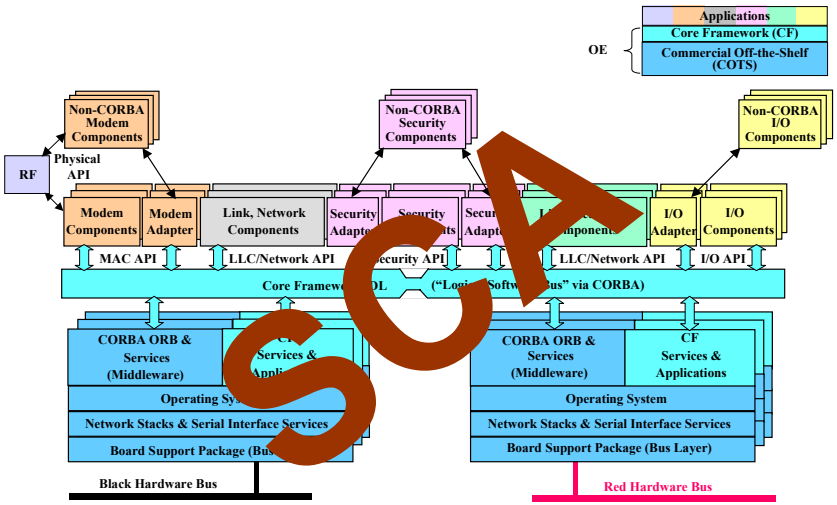
Indepth Coverage of the SCA Naming Service, Event Service, and Component Connections



Neli Hayes
Associate Technical Fellow
Principal Software Architect
Network Communications Systems
The Boeing Company, Anaheim, CA
persnaz.n.hayes@boeing.com
(714) 762-8768

BOEING®

Objectives

- Provide a brief overview of the OMG Naming and Event Services, along with an indepth look into the SCA-mandated subsets, related CF and component developer responsibilities, pertinent Domain Profile XML elements, Security Supplement restrictions and conflicts with the main SCA specification, and suggested approaches for resolving such conflicts for specific implementation architectures.
- Provide an overview of the Domain Profile XML elements for expressing connections between components, responsibilities of the CF providers with respect to establishing these connections, and responsibilities of component implementations that play the user or provider role in a connection.

Outline

Outline

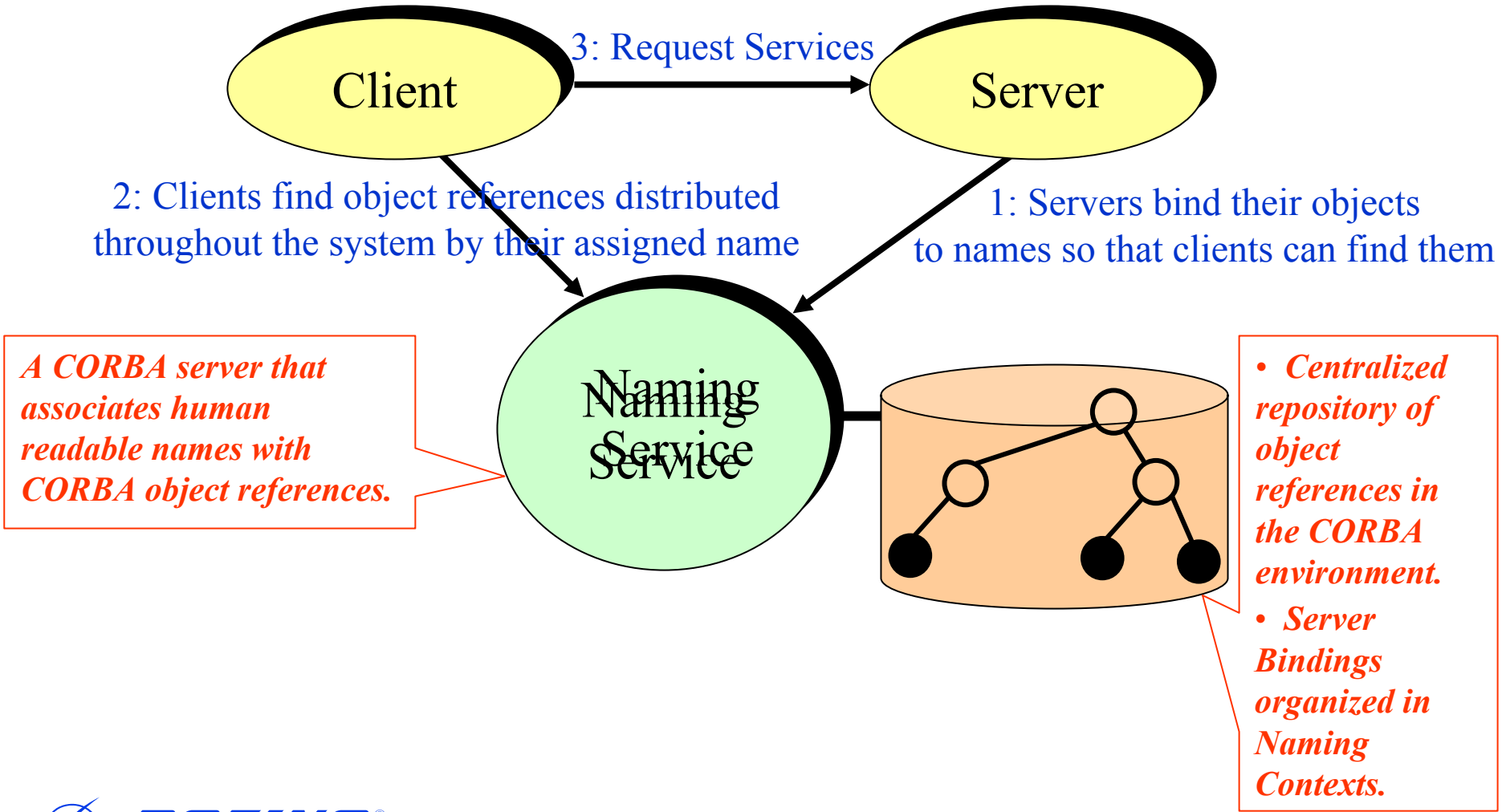
Outline

Outline

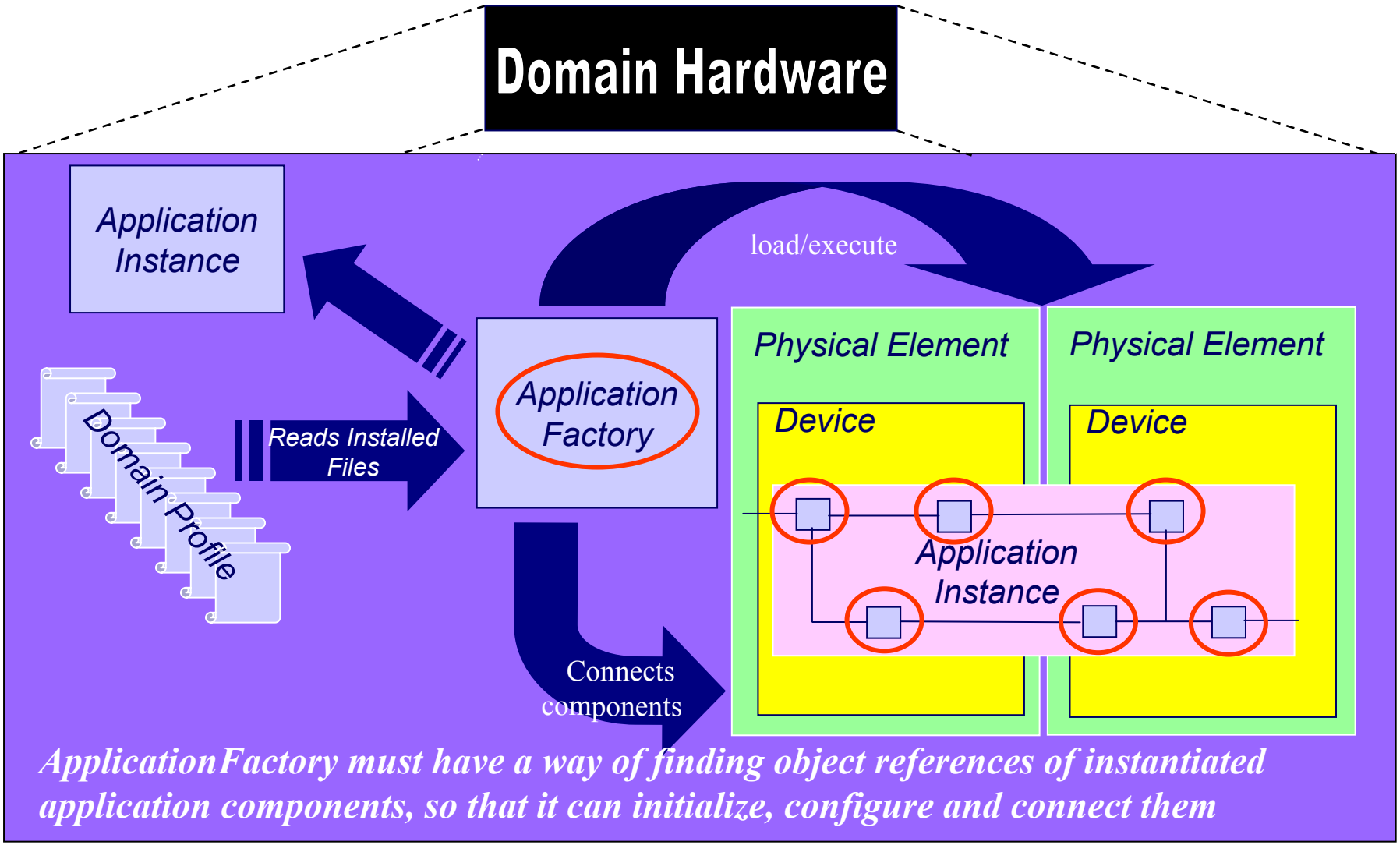
SCA Naming Service Usage for CF Providers and Component Developers

BOEING®

OMG Naming Service Architecture



Prominent SCA Naming Service Usage - During Application Instantiation



ApplicationFactory must have a way of finding object references of instantiated application components, so that it can initialize, configure and connect them

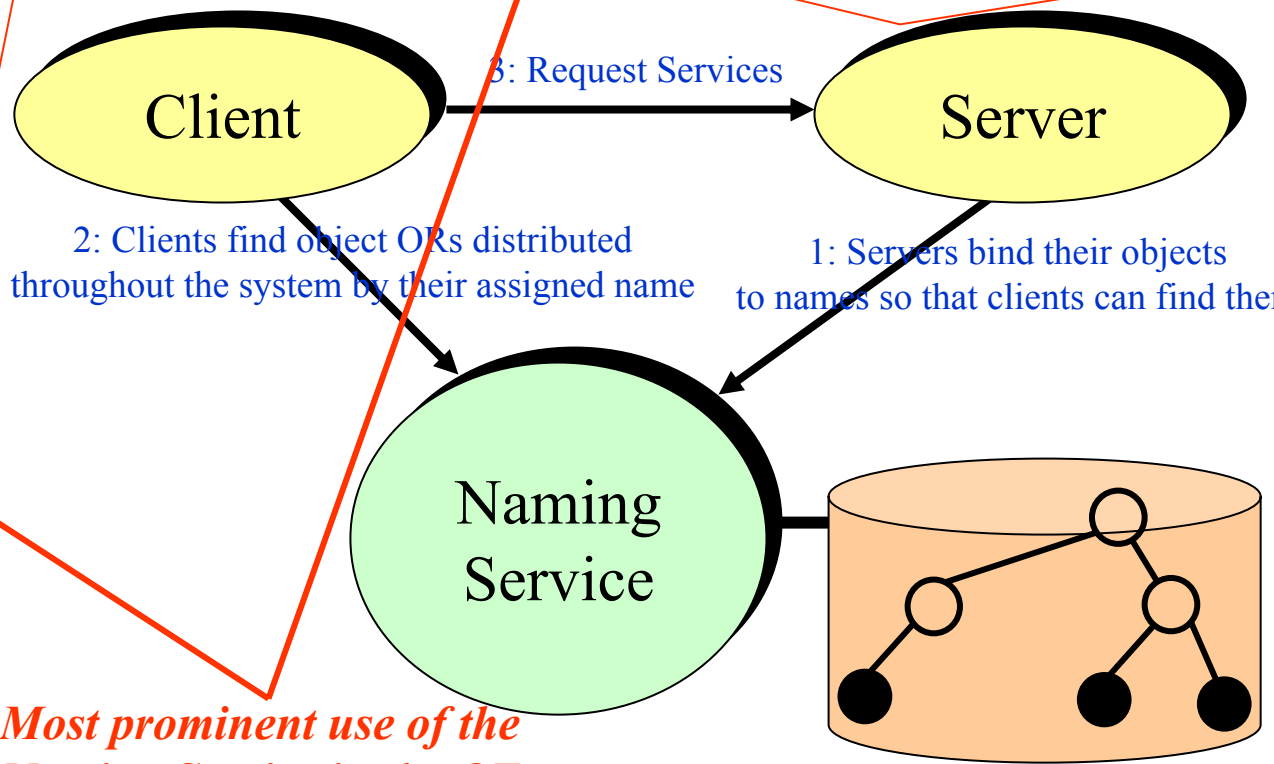
Naming Service Usage in the OE

Typical OE Naming Service Clients

- ***Application Installers (DomainManager client)***
- ***DeviceManagers (DomainManager client)***
- ***DomainManager: connect DeviceManager deployed components to services in CF domain***
- ***ApplicationFactory & Application: create, setup, connect, and tear down instantiated application components.***

Typical OE Naming Service Servers

- ***DomainManager***
- ***Deployed DeviceManager components***
- ***Instantiated application components***



Most prominent use of the Naming Service in the OE

BOEING®

CF's Usage of the Naming Service

- DeviceManager
  - Find the DomainManager object reference so that it can register itself and its deployed devices and services with the domain.
- DomainManager
  - Bind itself to the Naming Service so that others can find it (e.g. DeviceManagers, entities that need to install and instantiate applications in the domain).
  - Find object references of DCD components for processing DeviceManager connections.
- ApplicationFactory
  - Find object references of instantiated application components in order to initialize, configure and connect them.
- Application
  - Unbind object reference of application components from the NS.
  - Destroy ApplicationFactory-created naming contexts.

Component Developer's Usage of the Naming Service

- Bind the component's object reference to the Naming Service so that it can be found by the ApplicationFactory.
- The naming context to bind to is passed as an execparam to the component's entry point task, with an ID of *NAMING_CONTEXT_IOR*.
- The name binding to use for binding the component's object to is passed as an execparam to the component's entry point task, with an ID of *NAME_BINDING*.

Security Supplement and SCA Naming Service Restrictions and Conflicts

- Security Supplement Mandates
  - Only entities in the DomainManager process space are supposed to use the Naming Service *resolve* operation.
- Main SCA Specification Requirements
  - The Domain Profile provides the means for a DeviceManager to obtain the DomainManager object reference via the Naming Service.
  - DomainManager is to publish its object reference with the Naming Service so that the DeviceManagers and other entities in the system (e.g. application installers) can find it.
  - Application components must have access to the Naming Service in order to bind their object references to the Naming Service so that the ApplicationFactory can find them.

Security Supplement and SCA Naming Service Conflict Resolutions

- For systems that have to adhere to the SCA Security Supplement rules, the object reference of the DomainManager can be provided to DeviceManagers via other means, such as a constructor object reference parameter.
- In case of other domain entities that need access to the DomainManager (e.g. for application installations and instantiations), one possible platform implementation is to make the DomainManager object reference available through the DomainFinder and express connections to the DomainManager for persistent components via the connections element of the desired DeviceManager DCDs.
- Application components' access to the Naming Service can be restricted via providing a wrapper object that "looks" like the *CosNaming::NamingContext* interface, but, only provides what the application components need—the *bind ()* operation.

Required Naming Service Subset for SCA-Compliant OEs

- Based on OMG Naming Service Specification OMG Document formal/00-11/01.
- Must support *CosNaming bind(), bind_new_context(), unbind(), destroy(),* and *resolve().*

Naming Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
module CosNaming {

  typedef string Istring;

  struct NameComponent {
      Istring id;
      Istring kind;
  };

  typedef sequence<NameComponent> Name;

  enum BindingType { nobject, ncontext };

   struct Binding {
      Name binding_name;
      BindingType binding_type;
   };

 interface BindingIterator;
```

Always a Null "" string for SCA compliant components, such as DomainManager and instantiated application components naming contexts and name bindings

Mandatory Naming Service Subset to be supported by an SCA-Compliant OE

Naming Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
interface NamingContext {

enum NotFoundReason { missing_node, not_context, not_object };

exception NotFound { NotFoundReason why; Name rest_of_name; };
exception CannotProceed { NamingContext cxt; Name rest_of_name; };
exception InvalidName{};
exception AlreadyBound {};
exception NotEmpty{};

 void bind(in Name n, in Object obj)
   raises( NotFound, CannotProceed, InvalidName, AlreadyBound );
```

Mandatory Naming Service Subset to be supported by an SCA-Compliant OE

Components
DomainManager

```
void rebind(in Name n, in Object obj)
  raises(NotFound, CannotProceed, InvalidName);

void bind_context(in Name n, in NamingContext nc)
  raises(NotFound, CannotProceed, InvalidName, AlreadyBound);

void rebind_context(in Name n, in NamingContext nc)
  raises(NotFound, CannotProceed, InvalidName);
```

Naming Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
CF Object resolve (in Name n)
       raises(NotFound, CannotProceed, InvalidName);

CF void unbind(in Name n)
       raises(NotFound, CannotProceed, InvalidName);

    NamingContext new_context();

CF NamingContext bind_new_context(in Name n)
       raises( NotFound, AlreadyBound, CannotProceed, InvalidName );

CF void destroy() raises(NotEmpty);
    void list( in unsigned long how_many, out BindingList bl, out BindingIterator bi );
  };

  interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many, out BindingList bl);
    void destroy();
  };
```

Mandatory Naming Service Subset to be supported by an SCA-Compliant OE

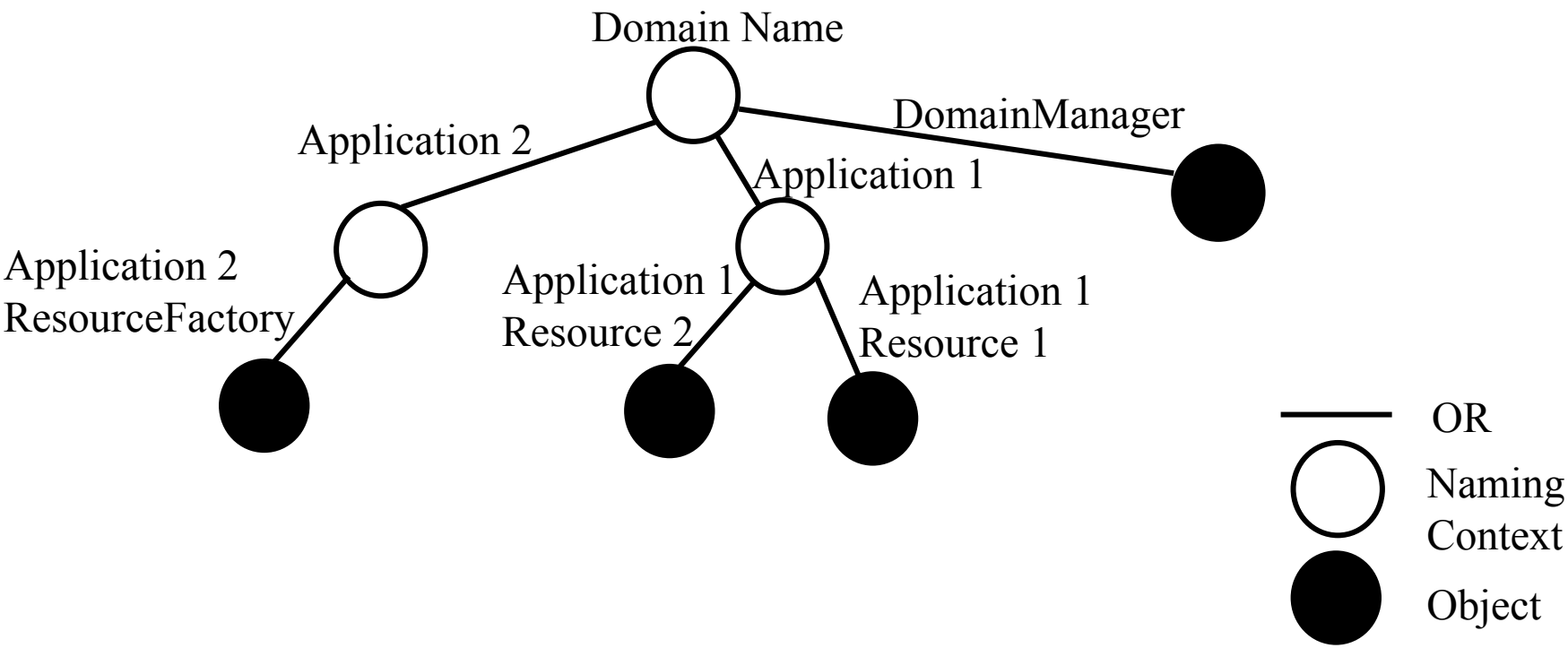
Naming Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
interface NamingContextExt: NamingContext {
  typedef string StringName;
  typedef string Address;
  typedef string URLString;

  StringName to_string(in Name n) raises(InvalidName);

  Name to_name(in StringName sn)
    raises(InvalidName);

  exception InvalidAddress {};

  URLString to_url(in Address addr, in StringName sn)
    raises(InvalidAddress, InvalidName);

  Object resolve_str(in StringName n)
    raises( NotFound, CannotProceed, InvalidName, AlreadyBound );
 };
};
```

SCA test tools should not rely on the existence of this interface in an SCA-compliant OE, as it is not required by the SCA.

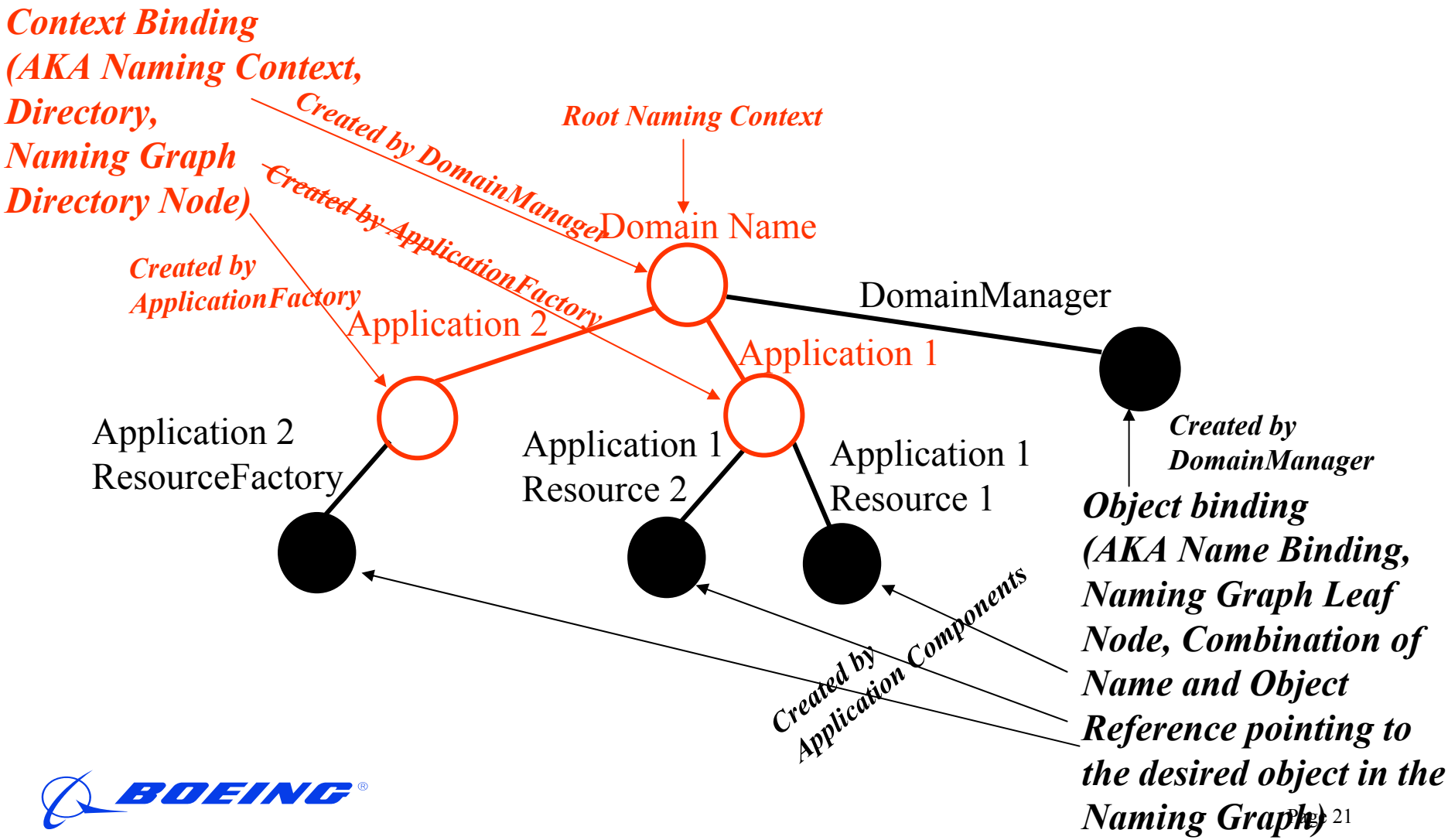
OE Naming Graph

(Major Context & Object Bindings)



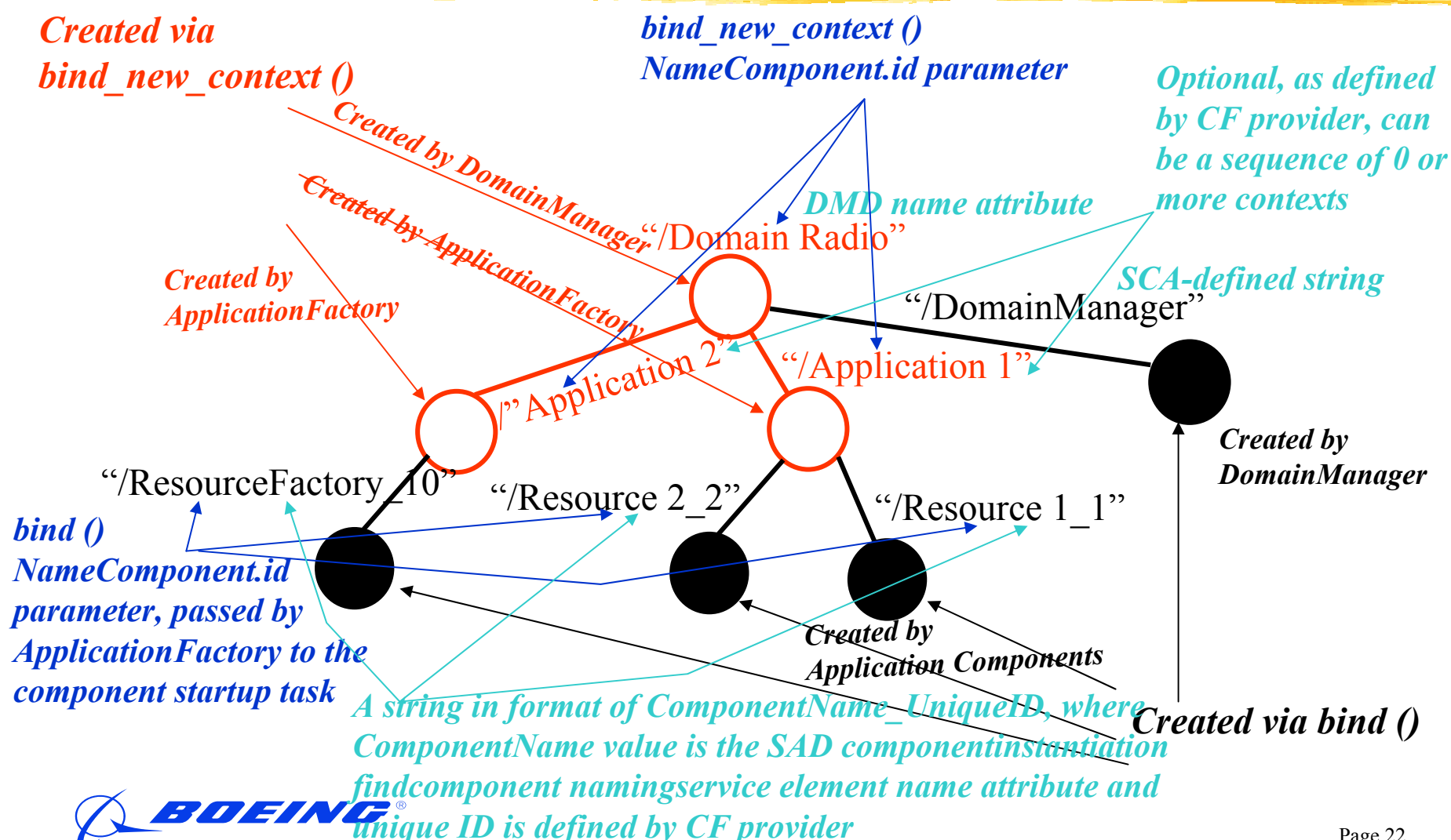
OE Naming Graph

(Major Context & Object Bindings)



OE Naming Graph

(Major Context & Object Bindings)



Domain Profile Naming Service Expression (DeviceManager's Device Configuration Descriptor)

```
<!ELEMENT deviceconfiguration
     ( description?
     , devicemanagersoftpkg
     , componentfiles?
     , partitioning?
     , connections?
     , domainmanager
     , filesystemnames?
     )>
<!ATTLIST deviceconfiguration
     id          ID          #REQUIRED
     name        CDATA       #IMPLIED>

<!ELEMENT domainmanager
( namingservice)>

<!ELEMENT namingservice EMPTY>
<!ATTLIST namingservice
name CDATA    #REQUIRED>
```

Covered in the Connections section of the presentation

How the DeviceManager finds the DomainManager object reference

Domain Profile Naming Service Expression (ApplicationFactory Software Assembly Descriptor)

```
<!ELEMENT softwareassembly
          ( description?
          , componentfiles
          , partitioning
          , assemblycontroller
          , connections?
          , externalports?
          )>
<!ATTLIST softwareassembly
          id          ID          #REQUIRED
          name        CDATA       #IMPLIED>

<!ELEMENT partitioning
     ( componentplacement
     |  hostcollocation
     )*>

<!ELEMENT componentplacement
     ( componentfileref
     , componentinstantiation+
     )>
```

Covered in the Connections section of the presentation

Deployment pattern of components and their component to host relationships for this application (e.g. how many times and in how many ways are various components of this application instantiated, subsets of which may be collocated on the same device or same address space)

How many times and in how many ways (what variances) is a component for this application instantiated

A particular instantiation of a component relative to a componentplacement

Domain Profile Naming Service Expression (ApplicationFactory Software Assembly Descriptor)

```
<!ELEMENT componentinstantiation
     ( usagename?
     , componentproperties?
     , findcomponent?
     )>
<!ATTLIST componentinstantiation
     id        ID        #REQUIRED>

<!ELEMENT findcomponent
     ( componentresourcefactoryref
             | namingservice
     )>
```

How the ApplicationFactory can find the object reference of a created component.

Via the name binding specified in the namingservice element

Domain Profile Naming Service Expression (DomainManager's DomainManager Configuration Descriptor)

```
<!ELEMENT domainmanagerconfiguration
     ( description?
     , domainmanagersoftpkg
     , services
     )>
<!ATTLIST domainmanagerconfiguration
     id       ID       #required
     name #CDATA       #required>

<!ELEMENT services
     ( service+
     ) >

<!ELEMENT service
     ( usesidentifier
     , findby
     )>

<!ELEMENT findby
     ( namingservice
     | domainfinder
     )>
```

One way to find the object reference of a service to be used by the DomainManager, via the name binding path specified in this namingservice element. Not an understandable use case for an SCA-compliant production program, as object references of services become available to the DomainManager when they register with the DomainManager. Also, services are not supposed to be used by the DomainManager until they have registered with the DomainManager. In such cases, one possible use of this findby element could be to ensure that the name of the registering service matches a domainfinder name and thus establishes that the registering service is also meant for DomainManager's use, such as a Log.

SCA Event Service Usage for CF Providers and Component Developers

BOEING®

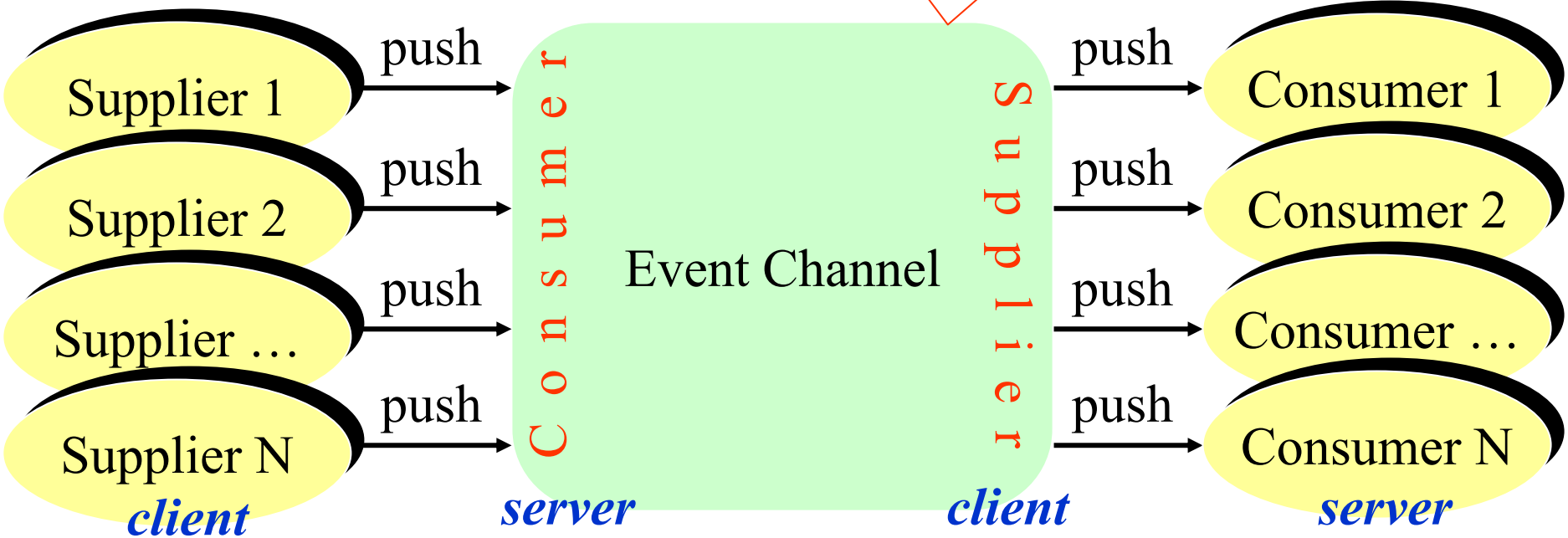
OMG Event Service Architecture

- Decoupled, asynchronous communication model between clients and servers.
  - Event producers (AKA suppliers/publishers) and consumers (AKA subscribers).
- Suppliers "publish" information without knowing who the consumers are.
- Consumers "subscribe" to information without regard to the information source.
- Suppliers are shielded from exceptions resulting from any of the consumer objects being unreachable or poorly behaved.

OMG Event Service Architecture (cont.)

Central Role in Event Service Specification

- *Supplier(s)/consumer(s) intermediary.*
- *Acts as consumer to supplier(s) and supplier to consumer(s).*



BOEING®

Event Service Usage in the OE

- Used by CF for
  - Connection of CF domain components to required event channels.
  - Connection/disconnection of application components to/from required event channels in the CF domain, during instantiation/tear down of an application in the CF domain.
  - Logging SCA-specified events to SCA-specified event channels.

Event Service Usage in the OE (cont.)

- OE provides two standard event channels:
  - **Incoming Domain Management Event Channel**
    - Allows the CF domain to become aware of changes in the domain.
      - ✓ e.g. device state changes.
  - **Outgoing Domain Management Event Channel**
    - Allows CF domain clients to become aware of changes in the domain.
      - ✓ e.g. Human Computer Interface (HCI) application receiving events pertaining to device/service/application software/instantiated application additions/removals from domain.
- Application developers allowed to setup other “non-standard” event channels.

Required Event Service Subset for SCA-Compliant OEs

- Based on OMG's Event Service specification.
  - Push interfaces (*PushConsumer* and *PushSupplier*) of the *CosEventComm* CORBA module as described in OMG Document formal/01-03-01: Event Service, v1.1.
  - Compatible IDL for the *CosEventComm* CORBA module in the OMG Document formal/01-03-02: Event Service IDL, v1.1.
- Canonical Push Model
  - Supplier(s) push events to the event channel which in turn pushes them to all registered consumer(s).

Event Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
module CosEventComm {

  exception Disconnected{};

  interface PushConsumer {
    void push (in any data) raises(Disconnected);
    void disconnect_push_consumer();
  };

  interface PushSupplier {
    void disconnect_push_supplier();
  };
...
};
```

Implemented by push consumers.
The object reference is registered with a push supplier.

The push supplier sends event data to the push consumer via push ().

Suppliers and consumers can disconnect from each other.

disconnect_push_consumer () disconnects the consumer from its supplier.
Invoked by the supplier or the entity that has connected the supplier to the consumer.

disconnect_push_supplier () disconnects the consumer from the supplier.
Invoked by the consumer or the entity that has connected the consumer to that supplier.

Implemented by push suppliers.

Event Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
module CosEventChannelAdmin {

  exception AlreadyConnected {};
  exception TypeError {};

  interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
  };

  interface ConsumerAdmin {
    ProxyPushSupplier obtain_push_supplier();
    …
  };

  interface SupplierAdmin {
    ProxyPushConsumer obtain_push_consumer();
    …
  };
```

Implemented by Event Service provider. All operations are used (i.e. invoked) by the CF (DomainManager and ApplicationFactory) for creating event channels and connecting event suppliers and event consumers to event channels (during DeviceManager component registration or ApplicationFactory application instantiation).

Invoked for consumers that want to connect to an event channel.

Invoked for suppliers that want to connect to an event channel.

These operations obtain the event channel intermediary's (i.e. known as the proxy push supplier and proxy push consumer) for the real consumer and the supplier to connect to. This way, the push consumer and the push supplier are not directly aware of each other. They are decoupled and shielded from each other.

Event Service IDL
(Mandatory subsets for SCA Compliant OEs)

```
module CosEventChannelAdmin {

interface EventChannel {
    ConsumerAdmin for_consumers();
    SupplierAdmin for_suppliers();
    void destroy();
  };
```

Permanently destroys the event channel, including any events that it has not yet delivered.

Also, destroys all administrative objects created by that channel (i.e. all ConsumerAdmin and SupplierAdmin objects) and all proxy objects created by those administrative objects (i.e. all ProxyPushConsumer and ProxyPushSupplier objects). Note: The above does not include the destruction of client-side object references to the administrative and proxy objects. It is the responsibility of the CF provider to destroy these client-side object references.

All connected consumers and suppliers are notified when their channel is destroyed (i.e. via PushConsumer::disconnect_push_consumer () and PushSupplier::disconnect_push_supplier ()).

Event Service IDL
(Mandatory subsets for SCA Compliant OEs)

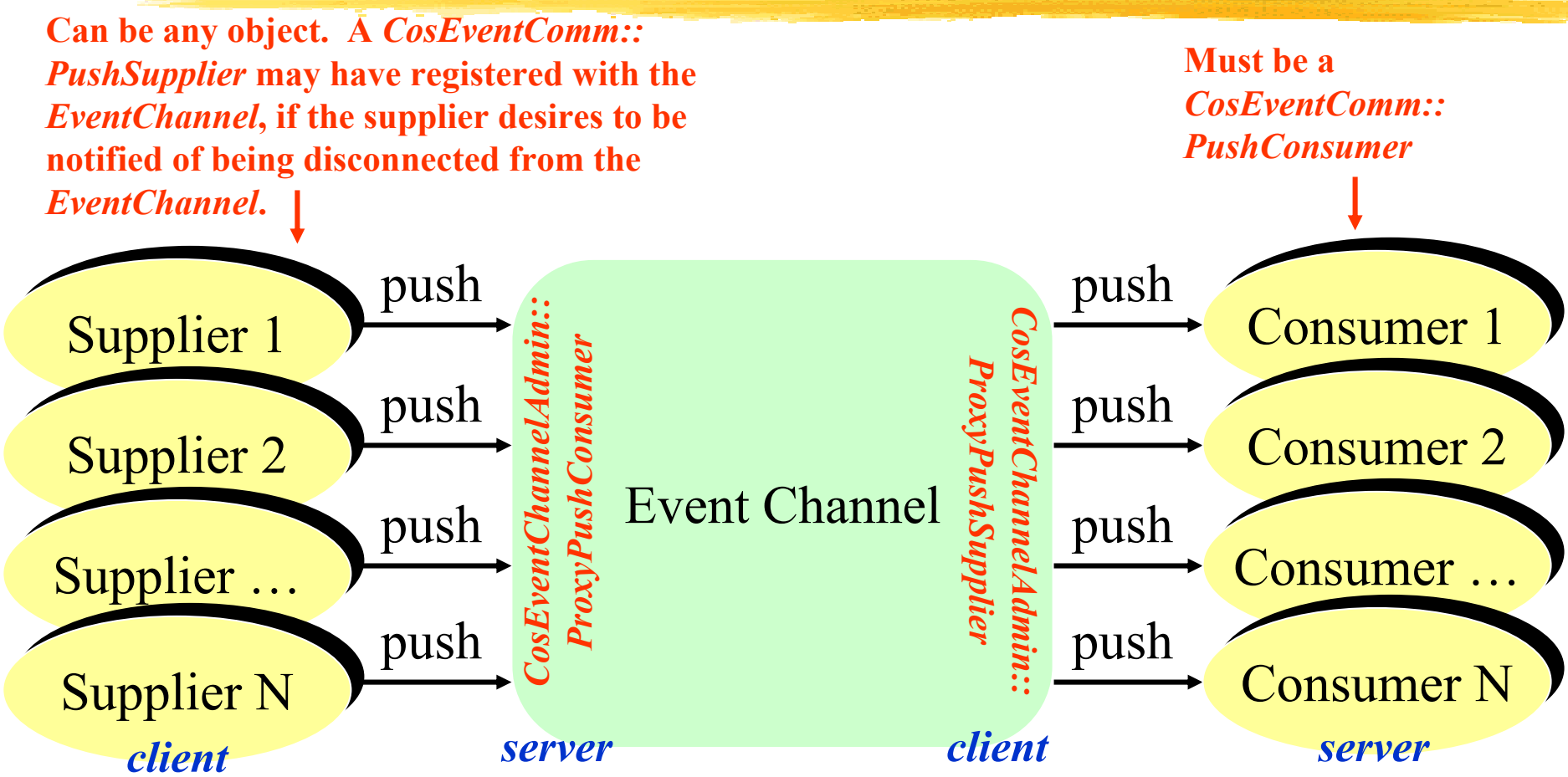
```
interface ProxyPushConsumer: CosEventComm::PushConsumer {
    void connect_push_supplier(
    in CosEventComm::PushSupplier push_supplier)
      raises(AlreadyConnected);
  };
```

The supplier is connected to the event channel (i.e. the event channel proxy push consumer that looks like the real consumer to the supplier).

```
interface ProxyPushSupplier: CosEventComm::PushSupplier {
    void connect_push_consumer(
    in CosEventComm::PushConsumer push_consumer)
      raises(AlreadyConnected, TypeError);
  };
};
```

The consumer is connected to the event channel (i.e. the event channel proxy push supplier that looks like the real supplier to the consumer).

Event Channel Interfaces Exposed to Suppliers and Consumers for Pushing Events



CF's Usage of the Event Service (DomainManager Construction)

- Creates the IDM and ODM event channels.
- Connects the DomainManager as a push supplier to the ODM event channel.
  - Creates a *CosEventChannelAdmin::SupplierAdmin* (via *CosEventChannelAdmin ::EventChannel::for_suppliers (),*
  - uses it (*CosEventChannelAdmin::SupplierAdmin::obtain_push_consumer ())* to obtain a *CosEventChannelAmdin::ProxyPushConsumer*, and
  - connects the DomainManager to the *EventChannel* as a push supplier (via *CosEventChannelAdmin::ProxyPushConsumer::connect_push_supplier ()).*
- Missing SCA Requirements
  - Even though the intent of the IDM event channel is to consume device state change events such that the domain (i.e. DomainManager and ApplicationFactory) becomes aware of what devices are or are not available for servicing application or DeviceManager component instantiations, the SCA does not explicitly state these requirements.
  - As such, an SCA-compliant CF provider does not have to do any more than creating the IDM event channel and connecting suppliers or consumers to it, as specified in DCD or SAD XML files.

CF's Usage of the Event Service (DomainManager Processing DeviceManager DCD Connections)

- If the user or provider in a connection is an event channel and it is not created yet then an event channel (*CosEventChannelAdmin::EventChannel*) is created.
- Depending on whether the event channel is the supplier (i.e. user or client) or the consumer (i.e. provider or server) in the connection, the *CosEventChannelAdmin::ProxyPushConsumer* or *CosEventChannelAdmin::ProxyPushSupplier* object is retrieved.
- The other end of the connection is connected to the event channel as a push consumer or a push supplier.

CF's Usage of the Event Service (DomainManager Processing DeviceManager DCD Connections)

- Event Channel Federation
  - Federation of event channels (i.e. one event channel acting as the push supplier and the other as a push consumer) in connections is up to the CF provider, as the SCA has no requirements in this area.
- When federation of event channels is not supported…
  - If the connection provider is an event channel, the user must ALWAYS support the *CF::PortSupplier* and *CF::Port* interfaces.
  - If the connection user is an event channel, the provider may or may not support the *CF::PortSupplier* interface.

CF's Usage of the Event Service (DomainManager Processing DeviceManager DCD Connections)

- When federation of event channels is not supported…
  - If the connection user supports the *CF::Port* interface, we have a two-way connection…
    - The user component's supplied port (via *CF::PortSupplier::getPort ()*) is connected to the event channel as a push supplier, and
    - The event channel's *CosEventchannelAdmin::ProxyPushConsumer* is connected to the user component via *CF::Port::connectPort (),* so that the user can push events to the event channel.
  - If the connection user is an event channel, we have a one-way connection…
    - The connection provider (regardless of whether or not it supports *CF::PortSupplier::getPort ()),* is connected to the connection user (i.e. the event channel) as a push consumer.

CF's Usage of the Event Service (Destruction of Event Channels created by the DomainManager)

- The destruction of the DomainManager-created event channels when no more consumers or producers are connected to them is left to the CF providers.
- The IDM and ODM event channels must remain alive for the DomainManager's lifetime.
- Depending on the particular implementation architecture, the CF provider may choose to leave other disconnected event channels alive during DomainManager's lifetime, due to these event channels becoming available to the entire domain (and not just the specifying DeviceManager DCD) and remaining in use, even after the parent DeviceManager unregisters from the domain.

CF's Usage of the Event Service (DomainManager Sending Events to the ODM Event Channel)

- During successful domain entity additions and removals
  - Application installations/uninstallations
  - DeviceManager/device/service registrations/unregistrations

CF's Usage of the Event Service (ApplicationFactory)

- Has access to the ODM event channel through the DomainManager.
- Uses the ODM event channel for recording successful application instantiation events.
- Identical requirements and processing as the DomainManager for connecting components during application instantiation.

CF's Usage of the Event Service (Application)

- Has access to the ODM event channel through the DomainManager.
- Uses the ODM event channel for recording successful application tear down events.
- Destruction of disconnected event channels during application tear down is left to CF provider. Depending on the particular implementation architecture, the CF may destroy the event channel or keep it in case it is in use by other running applications or other persistent domain components (i.e. DeviceManager deployed components connected to the same event channel).

Component Developers' Usage of the Event Service

- Push consumers must support the *CosEventComm::PushConsumer* interface and push suppliers must support the *CosEventComm::PushSupplier* interface so that…
  - They can be connected to event channels via the CF.
  - Push consumers can have events pushed to them.
  - They can be notified when getting disconnected from the event channel.

Domain Profile Event Service Expressions (DeviceManager DCD or ApplicationFactory SAD XML examples)

```
<!ELEMENT connections
    ( connectinterface*
    )>

<!ELEMENT connectinterface
    ( usesport
    , ( providesport
      | componentsupportedinterface
      | findby
      )
    )>
<!ATTLIST connectinterface
    id  ID              #IMPLIED>

<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
      | devicethatloadedthiscomponentref
      | deviceusedbythiscomponentref
      | findby
      )
    )>
```

```
<!ELEMENT providesport
    ( providesidentifier
    ,   ( componentinstantiationref
      | devicethatloadedthiscomponentref
      | deviceusedbythiscomponentref
      | findby
      )
    )>

<!ELEMENT componentsupportedinterface
    ( supportedidentifier
    , ( componentinstantiationref
      | findby
      )
    )>
```

The non-event channel end of the connection is always expressed using one of the many permutations of usesport, providesport, componentsupportedinterface or findby elements.

BOEING®

Domain Profile Event Service Expressions (DeviceManager DCD or ApplicationFactory SAD XML examples)

```
<!ELEMENT findby
    ( namingservice
    | domainfinder
    )>

<!ELEMENT domainfinder EMPTY>
<!ATTLIST domainfinder
    type        (filemanager | log | eventchannel | namingservice)   #REQUIREDCDATA
    name        CDATA       #IMPLIED
```

The event channel, as either a user or a provider in the connection, is always expressed using the findby domainfinder element.

The domainfinder type will be eventchannel and the domainfinder name will be "IDM_Channel", or "ODM_Channel" or any user-defined name for new event channels that ApplicationFactory or DomainManager must create. If name is not supplied, then the IDM event channel is assumed.

Domain Profile Expressions for Creating and/or Connecting to an Event Channel

- So, creating and connecting to a new event channel for a component is as easy as implementing the *CosEventComm::PushConsumer* or *CosEventComm::PushSupplier* interface and defining XML for a domainfinder findby element with an existing or new name for identifying the desired event channel. That is all.
- The CF will take care of creating the event channel, and establishing the connection between the component and the event channel, including all Event Service administrative details.

SCA Connections

SAD vs. DCD Connections

- There are two types of SCA connections:
  - Those expressed for connecting persistent components deployed to the domain by DeviceManagers.
    - The DeviceManager and its deployed devices and services to each other or to the rest of the domain.
    - Expressed in a DeviceManager's DCD XML file.
  - Those expressed for connecting components of an instantiated application.
    - Expressed in an application's SAD XML file.
- Though very similar, and using the same XML DTD, there are slight differences in processing the *connections* element of a DCD, than that of a SAD.
- These differences are hi-lited through the remaining slides.

Assembly Connections Description - *connections* XML Element

- SCA connections are expressed in the *connections* element of the parent assembly (i.e. the SAD or the DCD).
- Processed by ApplicationFactory or DomainManager.

```
<!ELEMENT connections
    ( connectinterface*
    )>
```

- The *connections* element provides the connections map between the assembly components.
- Each connection pairing in the assembly is expressed through a *connectinterface* element.

Single Connection Description - *connectinterface* XML Element

- For each connection, there is a user and a provider.
- The *connectinterface* element describes the user and the provider in a connection and how they are to connect to each other.
  - How to find the user and provider component?
    - What is their object reference?
  - Once found, how do they participate in this connection?
    - What is the object reference of the component's port or interface through which the connection is established?
- The connection user is always described by a *usesport* XML element.
- The connection provider is described via a *providesport*, *componentsupportedinterface*, or *findby* element.

```
<!ELEMENT connectinterface
    ( usesport
    , ( providesport
        | componentsupportedinterface
        | findby
        )
    )>
<!ATTLIST connectinterface
    id  ID              #IMPLIED>
```

connectinterface XML Element – Connection id

- The connectinterface id attribute is the connection id supplied to CF::Port::connectPort () for uniquely identifying the connection.
- If id is not provided, it is uniquely created by the CF.
  - By ApplicationFactory for SAD connections.
  - By the DomainManager for DCD connections.
- Not applicable if the connection user is an event channel.

```
<!ELEMENT connectinterface
    ( usesport
    , ( providesport
        | componentsupportedinterface
        | findby
        )
    )>
<!ATTLIST connectinterface
    id  ID              #IMPLIED>
```

Connection User - usesport XML Element

- Available methods of finding the object reference of the user component in the connection.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

usesport componentinstantiationref XML Element

- The user component is part of the assembly (id attribute of a componentinstantiation element in the SAD or DCD).
  - A DCD device or service created by the DeviceManager or a SAD application component created by the ApplicationFactory.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

```
<!ELEMENT componentinstantiationref EMPTY>
    <!ATTLIST componentinstantiationref
        refid     CDATA     #REQUIRED>
```

usesport devicethatloadedthiscomponentref XML Element

- Object reference of the domain logical device used by ApplicationFactory to load the specified assembly component.
- Applicable when a component (for instance a modem adapter) is pushing data/commands to a non-CORBA-capable device, such as a modem.
- Applicable only to SAD connections (not DCD connections).
  - ApplicationFactory is in charge of both creating SAD components and later connecting them. So, it has access to information required for processing this XML element.
  - At this time, there are no public CF IDL interfaces for the DomainManager to know (completely) how the DeviceManager created the DCD components.
  - DCD components are created by DeviceManagers and connected by the DomainManager.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

```
<!ELEMENT devicethatloadedthiscomponentref EMPTY>
    <!ATTLIST devicethatloadedthiscomponentref
        refid        CDATA        #REQUIRED>
```

usesport deviceusedbythiscomponentref XML Element

- Object reference of the domain logical device used by a component in the assembly.
- Used when a component is pushing/pulling data/commands to a device.
- Applicable only to SAD connections (not DCD connections).
  - ApplicationFactory is in charge of both creating SAD components and later connecting them. So, it has access to information required for processing this XML element.
  - At this time, there are no public CF IDL interfaces for the DomainManager to know (completely) how the DeviceManager created the DCD components.
  - DCD components are created by DeviceManagers and connected by the DomainManager.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

```
<!ELEMENT deviceusedbythiscomponentref EMPTY>
    <!ATTLIST deviceusedbythiscomponentref
                refid       CDATA    #REQUIRED
                usesrefid   CDATA    #REQUIRED>
```

usesport findby XML Element

- Object reference found via the Naming Service or the Domain Finder.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

```
<!ELEMENT findby
    ( namingservice
    | domainfinder
    )>
```

usesport findby namingservice XML Element

- The *namingservice name* attribute is used to search the CORBA Naming Service for the desired component.
- The name attribute denotes a complete naming context.
- In implementations that adhere to the SCA Security Supplement, one possible interpretation is that processing this element only applies to SAD connections.
  - All entities managed by the DomainManager use the same Naming Service.
  - Entities outside the DomainManager process space (i.e. DCD components created by a DeviceManager) cannot access the DomainManager Naming Service.

```
<!ELEMENT namingservice EMPTY
<!ATTLIST namingservice
        name            CDATA       #REQUIRED>
```

usesport findby domainfinder XML Element

- Applies to both SAD as well as DCD connections.
- Used to find the object reference of a specific *type*, possibly, with an optional *name*.
- If a *name* is not supplied…
  - *type* of *filemanager* – DomainManager's FileManager
    - Most likely not applicable to the usesport elements, as it does not support the CF::PortSupplier interface by default. Depends on the implementation architecture. More appropriate for a connection provider.
  - *type* of *namingservice* – domain Naming Service
    - Most likely not applicable to the usesport elements, as it does not support the CF::PortSupplier interface by default. Depends on the implementation architecture. More appropriate for a connection provider.
  - *type* of *log* – null reference
    - Not applicable to either connection user or providers. Why connect to a null reference.
  - *type* of *eventchannel* – Covered in the Event Channels section

```
<!ELEMENT domainfinder EMPTY>
<!ATTLIST domainfinder
    type        (filemanager | log| eventchannel | namingservice)   #REQUIREDCDATA
    name        CDATA       #IMPLIED
```

usesport findby domainfinder XML Element

- If a *name* is supplied…
  - The domain is searched for a reference of a specific type with the provided name.
- Possible extension to the *domainfinder* element
  - Depending on the implementation architecture, there could be many types of services not covered with the 4 basic types specified in the SCA, that would need to participate in DCD or SAD connections.
  - One way to express them using the *domainfinder* element is through extending the usage of the *namingservice* element and extending and restricting the behavior of *DomainManager::registerService ()* as follows:
    - Service names are unique in the entire domain, regardless of their type.
    - Use the *domainfinder type* of *namingservice* with the *domainfinder name* referring to the unique service name.
    - As services get registered with the DomainManager, they get added to the "domainfinder database", using their unique name.
    - Registration of services with duplicate names, regardless of their type is not allowed.

When the usesport XML Element Refers to an Event Channel

- Covered in the Event Channels section of the presentation.

usesport usesidentifier XML Element

- Identifies which "uses port" on the component participates in the connection relationship.
- Corresponds to an id for one of the component's ports in the component's SCD.
- The component must support the *CF::PortSupplier* interface.
- The CF calls the component's *CF::PortSupplier::getPort ()* with the supplied *usesidentifier*.
- The returned reference from *getPort ()* must support the *CF::Port* interface, as the CF will call its *CF::Port::connectPort (),* passing in the connection provider (identified via the *connectinterface providesport, componentsupportedinterface*, or *findby* methods) and the *connectinterface id* attribute implied or implicit value as the connection id.

```
<!ELEMENT usesport
    ( usesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

```
<!ELEMENT usesidentifier (#PCDATA)>
```

Connection Provider - *providesport* XML Element

- Identical XML expression and CF processing as the *usesport* XML element for finding the object reference of the participating component and its "provides port".
- The retrieved provides port object reference is passed to the connection's "uses port" *CF::Port::connectPort ()* operation.
- The *providesidentifier* corresponds to a *repid* attribute for one of the component *ports* elements, in the component's SCD.

```
<!ELEMENT providesport
    ( providesidentifier
    ,   ( componentinstantiationref
        | devicethatloadedthiscomponentref
        | deviceusedbythiscomponentref
        | findby
        )
    )>
```

Connection Provider - componentsupportedinterface XML Element

- Specifies a component that has an interface (a supportsinterface) that can satisfy an interface connection to the corresponding "uses port".
- Identical XML expression and CF processing for finding the component and it's participating supported interface, as the *usesport* and *providesport* XML elements.
- The component is identified by a *componentinstantiationref* or *findby* element.
- Same Security Supplement restrictions specified in the *usesport* section apply to *findby namingservice* elements.

```
<!ELEMENT componentsupportedinterface
        ( supportedidentifier
        , ( componentinstantiationref
          | findby
          )
        )>
```

componentsupportedinterface supportedidentifier XML Element

- The *supportedidentifier* element identifies the component's supported interface which participates in the connection.
- Corresponds to the *repid* attribute of one of the component's *supportsinterface* elements, in the component's SCD.
- A reference to this interface is retrieved via the retrieved component's *CF::PortSupplier::getPort (), with value of the supportedidentifier element.*

```
<!ELEMENT componentsupportedinterface
        ( supportedidentifier
        , ( componentinstantiationref
          | findby
          )
        )>
```

BOEING®

Connection Provider - *findby* XML Element

- If the connection provider is not an event channel…
  - There is no port or interface reference to retrieve. Once the component is retrieved, we are done with determining the providing end of the connection.
  - Identical XML expression and CF processing to the *findby* element for *usesport*, *providesport*, or *componentsupported* interface XML elements.
- If the connection provider is an event channel…
  - Covered in the Event Channels section of the presentation.
- Same Security Supplement restrictions specified in the *usesport* section apply to *findby namingservice* elements.

```
<!ELEMENT connectinterface
    ( usesport
    , ( providesport
        | componentsupportedinterface
        | findby
        )
    )>
<!ATTLIST connectinterface
    id  ID              #IMPLIED>
```

BOEING®

SCA Connections - Component Developer Responsibilities

- A component's supportsinterface(s) and/or port(s) are fully expressed in the component's Software Component Descriptor file (SCD), via the *componentfeatures supportsinterface* and *ports* XML elements.
- For (non-event channel) components that play the user role in the connection…
  - Must support the *CF::PortSupplier* interface.
  - The object returned from *CF::PortSupplier::getPort ()* must support the *CF::Port* interface.
- For (non-event channel) components that play the provider role in the connection…
  - In case of *providesport* or *componentsupportedinterface* XML elements…
    - Must support the *CF::PortSupplier* interface.

SCA Connections – Assembly Writer Responsibilities

- Correct expression of a *connectinterface* element for each user and provider pair in the connections map.

References

References

- http://jtrs.army.mil/sections/technicalinformation/fset_technical_sca.html
  - *Software Communication Architecture Specification*, V3.0
  - *Software Communications Architecture Security Supplement*, V3.0
- www.omg.org
  - OMG Interoperable Naming Service Specification, OMG Document formal/00-11-01
  - OMG Event Service Specification, OMG Document formal/01-03-01