG Specifications (as of Jun, 2000):
  - CORBA 2.3.1: orbos/99-10-07 (2.4 soon!)
  - CCM: ptc/99-10-04
  - PSS 2.0: orbos/99-07-07
- Douglas C. Schmidt’s work on CCM:
  - middleware2000.ps.gz
  - COMPSAC.ps.gz
  - CBSE.pdf