Tutorial on CORBA Component Model (CCM)

Nanbor Wang                        Craig Rodrigues
Washington University          BBN Technologies
St. Louis

July 6, 2003
Overview

• The purpose of this tutorial is to
  – present the motivation of CCM
  – introduce features most relevant to distributed, real-time, embedded applications
  – present common patterns for implementing important operations for CCM components

• but not to
  – enumerate through all the mapping rules
  – provide detailed references of all interfaces
  – make you capable of implementing a CCM framework
Motivation for and Overview of CORBA Component Model
Where We Started From: Object-Oriented Programming

- Object-Oriented programming simplified software development through higher level abstractions (i.e. associating related data and operations)
- Applying OO to network programming (Distributed Object Computing, CORBA, RMI, etc.) simplified distributed systems development
- We now have more robust software and more powerful distributed systems
Motivations for Applying OO to Network Programming

• Abstract away lower-level OS and protocol specific details for network programming
• Create distributed systems which are easier to model and build
• Result: robust distributed systems built with OO middleware: CORBA, RMI, etc.
Overview of CORBA

- CORBA shields applications from heterogeneous platform dependencies
  - e.g., languages, operating systems, networking protocols, hardware

- It simplifies development of distributed applications by automating/encapsulating
  - Object location
  - Connection & memory mgmt.
  - Parameter (de)marshaling
  - Event & request demultiplexing
  - Error handling & fault tolerance
  - Object/server activation
  - Concurrency
  - Security

- CORBA defines interfaces, not implementations
Example: Applying OO to Network Programming

- CORBA IDL specifies *interfaces* with operations
  - interfaces map to objects in programming languages (C++, Java)

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

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

- Operations in interfaces can be on local or remote objects
Shortcomings of Traditional OO-based CORBA Middleware?

- CORBA does not specify how “assembly” and “deployment” of object implementations should be done to create larger applications
  - Proprietary infrastructure and scripts are usually written to facilitate this
- CORBA IDL does not provide a way to logically group together related interfaces to offer a specific service
  - CORBA IDL does not offer such a feature, so such “bundling” must be done by the developer
Caveat: Limitations of CORBA 2.x Specification

- Requirements of non-trivial applications:
  - Collaboration of multiple objects and services
  - Wide-spread deployment on diverse platforms
- Limitations – Lack of standards
  - Server configuration
  - Object/service configuration
  - Application configuration
  - Object/service deployment
- Consequences – tight couplings at various layers
  - Brittle, non-scalable implementation
  - Hard to adapt and maintain
  - Increase time-to-market
The Emergence of Component Middleware

- Components give standard mechanisms for “assembly” and “deployment” of applications
- Components aggregate together related interfaces into logical units which are reusable
- Containers provide execution environment for components
- Containers communicate via a middleware bus
The CORBA Component Model (CCM)

• Supporting mechanisms
  – **Component Server**: a generic server process for hosting containers and components/homes
  – **Component Implementation Framework**: automate the implementation of many component features
  – **Packaging and Assembling tools**: for collecting implementations and configurations information into deployable assemblies
  – **Deployment mechanism**: automate the deployment of component assemblies to component servers

• Goals: Separating configuration concerns into aspects:
  – Server configuration
  – Object/service configuration
  – Application configuration
  – Object/service deployment
Component Features
Interface and Component Designing Stage

**Interface Design**
- Interface IDL Definitions

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

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

**Component Packaging**
- Component DLLs
  - Packaging Tools
  - XML Component Descriptors (.ccd)
  - XML Softpkg Descriptors (.csd)
  - XML Component & Home Properties
  - Component Packages (Zipped archives *.car)

**System Deployment**
- Running Applications
  - Deployment Tools
    - Component Configurations
    - Target Platform Properties
  - Assembly Packages (Zipped archives *.aar)
    - XML Assembly Descriptors (.cad)
  - Assembling Tools
    - XML Component & Home Properties
    - Application Assembly

**Application Assembly**
- BBN TECHNOLOGIES
  - A Verizon Company
Application Development via Composition

- **Rate Generator**
  - Sends periodic “Pulse” events to subscribers

- **Positioning Sensor**
  - Refreshes cached coordinates available thru **MyLocation** facet
  - Notifies subscribers via “Ready” events

- **Displaying Device**
  - Reads current coordinates via its **GPSLocation** receptacle
  - Updates display

A typical use case for industrial/automotive/avionics control
A CORBA Component

interface rate_control
{
    void start ();
    void stop ();
};

component RateGen
    supports rate_control
{
};

interface RateGen :
    Components::CCMObject,
    rate_control
{
};

• Context: To support development via composition
• Limitations of CORBA objects
  – Merely identify interfaces
  – No direct relation with reusable/deployable implementations
• Goals
  – Define a unit of reuse and implementation
  – Encapsulate an interaction and configuration model
• A component is a new CORBA meta-type
  – Extension of Object
  – Has an interface, and an object reference
• Could inherit from a single component type
• Could supports multiple interfaces
Managing Component Lifecycle

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

• Problems:
  – No standard way to manage component’s lifecycle
  – Need standard mechanisms to strategize lifecycle management

• CCM Solution:
  – Integrating Lifecycle service into component definitions
  – Using different component home’s to provide different lifecycle managing strategies
A CORBA Component Home

- "home" is a new CORBA meta-type
  - Has an interface, thus is identified by object reference
- Manages a unique component type
  - More than one home type can manage the same component type
  - A component instance is managed by one home instance
- Standard factory & finder operations
- Can have arbitrary user-defined operations
A Quick Example
Component and Home for HelloWorld

interface Hello
{
    void sayHello
    (in string username);
};

component HelloWorld supports Hello
{
};

home HelloHome manages HelloWorld
{
};
int main (int argc, char *argv[])
{
    CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
    CORBA::Object_var obj = orb->resolve_initial_references("NameService");
    CosNaming::NamingContextExt_var nc = CosNaming::NamingContextExt::_narrow (obj);
    obj = nc->resolve_str("HelloHome");
    HelloHome_var hh = HelloHome::_narrow (obj);
    HelloWorld_var hw = hh->create();
    hw->sayHello("Simon");
    hw->remove();
    return 0;
}

$>./hello-client
Hello World! -- from Simon.
More on Component Features
Components Can Have Different Views

• Context:
  – Components need to collaborate with several different kinds of components/systems
  – These collaborating components/systems may understand different interface types

• Problems:
  – Difficult to extend an interface
  – No standard way to acquire new interfaces

• CCM Solution:
  – Define facets, aka., provided interfaces
Component Facets

- Component facets:
  - Facets give offered operation interfaces
  - Specified with “provides” keyword

```cpp
interface position
{
  long get_pos ();
};

component GPS
{
  provides position
    MyLocation;
  ...
};

interface GPS : Components::CCMObject
{
  position
    provide_MyLocation ();
  ...
};
```
Using Other Components

• Context:
  – Components need to collaborate with several different kinds of components/systems
  – These collaborating components/systems may provide different types of interface

• Problems:
  – No standard way to specify capability to handle, or dependency to use other interfaces
  – No standard way connect an interface to a component

• CCM Solution:
  – Define receptacles
Component Receptacles

- Specifies a way to connect an interface to this component
- Specified with “uses” keyword

```cpp
component NavDisplay
{
  ...
  uses position GPSLocation;
  ...
};

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

• Context:
  – Components may also communicate using anonymous publishing/subscribing message passing syntax

• Problems:
  – Non-trivial to extend existing interface to support event passing
  – Standard CORBA Event Service is non-typed \(\rightarrow\) no type-checking connecting publishers-consumers
  – No standard way to specify component’s capability to generate and process events

• CCM Solution:
  – Standard eventtype/eventtype consumer interface
  – Event sources
  – Event sinks
Component Events

```
eventtype tick
{
    public rateHz rate;
};

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

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

- Events are IDL valuetypes
- Defined with the new `eventtype` keyword
Component Event Sources

component RateGen
{
    
    publishes tick Pulse;
    emits tick trigger;
    ...
}

interface RateGen : Components::CCMObject
{
    Components::Cookie subscribe_Pulse
        (in tickConsumer c);
    tickConsumer unsubscribe_Pulse
        (in Components::Cookie ck);
    ...
}

- Event sources:
  - Named connection points for event production
  - Two kinds: Publisher & Emitter
    - publishes = multiple consumers
    - emits = only one consumer

- Event delivery
  - Consumer subscribes/connects directly
  - Container mediates access to CosNotification channels or other event delivery mechanism
Component Event Sinks

- **Event sinks**
  - Named interface specifies which events may be pushed
  - Event sink can subscribe to multiple event sources
  - No distinction between emitter and publisher

```cpp
class NavDisplay
{
    ...;
    consumes tick Refresh;
};

interface NavDisplay : Components::CCMObject
{
    ...;
    tickConsumer get_consumer_Refresh ();
    ...;
}
```
The Need to Configure Components

• Context:
  – To make component implementations more adaptable, components should be reconfigurable

• Problems:
  – Should not commit to a configuration too early
  – No standard way to specify component’s configurable knobs
  – Need standard mechanisms to configure components

• CCM Solution:
  – Use component attributes for component configurations
  – Configuration mechanisms
Component Attributes

- Named configurable properties
  - Intended for component configuration
    - e.g., optional behaviors, modality, resource hints, etc.
  - Could raise exceptions
  - Exposed through accessors and mutators

```cpp
typedef unsigned long rateHz;

component RateGen
    supports rate_control
    {
        attribute rateHz Rate;
    };
```

```cpp
interface RateGen : Components::CCMObject, rate_control
    {
        attribute rateHz Rate;
    };
```
Recap – Component Features

- IDL3 definition of a component from a “client-view”
  - What the component life cycle operations are (i.e., `home`)
  - What a component offers to other components
  - What a component requires from other components
  - What collaboration modes are used between components
    - Synchronous via operation invocation
    - Asynchronous via event notification
  - Which component properties are configurable
- Maps to “Equivalent IDL2 Interfaces”
Configuring and Connecting Components

• Context:
  – Components need to be configured and connected together to form application

• Problems:
  – Components have different ports of different types and names
  – Non-scalable to generate code to connect a specific set of components

• CCM Solution:
  – Provide introspection interface to discover component capability
  – Provide generic port operations to compose/configure components
### 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 ports operations for provides, uses, subscribes, emits, and consumes.
  - Apply the “Extension Interface Pattern”
  - Used by deployment tools
  - Light Weight CCM spec won’t include equivalent IDL2 operations
Example of Connecting Components

- Interface → Receptacle
  
  ```
  objref = GPS->provide ("MyLocation");
  ```

- Event Source → Event Sink
  
  ```
  consumer = NavDisplay->get_consumer ("Refresh");
  ```

  ```
  GPS->subscribe ("Ready", consumer);
  ```

  ```
  NavDisplay->connect ("GPSLocation", objref);
  ```
Component Runtime Environment
CCM Server-side Features

- CCM is all about component – server - application configuration
- CORBA 2.x specifications lack higher level abstractions of servant usage models
- Require programmatic configuration (more often with boiler plate-like code)
- Apply meta-programming techniques
  - Reusable run-time environment
  - Drop in and run
  - Transparent to clients
The Container Model

- A framework in component servers

- Built on the Portable Object Adaptor
  - Automatic activation / deactivation
  - Resource usage optimization

- Provides simplified interfaces for CORBA Services
  - Security, transactions, persistence, and events

- Uses callbacks for instance management
  - Session states, activation, deactivation, etc.
Container Managed CORBA Policies

- Goal: decouple runtime configuration from the component implementation & configuration
- Specified by the component implementors using XML-based metadata
- Implemented by the container, not the component
- CORBA Policy declarations defined for:
  - Servant Lifetime
  - Transaction
  - Security
  - Events
  - Persistence
Implementing CORBA Components
Requirements for Implementing Components

Component and home Definitions

- Component implementations need to support introspection, navigation and manage connections.

- Different implementation may assume different run-time requirements

- Different run-time requirements use different container interfaces

Component and home servants:
- Navigation interface operations
  - Receptacles interface operations
  - Events interface operations
- CCMObject interface operations
  - CCMHome interface operations
- Implied equivalent IDL2 port operations
- Application-related operations (in facets, supported interfaces, event consumers)
Difficulties with Implementing Components

Problem:
Generic lifecycle and initialization server code must be handwritten.

- *Ad hoc* design
- Code bloat
- No reuse
Component Implementation Framework (CIF)

- CIF defines rules and tools for developing component implementations
  - Local executors

- Extends CCM-related declarations in IDL files.

- Describes component implementations.

- Automate most component implementation
Facilitating Component Implementation

Solution: CIDL is part of CCM strategy for managing complex applications.

- Helps separation of concerns.
- Helps coordination of tools.
- Increases the ratio of generated to hand-written code.
- Server code is now generated, startup automated by other CCM tools.
Connecting Components and Containers with CIDL

- CIDL compiler generates infrastructure code which connects together component implementations and container which hosts them
- Infrastructure code in container intercepts invocations on executors for managing activation, security, transactions, persistency, and so on
- CIF defines “executor mappings”
Examples on Component Implementations
### Simple HelloWorld & HelloHome Executors

**Interface Hello**

```cpp
interface Hello
{
    void sayHello (in string username);
};
```

**Component HelloWorld**

```cpp
component HelloWorld
{...}
```

**Home HelloHome**

```cpp
home HelloHome manages HelloWorld
{...}
```

**Class HelloWorld_Impl**

```cpp
class HelloWorld_Impl
: public virtual CCM_HelloWorld,
 public virtual CORBA::LocalObject
{
    public:
    HelloWorld_Impl () {}
    ~HelloWorld_Impl () {}
    void sayHello (const char *username)
    {
        cout << "Hello World for "
        << username
        << endl;
    }
};
```

**Class HelloHome_Impl**

```cpp
class HelloHome_Impl
: public virtual CCM_HelloHome,
 public virtual CORBA::LocalObject
{
    public:
    HelloHome_Impl () {} // Implement lifecycle management strategy of HelloWorld component
    ~HelloHome_Impl () {} // Implement lifecycle management strategy of HelloWorld component
    Components::EnterpriseComponent_ptr create ()
    {
        return new HelloWorld_Impl ();
    }
};
```

- Implements behaviors of HelloWorld component
- Implement a lifecycle management strategy of HelloWorld component
Component Entry Point Example

```cpp
extern "C" {
Components::HomeExecutorBase_ptr create_HelloHome ()
{
    return new HelloHome_impl;
}
}
```

- Container calls this method to create a home executor
- `extern "C"` required to prevent C++ name mangling, so function name can be resolved in DLL
Implementing Ports Mechanisms

- Stuff that get invoked upon
  - Facets
  - Event sinks

- Stuff that the component invokes
  - Receptacles
  - Event sources
interface position
{
    long get_pos ();
};

cOMPOnent GPS
{
    provides position
        MyLocation;
    ...
};

interface GPS :
    Components::CCMObject
{
    position
        provide_MyLocation ();
    ...
};

local interface GPS_Executor:
    CCM_GPS,
    CCM_position,
    Components::SessionComponent
{
    // Monolithic Executor Mapping
}

class GPS_Executor_Impl :
    public virtual GPS_Executor,
    public virtual CORBA::LocalObject
{
    public:
        ...
    virtual CCM_position_ptr
        get_MyLocation ()
    { return this; }
    virtual CORBA::Long
        get_pos ()
    { return cached_current_location; }
    ...
};
**Component Event Sinks**

- **Event sinks**
  - Clients can acquire consumer interfaces, similar to facets
  - CIDL generates event consumer servants
  - Executor mapping defines push operations directly

```cpp
component NavDisplay
{

  consumes tick Refresh;
}

interface NavDisplay :
  Components::CCMObject
{

  tickConsumer
  get_consumer_Refresh();
}

class NavDisplay_Executor_Impl :
  public virtual CCM_NavDisplay,
  public virtual CORBA::LocalObject
{
  public:

  virtual void push_Refresh (tick *ev) {
    this->refresh_reading ();
  }

  ...
}
```
Initialize Component Specific Context

- Component-specific context manages connections and subscriptions
- Container passes component its context via either
  - `set_session_context`
  - `set_entity_context`

```cpp
class GPS_Executor_Impl :
    public virtual GPS_Executor,
    public virtual CORBA::LocalObject
{
    private:
        CCM_GPS_Context_var context_;
    public:
        ...
        void set_session_context
        (Components::SessionContext_ptr c)
        {
            this->context_ =
                CCM_GPS_Context::narrow (c);
        }
        ...
};
```
Using Receptacle Connections

- Component-specific context manages receptacle connections
- Executor acquires the connected reference from the context

```cpp
class NavDisplayExecutor_Impl : public virtual CCM_NavDisplay,
public virtual CORBA::LocalObject
{
public:
...
virtual void refresh_reading (void) {
position_var cur =
this->context_->
get_connection_GPSLocation ();
long coord = cur->get_pos ();
...
}
...}
```
Pushing Events from a Component

- Component-specific context manages consumer subscriptions (for publishers) and connections (for emitters)
- Component-specific context also provides the event pushing operations and relays events to consumers

```cpp
class RateGenExecutor_Impl : public virtual CCM_RateGen, public virtual CORBA::LocalObject
{
  Components::Cookie subscribe_Pulse
    (in tickConsumer c);
  tickConsumer
    unsubscribe_Pulse
    (in Components::Cookie ck);
  ...
};
```
Component Packaging, Assembly, and Deployment
Component Packaging Stage

- Packaging: bundling a component implementation with associate metadata

System Deployment

Component Configurations

Target Platform Properties

Application Assembly

XML Component & Home Properties

XML Component Descriptors (.ccd)

Component DLLs

Language Tools

Servants, Executors, Contexts

CIDL Compiler

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)

XML Component & Home Properties

Component IDL Definitions

Component IDL Definitions

IDL Compiler

Interface IDL Definitions

Interface Design

Component Design

Component Implementation

XML Component & Home Properties

XML Softpkg Descriptors (.csd)

XML Assembly Descriptors (.cad)

Component Packages (Zipped archives *.car)

Assembly Packages (Zipped archives *.aar)

Assembling Tools

Packaging Tools

XML Component & Home Properties

Component Packages (Zipped archives *.car)

Component Descriptors (.ccd)
Component Packages

- Goals
  - Configure components, containers, servers
  - Extract these aspects into metadata

- That’s a lot of stuff to be bundled together and moved around
- “Classic” CORBA: No standard means of
  - Configuration
  - Distribution
  - Deployment

- Packaging of components
  - Components are packaged into a self-descriptive package as a compressed archive

- XML descriptors provide metadata that describe
  - The content of a package
  - The capability of components
  - The dependencies to other software artifacts
    - Other components
    - 3rd party DLLs
    - Valuefactories
Application Assembling Stage

- Assembly: A collection of components packages and metadata that specify the composition of an application
Component Assembling

- **Goals**
  - Configure components, containers, servers, and applications
  - Extract these aspects into metadata
  - Provide higher level of modeling

- “Classic” CORBA: No standard means of
  - Configuration
  - Distribution
  - Deployment

- An assembly descriptor specifies:
  - Component implementations
  - Component/home instantiations
  - Interconnections
<componentfiles>
  <componentfile id="com-RateGen">
    <fileinarchive name="RateGen.csd"/>
  </componentfile>

  <componentfile id="com-GPS">
    <fileinarchive name="GPS.csd"/>
  </componentfile>

  <componentfile id="com-Display">
    <fileinarchive name="NavDisplay.csd"/>
  </componentfile>
</componentfiles>
Component Home/Instances Installation Specifications

An assembly descriptor specifies how & where homes & components should be instantiated

A component property file (.cpf) can be associated with a home or a component instantiation to override default component properties
Assembly descriptors also specify how component instances are connected together.
Two Deployment Examples

- Making configuring, assembling, & deploying of applications easy
  - Component configurations
  - Component implementations
  - Inter-connections
  - Logical location constraints

RemoteDisplayGUI.cad

DuelDisplay.cad
Deployment Stage

- **Deploy:** Realization of a single component or an assembly specification
Application Deployment

• Deployment tools
  – Have knowledge of target platforms
  – Map locations in assembly to physical nodes
  – Manage available resources for applications
  – Use standard CCM interfaces defined in module `Components::Deployment` to realize an assembly
Where Do We Go from Here?
Summary

• The CORBA Component Model
  – Extend CORBA object model to support application development via composition
  – CIF defines ways to automate the implementation of many component features
  – Defines standard runtime environment with Containers and Component Servers
  – Specifies packaging and deployment framework

• Separating configuration concerns
  – Server configuration
  – Object/service configuration
  – Application configuration
  – Object/service deployment
CCM Status

• Available CCM Implementations
  – Component Integrated ACE ORB (CIAO) by Washington University and Vanderbilt University
  – Enterprise Java CORBA Component Model (EJCCM) by Computational Physics, Inc.
  – K2 CCM by iCMG (commercial product)
  – MICO CCM by Frank Pilhofer
  – QoS Enabled Distributed Object (Qedo) by FOKUS
  – OpenCCM by ObjectWeb (Java based)

• Modeling and Assembling tools
  – Cadena from Kansas State University
  – GME from ISIS, Vanderbilt University
CCM Related Specifications

- Light Weight CCM hosted by RTESS
  - realtime/03-05-05
- QoS for CCM RFP hosted by MARS
  - mars/03-06-12
- Stream for CCM RFP hosted by MARS
  - mars/03-06-11
- UML Profile for CCM hosted by MARS
  - mars/03-05-09
- Deployment and Configuration hosted by MARS
  - ptc/03-06-03

- CCM for distributed, real-time and embedded applications
  - CIAO and Qedo
  - Light weight CCM