Using the Lightweight CORBA Component Model to Develop Distributed Real-time & Embedded Applications


Dr. Douglas C. Schmidt & William R. Otte

schmidt@dre.vanderbilt.edu & wotte@dre.vanderbilt.edu

http://www.dre.vanderbilt.edu

Electrical Engineering & Computer Science
Vanderbilt University Nashville, Tennessee

Other contributors include Jai Balasubramanian, Kitty Balasubramanian, Gan Deng, Tao Lu, Bala Natarajan, Jeff Parsons, Frank Pilhofer, Craig Rodrigues, & Nanbor Wang
Motivation & Overview of Component Middleware

www.cs.wustl.edu/~schmidt/cuj-16.doc
Where We Started: Object-Oriented Programming

- Object-oriented (OO) programming simplified software development through higher level abstractions & patterns, e.g.,
  - Associating related data & operations
  - Decoupling interfaces & implementations

Well-written OO programs exhibit recurring structures that promote abstraction, flexibility, modularity, & elegance
Next Step: Distributed Object Computing (DOC)

• Apply the Broker pattern to abstract away lower-level OS & protocol-specific details for network programming

• Create distributed systems which are easier to model & build using OO techniques

• Result: robust distributed systems built with distributed object computing (DOC) middleware
  
  – e.g., CORBA, Java RMI, etc.

We now have more robust software & more powerful distributed systems
Overview of CORBA 2.x Standard

• CORBA 2.x is DOC middleware that shields applications from dependencies on heterogeneous platforms
  • e.g., languages, operating systems, networking protocols, hardware

• CORBA 2.x automates
  – Object location
  – Connection & memory mgmt.
  – Parameter (de)marshaling
  – Event & request demultiplexing
  – Error handling & fault tolerance
  – Object/server activation
  – Concurrency & synchronization
  – Security

CORBA 2.x defines interfaces & policies, but not implementations
Example: Applying OO to Network Programming

• CORBA 2.x IDL specifies *interfaces* with operations
  – Interfaces map to objects in OO programming languages
    • e.g., C++, Java, Ada95, etc.

```idl
interface Foo
{
    void bar (in long arg);
};
```

```cpp
class Foo : public virtual CORBA::Object
{
    virtual void bar (CORBA::Long arg);
};
```

– Operations defined in interfaces can be invoked on local or remote objects
Drawbacks of DOC-based CORBA 2.x Middleware

CORBA 2.x application development is unnecessarily tedious & error-prone

- CORBA 2.x IDL doesn’t provide a way to group together related interfaces to offer a service family
  - Such “bundling” must be done by developers via CORBA idioms & patterns
- CORBA 2.x doesn’t specify how configuration & deployment of objects should be done to create complete applications
  - Proprietary infrastructure & scripts are written by developers to enable this
Example: Limitations of CORBA 2.x Specification

- Requirements of non-trivial DRE systems:
  - Collaboration of multiple objects & services
  - Deployment on diverse platforms
- CORBA 2.x limitations – lack of **standards** for
  - Server/node configuration
  - Object/service configuration
  - Application assembly
  - Object/service deployment
- Consequences:
  - Brittle, non-scalable implementation
  - Hard to adapt & maintain
  - Increased time-to-market
Solution: Component Middleware

- Creates a standard “virtual boundary” around application component implementations that interact only via well-defined interfaces
- Define standard container mechanisms needed to execute components in generic component servers
- Specify the infrastructure needed to configure & deploy components throughout a distributed system

```xml
<ComponentAssemblyDescription id="a_HUDDisplay"> ...
  <connection>
  <name>GPS-RateGen</name>
  <internalEndPoint><portName>Refresh</portName><instance>a_GPS</instance></internalEndPoint>
  <internalEndPoint><portName>Pulse</portName><instance>a_RateGen</instance></internalEndPoint>
  </connection>
  <connection>
  <name>NavDisplay-GPS</name>
  <internalEndPoint><portName>Refresh</portName><instance>a_NavDisplay</instance></internalEndPoint>
  <internalEndPoint><portName>Ready</portName><instance>a_GPS</instance></internalEndPoint>
  </connection> ...
</ComponentAssemblyDescription>
```
Overview of the Lightweight CORBA Component Model (CCM)
Capabilities of CORBA Component Model (CCM)

- **Component Server**
  - A generic server process for hosting containers & component/home executors

- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features

- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies

- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers

- Containers define operations that enable component executors to access common middleware services & runtime policies
Capabilities of CORBA Component Model (CCM)

• Component Server
  – A generic server process for hosting containers & component/home executors

• Component Implementation Framework (CIF)
  – Automates the implementation of many component features

• Component packaging tools
  – Compose implementation & configuration information into deployable assemblies

• Component deployment tools
  – Automate the deployment of component assemblies to component servers
Capabilities of CORBA Component Model (CCM)

- **Component Server**
  - A generic server process for hosting containers & component/home executors

- **Component Implementation Framework (CIF)**
  - Automates the implementation of many component features

- **Component packaging tools**
  - Compose implementation & configuration information into deployable assemblies

- **Component deployment tools**
  - Automate the deployment of component assemblies to component servers
Capabilities of CORBA Component Model (CCM)

• **Component Server**
  – A generic server process for hosting containers & component/home executors

• **Component Implementation Framework (CIF)**
  – Automates the implementation of many component features

• **Component packaging tools**
  – Compose implementation & configuration information into deployable assemblies

• **Component deployment tools**
  – Automate the deployment of component assemblies to component servers
# Available CCM Implementations

<table>
<thead>
<tr>
<th>Name</th>
<th>Provider</th>
<th>Open Source</th>
<th>Language</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Integrated ACE ORB (CIAO)</td>
<td>Vanderbilt University &amp; Washington University</td>
<td>Yes</td>
<td>C++</td>
<td><a href="http://www.dre.vanderbilt.edu/CIAO/">www.dre.vanderbilt.edu/CIAO/</a></td>
</tr>
<tr>
<td>Enterprise Java CORBA Component Model (EJCCM)</td>
<td>Computational Physics, Inc.</td>
<td>Yes</td>
<td>Java</td>
<td><a href="http://www.cpi.com/ejccm/">www.cpi.com/ejccm/</a></td>
</tr>
<tr>
<td>K2</td>
<td>iCMG</td>
<td>No</td>
<td>C++</td>
<td><a href="http://www.icmgworld.com/products.asp">www.icmgworld.com/products.asp</a></td>
</tr>
<tr>
<td>MicoCCM</td>
<td>FPX</td>
<td>Yes</td>
<td>C++</td>
<td><a href="http://www.fpx.de/MicoCCM/">www.fpx.de/MicoCCM/</a></td>
</tr>
<tr>
<td>OpenCCM</td>
<td>ObjectWeb</td>
<td>Yes</td>
<td>Java</td>
<td>openccm.objectweb.org/</td>
</tr>
<tr>
<td>QoS Enabled Distributed Object (Qedo)</td>
<td>Fokus</td>
<td>Yes</td>
<td>C++</td>
<td><a href="http://www.qedo.org">www.qedo.org</a></td>
</tr>
<tr>
<td>StarCCM</td>
<td>Source Forge</td>
<td>Yes</td>
<td>C++</td>
<td>sourceforge.net/projects/starccm/</td>
</tr>
</tbody>
</table>
CCM Compared to EJB, COM, & .NET

- Like Sun Microsystems’ Enterprise Java Beans (EJB)
  - CORBA components created & managed by homes
  - Run in containers that manage system services transparently
  - Hosted by generic application component servers
  - But can be written in more languages than Java

- Like Microsoft’s Component Object Model (COM)
  - Have several input & output interfaces per component
  - Both point-to-point sync/async operations & publish/subscribe events
  - Component navigation & introspection capabilities
  - But has more effective support for distribution & QoS properties

- Like Microsoft’s .NET Framework
  - Could be written in different programming languages
  - Could be packaged to be distributed
  - But runs on more platforms than just Microsoft Windows
Comparing Application Development with CORBA 2.x vs. CCM
CORBA 2.x User Roles

- Object interface designers
- Server developers
- Client application developers
CORBA 2.x Application Development Lifecycle

Specification of IDL interfaces of objects

Implement servants & write all the code required to bootstrap & run the server

Interface Design

IDL Definitions

IDL Compiler

Stubs & Skeletons

Application Development & Deployment

Object Implementations

“Other” Implementations

Language Tools

Libraries

Applications

CORBA 2.x supports programming by development (engineering) rather than programming by assembly (manufacturing)
CCM User Roles

- Component designers
- Component clients
- Composition designers
- Component implementers
- Component packagers
- Component deployers
- Component end-users
CCM Application Development Lifecycle

Interface IDL Definitions

Component Design

Component Implementation

Component Packaging

Specification of IDL
2 types, e.g., supported interfaces
CCM Application Development Lifecycle

Specification of IDL 3 types, e.g., provided & required interfaces
Implementation of component *executors*, plus association of components with component executors & their homes via the *Component Implementation Definition Language (CIDL)*.
Grouping of component implementation artifacts & metadata descriptors into component packages.

CCM Application Development Lifecycle
Specification of component interconnections & composition of component assembly packages
CCM Application Development Lifecycle

Interface Design
- Interface IDL Definitions
  - IDL Compiler

Component Design
- Component IDL Definitions
  - Stubs & Skeletons
  - CIDL Compiler

Component Implementation
- Object Implementations
- Servants, Executors, Contexts
- Language Tools

Component Packaging
- Component DLLs
  - Packaging Tools
  - Component Domain Descriptors (.ccd)
  - Component Package Descriptors (.cpd)
  - Component Assembly Descriptors
    - Component Assembly Descriptor (* .cid)

Component & Home Properties
- Component Deployment Descriptors (.cid)
- Component Implementation Descriptors (.iad)
- Monolithic Component Description

Deployment Planning
- Deployment Plan Descriptors (.cdp)
  - Deployment Planning Tools
    - Deployment Planning Descriptor (.cdp)
    - Component Package Descriptors (.cpd)
    - Component Assembly Tools
      - Component & Home Properties
        - Component Assembly Descriptor

Specification of deployment target domain & configuration of component assembly

Running Applications
- Deployment Tools

System Deployment
- Assembly Tools
  - Component & Home Properties
    - Component Deployment Descriptors (.cdp)
The diagram illustrates the CCM Application Development Lifecycle (ADL). The lifecycle is divided into four main phases: Interface Design, Component Design, Component Implementation, and Component Packaging.

1. **Interface Design**: This phase involves defining the interface using Interface IDL Definitions. The IDL Compiler is used to generate Stubs & Skeletons, which are then compiled into Component IDL Definitions.

2. **Component Design**: Component IDL Definitions are then refined into Component CIDL Definitions. The CIDL Compiler is used to generate Language Tools, Servants, Executors, and Contexts.

3. **Component Implementation**: This phase involves implementing the components. Object Implementations are generated using an IDL Compiler, and objects are realized using Servants, Executors, Contexts.

4. **Component Packaging**: The implemented components are packaged using Package Tools. Component DLLs are generated, and the Component & Home Properties are realized using Component & Home Properties.

The lifecycle is further divided into System Deployment and Deployment Planning.

- **System Deployment**: This phase involves running applications and deploying them onto target nodes. Deployment Tools are used to generate Deployment Plan Descriptors (.cdp), which are used by Deployment Tools to assemble deployment plans.

- **Deployment Planning**: This phase involves planning and generating deployment plans. Deployment Planning Tools are used to generate Deployment Plan Descriptors (.cdp), which are then deployed using Deployment Planning Tools.

The diagram highlights the process of deploying component assembly packages onto target nodes according to a deployment plan.
CCM Application Development Lifecycle

CCM makes explicit steps performed implicitly in CORBA 2.x
CORBA Component Model (CCM) Features
Example CCM DRE Application

- **Rate Generator**
  - Sends periodic *Pulse* events to consumers

- **Positioning Sensor**
  - Receives *Refresh* events from suppliers
  - Refreshes cached coordinates available thru *MyLocation* facet
  - Notifies subscribers via *Ready* events

- **Display Device**
  - Receives *Refresh* events from suppliers
  - Reads current coordinates via its *GPSLocation* receptacle
  - Updates display

Avionics example used throughout tutorial as typical DRE application

Component Server

$CIAO_ROOT/examples/OEP/Display/
Unit of Business Logic & Composition in CCM

- **Context**
  - Development via *composition*

- **Problems**
  - CORBA 2.x object limitations
    - Objects just identify interfaces
    - No direct relation w/implementations

- **CCM Solution**
  - Define CORBA 3.0 `component` meta-type
    - Extension of CORBA 2.x `object` interface
    - Has interface & object reference
    - Essentially a stylized use of CORBA interfaces/objects
      - i.e., CORBA 3.x IDL maps onto equivalent CORBA 2.x IDL
A CORBA component can contain *ports*:

- **Facets** (*provides*)
  - Offers operation interfaces
- **Receptacles** (*uses*)
  - Required operation interfaces
- **Event sources** (*publishes & emits*)
  - Produced events
- **Event sinks** (*consumes*)
  - Consumed events
- **Attributes** (*attribute*)
  - Configurable properties

Each component instance is created & managed by a unique component *home*
Monolithic Component Implementation

• Executable piece of software
  – One or more “implementation artifacts” (e.g., .exe, .so, .o, .class)
  – Zero or more supporting artifacts (e.g., configuration files)

• May have hardware or software requirements/constraints
  – Specific CPU (e.g., x86, PPC, SPARC)
  – Specific OS (e.g., Windows, VxWorks, Linux, Solaris)
  – Hardware devices (e.g., GPS sensor)

• Described by metadata, e.g., *.ccd, *.iad, & *.cid files
Assembly-based Component Implementation

• Set of interconnected (sub)components
• Hardware & software independent
  – Reuses subcomponents as “black boxes,” independent of their implementation
• Implements a specific (virtual) component interface
  – i.e., external ports & attributes are “mapped” to internal subcomponents
• Assemblies are fully reusable
  – Can be “standalone” applications or reusable components
• Assemblies are hierarchical
  – i.e., can be used in an encompassing assembly
  – Note recursion here…
• Described by metadata, e.g., *.ccd & *.cid files
Simple CCM Component Example

• Roles played by CCM component
  – Define a unit of composition, reuse, & implementation
  – Encapsulate an interaction & configuration model

• A CORBA component has several derivation options, i.e.,
  – It can inherit from a single component type
    component E : D {}
  – It can support multiple IDL interfaces

```
// IDL 3
interface rate_control
{
    void start ();
    void stop ();
};

component RateGen
    supports rate_control {};

// Equivalent IDL 2
interface RateGen :
    ::Components::CCMOBJECT,
    rate_control {};
```

```
interface A {}
interface B {}
component D supports A, B {};
```
Managing Component Lifecycle

• Context
  – Components need to be created by the CCM run-time

• Problems with CORBA 2.x
  – No standard way to manage component’s lifecycle
  – Need standard mechanisms to strategize lifecycle management

• CCM Solution
  – Integrate lifecycle service into component definitions
  – Use different component home’s to provide different lifecycle managing strategies
    • Based on Factory & Finder patterns
A CORBA Component Home

- **home** is new CORBA meta-type
  - A **home** has an interface & object reference

- Manages one type of component
  - More than one home type can manage same component type
  - However, a component instance is managed by one home instance

- Standard *factory* & *finder* operations
  - e.g., `create()`

- **home** can have user-defined operations

```idl
// IDL 3
home RateGenHome manages RateGen
{
  factory create_pulser
  (in rateHz r);
};

// Equivalent IDL 2
interface RateGenHomeExplicit
: Components::CCMHome {
  RateGen create_pulser
  (in rateHz r);
};
interface RateGenHomeImplicit
: Components::KeylessCCMHome {
  RateGen create ();
};
interface RateGenHome :
  RateGenHomeExplicit,
  RateGenHomeImplicit {};
```
A Quick CCM Client Example
Component & Home for Simple **HelloWorld**

```java
interface Hello {
    void sayHello (in string username);
};
interface Goodbye {
    void sayGoodbye (in string username);
};
component HelloWorld supports Hello {
    provides Goodbye Farewell;
};
home HelloHome manages HelloWorld {};
```

- IDL 3 definitions for
  - Component: **HelloWorld**
  - Managing home: **HelloHome**
- Example in $CIAO_ROOT/docs/tutorial/Hello/
The Client OMG IDL Mapping

- As we’ve seen, each OMG IDL 3.0 construction has an equivalent in terms of OMG IDL 2.x

- Component & home types are viewed by clients through the CCM client-side OMG IDL mapping

- This mapping requires no change in CORBA’s client programming language mapping
  
  - i.e., clients still use their favorite IDL-oriented tools, such as CORBA stub generators, etc.

- Clients need not be “component-aware”
  
  - i.e., they can just invoke interface operations
Simple Client for **HelloWorld** Component

```c
1 int
2 main (int argc, char *argv[])
3 {
4    CORBA::ORB_var orb = 
5       CORBA::ORB_init (argc, argv);
6    CORBA::Object_var o = 
7       orb->resolve_initial_references
8       ("NameService");
9    CosNaming::NamingContextExt_var nc = 
10       CosNaming::NamingContextExt::_narrow (o);
11    o = nc->resolve_str ("myHelloHome");
12    HelloHome_var hh = HelloHome::_narrow (o);
13    HelloWorld_var hw = hh->create ();
14    hw->sayHello ("Dennis & Brian");
15    hw->remove ();
16    return 0;
17 }
```

- Lines 4-10: Perform standard ORB bootstrapping
- Lines 11-12: Obtain object reference to home via Naming Service
- Line 13: Use home to create component
- Line 14: Invoke remote operation
- Line 15: Remove component instance
- Clients don’t always need to manage component lifecycle directly

```bash
$ ./hello-client # Triggers this on the server:
Hello World! -- from Dennis & Brian.
```
CCM Component Features in Depth

www.cs.wustl.edu/~schmidt/cuj-17.doc
Components Can Offer Different Views

• Context
  – Components need to collaborate with other types of components
  – These collaborating components may understand different interfaces

• Problems with CORBA 2.x
  – Hard to extend interface without breaking/bloating it
  – No standard way to acquire new interfaces

• CCM Solution
  – Define facets, a.k.a. provided interfaces, that embody a view of the component & correspond to roles in which a client may act relatively to the component
    • Represents the “top of the Lego”
Component Facets

- Facet characteristics:
  - Define *provided* operation interfaces
  - Specified with *provides* keyword
    - Logically represents the component itself, not a separate entity contained by the component
  - However, facets have independent object references obtained from *provide_*() factory operation
    - Can be used to implement *Extension Interface* pattern

```idl
// IDL 3
interface position
{
    long get_pos ();
};
component GPS
{
    provides position MyLocation;
    ...
};

// Equivalent IDL 2
interface GPS
    : Components::CCMObject
{
    position
        provide_MyLocation ();
    ...
};
```
The *Extension Interface* design pattern (POSA2) allows multiple interfaces to be exported by a component to prevent

- breaking of client code &
- bloating of interfaces

when developers extend or modify component functionality.
Using Other Components

• Context

  – Components need to collaborate with several different types of components/applications
  – These collaborating components/applications may provide different types of interfaces

• Problems with CORBA 2.x

  – No standard way to specify interface dependencies
  – No standard way to connect an interface to a component

• CCM Solution

  – Define receptacles, a.k.a. required interfaces, which are distinct named connection points for potential connectivity

  • Represents the “bottom of the Lego”
Component Receptacles

- Receptacle characteristics
  - Define a way to connect one or more *required* interfaces to this component
  - Specified with `uses (multiple)` keyword
  - Can be *simplex* or *multiplex*

- Connections are established *statically* via tools during deployment phase
- Connections are managed *dynamically* at run-time by containers to offer interactions with clients or other components via callbacks
- CCM also enables connection establishment during run-time

```idl
// IDL 3
class NavDisplay {
    ...
    uses position GPSLocation;
    ...
};

// Equivalent IDL 2
interface NavDisplay : Components::CCMObject {
    ...
    void connect_GPSLocation (in position c);
    position disconnect_GPSLocation();
    position get_connection_GPSLocation ();
    ...
};
```
Event Passing

• Context
  – Components often want to communicate using publisher/subscriber message passing mechanism

• Problems with CORBA 2.x
  – Standard CORBA Event Service is dynamically typed, i.e., there’s no static type-checking connecting publishers/subscribe
  – Non-trivial to extend request/response interfaces to support event passing
  – No standard way to specify an object’s capability to generate & process events

• CCM Solution
  – Standard `eventtype` & `eventtype` consumer interface (which are based on `valuetypes`)
  – Event sources & event sinks (“push mode” only)
Component Events

// IDL 3

eventtype tick
{
    public rateHz Rate;
};

// Equivalent IDL 2

valuetype tick : Components::EventBase
{
    public rateHz Rate;
};

interface tickConsumer :
    Components::EventConsumerBase
{
    void push_tick
        (in tick the_tick);
};

- Events are implemented as IDL valuetypes
- Defined with the new IDL 3 eventtype keyword
  - This keyword triggers generation of additional interfaces & glue code
Component Event Sources

• Event source characteristics
  – Named connection points for event production
  – Two kinds of event sources: *publisher* & *emitter*
  • *publishes* = may be multiple consumers
  • *emits* = only one consumer

– Two ways to connect with event sinks
  1. Consumer connects directly
  2. CCM container mediates access to CosNotification/CosEvent channels or other event delivery mechanism (e.g., OMG DDS, RtEC, etc.)

```idl
// IDL 3
component RateGen
{
  publishes tick Pulse;
  emits tick Trigger;
  ...
};

// Equivalent IDL 2
interface RateGen :
Components::CCMObject {
Components::Cookie
  subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
  unsubscribe_Pulse
    (in Components::Cookie ck);
  ...
};
```
Component Event Sinks

- Event sink characteristics
  - Named connection points into which events of a specific type may be pushed
  - Multiple event sinks of same type can subscribe to the same event sources
  - No distinction between emitter & publisher
  - Connected to event sources via object reference obtained from `get_consumer_*()` factory operation

```
// IDL 3
component NavDisplay
{
  ...
  consumes tick Refresh;
};

// Equivalent IDL 2
interface NavDisplay : Components::CCMObject
{
  ...
  tickConsumer
    get_consumer_Refresh();
  ...
};
```
CCM Events

- Context
  - Generic event `push()` operation requires a generic event type

- Problem
  - User-defined event types are not generic

- CCM Solution
  - `EventBase` abstract valuetype

```cpp
// IDL 2
valuetype tick :
  Components::EventBase {...};

interface tickConsumer :
  Components::EventConsumerBase {...};

// C++ mapping
class tickConsumer : // ...
{
  virtual void push_event
    (Components::EventBase *evt);

  ...
};
```

Enables both statically- & dynamically-typed event passing
Connecting Components

• Context
  – Components need to be connected together to form complete applications

• Problems
  – Components can have multiple ports with different types & names
  – It’s not scalable to write code manually to connect a set of components for a specific application

• CCM Solutions
  – Provide introspection interface to discover component capability
  – Provide generic port operations to connect components using external deployment & configuration tools
  – Represents snapping the lego bricks together
Generic Port Operations

<table>
<thead>
<tr>
<th>Port</th>
<th>Equivalent IDL2 Operations</th>
<th>Generic Port Operations (CCMObject)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facets</td>
<td>provide_name ();</td>
<td>provide (&quot;name&quot;);</td>
</tr>
<tr>
<td>Receptacles</td>
<td>connect_name (con); disconnect_name ();</td>
<td>connect (&quot;name&quot;, con); disconnect (&quot;name&quot;);</td>
</tr>
<tr>
<td>Event sources (publishes only)</td>
<td>subscribe_name (c); unsubscribe_name ();</td>
<td>subscribe (&quot;name&quot;, c); unsubscribe (&quot;name&quot;);</td>
</tr>
<tr>
<td>Event sinks</td>
<td>get_consumer_name();</td>
<td>get_consumer (&quot;name&quot;);</td>
</tr>
</tbody>
</table>

- Generic port operations for provides, uses, subscribes, emits, & consumes keywords are auto-generated by the CIDL compiler
  - Apply the Extension Interface pattern
  - Used by CCM deployment & configuration tools
  - Lightweight CCM spec doesn’t include equivalent IDL 2 operations
Example of Connecting Components

CCM components are connected via deployment tools during launch phase

- Facet $\rightarrow$ Receptacle
  
  $$\text{objref} = \text{GPS}$\rightarrow$provide ($"\text{MyLocation}"$);
  
  $\text{NavDisplay}$-$\rightarrow$connect ($"\text{GPSLocation}$", objref);

- Event Source $\rightarrow$ Event Sink
  
  consumer = $\text{NavDisplay}$-$\rightarrow$get_consumer ($"\text{Refresh}$")
  
  $\text{GPS}$-$\rightarrow$subscribe ($"\text{Ready}$", consumer);

Connected object references are managed by containers
Component Deployment & Configuration
Overview of Deployment & Configuration Process

- **Goals**
  - Ease component reuse
  - Build complex applications by assembling existing components
  - Deploy component-based application into heterogeneous domain(s)

- **Separation of concerns & roles**
  - Component development & packaging
  - Application assembly
  - Application configuration
  - Application deployment
  - Server configuration
Component Configuration Problem

Component middleware & applications are characterized by a large configuration space that maps known variations in the application requirements space to known variations in the solution space.

- Components interact with other software artifacts & environment to achieve specific functions
  - e.g., using a specific run-time library to encrypt & decrypt data
- Some prior knowledge of the run-time environment may be required during development
  - e.g., rates of certain tasks based on the functional role played
- Need to configure the middleware for specific QoS properties
  - e.g., transport protocols, timeouts, event correlation, concurrency/synchronization models, etc.
- Adding environment & interaction details with the business logic leads to overly tight coupling
**Concept**

- Configure run-time & environment properties late in the software lifecycle, i.e., during the deployment process

**Solution**

- **Well-defined exchange formats** to represent configuration properties
  - Can represent a wide variety of data types
  - Well-defined semantics to interpret the data
- **Well-defined interfaces** to pass configuration data from “off-line” tools to components
- **Well-defined configuration boundary** between the application & the middleware
Component Deployment Problem

• Component implementations are usually hardware-specific
  – Compiled for Windows, Linux, Java – or just FPGA firmware
  – Require special hardware
    • e.g., GPS sensor component needs access to GPS device via a serial bus or USB
    • e.g., Navigation display component needs … a display
      – not as trivial as it may sound!
• However, computers & networks are often heterogeneous
  – Not all computers can execute all component implementations
• The above is true for each & every component of an application
  – i.e., each component may have different requirements
OMG Component Deployment & Configuration Spec (1/2)

Goals of D&C Phase

• Promote component reuse
• Build complex applications by assembling existing components
• Automate common services configuration
• Declaratively inject QoS policies into applications
• Dynamically deploy components to target heterogeneous domains
• Optimize systems based on component configuration & deployment settings
OMG Component Deployment & Configuration Spec (1/2)

OMG D & C Spec (PIM & PSMs)

XML Schema Generation

IDL Generation

Interchange Formats

Deployment requirements

OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)
CCM Deployment Solution

• Well-defined exchange format
  – Defines what a software vendor delivers
  – Requires “off-line” data format that can be stored in XML files

• Well-defined interfaces
  – Infrastructure to install, configure, & deploy software
  – Requires “on-line” IDL data format that can be passed to/from interfaces

• Well-defined software metadata model
  – Annotate software & hardware with interoperable, vendor-independent, deployment-relevant information
  – Generate “on-line” & “off-line” data formats from models
    • e.g., CoSMIC at www.dre.vanderbilt.edu/cosmic
Deployment & Configuration “Segments”

<table>
<thead>
<tr>
<th>PIM</th>
<th>Data Model</th>
<th>Run-time Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Software</td>
<td>Metadata to describe component-based applications &amp; their requirements</td>
<td>Repository Manager interfaces to browse, store, &amp; retrieve such metadata</td>
</tr>
<tr>
<td>Target</td>
<td>Metadata to describe heterogeneous distributed systems &amp; their capabilities</td>
<td>Target Manager interfaces to collect &amp; retrieve such metadata &amp; commit resources</td>
</tr>
<tr>
<td>Execution</td>
<td>Metadata to describe a specific deployment plan for an application into a distributed system</td>
<td>Execution Manager interfaces to prepare environment, execute deployment plan on target, manage lifecycle</td>
</tr>
</tbody>
</table>

- Data model
  - Metadata, usually in XML format
- Run-time model
  - Deployment interfaces (similar to CORBA services)
D&C Actors

- **Development**
  - Specifier/Developer
  - Assembler
  - Packager

- **Target**
  - Domain Administrator

- **Deployment**
  - Repository Administrator
  - Planner
  - Executor
  - Actors are abstract
    - Usually humans & software tools

These actors & stages are simply making explicit existing processes.
Component Packaging

**Goal:** Associate *component implementation(s)* with *metadata*
Component-based Software: Component

• Component
  – Modular
  – Encapsulates its contents
  – Replaceable “black box”, conformance defined by interface compatibility

• Component Interface
  – “Ports” consist of provided interfaces (facets) & required (used) interfaces (receptacles)
  – Attributes

• Component Implementation
  – “Monolithic” (i.e., executable software) or
  – “Assembly-based” (a set of interconnected subcomponents)
Monolithic Component Implementation

- Executable piece of software
  - One or more “implementation artifacts” (e.g., .exe, .so, .o, .class)
  - Zero or more supporting artifacts (e.g., configuration files)

- May have hardware or software requirements/constraints
  - Specific CPU (e.g., x86, PPC, SPARC)
  - Specific OS (e.g., Windows, VxWorks, Linux, Solaris)
  - Hardware devices (e.g., GPS sensor)

- Described by *.ccd, *.iad, & *.cid files
Assembly-based Component Implementation

- Set of interconnected (sub)components
- Hardware & software independent
  - Reuses subcomponents as “black boxes,” independent of their implementation
- Implements a specific (virtual) component interface
  - i.e., external ports & attributes are “mapped” to internal subcomponents
- Assemblies are fully reusable
  - Can be “standalone” applications or reusable components
  - Assemblies are hierarchical
    - i.e., can be used in an encompassing assembly
    - Note recursion here…
- Described by *.ccd & *.cid files
Component Package

- A set of alternative, replaceable implementations of the same component interface
  - e.g., implementations for Windows, Linux, and/or JVM
- Can be a mix of monolithic & assembly-based implementations
  - e.g., a parallel, scalable implementation for a Solaris symmetric multiprocessor or a single monolithic Java component
- Implementations may have different “quality of service” (QoS)
  - e.g., latency, resolution, security
- “Best” implementation is chosen at deployment time by Planner
  - Based on available hardware & QoS requirements
Component Packaging Tools

• Goals
  – Extract systemic properties into metadata
  – Configure components, containers, target environment, & applications
  – Provide abstraction of physical information, e.g., OS version, location of DLLs, etc.

• CCM component packages bring together
  – Multiple component implementations
  – Component properties
  – Descriptors (XML Files)

• Descriptors provide metadata that describe contents of a package, dependencies on other components, 3rd party DLLs, & value factories
Application Assembly

Goal: Group packages & metadata by specifying their interconnections

Interface Design
- Interface IDL Definitions
- IDL Compiler

Component Design
- Component IDL Definitions
- Component CIDL Definitions
- Stub & Skeleton Definitions
- Component IDL Compiler

Component Implementation
- Component CIDL Definitions
- Object Implementations
- Language Tools
- Component DLLs
- Component Interface Descriptors (.ccd)
- Servants, Executors, Contexts

Component Packaging
- Component & Home Properties
- Component Interface Descriptors (.ccd)
- Packaging Tools
- Deployment Planning Tools

System Deployment
- Deployment Plan Descriptor (.cdp)
- Deployment Planning Tools
- Component Domain Descriptor (.cdd)
- Assembly Tools
- Component Package Descriptors (.cpd)
- Component Assembly Descriptors
- Component & Home Properties

Application Assembly
- Component Implementation Descriptors (.cid)
- Component Implementation Descriptors (.cpd)
- Running Applications
Application Assembly Tools

• Goals
  – Compose higher level components from set of (sub)components
  – Store composition & connection information as metadata
  – Provide abstraction of logical information, e.g., interconnections
• Component assembly description specifies:
  – Subcomponent packages
  – Subcomponent instantiation & configuration
  – Interconnections
  – Mapping of ports & properties to subcomponents
• “Pure metadata” construct (no directly executable code, hardware-agnostic)
Deployment Planning

Goal: Map application assembly onto *target environment* via *deployment plan*
Deployment Planning Tools

• Goals
  – Concretize deployment metadata
  – Using Deployment Domain to describe deployment environment

• Component Deployment Plan description:
  – Flatten the assembly hierarchy -- an assembly of monolithic components
  – Deployment details – locations to deploy components
  – Interconnections
  – Mapping of ports & properties to subcomponents
Deployment

Goal: Deploy/execute application/components according to *deployment plan*
Deployment Infrastructure Overview

• Goals
  – Realize a deployment plan on its target deployment platform

• Deployment phase includes:
  – Performing work in the target environment to be ready to execute the software (such as downloading software binaries)
  – Install component instances into the target environment
  – Interconnecting & configuring component instances
Deployment Infrastructure Overview (1/2)

- **Repository Administrator**
  - **RepositoryManager**: Manages packages in repository.

- **Planner**
  - **DeploymentPlan**: Plans deployment of application based on resource data from resourceDataProvider. Resolves the package using searchPath. Produces a compatible deployment plan.

- **Executor**
  - **ExecutionManager**: Prepares and finds packages. Uses plan to execute it in the target environment.

- **TargetManager**
  - **DomainApplicationManager**: Manages domain applications.
    - **DomainApplication**: Prepares plans for deployment based on resource data from resourceDataProvider.
    - **NodeManager**: Manages nodes in the deployment plan.
      - **NodeApplicationManager**: Manages node applications.

- **NodeApplication**
  - Spawns applications for each node in the deployment plan.
Deployment Infrastructure Overview (2/2)

- **Repository Manager**
  - Database of components that are available for deployment ("staging area")

- **Target Manager**
  - Retrieval of target data (i.e., available nodes & resources)
  - "Execution" Runtime Model

- **Execution Manager**
  - Execution of an application according to a "Deployment Plan"

- **Domain Application Manager**
  - Responsible for deploying an application at the domain level

- **Domain Application**
  - Represents a "global" application that was deployed across nodes
  - "Component Software" Runtime Model

- **Node Application Manager**
  - Responsible for deploying a locality constrained application onto a node
  - "Target" Runtime Model

- **Node Application**
  - Represents a portion of an application that’s executing within a single node

www.cs.wustl.edu/~schmidt/PDF/DanCE.pdf
Deployment Infrastructure: Repository Manager

- Database of components
  - Metadata (from Component Data Model)
  - Artifacts (i.e., executable monolithic implementations)
- Applications can be configured
  - e.g., to apply custom policies, e.g., "background color" = "blue"
- Applications are installed from packages
  - ZIP files containing metadata in XML format & implementation artifacts
- CORBA interface for installation of packages, retrieval, & introspection of metadata
- HTTP interface for downloading artifacts
  - Used by Node Managers during execution
Deployment Infrastructure: Target Manager

- Singleton service, i.e., one TargetManager per domain
- Retrieval of available or total resource capacities
- Allocation & release of resources (during application deployment)
- No “live” monitoring of resources implied (optional)
  - Assumption: all resources are properly allocated & released through this interface
- Allows “off-line” scenarios where the possibility & the effect of deploying applications is analyzed
  - e.g., “Given this configuration, is it possible to run this set of application components simultaneously? How?”
Deployment Infrastructure: Execution Manager

- Singleton service, i.e., one *ExecutionManager* per domain
- A “daemon-like” process always running in each domain
- User-visible front-end for executing a global (domain-level) deployment plan
  - Deployment plan results from planning for the deployment of an application, based on a specific set of nodes & resources
- Has information on all *NodeManagers* in the domain
- Instructs *NodeManagers* to execute respective per-node pieces of an application
Deployment Infrastructure: Node Manager

- Mirrors the *ExecutionManager*, but is limited to one node only
- A “daemon-like” process that is always running on each individual node
- Responsible for deploying local (node-level) deployment plan
Execution/Node Managers Interaction

- `ExecutionManager` computes per-node Deployment Plan
  - “Virtual” assemblies of components on the same node
  - Described using the same data structure
- All parts are sent to their respective `NodeManager`
  - Processing can be concurrent
- `ExecutionManager` then sends “provided” references to their users
- Transparent to “Executor” user
Launch Application: Domain vs. Node

- **Domain** provides functionality at the domain level
- **Node** provides similar functionality, but restricted to a Node
- **ApplicationManager**
  - `startLaunch()` & `destroyApplication()` operations
- **Application**
  - `finishLaunch()` & `start()` operations
Deployment Actors

Actors — usually, humans aided by software tools
Deployment Actors: Repository Administrator

- Receives component package from software vendor
- Installs package into repository, using Repository Manager
  - Assigns “installation name”
  - Optionally applies custom configuration properties
    - i.e., sets default values for an application’s external attributes (can be overridden during deployment)
  - Optionally sets “selection requirements”
    - Will be matched against implementation capabilities (during planning)
- Maintains repository contents
  - Browsing repository, updating packages, deleting packages …
Deployment Actors: Planner

- Accesses application metadata from Repository Manager
  - Resolving referenced packages
- Accesses resource metadata from Domain through Target Manager
  - Live “on-line” data or simulated “off-line” data
- Matches requirements against resources
- Makes planning decisions
  - Selects appropriate component implementations
  - Places monolithic component instances onto nodes, assembly connections onto interconnects & bridges
- Produces Deployment Plan
  - “Off-line” plans can be stored for later reuse
Deployment Actors: Executor

• Passes Deployment Plan to Execution Manager
• Separate “Preparation” & “Launch” phases
  – Preparation readies software for execution
    • Usually involves loading implementation artifacts to nodes via Node Manager
    • May (implementation-specific) also involve pre-loading artifacts into memory, e.g., for faster launch
  – Launch starts application
    • Instantiating & configuring components
    • Interconnecting components
    • Starting components
The **Display** component is an assembly component

When we deploy it, only the "monolithic" components will be actually deployed

"Deployer actor" can specify which "monolithic" component(s) maps to which nodes, as specified by the `ComponentDeploymentPlan (.cdp)` descriptor

We deploy three components to two nodes
Deployment Example: Prepare Plan

- Before calling `preparePlan()`, `ExecutionManager` should be running & two `NodeManagers` should be running on Alice & Bob nodes
- Retrieve Component Packages from the `Component Repository`
- `RepositoryManager` parses XML metadata into an in-memory representation
- `RepositoryManager` creates global deployment plan & passes it to `ExecutionManager` to `preparePlan()`, which delegates to `DomainApplicationManager`
- `DomainApplicationManager` splits it into multiple local plans
- Contacts the two `NodeManagers` residing in Alice & Bob nodes to create appropriate `NodeApplicationManagers` & dispatch individual local plans
Deployment Example: Start Launch

- **Executor** initiates launching of the application
- **DomainApplicationManager** creates a **DomainApplication** object
  - Facilitates application launch by contacting individual **NodeApplicationManagers**
- **NodeApplicationManagers** residing in Alice & Bob nodes will create a **NodeApplication** individually
• Executor notifies \textit{DomainApplication} of completion of application launch
• \textit{DomainApplication} notifies \textit{NodeApplications} running on Alice & Bob nodes to complete application launch
• Connections between components are made at this stage
• Optional “\textit{start}” parameter could be given to indicate whether actually “start” the application (i.e., \texttt{SetSessionContext()}, etc)
Deployment Example: Application Teardown

- **Executor** initiates tear-down by first terminating running applications under its control
  - **DomainApplicationManager** ensures tear down of **NodeApplications** running on both Alice & Bob nodes
- It then tears down both managers in Alice & Bob nodes
Wrapping Up
Tutorial Summary

• **CCM spec**
  - Extends the CORBA object model to support application development via composition
  - CORBA Implementation Framework (CIF) defines ways to automate the implementation of many component features
  - Defines standard run-time environment with Containers & Component Servers
  - Specifies deployment & configuration framework

• **Deployment & Configuration** specification separates key configuration concerns
  - Server configuration
  - Object/service configuration
  - Application configuration
  - Object/service deployment
Additional Information on CORBA & CCM

OMG specifications pertaining to CCM

- CORBA Component Model (CCM)
  - ptc/02-08-03
- Lightweight CCM
  - ptc/04-02-03
- QoS for CCM RFP
  - mars/03-06-12
- Streams for CCM RFP
  - mars/03-06-11
- UML Profile for CCM
  - mars/03-05-09
- Deployment & Configuration (D&C)
  - ptc/05-01-07

Books pertaining to CCM

- *CORBA 3 Fundamentals & Programming*, Dr. John Siegel, published at John Wiley & Sons

Web resources pertaining to CCM

- “The CCM Page” by Diego Sevilla Ruiz
  - www.ditec.um.es/~dsevilla/ccm/
- OMG CCM specification
  - www.omg.org/technology/documents/formal/components.htm
- CUJ columns by Schmidt & Vinoski

Complete Lightweight CCM tutorial: www.cs.wustl.edu/~schmidt/OMG-CCM-Tutorial.ppt
Overview of CIAO & Future R&D Directions
Overview of CIAO

- **Component Integrated ACE ORB**
  - Lightweight CCM implementation atop TAO
  - Supports component-oriented paradigm for DRE applications
    - Provides Real-time CORBA policies & mechanisms required for DRE applications
    - Key DRE aspects are supported as first-class metadata
- First official release (CIAO 0.4) was at end of December 2003
- Latest release is downloadable from [deuce.doc.wustl.edu/Download.html](http://deuce.doc.wustl.edu/Download.html)
CIAO Status

- Components can be built as shared libs or static libs
- Component server supported
- MDD tools to install, host, load, & manage component implementations are available
- The CIAO Deployment and Configuration Engine (DAnCE) provides support for component assemblies in compliance with ptc/02-08-03
- CIAO also supports Real-time CCM extensions
  - www.cs.wustl.edu/~schmidt/CIAO.html

- Support for IDL 3 (component, home & related keywords) & most CIDL features have been added
- Support for all types of ports: facets (provides), receptacles (uses, uses multiple), event sources (emits, publishes) & event sinks (consumes)
- Support for the Session container via CIDL compiler
CIAO Next Steps

• Deployment & Configuration (Leads: Gan Deng & Will Otte)
  – Implementing the new deployment & configuration specification, ptc/03-07-02, necessary for DARPA ARMS program
  – Changes to the deployment & assembly toolset to support lightweight components, as prescribed by ptc/04-02-03

• Core CCM Infrastructure (Leads: Johnny Willemsen & Nanbor Wang)
  – Additional support for Real-time CORBA Policies at the ORB level & object level
    • i.e., at the object reference level of a component receptacle
  – Integration of different event propagation mechanisms (such as Event & Notification Services) within the container
  – Compliant with Lightweight CCM specification

• Modeling tool support for CIAO (Leads: Kitty Balasubramanian & Jeff Parsons)
  – See www.dre.vanderbilt.edu/cosmic for details
How to Learn about CCM & CIAO Programming

- Examples available with the distribution
  - CIAO/docs/tutorial/Hello, a simple example that illustrates the use of some basic CCM concepts
  - CIAO/examples/OEP/BasicSP
    - A simple example that shows the interaction between 4 components
  - CIAO/examples/OEP/Display
    - Similar to the BasicSP, but has an additional feature showing integration with Qt toolkit
- Step-by-step to create & deploy components based on CIAO available at
  - CIAO/examples/Hello
- “Quick CORBA 3”, Jon Siegel, John Wiley & Sons provides a quick start
- C/C++ User Journal articles with Steve Vinoski