

PocoCapsule™ CORBA

Component Framework for
POA Server, Event/Notification, DDS, SCA, and RTC

Developer Guide

Release 1.0
October 8, 2007

Pocomatic Software LLC

Foster City, CA USA

PocoCapsule™ CORBA C++ Developer Guide

Release 1.0
October 8, 2007
Foster City, CA. USA

Copyright© 2007 by Pocomatic Software LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of Pocomatic Software LLC.

This document is available at:

<http://www.pocomatic.com/docs/pococpp-corba-dev-guide.pdf>

Table of Content

Table of Content.....	2
1. Introduction.....	4
1.1 The need for a component framework.....	4
1.2 CCM: More trouble than worth.....	5
1.3 PocoCapsule: Simple is better.....	6
1.4 DSM: Simple vs Comprehensive.....	7
1.5 About this document.....	8
2. An Hello Example.....	10
2.1 Defining the object interface.....	10
2.2 Implementing the server object.....	11
2.3 Declarative deployment descriptor.....	12
2.4 Dynamic invocation proxies.....	14
2.5 The Container Server.....	15
2.6 Playing this example.....	16
2.7 Changing Server Structure/Configuration.....	17
3. Simple Server Applications.....	20
3.1 DSM and PocoCapsule/CORBA.....	20
3.2 PocoCapsule/CORBA Schema Overview.....	21
3.3 The <orb> element.....	23
3.4 The <object> element.....	24
3.4.1 <bean> or <ref> child element of <object>.....	25
3.4.2 <i>impl-ref</i> attribute of <object>.....	25
3.4.3 <i>impl-type</i> and <i>interface</i> attributes of <object>.....	26
3.4.4 <i>id</i> attributes of <object>.....	27
3.4.5 <i>oid</i> attribute of <object>.....	27
3.4.6 <i>uri</i> attribute of <object>.....	27
4. POA Server Applications.....	30
4.1 POA and POA Policies.....	30
4.2 The <poa> element.....	33
4.2.1 <polices> element.....	33
4.2.2 <property> element.....	34
4.2.3 <object> element.....	35
4.2.4 <i>id</i> attribute.....	35
4.2.5 <i>name</i> attribute.....	35
4.3 The POA Request Processing Policy.....	35
4.3.1 Use Activated Object Map Only.....	36
4.3.2 Use Default Servant.....	36
4.3.3 Use Servant Manager (Activator).....	37
4.3.4 Use Servant Manager (Locator).....	38
4.3.5 Transparent Location Forward.....	39
4.4 Customize Server Models.....	40
5. Event/Notification Applications.....	44

5.1 The OMG Event/Notification Service.....	44
5.1.1 Connection model	44
5.1.2 Event pushing operation(s) and message types	46
5.1.3 POEM vs EVIL.....	46
5.2 Event Consumer Implementation.....	47
5.3 The <event-consumer-adaptor> element.....	48
5.3.1 <event-consumer> element.....	51
5.3.2 <admin>, <channel>, <service> elements	52
5.3.3 <props> element and event filter.....	53
5.4 Event Supplier Implementation.....	54
5.5 The <event-supplier-adapter> element.....	57
5.5.1 <event-supplier> element.....	59
5.5.2 <admin>, <channel>, <service> elements	60
5.5.3 <props> element.....	61
6. DDS Applications.....	62
6.1 OMG Data Distribution Service (DDS)	62
6.2 DDS Data Model, Typed Reader and Writer	64
6.3 DDS Data Emitter Application	65
6.4 DDS Data Reader Listener Implementation	66
6.5 PocoCapsule DDS Application Deployment Model.....	67
6.6 The <dds-topic> element.....	70
6.7 The <dds-publisher> and <dds-writer> elements.....	70
6.8 The <dds-subscriber> and <dds-reader> elements	72
Appendix A. SDR and JTRS-SCA.....	76
Appendix B. Robotic Component and OMG-RTC	82
Appendix C. CORBA Overview.....	86
C.1 CORBA Architecture.....	86
C.2 Characteristics of CORBA Architecture	87
C.3 CORBA Object Interfaces	88
C.4 CORBA Object References.....	88

1. Introduction

1.1 The need for a component framework

Handcrafting a large scale distributed application from scratch is hard, expensive, risky, and bound for heavy maintenance burdens. The goal of the CORBA architecture was to reduce these pains by factoring out reusable low level functions/services in order for application developers to focus on high level business logic and domain issues.

With a CORBA middleware, an iteration cycle in developing a distributed application can be divided into the following three sequential phases:

- ***System partition and interfaces design***: In this phase, architects/developers partition the application into business logic units (modules and objects) and define their interfaces using the OMG IDL¹.
- ***Implement business logic***: In this phase, developers write business logic implementations for the partitioned units. These implementations support all operations of their respective IDL interfaces.
- ***Deployment and configuration (D&C)***: In this phase, developers assemble individual business logic units together and enlist them to the underlying ORB runtime and CORBA common services (such as OMG Event/Notification and DDS) with desired policy or QoS configuration settings.

CORBA indeed simplified the first two phases of this development iterate cycle by making distributed applications similar to conventional well understood non-distributed OO applications. Unfortunately, CORBA also brought in considerable framework level complexities and is far from satisfactory on simplifying the third phase. The programming models (APIs) of ORB/POA server and most common services (such as Event/Notification and DDS) are heavily system or framework oriented. These programming models are awfully complex to most domain/business application developers, even cumbersome to most system application developers, middleware experts, and ORB and OMG common services implementers themselves.

This situation² calls for a component framework that could substantially simplify CORBA applications for domain experts whose skills and interests are not low level middleware plumbing but solving high level domain issues and building sophisticated and/or intelligent business applications. This framework should even allow domain experts with little programming skill to quickly, comfortably, and flexibly assemble, deploy, and configure sophisticated CORBA applications.

¹ In MDA, developers could use UML as modeling language and generate CORBA IDL using tools.

² This is a generic situation, also observed in J2EE, WebServices, high performance computing (HPC), robotic motion control applications, or vertically in enterprise (EAI, ERP), healthcare (HL7, EHR/HISA), intelligent traffic and/or vehicle control systems (ITS), datacom and telecom (TMN, IN/AIN, Parlay/OSA, SIP, VoIP, and NMS/OSS/BSS), defense (SDR, ATC, C4I, JBI, and OACE).

1.2 CCM: More trouble than worth

The CORBA Component Model (CCM) was supposed to be the solution, however, made the situation much worse. CCM significantly increases the complexity of all three phases of the CORBA application development cycle with much more contrived framework complexities and imposes a steep learning curve even for CORBA experts, not to mention the dreadful entry barrier for business application developers.

CCM also introduces a compliance barrier that not only tightly locks in compliant component implementations, but also locks out almost all existing CORBA business logic implementations (servants) and even certain client applications³ and important OMG common services. To be reusable in CCM, these CORBA application implementations and common services have to be refactored or even completely redefined and reimplemented. All of these largely defeat the very purpose of having this component framework.

For instance, numerous scholastic publications claimed that CCM greatly reduced lines of code of OMG Event/Notification Service applications. Nevertheless, as a matter of fact, the “*Events as Valuetypes of IdL*” (EVIL) model of CCM not only forces developers to write more rather than less plumbing code⁴, but also rules out most existing Event/Notification Service implementations⁵ and almost all already deployed, standard based Event/Notification Service applications⁶. These standards/specifications as well as their service and application implementations had to be significantly rewritten to be compliant to and interoperable with the CCM EVIL. Even if additional plumbing code and such a massive reengineering exercise were acceptable, it would still incur considerable performance overhead and significant network bandwidth penalty because the EVIL model requires valuetypes representing complex events/alarms to be inserted into and transferred as *CORBA::Anys*.

CCM was largely influenced by techniques developed during later 90’s in the Apache Avalon project and EJB 1.0. These techniques have been abandoned in the mainstream industry for years and the EJB 1.x/2.x has become a classic negative example of architecture design. Nevertheless, taking either an ostrich attitude or a “if you can’t convince them (users), confuse them!” tactic, CCM is still constantly being hyped with buzzwords such as “advanced”, “mission-critical”, and “lightweight”, even though its techniques are out-of-date, its implementations are vulnerable and heavy. Whenever being criticized for its heaviness and complexity, CCM always responds with its favorite equation claiming that advanced component frameworks for distributed mission-critical

³ For instance, in some CCM implementations, it becomes very difficult to develop OMG asynchronous method invocation (AMI) client applications to access a CCM server.

⁴ For instance, event valuetype factories, factory registration, and double dispatch.

⁵ Among a handful (more than 6) Event/Notification Service implementation only one (VisiNotify) is able to support interoperable valuetype pass-through without non-portable vendor specific workaround (such as link or load user application specific valuetype factory into the channel executable).

⁶ Examples of such applications include, for instance, almost all CORBA/TMN applications based on 3GPP, TMF and ITU-T standards/specifications.

applications deserve to be heavy, complex, recondite, and hard to use, and their simplifications could only yield less featured inferior editions.

1.3 PocoCapsule: Simple is better

PocoCapsule/CORBA overturns the CCM's equation above definitely in the "David vs Goliath" face-off. It reiterates the classic "less is more" aphorism by demonstrating an intuitive, easy to use, and lightweight while substantially powerful and reliable component and D&C framework for CORBA server applications, OMG Event/Notification service applications, OMG-DDS applications, JTRS-SCA SDR applications, and OMG-RTC robotic component applications, etc.

PocoCapsule/CORBA does not force applications to stick to any particular component model. It accepts virtually any C++ objects, referred to as *plain old C++ objects* (POCO), as components and is therefore neutral to all component models. Business logic object implementations and common services, which were designed, developed, and built with or without knowledge of PocoCapsule/CORBA framework, can all be seamlessly used and manipulated as components by PocoCapsule/CORBA, and can still be tested and reused in their respective frameworks without PocoCapsule as well. These components include, for instance, existing CORBA 2.x business logic implementations (servants), CORBA 3.x and CCM components, JTRS-SCA core framework (CF) resources and devices, OMG-RTC components, and existing and new OMG common services such as OMG Event/Notification Service and DDS and their application object implementations (consumers, suppliers, data readers and writers). This neutral component model implies:

- There is no compliance barrier. Legacy implementations are seamlessly supported.
- There is no ramp up time and no learning curve⁷ for another contrived component model.
- The straightforwardness of design and implementation phases of CORBA 2.x application development are retained.
- Existing and matured assets (various vertical domain standards and implementations of ORBs, OMG common services, and applications) and investments (trainings) can mostly be preserved and are reusable explicitly with minimum cost, impact, uncertainty, and risk.

For the third development phase, the declarative assembly and deployment model of PocoCapsule/CORBA largely melts away plumbing complexities and eliminates the need for domain/business application developers to deal with or even to learn those low level system details. Many CORBA application scenarios and common services used to be notoriously obscure to CORBA experts and dreadful obstacles to CCM now become fairly obvious and handy even to CORBA novices.

⁷ A CORBA, CCM and COS services (Naming and Event/Notification) training course offered by one CCM advocate took 15 weeks. In comparison, it needs less than 15 hours for a CORBA novice to be able to master much more features in PocoCapsule/CORBA.

Although CCM component deployment is also declarative, its schemas are awfully complicated and are not to be used manually. The typical CCM excuse is that tools would emerge and solve this problem eventually, which was exactly what the old 1.x and 2.x EJB claimed, promised, and failed. This complexity is also a discouraged high entry barrier for third party tool vendors. The consequence is that advanced tools have hardly emerged. A number of primitive tools have almost no help on reducing this complexity but only introduce additional complexities, steeper learning curves, tighter vendor lock-in barriers, heavier bloated code generations, and more obstacles and difficulties for application debugging, diagnosis, testing, integration, and maintenance.

In contrast, PocoCapsule/CORBA's assembly and deployment schemas are significantly simple, straightforward, and intuitive that can quickly be grasped by average developers, easily be used manually, and completely eliminates the entry barrier for application developers as well as third party UI tool vendors.

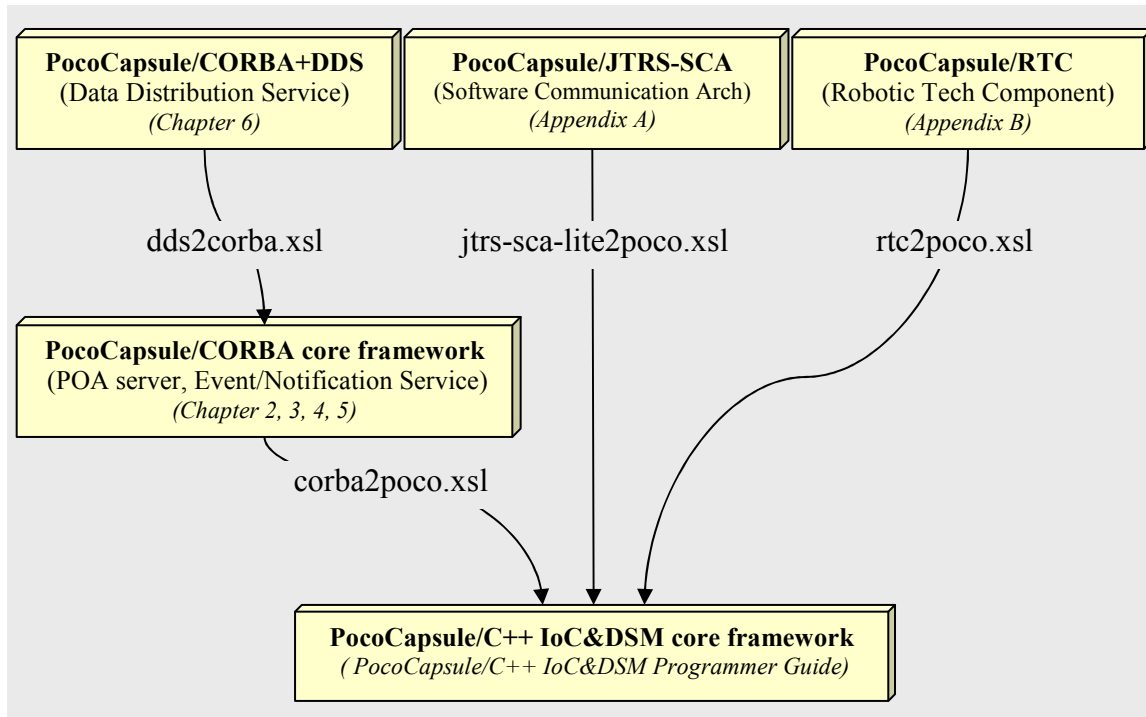
1.4 DSM: Simple vs Comprehensive

Another classic CCM equation claims that a component framework meeting comprehensive requirements of diverse applications will have to sacrifice its simplicity. According to this equation, a framework would either have to define a complicated and bloated model that comprehensive domain requirements or have to ignore high level domain requirements and force developers to use a rigid low level model,

Contrast to this claim, PocoCapsule/CORBA is simple, slim, and intuitive while still able to provide comprehensive solutions for diverse applications in arbitrary object or component models. PocoCapsule/CORBA achieves the seemingly contradicted combination of *simplicity*, *intuitiveness*, *expressiveness*, and *comprehensiveness* by defining a minimum but customizable core schema. This allows users or third parties to define their own *domain specific models* (DSM)⁸ as extensions of the core deployment model. Domain requirements from diverse applications, alternative component models, and/or higher abstraction levels can be expressed in user or third party defined high level DSMs. The core framework is able to accept any such DSMs based on their schema transformation templates (in form of XSLT style sheets). This allows users or third parties to easily, quickly, declaratively, and portably extend and customize the PocoCapsule core framework into various higher level, structured, and more expressive deployment frameworks.

In fact, all deployment models within PocoCapsule/CORBA framework suite mentioned previously as well as the PocoCapsule/CORBA core framework itself are merely some DSM variations based on the PocoCapsule/C++ IoC&DSM core framework as illustrated in the following diagram and to be discussed in their indicated chapters or document:

⁸ See more DSM discussions on <http://www.dsmforum.org>



These domain specific models can immediately be used by applications or further customized into higher level and more expressive DSM variations.

1.5 About this document

This document serves as a developer guide and manual of PocoCapsule/CORBA component framework suite. The following topics and contents are to be covered:

- Basic CORBA servers using servant (inherited from skeleton) and native (not inherited from skeleton) implementations are discussed in chapters 2, 3, and also illustrated by the examples/corba/hello and the examples/corba/hello-tie examples.
- Sophisticated CORBA POA servers with user specified POA policies, and usage of default servant and/or servant managers are covered in chapters 4 and also the examples/corba/poa-server example.
- CORBA servers deployed in user defined domain-specific-languages are illustrated in chapter 4 and also in the examples/corba/dsm-server example.
- CORBA Event/Notification publisher and subscriber applications with typed, structured, sequence and untyped events with or without filter(s) are discussed in chapter 5 and illustrated by the examples/corba/event example.
- OMG Data Distribution Service data reader and writer applications are discussed in chapter 6 and illustrated in the examples/corba/dds example.
- Assembling software defined radio (SDR) applications in various schemes, including JTRS-SCA core framework (CF) component model and JTRS-SCA application assembly schema, are discussed in appendix A and illustrated in the examples/corba/jtrs-sca, the examples/basic-ioc/sdr, and the examples/basic-ioc/dsm-sdr examples.

- Assembling robotic component applications in various schemes, including OMG-RTC component model, is discussed in appendix B and illustrated in the examples/corba/rtc and the examples/basic-ioc/robot-vehicle examples.

Although many of the CORBA application scenarios and common service usages shown above were traditionally challenging for average CORBA experts, they are straightforward and even obvious for beginners with PocoCapsule/CORBA. Besides, this document is more focused on CORBA application assembly, deployment, and configuration but less about their implementations. Therefore, instead of requiring expert level background on CORBA server object implementations and programming models of the services above, this document only assumes readers to have basic concept level comprehension on CORBA object model, such as what summarized in appendix C, and entry level experience of CORBA IDL and C++ mapping.

2. An Hello Example

In this chapter, a simple client/server example is used to illustrate the full CORBA application development cycle, especially the usage of PocoCapsule/CORBA framework in the deployment phase.

The server object in this example is designed and implemented as a plain old CORBA 2.x application. This immediately avoids those harmful and contrived complexities of CORBA 3.x and/or CCM techniques which have largely been abandoned by the mainstream software industry.

The source code of this example is available in the `examples/corba/hello` and `examples/corba/hello-tie` directories of the PocoCapsule/C++ installation.

2.1 Defining the object interface

In OMG CORBA object model, interfaces of objects are defined using the programming language neutral *interface define language* (IDL). For instance, the object interface of this example is defined as follows:

```
module sample {
    interface Greeting {
        string hello(in string msg);
    };
};
```

The IDL interface definition above is intuitive for average C++ and/or Java application developers. Conceptually, it could be understood as merely a language independent expression of the following C++ pure virtual class definition:

```
class sample {
public:
    class Greeting : public CORBA::Object {
public:
        virtual char* hello(const char* msg) = 0;
        ...
    };
};
```

In fact, IDL to C++ mapping utilities/tools indeed generate the client-side stub class `sample::Greeting` more or less similar to this conceptual mapping⁹.

⁹ An actual client stub mapping does not have to be a pure virtual class to avoid one layer of not that necessary abstraction.

2.2 Implementing the server object

The second development phase is to implement the “business logic” of this Greeting service object. There are two portable implementation styles standardized by OMG, namely the “*servant inheritance*” and the “*TIE delegation*”, and are referred to as “*servant*” and “*native*” respectively in PocoCapsule/CORBA.

In the “*servant*” style, the object’s server implementation (known as the servant) C++ class should support all operations defined in the IDL interface and should have the so-called POA server skeleton as its direct or indirect parent class. This POA server skeleton is generated from the user defined object IDL using the tool that comes with the underlying ORB (such as the `idl2cpp` of VisiBroker/C++ or `tao_idl` of TAO). In this example, the generated POA server skeleton C++ class is the `POA_sample::Greeting`. The “*servant*” implementation class `MyGreetingServantImpl` is therefore implemented, for instance, as follows:

```
class MyGreetingServantImpl : public POA_sample::Greeting
{
public:
    char* hello(const char* s) {
        printf("server received a %s\n", s);
        return CORBA::string_dup("hello greeting from server");
    }
};
```

In the “*native*” style, the object’s server implementation C++ class should support all operations defined in the IDL interface as in the case of “*servant*” style, however, not necessary to have the so-called POA server skeleton in the parent class hierarchy. In this example, a “*native*” implementation class `MyGreetingNativeImpl` is simply implemented as follows:

```
class MyGreetingNativeImpl // public POA_sample::Greeting
{
public:
    char* hello(const char* s) {
        printf("server received a %s\n", s);
        return CORBA::string_dup("hello greeting from server");
    }
};
```

The compiled binary object file(s) of the implementation above, either the servant or the native class, can be directly linked into the server application main executable later on, or can be linked into a standalone shared or dynamic loadable library and then linked or loaded by the main server executable.

Notably, the object interface and its implementations in either the servant or the native form are designed, implemented, and built as ordinary CORBA 2.x application components. Developers do not need to have knowledge or concern about PocoCapsule/CORBA and can assume the resulting component binary (shared/dll library or object file) is going to be used within a plain old CORBA 2.x runtime environment.

2.3 Declarative deployment descriptor

The third development phase is deployment, namely to enlist instances of server object implementation classes to the ORB and retrieve the object references or URLs for clients. In PocoCapsule/CORBA, this is done by writing a deployment descriptor that describes the desired deployment setup.

A PocoCapsule/CORBA deployment descriptor describes the structure or arrangement of a given CORBA application in a *declarative* style, to be distinct from the traditional *imperative* programming style. A *declarative* program describes “*what*” are the intended structures and dependencies without specifying procedures to construct them. An *imperative* program instructs “*how*” to proceed step by step in performing given operations.

Imperative programming style is familiar to most OO professionals and is widely used in component implementations and traditional manual application assembly and deployment. However, it is not the most expressive way to describe high level service or system level deployment compositions. Such compositions are best to be comprehended and expressed by their deployed structures rather than by their deployment procedures.

For the *servant* style implementation, the declarative deployment descriptor XML document¹⁰ looks like the following:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE corba-application-context
  SYSTEM "http://www.pocomatic.com/corba-application-context.dtd">

<?xml-transform type="text/xsl"
  href=http://www.pocomatic.com/corba2poco.xsl?>

<corba-application-context>
  <bean id="my-impl" class="MyGreetingServantImpl"/>

  <orb id="my-orb">
    <object uri="my-server"
      impl-ref="my-impl"/>
  </orb>

  <load library="./GreetingImpl.$dll"/>
  <load library="./reflx.$dll"/>
</corba-application-context>
```

Similarly, for the *native* (namely, the TIE delegation) style implementation, the deployment descriptor XML document looks like the following:

¹⁰ In many component frameworks, deployment descriptors have to be XML document files, with specific file names and/or extension names, and packaged inside ZIP files. PocoCapsule is much flexible and extendable on deployment descriptor formats and does not impose those unnecessary restrictions above. PocoCapsule/CORBA deployment descriptors can be expressed in either XML or pre-parsed encoded text (or even binary) formats and can be provided in forms of files (with arbitrary user specified names), embedded literal strings, LDAP entries, database records, requests or replies received over network connections, etc. It is merely for intuitiveness, deployment descriptors in this and all other out-of-the-box examples are created, edited, and used in form of XML documents.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE corba-application-context
  SYSTEM "http://www.pocomatic.com/corba-application-context.dtd">

<?xml-transform type="text/xsl"
  href="http://www.pocomatic.com/corba2poco.xsl?>

<corba-application-context>
  <bean id="my-impl" class="MyGreetingNativeImpl"/>

  <orb id="my-ORB">
    <object uri="my-server"
      impl-ref="my-impl"
      impl-type="native"
      interface="sample::Greeting"/>
  </orb>

  <load library="./GreetingImpl.$dll"/>
  <load library="./reflx.$dll"/>
</corba-application-context>

```

The *deployment descriptor*, either the first (for *servant* implementation) or the second (for *native* implementation), describes the following server components and setup:

- A server object implementation is declared as a singleton *<bean>* instance¹¹ of the class *MyGreetingServantImpl* or *MyGreetingNativeImpl*. By this declaration, this instance will be instantiated on demand using its constructor with no parameter, and can be referred within the application context via its *id* “my-impl”.
- An ORB instance, with *id* of “my-ORB”. By this declaration, this ORB instance will immediately be initiated as a singleton.
- Explicitly under this ORB, a CORBA object is declared (enlisted) uses the declared “my-impl” *servant* or *native* instance above as its implementation. The *uri* attribute of this object equals to “my-server” which will be used as the URI of its corbaloc URL. By this declaration, this CORBA object will immediately be activated as a singleton.
- When deploying the *native* style implementation, the *impl-type* attribute of *<object>* element should be specified as “native” (the default value of this attribute is “servant”).
- When deploying the *native* style implementation, the *interface* attribute of *<object>* element should be specified as the full scope name of the server IDL interface, namely the “sample::Greeting” in this example.

This example assumes all application modules are built as libraries and are dynamically loaded during the deployment¹². Therefore, this descriptor specifies:

¹¹ The *<bean>* element is defined in the poco-application-context.dtd. The lifecycle management of eager singleton beans is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

¹² PocoCapsule/C++ container will always load declared libraries at the beginning of the deployment of a application context. See the document *PocoCapsule/C++ IoC&DSM Developer Guide* for details.

- The container should load listed dynamic modules (libraries) before deploying the described application context.

The first library is *GreetingImpl.\$dll* in the local directory. The extension name “*\$dll*” will be substituted as “*dll*” on Windows and “*so*” or “*sl*” on Unix. The *GreetingImpl.\$dll* library contains the compiled binary of *MyGreetingServantImpl* or *MyGreetingNativeImpl* class (or both). The second library *reflx.\$dll* contains the compiled binary of dynamic invocation proxies for container to perform “*inversion of control*” on target component implementations (see next section).

Also, in this PocoCapsule/CORBA deployment descriptor, a DOCTYPE of DTD and a XML processing instruction referring to a XSLT style sheet are specified. These details will be discussed in the next chapter.

2.4 Dynamic invocation proxies

The PocoCapsule/C++ core framework and the server component implementations above in form of plain old C++ object (POCO) classes are developed independently without knowing each other. However, when deploying these implementations, the core framework has to invoke operations defined on these POCO classes, such as its constructor. This is achieved in PocoCapsule/C++ IoC framework through dynamic invocation proxies.

A unique technique of PocoCapsule/C++ IoC framework is to generate dynamic invocation proxies from deployment descriptors. This scenario is opposite to “reflection” and therefore is referred to as “*projection*”. Unlike CASE tools or MDA code generators, PocoCapsule/C++ only generates the invocation proxies implied by metadata of deployment descriptors rather than the procedure code with actual invocation parameters to be performed by these descriptors. Hence, the proxy code does not need to be regenerated under method argument value changes of the deployment descriptor.

The utility tool in PocoCapsule/C++ for this “projection” code generation is *pxgenproxy*. In this example, the proxy code is generated from the descriptor file (happens to be named as *setup.xml* here) as follows:

```
% pxgenproxy setup.xml -h=hello.h
```

The generated source code of proxies will be written to a file (happens to be *setup_reflx.cc* in this example). More detailed description on *pxgenproxy* utility can be found in “*PocoCapsule/C++ IoC & DSM Framework Developer Guide*”¹³.

Unlike CASE, MDA, and IDL that either require application developers to manually insert application code into the generated code, or require application code to extend and

¹³ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

implement or invoke generated interfaces/stubs, the PocoCapsule/C++ generated code are completely unknown to applications and largely transparent to application developers.

Application developers only need to look into the generated proxy code if an operation invocation signature mismatch error was reported during its compilation, which is most likely caused by bugs inside the user supplied deployment descriptor. However, the PocoCapsule/CORBA framework's high level domain specific declarative schema is able to catch most of such low-level errors by the schema validation. Therefore, with PocoCapsule/CORBA, application developers rarely need to look into generated code and can simply compile and build it into a shared/dynamic library or directly link it with the application main executable. In this example, the shared library approach is used and the library is named as `reflx.so` (or `reflx.dll` on windows).

2.5 The Container Server

The Greeting object server object implementation should be deployed, with the deployment descriptor, to a runtime environment, also known as a container. In PocoCapsule and most other IoC containers, the term “container” merely refers to a application context factory object, which can be embedded in any C++ or Java applications. In this example with PocoCapsule/C++ IoC, this factory is embedded in a tiny C++ `main()` function as follows:

```
#include <pocoapp.h>
...

int main(int argc, char** argv)
{
    POCO_AppContext::initDefaultAppEnv(argc, argv);

    // create the server context representation
    POCO_AppContext* ctxt
        = POCO_AppContext::create("setup.xml", "file");

    // establish the server structure
    ctxt->initSingletons();

    // start the ORB runner
    printf("Server is ready, with URL:\n");
    ctxt->getBean("pocomatic.dispatcher:my-orb");

    return 0;
}
```

This mini container is application independent. Therefore, it is used by many other examples in the PocoCapsule/C++ installation, and can be used by real production applications as well. The `POCO_AppContext` class in the code is documented in “*PocoCapsule/C++ IoC & DSM framework developer guide*”¹⁴. In a brief description, the code above in this main executable does:

¹⁴ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

- The `initDefaultAppEnv()` passes the command line arguments to an underlying POCO_AppEnv singleton instance. These argument will then be passed to the PocoCapsule/C++ engine as well as `ORB_init()`.
- An application context representation is created from the given assembly/deployment descriptor (happens to be named as `setup.xml` in this example).
- Establish the server structure from the context by instantiating all eager singleton components (see PocoCapsule/C++ IoC & DSM). This includes declared ORB instance(s) and all declared CORBA objects.
- Then, to start the ORB request dispatch loop and print out all published corbaloc URLs, the server retrieves the reference of a lazy instantiation bean of `id` equals to `"pocomatic.dispatcher:user-declared-orb-id"` (in this example, the user declared orb id is `"my-orb"`) from the context. This effectively invokes a function implicitly declared as this bean's factory. As to be described in more detail, URLs are automatically published by PocoCapsule/CORBA if an `<object>` element is declared with a specified `uri` attribute. The value of this `uri` attribute will be used as the URI of the published URL.

2.6 Playing this example

The first thing to do in playing this example is to start the server. As said above, the server will print out the URL of the server object with the URI (i.e. `"my-server"`) specified in deployment descriptor `setup.xml`:

```
% server
Server is ready, with URL:
corbaloc::192.168.2.2:2809/my-server
```

To see what objects and methods are inversely called back by the container behind the scene, one can restart the server with the `"-Dpococapsule.trace.enable=true"` command line option. This command line argument will then be passed to the PocoCapsule/C++ engine through the application context to turn on the tracing option. The trace messages are printed on the stderr, as highlighted in the following result:

```
% server -Dpococapsule.trace.enable=true
PCO_CORBAHelper::ORB_init(...) = (CORBA::ORB_ptr) (0x86daa50)
((CORBA::ORB_ptr) 0x86daa50)->resolve_initial_references(...)
= (CORBA::Object_ptr) (0x8700494)
PortableServer::POA::_narrow(...) = (PortableServer::POA_ptr) 0x86ffc48
CORBA::release((CORBA::Object_ptr) (0x8700494))
((PortableServer::POA_ptr) 0x86ffc48)->the_POAManager()->activate()
new MyGreetingServantImpl() = (MyGreetingServantImpl*) 0x8707090
PCO_CORBAHelper::export_object(...) = (CORBA::Object_ptr) 0x87073ac
Server is ready, with URL:
PCO_CORBAHelper::run(...)
corbaloc::192.168.2.2:2809/my-server
```

The client invokes the *hello()* method of this server object by calling the method on its stub. This stub is a *sample::Greeting* C++ object, and can be obtained from the ORB based on the known server object URL as in the following code:

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::String_var url = (const char*)"corbaloc::192.168.2.2:2809/my-server

// destringfy the url into a client stub
sample::Greeting_var stub =
    sample::Greeting::_narrow(orb->string_to_object(url.in()));

// make an invocation on the stub
CORBA::String_var reply = stub->hello("hello from client");

// print out what returned from the server
printf("client receives %s\n", reply.in());
```

2.7 Changing Server Structure/Configuration

With the PocoCapsule/CORBA declarative deployment, a server application can easily be restructured without rewriting/modifying a single line of code. Assuming the developer wants the server to have the following setting:

- To process requests in *single thread*, rather than concurrently,
- To have a *persistent lifespan* (the meaning of this will be described in a later chapter).

This setting can easily and intuitively be declared in the following deployment description (using the servant style implementation for instance):

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE corba-application-context
  SYSTEM "http://www.pocomatic.com/corba-application-context.dtd">

<?xml-transform type="text/xsl"
  href="http://www.pocomatic.com/corba2poco.xsl?>

<corba-application-context>
  <load library="./GreetingImpl.$dll"/>
  <load library="./reflx.$dll"/>

  <bean id="my-impl" class="MyGreetingServantImpl"/>

  <orb id="my-orb">
    <poa>
      <policies>
        <policy name="thread" value="single"/>
        <policy name="lifespan" value="persistent"/>
      </policies>

      <object uri="my-server" impl-ref="my-impl"/>
    </poa>
  </orb>
</corba-application-context>
```

The differences to the previous descriptor are highlighted. The Greeting object was previously declared to be activated (enlisted) on the (hidden) default object adapter (therefore, use default policies) is now declared to be activated (enlisted) on an object adapter (POA) with the user specified thread and lifespan policies.

This structure change implies additional dynamic invocation signatures for deployment, therefore, requires regenerate their proxies using *pxgenproxy*. After regenerating and rebuilding the reflection dynamic invocation proxy library *reflx.so* (or *.dll* on windows), this new server structure can be instantiated by simply restarting the server with the new deployment descriptor, without having to rebuild the server binary.

<this is a empty page>

3. Simple Server Applications

3.1 DSM and PocoCapsule/CORBA

The PocoCapsule/C++ IoC&DSM core framework¹⁵ is neutral to all component models. Therefore, it inherently supports plain old C++ native objects as well as plain old CORBA servant objects as components. However, although being very simple, straightforward, and generally applicable, the IoC schema of this core framework is primarily explicit descriptions of low level API invocations in XML, and therefore is verbose, not expressive, and even error prone in large deployment¹⁶.

Instead of defining a bloat schema that meets all domain requirements or ignoring high level domain requirements and forcing developers to use a rigid low level schema, the PocoCapsule/C++ IoC&DSM framework define a very simple but customizable core schema. This allows users or third parties to define their own *domain specific modelings* (DSM) as extensions of the core deployment model. PocoCapsule/CORBA is such a DSM extension that adds several new deployment elements specific to CORBA server, Event/Notification, and even DDS and RTC applications.

Different from other schema extension solutions provided in other IoC frameworks¹⁷, the DSM support in PocoCapsule/C++ is declarative and based on ubiquitous, standardized, and portable technique that is completely free of any proprietary plug-in API and XML DOM/SAX model programming. As described in more detail in the *PocoCapsule/C++ IoC&DSM Developer Guide*¹⁸, a DSM in PocoCapsule/C++ IoC&DSM is defined by its document type definition (DTD) and its schema transformation templates XSLT style sheet.

For PocoCapsule/CORBA, the DSM deployment schema is defined in the *corba-application-context.dtd*, and transformation templates are declared in the *corba2poco.xsl* XSLT style sheet. A PocoCapsule/CORBA application deployment descriptor should always:

- declare DOCTYPE corba-application-context with the system id:
 “*http://www.pocomatic.com/corba-application-context.dtd*”
- declare transform process instruction with href equals to:
 “*http://www.pocomatic.com/corba2poco.xsl*”
- use `<corba-application-context>` as the document node.

¹⁵ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

¹⁶ This is one example of the so-called “XML Hell”.

¹⁷ Such as the “Extensible XML Authoring” of Spring Framework 2.0.

¹⁸ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

This is illustrated by the following PocoCapsule/CORBA application deployment descriptor skeleton:

```
<?xml ...?>
<!DOCTYPE corba-application-context
  SYSTEM http://www.pocomatic.com/corba-application-context.dtd>

<?xml-transform
  type="text/xsl"
  href="http://www.pocomatic.com/corba2poco.xsl"?>

<corba-application-context>
...
</corba-application-context>
```

Note: both the schema DTD file and the transforming XSLT style sheet mentioned above are built in the PocoCapsule/CORBA framework. Therefore, they are always resolved from the PocoCapsule/CORBA local installation, unless the descriptor documents are opened by a XML document viewer (such as a web browser).

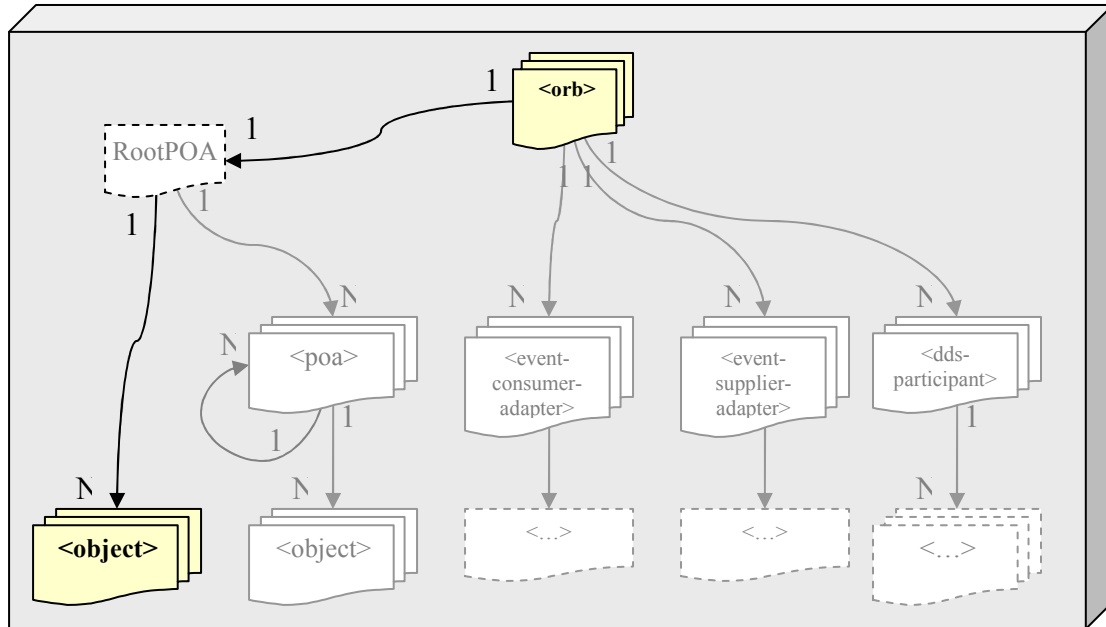
Also, PocoCapsule/CORBA encapsulates some of its implementations within an ORB adapting library (or libraries for different ORBs). This ORB adapting library supports a portable layer to the underlying ORB, the URL manage agent, the high level interface to POA, as well as to Event/Notification Service, etc. The generated proxy code will invoke methods on these POCOs, and therefore, should be linked with such a library for the specific ORB.

In the current release, two ORBs are supported, namely VisiBroker and TAO. Their ORB adapting libraries are listed below:

<i>Library Name (unix)</i>	<i>Library Name (Windows)</i>	<i>Usage</i>
libpoco2vb.so	poco2vb.dll	Used by dynamic invocation proxies to do IoC on VisiBroker ORB (and POA). Also, a corbaloc URL manager (agent) is embedded.
libpoco2tao.so	poco2tao.dll	Used by dynamic invocation proxies to do IoC on TAO ORB (and POA). Also, a corbaloc URL manager (agent) is embedded.

3.2 PocoCapsule/CORBA Schema Overview

A CORBA application utilizes functions of an underlying embedded CORBA ORB engine. Programmatically, functions of this ORB are partitioned in an ORB object and other programmatic artifacts as child or grandchild objects. These artifacts are explicitly or implicitly created from the ORB, such as activated CORBA object, object references, CORBA object adapters, connections or subscriptions to OMG Event/Notification services (supplier or consumer adapters). Therefore, conceptually, CORBA application structures can be characterized as an “upside down” tree rooted from an ORB instance, as shown in the following diagram:



In conventional programs, or equivalently in primitive IoC descriptions, this application tree structure is flattened and buried in the wiring logic, not intuitively visible from the code structure or declaration document structure. To wire-up this structure, one also has to explicitly know and use factory API on each node.

The PocoCapsule/CORBA models this application tree structure explicitly in its deployment description document structure, as follows:

- PocoCapsule extends the primitive PocoCapsule/C++ IoC&DSM core schema with several new XML elements, such as `<orb>`, `<object>`, `<poa>`, `<event-consumer-adapter>`, `<event-supplier-adapter>`, and `<dds-participant>`, to describe artifacts in CORBA applications
- A XML document with elements above as nodes conceptually describes a CORBA application structure. An XML element in this document denotes a programmatic object created from the programmatic object represented by its parent element in the actual CORBA application.
- Tag names of a parent and its child CORBA element implicitly determine the object instantiation/creation method to be used by the container.
- Furthermore, the attributes and child elements implicitly determine arguments of the instantiation/creation method.

A PocoCapsule/CORBA deployment descriptor does not explicitly describe how to wire-up the CORBA application structure, but models the structure. This not only makes CORBA application structures become intuitively visible, but also has their detail wiring operations completely concealed in the framework. This largely reduces the possibility of wrong method name and invalid input parameters in performing the corresponded underlying function calls and also largely eliminates the learning curve of system level

APIs of CORBA ORB, POA, Event/Notification Service, and DDS, for domain/business logic focused developers.

In this chapter and the next few chapters, the PocoCapsule/CORBA application deployment schema is to be explored in the following order:

- This chapter introduces the `<orb>` and `<object>` elements, and discusses the simple CORBA server model and URLs of CORBA objects.
- The Chapter 4 discusses `<poa>` element, as well as various sophisticated CORBA server models.
- The chapter 5 discusses the `<event-consumer-adapter>` and `<event-supplier-adapter>` elements, as well as integration with the OMG Event/Notification service..
- The Chapter 6 discusses the `<dds-participant>` element.
- The Chapter 7 discusses JTRS-SCA and OMG-RTC application deployment models.

3.3 The `<orb>` element

An `<orb>` element declares a root of a CORBA ORB application structure. All other CORBA elements are defined as its children or grandchildren. The `<orb>` element is formally defined in the *corba-application-context.dtd* as follows:

```
<!ELEMENT orb (args?,
              (object|poa|
               event-consumer-adapter|event-supplier-adapter|
               dds-participant)*)>

<!ATTLIST orb id ID #IMPLIED>

<!ELEMENT args (arg)*>
<!ELEMENT arg EMPTY>
<!ATTLIST arg value CDATA #REQUIRED>
```

Child elements `<object>`, `<poa>`, `<event-consumer-adapter>`, `<event-supplier-adapter>`, and `<dds-participant>` of `<orb>` further declare fine grained substructures or/and components of the CORBA application.

An `<orb>` element is transformed to an eager singleton `<bean>` of the PocoCapsule/C++ core schema¹⁹ with the same *id* attribute. Therefore, if the *id* attribute is not omitted, this bean can be retrieved from the application context using the *getBean()* method (see *PocoCapsule/C++ IoC&DSM programmer guide*) and the return type is `CORBA::ORB_ptr`.

An `<orb>` element also implies two hidden beans in the transformed result. The first implied bean is the root POA bean, which will be used to activate objects for simple applications, as well as used as the factory bean for creating other POA beans declared by users. The second implied bean is the ORB dispatcher. The constructor of this bean prints

¹⁹ The `<bean>` element is defined in the *poco-application-context.dtd*. The lifecycle management of eager singleton beans is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

out all declared CORBA object URLs (to be described later) and then blocks itself on the request dispatching loop until the ORB shutdown. The id of the ORB runner bean is *pocomatic.dispatcher.<the-orb's-id>*.

The *<args>* child element of *<orb>* specifies an *argv* list for initializing the declared ORB. For instance, the ORB declared in the following XML segment will be initialized with *argc=2* and *argv[] = {"-ORBInitRef", "ABC="corbaloc::192.168.2.1:1234/abc"}*.

```

...
<orb>
  <args>
    <arg value="-ORBInitRef"/>
    <arg value="ABC=corbaloc::192.168.2.1:1234/abc"/>
  </args>
  ...
</orb>

```

If this *<orb>* element is declared without *<args>* subnode, then the *argv* list set on the environment object used by the application context will be used to initialize the ORB. For instance:

```

int main(int argc, char** argv)
{
    POCO_AppEnv env =
        POCO_AppContext::initDefaultAppEvent(argc, argv);
    ...
}

```

or equivalently the:

```

int main(int argc, char** argv)
{
    POCO_AppEnv env =
        POCO_AppContext::getDefaultAppEvent();
    env->setArray("pococapsule.init.argv", argc, argv);
    ...
}

```

will have the *main()* command line *argv* list passed to the ORB initialization method, if the *env* variable is used to create the application context (see *PocoCapsule/C++ IoC&DSM programmer guide*).

3.4 The *<object>* element

The *<object>* element declares a CORBA object. Simple CORBA applications can use this element as children of an *<orb>* element to declare simple CORBA server objects that will be manipulated with default policies (server object manipulating policies will be discussed in the next chapter). The *<object>* element is formally defined as follows:

```

<!ELEMENT object (bean|ref)?>
<!ATTLIST object impl-ref IDREF #IMPLIED>
<!ATTLIST object impl-type (servant|native) "servant">
<!ATTLIST object interface CDATA #IMPLIED>
<!ATTLIST object id ID #IMPLIED>
<!ATTLIST object oid CDATA #IMPLIED>
<!ATTLIST object uri CDATA #IMPLIED>

<!-- <bean> and <ref> are elements defined in the core schema -->

```

An *<object>* is also transformed to an eager singleton *<bean>* of the PocoCapsule/C++ core schema²⁰. This *<bean>* instance can be retrieved from the application context using its declared *id*. The type of the returned pointer is *CORBA::Object_ptr*.

Attributes and child elements of the *<object>* element are defined in the following sections.

3.4.1 *<bean>* or *<ref>* child element of *<object>*

An *<object>* bean can optionally have a *<bean>* or *<ref>* child element that declares the implementation instance of the object. For example, the following XML segment declares a CORBA server object to be activated using an instance of *MyServantImpl* as servant.

```

<orb>
  <object id="my-server">
    <bean class="MyServantImpl"/>
  </object>
</orb>

```

The following XML segment is equivalent to the above one:

```

<orb>
  <object id="my-server">
    <ref bean="my-servant-impl"/>
  </object>
</orb>

<bean id="my-servant-impl" class="MyServantImpl"/>

```

3.4.2 *impl-ref* attribute of *<object>*

The *impl-ref* attribute of *<object>* is a short cut alternative of the *<ref>* subnode of an *<object>*. For instance, the following XML segment is equivalent to the previous two declarations:

²⁰ The *<bean>* element is defined in the *poco-application-context.dtd*. The lifecycle management of eager singleton beans is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

```
<orb>
  <object id="my-server" impl-ref="my-servant-impl"/>
</orb>

<bean id="my-servant-impl" class="MyServantImpl"/>
```

Note: The *<bean>* and the *<ref>* elements in the two examples are elements defined in the PocoCapsule/C++ core framework and documented in the “*PocoCapsule/C++ IoC&DSM Developer Guide*”²¹.

3.4.3 *impl-type* and *interface* attributes of *<object>*

By default, implementations of CORBA objects are C++ classes inherited from POA skeletons generated from their respective IDL interfaces. Because the common superclass of all POA skeletons is the *PortableServer::ServantBase*, therefore, implementations of this category are referred to as “*servant*” classes in PocoCapsule/CORBA.

CORBA also supports an alternative approach that does not require implementations to subclass from POA skeletons but only need to implement all methods declared in their respective IDL interfaces. Implementations of this category are referred to as “*native*” classes in PocoCapsule/CORBA.

The *impl-type* attribute of an *<object>* element specifies the category of the given implementation bean. The default value of this attribute is “*servant*”. To use a native implementation, this value should be set to “*native*”.

The *interface* attribute specifies the full scope name of the IDL interface. This attribute is ignored for “*servant*” implementation but mandatory for “*native*” implementation.

The following XML segment declares a simple server object of the *native* implementation category:

```
<orb>
  <object id="my-server"
    impl-ref="my-native-impl"
    impl-type="native"
    interface="sample::Foo"/>
</orb>

<bean id="my-native-impl" class="MyNativeImpl"/>
```

²¹ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

3.4.4 *id* attributes of `<object>`

An `<object>` element is transformed to a POCO bean with the same *id* attribute. This POCO bean is a CORBA object reference. If the *id* attribute is specified in the `<object>` element, this bean, namely the CORBA object reference, can be retrieved from the application context using `getBean()` method and safely casted to `CORBA::Object_ptr` (see *PocoCapsule/C++ IoC&DSM programmer guide*).

For example the CORBA object reference of the CORBA server object in the example above can be retrieved as follows:

```
...
POCO_AppContext* ctxt = POCO_AppContext::create("setup.xml", "file");

// retrieve the object reference from the application context
CORBA::Object_ptr ref = (CORBA::Object_ptr) ctxt->getBean("my-server");
...
```

3.4.5 *oid* attribute of `<object>`

The *oid* attribute of `<object>` declares the user specified object id to be used by the ORB engine in creating the object reference as well as activating the server object. If this attribute is omitted, the ORB will try to use an id generated by itself (referred to as “*system ID*”).

Note: A simple application declaring an `<object>` directly under `<orb>` node implies “*system ID*”. Therefore, users should not specify the *oid* attribute for that `<object>` element.

3.4.6 *uri* attribute of `<object>`

The *uri* attribute specifies the URI (a.k.a key string) of this object's corbaloc URL. If this attribute is omitted, no URL for this object will be generated.

For instance, the following XML segment declares a server object to be activated and also specifies the URI of its corbaloc URL to be “**my-server**”:

```
<orb>
  <object uri="my-server">
    <bean class="MyServerImpl"/>
  </object>
</orb>
```

The corbaloc URL of the object declared above will be:

```
corbaloc::agent-ip:agent-port/my-server
```

Here, the *agent-ip* and the *agent-port* are the IP address (or host name) and TCP port number of the embedded corbaloc agent.

By default, PocoCapsule/CORBA will use the primary IP address of the host. In the case of a host has multiple IP addresses, an alternative one from them can be specified using the “*-Dcom.pocomatic.URLAgentHost=agent_ip*” command line option as in the next example.

Also, the default agent port is 2809, as specified by OMG. An alternative port number can also be specified using the “*-Dcom.pocomatic.URLAgentPort=agent_port*” command line option as in the following example.

```
% server -Dcom.pocomatic.URLAgentHost=192.168.2.3 \  
-Dcom.pocomatic.URLAgentPort=12345
```

Here, the command line argv and argc are assumed to be passed to the environment used in creating the application context (see examples in examples/corba directory and *PocoCapsule/C++ IoC&DSM Developer Guide*).

A corbaloc URL can explicitly be used to generate an object reference of its server object by a remote client application, for instance, as follows (assuming the <host> is 192.168.2.2, and the port is 2809):

```
...  
CORBA::Object_var ref  
= orb->string_to_object("corbaloc::192.168.2.2:2809/my-server");
```

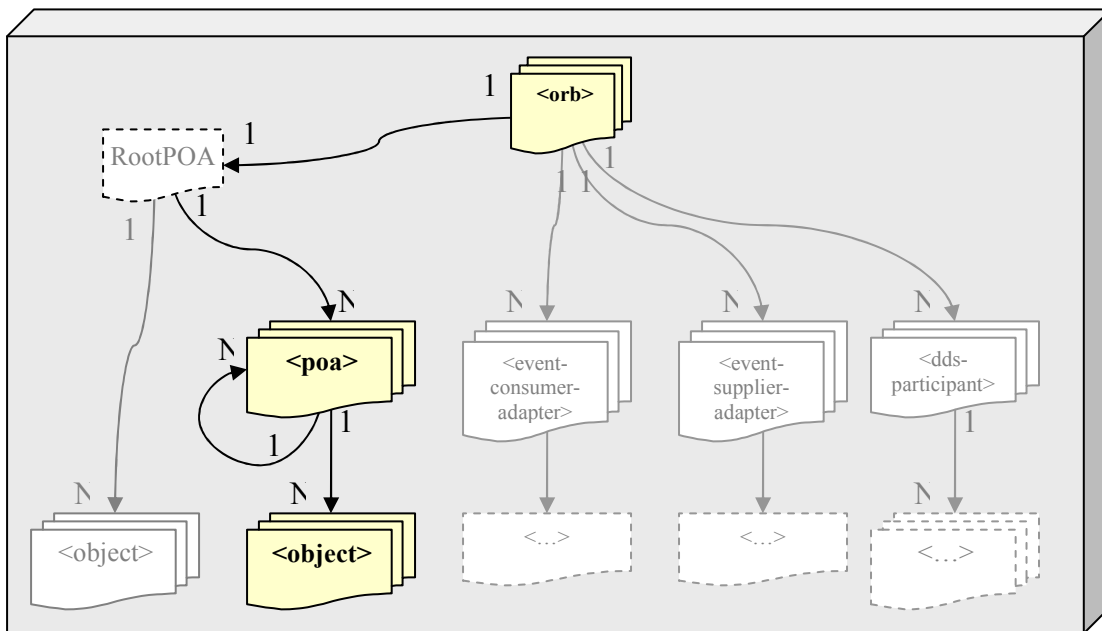
If an <orb> element is declared with an *id* attribute, an ORB dispatcher object will be generated by the schema transformer. The *id* of this ORB dispatcher bean automatically becomes *pocomatic.dispatcher.<the-orb's-id>*. This bean is designed not to be initialized until it is demanded. Therefore, using the *getBean()* to retrieve this bean from the application context will cause it to be instantiated. The constructor of this bean prints out all declared CORBA object URLs (to be described later) and then blocks itself on the request dispatching loop until the ORB shuts down.

<this is an empty page>

4. POA Server Applications

4.1 POA and POA Policies

Simple application server objects directly declared under an `<orb>` element will be activated on an internal hidden object adapter (the root POA) using default policies. Sophisticated server models and objects that intend to use alternative policies need to be deployed on user explicitly declared *object adapter* known as *portable object adaptor* (POA) with user specified policies. Such a server structure with ORB, POA and objects is illustrated in the following diagram:



In PocoCapsule/CORBA, a POA is simply declared as a `<poa>` element with a `<policies>` child element listing policies of the POA as `<policy>` elements, as in the following example:

```

...
<orb>
  <poa>
    <policies>
      <policy name="..." value="..." />
      ...
    </policies>
    ...
  </poa>
  ...
</orb>
...

```

PocoCapsule/CORBA supports all OMG standardized POA policies summarized in the following table:

<i>The name attribute of <policy></i>	<i>The value attribute of <policy> (and its implied value)</i>	<i>Semantic meaning</i>	<i>Useful scenarios</i>
<i>“lifespan”</i>	<i>“transient”</i> (implied, unless specified elsewhere)	Object references (IORs) from a transient <poa> are time stamped. They will become invalid after server restart, even if the server reuses the original transport endpoint (IP and port), name , and object oid attributes.	Using “transient” policy can prevent clients from rebinding to a new server object whose transport address and internal adapter name and object id coincident to a previous (already deactivated) object.
	<i>“persistent”</i>	Object references (IORs) from a persistent <poa> are not time stamped. They will remain valid after server restart, if the server is restarted at the original transport endpoint (IP and port), and reuse the same name , and the object oid attributes.	Using “persistent” policy allows clients to be able to rebind to the object after its server restart on the same transport address and the object is reactivated on the same adapter with same oid attribute.
<i>“request-processing”</i>	<i>“use-aom-only”</i> (implied)	On dispatching a request to its servant implementation instance, the POA only uses Activated Object Map (AOM) to resolve the object id (oid) to servant instance mapping.	Simple and straightforward, but may not scale and sufficient in sophisticated cases.
	<i>“use-servant-manager”</i>	On dispatching a request to its servant implementation instance, the POA uses AOM to resolve the object id (oid) to servant instance mapping (if the servant-retention policy is “retain”). If this fails to resolve the servant instance, the POA will use the user specified servant manager to resolve the servant. <i>A servant manager should be injected to this POA, using the set_servant_manager property setter.</i>	<ul style="list-style-type: none"> • To support object activation on demand. • To implement application manipulated object pool mapping. • To do object location forward.
	<i>“use-default-servant”</i>	On dispatching a request to its servant implementation instance, the POA uses AOM to resolve the object id (oid) to servant instance mapping. If this fails to resolve the servant instance, the POA will dispatch this request to a servant (i.e. the default servant) specified by the developer.	<ul style="list-style-type: none"> • To process requests based on information encapsulated in object ids. • To implement generic agent. <p><i>A default servant should be injected to this POA using the set_servant property setter</i></p>

"servant-retention"	"retain" (implied)	Objects, explicitly activated by application or implicitly by servant manager are retained by this POA in AOM, and will be used for subsequent invocations.	
	"non-retain"	An object to servant mapping only applies to current invocation, and not retained by POA.	Usually combine with "use-servant-manager" request-processing policy (servant-locator). <i>An <object> element under this <poa> should not have <bean>, <ref> sub-element or impl-ref attribute to specifies its implementation..</i>
"thread"	"orb-control" (implied)	ORB specified multi-thread mode, such as thread-pool or thread per-client-connection.	
	"single"	All requests to objects of this POA are serialized.	Mainly to support thread-unsafe implementations
	"main-thread"	Similar to "single" thread mode, except all request are handled by the same thread.	If the thread that has a previous set properties (thread local or a big stack size etc.) to be used for all requests.
"id-assignment"	"system-id" (implied)	The POA will implicitly generate object ids, if not specified by the application on activating object or creating object references. <i>The oid attribute of a <object> under this <poa> should not be specified.</i>	<ul style="list-style-type: none"> Usually for transient objects.
	"user-id"	Applications must explicitly specify object ids on activating object or creating object references.	<ul style="list-style-type: none"> Usually for persistent objects, or, To encapsulate information or marks in object id. <i>The oid attribute of a <object> under this <poa> must be specified.</i>
"id-uniqueness"	"unique-id" (implied, if request-processing policy is not "use-default-servant").	Object id and servant is 1-to-1 mapped.	<i>This policy should not be combined with "use-default-servant" request-processing policy.</i>

	“multiple-id” (implied, if request- processing policy is “use- default-servant”	Object id and servant can be N- to-1 mapped. Usually, this is useful with default servant.	This policy should be combined with <i>use-default- servant request-processing</i> policy.
“implicit- activation”	“no-implicit” (implied)	To prohibit implicit activation.	
	“implicit”	Allowing implicitly activated object on default POA.	

4.2 The <poa> element

In PocoCapsule/CORBA a POA is declared using a <poa> element. The schema definition of this element is as follows:

```

<!ELEMENT poa          (policies?, (property|ioc)*, (object|poa)*)>
<!ATTLIST poa         id      ID      #IMPLIED>
<!ATTLIST poa         name    CDATA   #IMPLIED>

<!ELEMENT policies    (policy*)>
<!ELEMENT policy      EMPTY>
<!ATTLIST policy      name    CDATA   #REQUIRED>
<!ATTLIST policy      value   CDATA   #REQUIRED>

<!ELEMENT property    (bean|ref|array)?>
<!ATTLIST property    name    CDATA   #REQUIRED>
<!ATTLIST property    type    (%type;|array) "bean">
<!ATTLIST property    class   CDATA   #IMPLIED>
<!ATTLIST property    ref     IDREF   #IMPLIED>
<!ATTLIST property    value   CDATA   #IMPLIED>
<!ATTLIST property    pass    (%syntax;) #IMPLIED>

<!-- <ioc> is element defined in PocoCapsule core schema -->

```

A <poa> element is also transformed to a <bean> element of the PocoCapsule/C++ core schema²². This bean is retrievable from application context using declared id. The type of return value is *PortableServer::POA_ptr*.

Child elements and attributes of <poa> are defined in the following sections.

4.2.1 <policies> element

The optional <policies> child element of a <poa> declares the set of policies of the object adapter. The valid name-value pairs of POA policies are listed in the POA policy table above, for instance:

²² The <bean> element is defined in the poco-application-context.dtd and is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

```

<!--
declare a POA, using default servant and persistent lifespan
-->
<poa>
  <policies>
    <policy name="request-processing" value="use-default-servant"/>
    <policy name="lifespan" value="persistent"/>
  </policies>

  <!-- create a URI -->
  <object uri="my-server-use-default-servant"/>
</poa>

```

4.2.2 <property> element

<property> element merely declares a single parameter post-instantiation setter IoC method (see *PocoCapsule/C++ IoC&DSM Developer Guide*²³) on the <poa> element. For instance:

```

<!--
declare a POA, using default servant and persistent lifespan
-->
<poa>
  <policies>
    <policy name="request-processing"
      value="use-default-servant"/>
  </policies>

  <!--
      inject the default servant to this POA using the
      set_servant() setter injection.
  -->
  <property name="set_servant">
    <bean class="GreetingImpl"/>
  </property>

  <!-- create a URI -->
  <object uri="my-server-use-default-servant"/>
</poa>

```

This is equivalent to the following expression:

²³ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

```

<!--
declare a POA, using default servant and persistent lifespan
-->
<poa>
  <policies>
    <policy name="request-processing"
           value="use-default-servant"/>
  </policies>

  <!--
           inject the default servant to this POA using the
           set_servant() setter injection.
  -->
  <ioc method="set_servant">
    <method-arg><bean class="GreetingImpl"/></method-arg>
  </ioc>

  <!-- create a URI -->
  <object uri="my-server-use-default-servant"/>
</poa>

```

4.2.3 <object> element

<object> can be used as a child elements of an <orb> as well as a <poa>. The schema of this element has been defined in the section 3.4. When an <object> element is used under a <poa>, the corresponding object reference or object activation will be created or activated on the user declared POA.

4.2.4 id attribute

A <poa> element is transformed to a <bean> element of PocoCapsule/C++ core schema²⁴ with the same *id* attribute, and the class attribute of this POCO bean is *PortableServer::POA* object. If the *id* attribute is specified in the <object> element, this bean, namely the POA reference, can be retrieved from the application context using *getBean()* method.

4.2.5 name attribute

This optional attribute specifies a name to be assigned to this POA. If this attribute is not specified, PocoCapsule/CORBA will automatically generate one.

4.3 The POA Request Processing Policy

The CORBA POA server model is largely determined by the POA request processing policy, which specifies how ORB runtime engine resolves the servant implementation for dispatching each received request.

A POA's request processing policy is specified in the <policies> child element of the <poa> element. The three valid values of this are:

²⁴ The <bean> element is defined in the poco-application-context.dtd and is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

- *use-aom-only*: Use the activate object map only
- *use-default-servant*: Use default servant
- *use-servant-manager* (retain): Use servant manager (activator)
- *use-servant-manager* (non-retain): Use servant manager (locator)

This section presents more examples and descriptions on them.

4.3.1 Use Activated Object Map Only

This is the default policy setting of POAs. A POA with this setting only uses the *active object map* to resolve the servant for dispatching a received request. Therefore, objects on this POA should be activated explicitly, as in the following example:

```
<!--
    a POA, using AOM only (this is also the default setting)
-->
<poa>
  <!--
    This is actually same as the default policy setting.
    Therefore, it is optional to explicitly set it here.
  -->
  <!--
  <policies>
    <policy name="request-processing" value="use-aom-only"/>
  </policies>
  -->

  <!-- explicitly activate an object -->
  <object uri="my-server-use-aom-only">
    <bean class="GreetingImpl">
  </object>
  ...
</poa>
```

4.3.2 Use Default Servant

Under this setting, it is optional to activate object explicitly. A default servant can be injected to the POA. If this POA receives a request to an object not being found in its *active object map*, it will dispatch the request to the default servant.

In the following example, a default servant is set up on the POA, and no object is explicitly activated. An object reference is created from that POA. When a client invokes a request on that reference, the request will be dispatched to the default servant.

```

<!--
    a POA, using default servant
-->
<poa>
  <policies>
    <policy name="request-processing" value="use-default-servant"/>
  </policies>

  <!-- inject the default servant -->
  <property name="set_servant">
    <bean class="GreetingImpl">
  </property>

  <!-- create a URI, but not activate the object -->
  <object uri="my-server-use-default-servant"/>
</poa>

```

4.3.3 Use Servant Manager (Activator)

Under this setting, it is optional to activate object explicitly on this POA. A servant activator should be injected to the POA. This servant activator should be a bean instance of class extended from *PortableServer::ServantActivator* and implement the *incarnate()* and *etherealize()* operations of its abstract parent, as in the following example:

```

class ServantActivatorImpl : public PortableServer::ServantActivator
{
public:
    PortableServer::Servant incarnate(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr      poa);

    void etherealize(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr      poa,
        PortableServer::Servant      servant,
        CORBA::Boolean               cleanup_in_progress,
        CORBA::Boolean               remaining_activations);
};

```

When this POA receives a request to an object that is not activated, the POA will call this servant activator's *incarnate()* operation, and use the returned servant to activate the object. Then, the POA will dispatch the request to the object's servant. The activated object will remain in the *active object map* until explicitly deactivated programmatically (some ORBs have a value added feature to deactivate idled objects automatically). The *etherealize()* operation will be invoked on this object's deactivation.

In the following example, a servant activator is injected to the POA, and no object is explicitly activated. An object reference is created from that POA. The servant activator's *incarnate()* will be invoked *at the first time* a client invokes a method on this reference, and the *etherealize()* will be invoked when the object is deactivated.

```

<!--
    a POA, using servant activator
-->
<poa>
  <policies>
    <policy name="request-processing" value="use-servant-manager"/>
    <policy name="servant-retention" value="retain"/>
  </policies>

  <!-- inject the servant activator -->
  <property name="set_servant_manager">
    <bean class="ServantActivatorImpl"/>
  </property>

  <!-- create a URL -->
  <object uri="my-server-use-servant-activator"/>
</poa>

```

4.3.4 Use Servant Manager (Locator)

Under this setting, the POA prohibits manually activating objects. A servant locator should be injected to the POA as a bean instance of a class inherited from *PortableServer::ServantLocator* and have the *preinvoke()* and *postinvoke()* operations of its parent implemented, as in the following example:

```

class ServantLocatorImpl : public PortableServer::ServantLocator
{
public:
    PortableServer::Servant preinvoke(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr      poa,
        const char*                   operation,
        PortableServer::ServantLocator::Cookie& cookie);

    void postinvoke(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr      poa,
        const char*                   operation,
        PortableServer::ServantLocator::Cookie& cookie,
        PortableServer::Servant      servant);
};

```

Every time, on processing a request, the POA will call this servant locator's *preinvoke()* method to get a one time only servant to handle the request, and will call the *postinvoke()* method afterwards.

In the following example, a servant locator is injected to the POA. An object reference is created from that POA. *Everytime* a client invokes a method on this reference, the servant locator's *preinvoke()* and *postinvoke()* will be called.

```

<!--
    a POA, using servant locator
-->
<poa>
  <policies>
    <policy name="request-processing" value="use-servant-manager"/>
    <policy name="servant-retention" value="non-retain"/>
  </policies>

  <!-- inject the servant activator -->
  <property name="set_servant_manager">
    <bean class="ServantLocatorImpl"/>
  </property>

  <!-- create a URL -->
  <object uri="my-server-use-servant-locator"/>
</poa>

```

4.3.5 Transparent Location Forward

A servant locator or activator not only can resolve servant implementations on demand by returning their local references, but also can transparently redirect clients to another remote CORBA object by throwing *ForwardRequest* exception containing its reference.

For instance, with a servant activator, instance of returning a servant reference, the *incarnate()* method throws an *ForwardRequest* exception as in the following example:

```

class ServantActivatorImpl : public PortableServer::ServantActivator
{
  Public:
    PortableServer::Servant incarnate(
      const PortableServer::ObjectId& oid,
      PortableServer::POA_ptr poa) {
      CORBA::Object_ptr new_obj = ...
      throw PortableServer::ForwardRequest(new_obj.in());
    }
    ...
};

```

Equivalently, a servant locator can also throw a *RequestForward* exception from the *preinvoke()* method as in the example below:

```
class ServantLocatorImpl : public PortableServer::ServantLocator {
public:
    PortableServer::Servant preinvoke(
        const PortableServer::ObjectId& oid,
        PortableServer::POA_ptr      poa,
        const char*                    operation,
        PortableServer::ServantLocator::Cookie& cookie) {
        CORBA::Object_ptr new_obj = ...
        throw PortableServer::ForwardRequest(new_obj.in())
    }
    ...
};
```

Using the mechanism above, an object server can transparently redirect (a.k.a forward) a client to another object server located in another executing process or even at another remote network node. This feature is fully transparent to client side applications, and is referred to as *location forward* in CORBA.

By default, once a client is redirected to another object, all subsequent invocations this client made on the original object reference will automatically be delivered to the new object through a direct connection.

4.4 Customize Server Models

Unlike EJB (pre-3.x) or CCM, the core framework of PocoCapsule/CORBA does not introduce a new server model but simply uses the native OMG POA as the primitive server model. Hence, all standardized and vendor value added POA server setup scenarios and functionalities are able to be supported straightforwardly.

Furthermore, PocoCapsule/CORBA does not enforce this core deployment model as the single possible option. Developers or third parties can easily extend and/or customize this core deployment model by defining their own domain specific model (DSM) schemas. These DSM schemas could have a higher abstraction level and are more expressive for specific applications than the relative low level generic core schema.

Different from other schema extension solutions provided in other IoC frameworks²⁵, the DSM support in PocoCapsule is declarative and based on widely used, standardized, and portable technology that is completely free of any proprietary plug-in API or any XML DOM/SAX programming. As described in more detail in the *PocoCapsule/C++ IoC&DSM Developer Guide*²⁶, a DSM in PocoCapsule/C++ IoC&DSM is defined by its document type definition (DTD) and its schema transformation templates XSLT style sheet. The DTD should be declared as the system id value of the DOCTYPE, and the XSLT file name should be declared as the *href* attribute of the xml-transform processing-instruction, as in the following example:

²⁵ Such as the “Extensible XML Authoring” of Spring Framework 2.0.

²⁶ <http://www.pocomatic.com/docs/pococpp-iocdsm-dev-guide.pdf>

```

<?xml ...?>
<!DOCTYPE my-application-context SYSTEM my-application-context.dtd>

<?xml-transform type="text/xsl" href="myapp2corba.xsl"?>

<my-application-context>
...
</my-application-context>

```

Here, the `myapp2corba.xsl` specifies the transformation mapping from the user defined DSM schema to the PocoCapsule/CORBA core schema. A segment of this XSLT style sheet is illustrated as follows:

```

<xsl:stylesheet version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method="xml" encoding="iso-8859-1" indent="yes"/>
<xsl:output doctype-system=
  "http://www.pocomatic.com/poco-application-context.dtd"/>
...
<xsl:template match="home-with-servant-locator">
  <poa>
    <policies>
      <policy name="request-processing" value="use-servant-manager"/>
      <policy name="servant-retention" value="non-retain"/>
    </policies>
    <xsl:apply-templates/>
  </poa>
</xsl:template>

...

<xsl:template match="servant-locator">
  <property name="_servant_manager">
    <bean>
      <xsl:attribute name="class">
        <xsl:value-of select="@class"/>
      </xsl:attribute>
    </bean>
  </property>
</xsl:template>

...

```

Here, a `<home-with-servant-locator>` element in the DSM is mapped to a `<poa>` element with desired policies. And a `<servant-locator>` element is transformed to a `<property>` setter IoC.

With this transformation template specified in the processing-instruction style sheet, the servant locator example in the previous section can be deployed using the following more expressive description segment:

```

<home-with-servant-locator>
  <servant-locator class="MyServantLocatorImpl"/>
</home-with-servant-locator>

```

This example can be found in the `examples/corba/dsm-server` directory of the PocoCapsule/C++ installation.

<this is an empty page>

5. Event/Notification Applications

5.1 The OMG Event/Notification Service

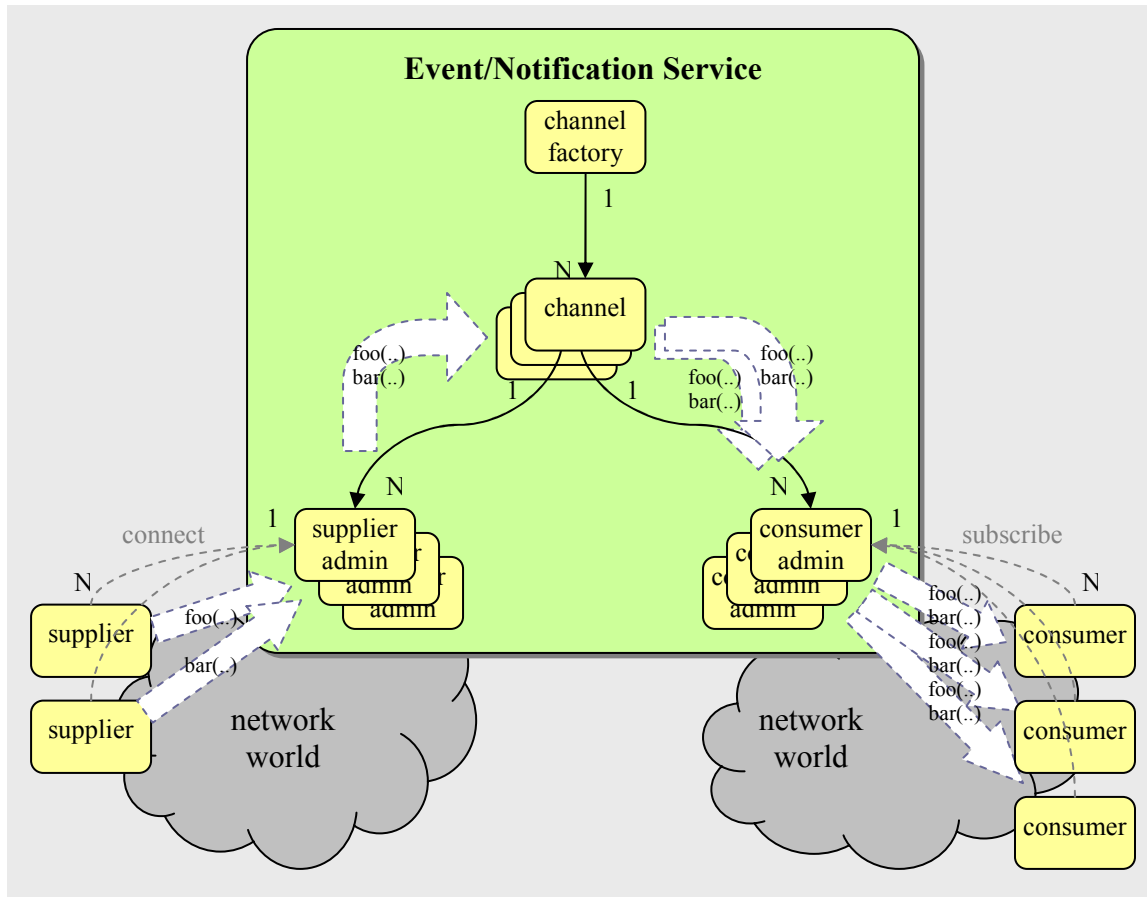
In addition to the client/server style invocations on remote objects, another important distributed application model is the publish/subscribe event distribution, in which the event suppliers (emitters) and consumers (listeners) are decoupled from each other. The OMG Event/Notification Service provides a middleware support for CORBA applications with this model. When using such a service, application developers need to deal with the following two aspects:

- The service's *connection model*: namely, how user implemented event suppliers and consumers are connected or subscribed to the service for sending or receiving events.
- The *event pushing operation(s)* and *message types*: namely, the operations invoked by suppliers to push events to the service, and the operations implemented by consumers to receive events pushed from the event service.

5.1.1 Connection model

The connection model of the OMG Event/Notification Service comes with truck load of low level plumbing details and incurs considerable boilerplate code. It is not only hard for novices and domain/business application developers to comprehend but also cumbersome for system focused CORBA experts. PocoCapsule/CORBA drastically simplifies this connection model by abstracting away all low level complexities behind a highly intuitive and expressive deployment model. Some Event/Notification Service usage scenarios, such as typed event service and filter, used to be notoriously obscure to CORBA experts and dreadful obstacles to CCM, become fairly obvious and handy even for CORBA novices in PocoCapsule/CORBA.

With PocoCapsule/CORBA, developers do not have to learn and know low level middleware details but only need to have a conceptual level knowledge of the OMG Event/Notification Service connection model. This concept level view can be summarized by the following diagram:



This diagram illustrates the following characteristics of this service:

- *Publish/subscribe*: User implemented event suppliers connect to and push event into the service, and user event consumers subscribe to and receive event from the service.
- *Service objects*: An Event/Notification service instance exports the following service objects: a channel factory, arbitrary number of event channels created from this factory, number of supplier admins and consumer admins created from their respective channels (each channel may create any number of admins).
- *Connections*: Each supplier admin can accept connections from multiple suppliers.
- *Subscriptions*: Each consumer admin can accept subscriptions from multiple consumers.
- *Message flow*: Unless dropped off by filters or QoS controls, events pushed into a supplier admin by connected suppliers will all be handed over to its channel, and then replicated to all consumer admins of that channel. Each admin further forwards events to all of its subscribed remote consumers.

The actual OMG Event/Notification Service is more complicated than this conceptual description. It consists of additional concepts and artifacts, such as proxies, filters and filter factories, not mentioned in the discussion above.

5.1.2 Event pushing operation(s) and message types

In the OMG Event/Notification Service, an event supplier pushes events into a remote service by invoking event pushing operations on a proxy object in the service. This method invocation scenario is the same as a client invoking a method on a remote CORBA server object. Similarly, a subscribed event consumer receives events from the service as event operation invocations in the same way as a CORBA server receiving remote client invocations.

Server interfaces of conventional client/server applications could be defined by either users or OMG (such as the Name Service server). Similarly, event receiving interfaces of event consumer applications (which are also event pushing interfaces used by suppliers) could also be defined by users or OMG.

Events as IDL invocations of user defined event pushing/receiving interfaces are referred to as “typed” events. Events as the “push”-some method invocations of the three OMG pre-defined event pushing/receiving interfaces are referred to as “structured”, “sequence”, or “untyped” events respectively. These event pushing/receiving interfaces and their respective event pushing operations are listed in the following table:

<i>Event type</i>	<i>event pushing/receiving interface and operation(s)</i>
typed	user defined IDL interface
	operations of user defined event interface
structured	CosNotifyComm::StructuredPushConsumer
	void push_structured_event(const CosNotification::StructuredEvent&);
sequence	CosNotifyComm::SequencePushConsumer
	void push_structured_events(const CosNotification::EventBatch&);
untyped	CosEventComm::PushConsumer
	void push(const CORBA::Any&);

5.1.3 POEM vs EVIL

The native OMG Event/Notification model discussed in the previous section is referred to as the *Plain Old Event Model (POEM)* of CORBA event applications. PocoCapsule/C++ explicitly supports the POEM without creating a new compliance barrier. This implies OMG Event/Notification Service implementations and POEM applications (emitters, listeners) developed with or without knowledge of PocoCapsule/CORBA can all be seamlessly used in PocoCapsule/CORBA and seamlessly interoperable with those not using PocoCapsule/CORBA.

In contrast, CCM defines an “*Events as Valuetypes of IdL*” (EVIL) event model that outlaws the POEM. EVIL is a rare and weird practice in existing CORBA POEM applications, due to the fact that this will not only enforce more plumbing code but also is not supported by almost all implementations of the OMG Event/Notification Service. Therefore, all existing POEM applications and almost all service implementations are neither reusable in EVIL nor interoperable with EVIL.

5.2 Event Consumer Implementation

Event consumers (listeners) are implementations that receive and consume events. The implementations of POEM event consumers are the same as implementations of conventional CORBA server objects described previously. Namely, implementations could either be servants inherited from POA skeletons of their respective event receiving interfaces or native C++ classes that do not inherit from skeletons but only implement their respective event push operations.

For instance, a structured event consumer should implement the OMG defined event pushing operation *push_structured_event(...)* as in the following “*native*” implementation:

```
class MyStructuredEventListenerNativeImpl
{
public:
    void push_structured_event(
        const CosNotification::StructuredEvent& event) {
        // business logic here to consume the received event
        ...
    }
    ...
};
```

Or as in the following “*servant*” implementation:

```
class MyStructuredEventListenerServantImpl
    : public POA_CosNotifyComm::StructuredPushConsumer
{
public:
    void push_structured_event(
        const CosNotification::StructuredEvent& event) {
        // business logic here to consume the received event
        ...
    }
    ...
};
```

Note: To shield domain/business application developers from unnecessary plumbing, PocoCapsule ignores and does not require “*native*” style implementations of structured, sequence, and untyped consumers to implement non-event-pushing operations that are declared in these OMG pre-defined event consumer interfaces²⁷. If an application does need to utilize these operations, it should implemented and deploy a “*servant*” style implementation.

Typed event consumer implementations are very similar, except their event pushing/receiving interfaces are not pre-defined by OMG but by users. For instance, the following user IDL module defines a “*typed*” event pushing/receiving interface:

²⁷ Such as `disconnect_xxx()` and `offer_change()` operations.

```

module sample {
    ...
    interface MyTypedEvent {
        void hello(in string greeting, in string sender);
        void headlinenews(in NewsBrief news, in string sender);
        void stockquote(in string id, in float quote);
    };
};

```

For this user defined event pushing/receiving interface, a “*native*” typed event consumer could be simply a class that supports all its operations as follows:

```

class MyTypedEventListenerNativeImpl
{
public:
    void hello(String greeting, String sender) { ... }
    void headlinenews(sample.NewsBrief news, String sender) { ... }
    void stockquote(String id, float quote) { ... }
};

```

A corresponded “*servant*” style “*typed*” event consumer implementation is similar except that the implementation class inherits from the interface’s POA skeleton as highlighted in the following:

```

class MyTypedEventListenerNativeImpl : public POA_sample::MyTypedEvent
{
public:
    void hello(String greeting, String sender) { ... }
    void headlinenews(sample.NewsBrief news, String sender) { ... }
    void stockquote(String id, float quote) { ... }
};

```

5.3 The <event-consumer-adaptor> element

As illustrated in the previous section, writing a consumer implementation, either “*native*” or “*servant*”, is the same as writing a normal server object implementation. Then, to be able to receive intended events, the consumer implementation needs to be subscribed to a desired service channel.

PocoCapsule/CORBA abstracts away all plumbing complexities of OMG Event/Notification Service subscription/connection model by the <event-consumer-adaptor> deployment description element.

For instance, to subscribe the structured event consumer above to an OMG Event/Notification service, the simplest form of description using <event-consumer-adaptor> could be:

```

<!-- subscribe a listener to structured event service -->
<event-consumer-adapter type="structured">
  <event-consumer impl-type="native">
    <bean class="MyStructuredEventListenerNativeImpl"/>
  </event-consumer>
</event-consumer-adapter>

```

Similarly, to subscribe the typed event consumer above to an OMG Typed Event/Notification service, the description could be:

```

<!-- subscribe a listener to typed event service -->
<event-consumer-adapter type="typed">
  <event-consumer impl-type="native" interface="sample::MyTypedEvent">
    <bean class="MyTypedEventListenerNativeImpl"/>
  </event-consumer>
</event-consumer-adapter>

```

The `<event-consumer-adapter>` element is defined in the `corba-application-context.dtd` as follows:

```

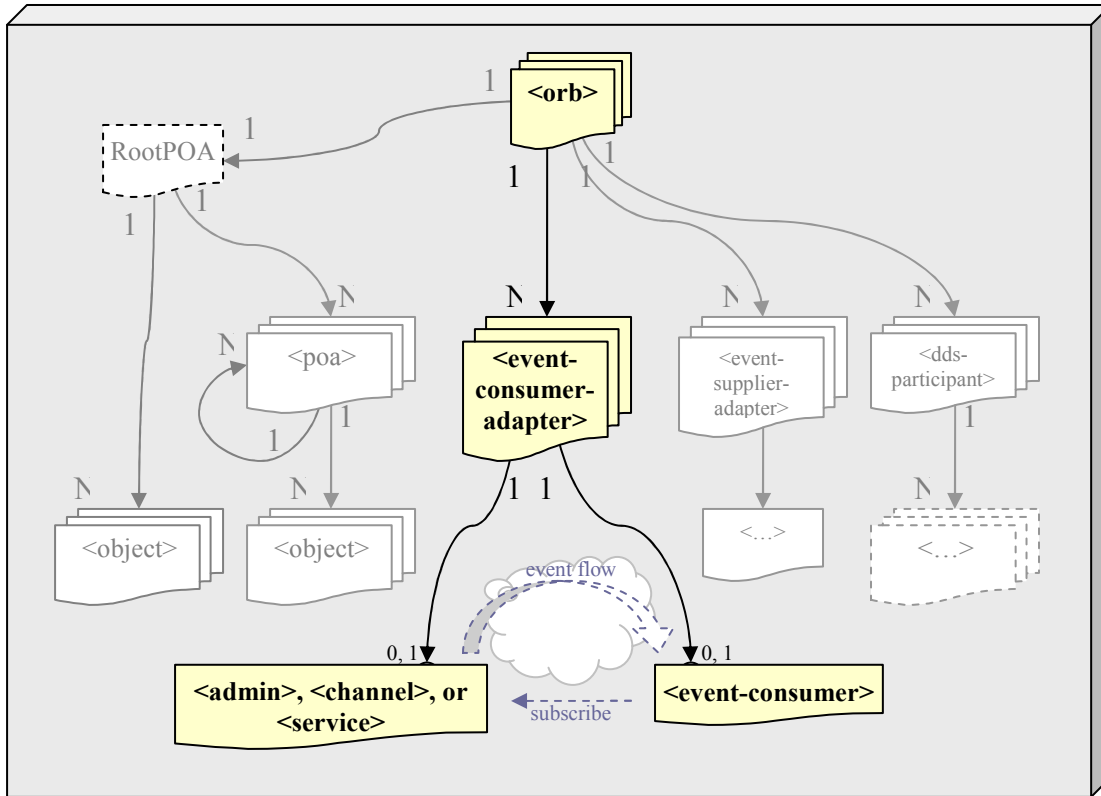
<!ENTITY %conn-ports "admin|channel|service">
<!ENTITY %event-types "typed|structured|sequence|untyped">
<!ENTITY %impl-kinds "object|servant|native">

<!ELEMENT event-consumer-adapter ((%conn-ports;)?, event-consumer, props?)>
<!ATTLIST event-consumer-adapter id ID #IMPLIED>
<!ATTLIST event-consumer-adapter type (%event-types;) #REQUIRED>
<!ATTLIST event-consumer-adapter topic CDATA #IMPLIED>

<!ELEMENT event-consumer (bean|ref)>
<!ATTLIST event-consumer impl-ref IDREF #IMPLIED>
<!ATTLIST event-consumer impl-type (%impl-kinds;) #REQUIRED>
<!ATTLIST event-consumer interface CDATA #IMPLIED>

```

The following diagram illustrates the `<event-consumer-adapter>` element and some of its child elements as part of the PocoCapsule/CORBA core schema hierarchy and the described Event/Notification Service consumer subscription and event flow.



Attributes of the `<event-consumer-adapter>` are:

- **id** attribute: An `<event-consumer-adapter>` element is actually transformed to an eager singleton `<bean>` instance of type `CORBA::Object` with this `id`²⁸. This bean is actually a proxy reference of a downstream end of the channel. This proxy reference supports the consumer interface of the specified event type.
- **type** attribute: specifies the type of the event consumer. In this release, the valid values of this attribute are “*typed*”, “*untyped*”, “*structured*” or “*sequence*”.
- **topic** attribute: This attribute can be used for typed, structured, or sequence consumers and is ignored for untyped consumer. It provides a primitive event channel multiplex mechanism²⁹.
 - For a typed consumer, the value of this attribute is the “*key*” or repository id used to subscribe to the channel. Only events from a supplier connected with compatible key will be forwarded to the consumer.
 - For a structured or a sequence event consumer, this attribute should be in the format of `domain_name:type_name`. If it is specified, it will be used to filter out events with `domain_name` or `type_name` field fail to match the specified pattern in the topic.

²⁸ The `<bean>` element is defined in the `poco-application-context.dtd`. The lifecycle management of eager singleton beans is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

²⁹ This feature is subject to the support of the `subscription_change()` method of the underlying Event/Notification Service implementation.

<i>event type</i>	<i>event topic</i>
typed	the event interface repository id, or key (default “*”)
untyped	N/A
structured	[domain_name][:type_name] (default value is “%ALL:%ALL”)
sequence	[domain_name][:type_name] (default value is “%ALL:%ALL”)

Child elements of `<event-consumer-adapter>` are described below.

5.3.1 `<event-consumer>` element

This child element of `<event-consumer-adapter>` specifies the consumer implementation to be subscribed to the event service for receiving intended events. Attributes of this element are similar to that of the `<object>` element discussed previously. This similarity is natural because an `<event-consumer>` is no more than a CORBA object that is to be invoked remotely by the event service to push in events.

- ***impl-ref***: This attribute specifies the implementation bean. This is a form of shortcut of the `<ref>` child element.
- ***impl-type***: This attribute specifies the category of implementation bean referred by the *impl-ref* attribute, child element `<bean>`, or `<ref>`. The valid values of this attribute are “*native*”, “*servant*”, or “*object*” with the following meanings:
 - “*native*”: The implementation is a “*native*” consumer (listener) bean that supports all event pushing operations of the interface. If this is a typed consumer (namely, the *type* attribute of the parent `<event-consumer-adapter>` is “typed”), the *interface* attribute of this element should be used to specify the full scope name of the event interface, such as “*sample::MyTypedEvent*” in the previous example.
 - “*servant*”: The implementation is a “*servant*” that inherits the POA skeleton interface of the event interface and implements all operations defined by the interface.
 - “*object*”: The implementation is a consumer reference bean (of C++ class type able to be implicitly casted to `CORBA::Object`) which supports the required event interface. For instance, the *impl-ref* attribute or the `<ref>` child element is the *id* attribute of an `<object>` or `<event-supplier-adapter>` element.
- ***interface***: this attribute is only used for “*typed*” “*native*” consumer, and is used to specifies the full scoped interface name of the user defined event pushing/receiving interface, as in the following example:

```

<!-- subscribe a listener to typed event service -->
<event-consumer-adapter type="typed">
  <event-consumer impl-type="native"
    interface="sample::MyTypedEvent">
    <bean class="MyTypedEventListenerNativeImpl"/>
  </event-consumer>
</event-consumer-adapter>

```

This attribute is ignored for other types of events (namely the “*structured*”, “*sequence*”, or “*untyped*” events) and is also ignored for other type of implementations (namely the “*servant*” or “*object*” implementations).

5.3.2 <admin>, <channel>, <service> elements

By default, the consumer is subscribed to the default consumer admin of the default channel (or default typed channel)³⁰ of the service resolved from the ORB using service name “*NotificationService*”.

If an alternative admin, channel, or service is to be used, then the <event-consumer-adapter> should be declared with an <admin>, <channel>, or <service> child element. The rules here are:

- If the <admin> element is declared, the consumer will be subscribed to the specified consumer admin.
- If the <channel> element is declared, the consumer will be subscribed to the default consumer admin of the specified channel.
- If the <service> element is declared, the consumer will be subscribed to the default consumer admin of the default channel³¹ of the specified service.

These three elements are defined as follows:

```

<!ELEMENT admin      ((bean|ref)?)>
<!ATTLIST admin     ior      CDATA  #IMPLIED>
<!ATTLIST admin     ref      IDREF  #IMPLIED>

<!ELEMENT channel   ((bean|ref)?)>
<!ATTLIST channel   ior      CDATA  #IMPLIED>
<!ATTLIST channel   ref      IDREF  #IMPLIED>

<!ELEMENT service   ((bean|ref)?)>
<!ATTLIST service   ior      CDATA  #IMPLIED>
<!ATTLIST service   ref      IDREF  #IMPLIED>

```

³⁰ A default channel is the first channel in the channel sequence returned from the `get_all_channels()` or `get_all_typed_channels()` operations on the service (namely the channel or typed channel factory). If this channel does not exist, PocoCapsule/CORBA will create one.

³¹ A default channel is the first channel in the channel sequence returned from the `get_all_channels()` or `get_all_typed_channels()` operations on the service (namely the channel or typed channel factory). If this channel does not exist, PocoCapsule/CORBA will create one.

Its attributes and child elements are defined as:

- *ior* attribute: The IOR (or URL) of the correspondent admin, channel, or service reference.
- *ref* attribute and *<bean>* or *<ref>* child elements: Specifies a bean of client stub of the respective CORBA object (namely, consumer admin, channel, or service).

For instance, to subscribe the consumer to a channel with a given corbaloc URL, the deployment descriptor could be expressed as follows

```

<!-- subscribe a listener to typed event service -->
<event-consumer-adapter type="typed">
  <channel ior="corbaloc::192.168.2.3:12345/my-channel"/>
  <event-consumer impl-type="native"
    interface="sample::MyTypedEvent">
    <bean class="MyTypedEventListenerNativeImpl"/>
  </event-consumer>
</event-consumer-adapter>

```

5.3.3 *<props>* element and event filter

The *<props>* child element of *<event-consumer-adapter>* is similar to the *<policies>* of the *<poa>* element. Like *<poa>*'s *<policies>* is to configure the QoS policies supported by the POA (namely all objects from that POA), the *<props>* is mostly to support QoS configuration of the given *<event-consumer-adapter>*. This element is mostly reserved for future enhancement. In the current release, the only feature supported through this element is the consumer side event filter configuration.

The *<props>* element is defined as follows:

```

<!ELEMENT props      (prop)*>
<!ELEMENT prop      EMPTY>
<!ATTLIST prop      name   CDATA  #REQUIRED>
<!ATTLIST prop      value  CDATA  #REQUIRED>

```

The only supported value of the *name* attribute in this release is “filter”, with the filter expression as the value of the *value* attribute.

For instance, the following *<props>* element configures an event filter for a *typed* consumer:

```

<!-- subscribe a listener to typed event service with filter -->
<event-consumer-adapter type="typed">
  <event-consumer impl-type="native"
    interface="sample::MyTypedEvent">
    <bean class="MyTypedEventListenerNativeImpl"/>
  </event-consumer>

  <props>
    <prop name="filter"
      value="($type_name == 'headlinenews')
        or ($type_name == 'stock_quote')"/>
  </props>
</event-consumer-adapter>

```

The corresponded filter for a *structured* event consumer can be configured as follows:

```

<!-- subscribe a listener to structured event service with filter -->
<event-consumer-adapter type="structured">
  <event-consumer impl-type="native">
    <bean class="MyStructuredEventListenerNativeImpl"/>
  </event-consumer>

  <props>
    <prop name="filter"
      value="($event_name == 'headlinenews')
        or ($event_name == 'stock_quote')"/>
  </props>
</event-consumer-adapter>

```

5.4 Event Supplier Implementation

Event suppliers push events into event service(s) in the same way that clients invoke operations on remote server objects to send requests. For suppliers, these event pushing invocations are made on proxy consumers in the remote server of the Event/Notification service(s). These proxy consumers support the desired event pushing interface as discussed previously and listed below:

Event type	event pushing/receiving interface and operation(s)
typed	user defined IDL interface
	operations of user defined event interface
structured	CosNotifyComm::StructuredPushConsumer
	void push_structured_event(const CosNotification::StructuredEvent&);
Sequence	CosNotifyComm::SequencePushConsumer
	void push_structured_events(const CosNotification::EventBatch&);
Untyped	CosEventComm::PushConsumer
	void push(const CORBA::Any&);

For instance, to push a structured event to an intended event service channel, an event supplier simply obtains the reference to a proxy consumer object from the service and invoke the *push_structured_event()* on the reference as follows:

```

CORBA::Object_var ref = ...
proxy = CosNotifyComm::StructuredPushConsumer::_narrow(ref.in());

CosNotification::StructuredEvent event = ...

proxy->push_structured_event(event);

```

Similarly, to push a typed event to a typed event service channel, the event supplier first needs to get the reference of a proxy consumer from the remote event service, and then retrieves the corresponded typed proxy (also known as *the <I> interface*) from that proxy consumer using its *get_typed_consumer()* operation. The returned typed proxy should support and be able to be narrowed down to the user defined typed interface. This is illustrated as follows:

```

CosTypedEventComm::TypedPushConsumer_ptr proxy = ...

sample::MyTypedEvent_var the_i_intf
    = sample::MyTypedEvent::_narrow(proxy->get_typed_consumer());

the_i_intf->stock_quote("IBM", 95.5);

```

The examples above show that it is straightforward for the supplier applications to push events to the intended event service channels if the references of the desired proxy consumers are available. However, getting references of these proxy consumers from a given event service involves many framework level programming details that should not be the focus of business/domain application developers.

PocoCapsule/CORBA deployment model hides these low level details by creating these proxy consumers based on given declarative deployment descriptions and optionally destroying them on the termination of the local application deployment contexts.

There are two approaches an application can obtain the proxy consumer references created by PocoCapsule/CORBA container. In the first approach, applications resolve these references from the deployed application contexts using the *getBean()* operation. This approach, although straightforward, would tightly couple the application to the framework and therefore should be avoided.

In the second approach, event supplier applications do not resolve proxy consumer references from the container but let the container inject them back through the following OMG standardized consumer reference injection operations (also known as “setter methods” or “receptacle ports”):

<i>Event type</i>	<i>Consumer reference inject(setter, receptacle) operation</i>
Typed	<code>void connect_typed_push_consumer (CosTypedEventComm::TypedPushConsumer_ptr);</code>
Untyped	<code>void connect_any_push_consumer (CosEventComm::PushConsumer_ptr);</code>

Structured	<code>void connect_structured_consumer(CosNotifyComm::StructuredPushConsumer_ptr);</code>
Sequence	<code>void connect_sequence_consumer(CosNotifyComm::SequencePushConsumer_ptr);</code>

Note: a proxy consumer reference retrieved from the application context using `getBean()` is of type `CORBA::Object_ptr`. It needs to be narrowed to the event consumer interface by application. On the other hand, the proxy consumer references injected via above methods are all type narrowed already.

This approach of resolving dependency reference for applications is known as “dependency injection”.

As an example, a structured event supplier with such a proxy consumer reference injection operation could look like:

```
class MyStructuredEventEmitterImpl {
    // an external dependency
    CosNotifyComm::StructuredConsumer_var _proxy;

public:
    // an method for PocoCapsule/CORBA to inject the proxy consumer
    void connect_structured_push_consumer(
        CosNotifyComm::StructuredPushConsumer_ptr proxy) {
        _proxy = proxy;
        CORBA::Object::_duplicate(proxy);
    }

    ...
    // arbitrary business logic methods,
    // to push event to the proxy consumer.
    void blabla(...) {
        ...
        _proxy->push_structured_event(event);
        ...
    }
};
```

Here, the event supplier class implements the proxy consumer reference injection method: `connect_structured_push_consumer(CosNotifyComm::StructuredPushConsumer_ptr)`.

Similarly, a typed event supplier implementation with such a proxy consumer reference injection operation could look like:

```

class MyTypedEventEmitterImpl {
    // an external dependency
    sample::MyTypedEvent_var _the_i_intf;

public:
    // an method for PocoCapsule/CORBA to inject the proxy consumer
    void connect_structured_push_consumer(
        CosTypedEventComm::TypedPushConsumer_ptr proxy) {
        CORBA::Object_var ref = proxy->get_typed_consumer();
        _the_i_intf = sample::MyTypedEvent::_narrow(ref.in());
    }

    ...
    // arbitrary business logic methods,
    // to push event to the proxy consumer through the <I> interface.
    void blabla(...) {
        ...
        _the_i_intf->stockquote("IBM", (CORBA::Float)95.5);
        ...
    }
};

```

Here, the typed event supplier class implements the proxy consumer reference injection method: *connect_structured_push_consumer(CosTypedEventComm::TypedPushConsumer_ptr)*.

For structured, sequence, and untyped event suppliers, injected proxy consumer references are also event interfaces themselves. Therefore, these supplier implementations can explicitly invoke the event operation(s) on these references. These event operations are the same as the event operations of correspondent event consumers, which have been listed in the previous section.

For a typed event supplier, the injected proxy consumer reference is not the actual typed event interface, but a control interface, for operations such as QoS setting and disconnecting from the service. The correspondent typed event proxy interface can be retrieved from this (control) proxy reference using the *get_typed_consumer()* method and then narrowed the returned reference to the desired typed stub.

5.5 The **<event-supplier-adapter>** element

PocoCapsule/CORBA abstracts away all plumbing complexities of OMG Event/Notification Service connection model by the *<event-supplier-adapter>* deployment description element.

For instance, to connect the structured event supplier above to an OMG Event/Notification service, the simplest form of description using *<event-supplier-adapter>* could be:

```

<event-supplier-adapter type="structured">
  <event-supplier impl-type="native">
    <bean class="MyStructuredEventEmitterNativeImpl"/>
  </event-supplier>
</event-supplier-adapter>

```

Similarly, to connect the typed event supplier above to an OMG Typed Event/Notification service, the description could be:

```

<event-supplier-adapter type="typed">
  <event-supplier impl-type="native">
    <bean class="MyTypedEventEmitterNativeImpl"/>
  </event-supplier>
</event-supplier-adapter>

```

In each of the two examples above, an `<event-supplier-adapter>` is used to declare that the adapter should connect to a structured or typed event service³², create a proxy consumer and inject its reference to the specified `<event-supplier>` implementations.

The `<event-supplier-adapter>` element is defined in `corba-application-context.dtd` as follows:

```

<!ENTITY %conn-ports "admin|channel|service">
<!ENTITY %event-types "typed|structured|sequence|untyped">
<!ENTITY %impl-kinds "object|servant|native">

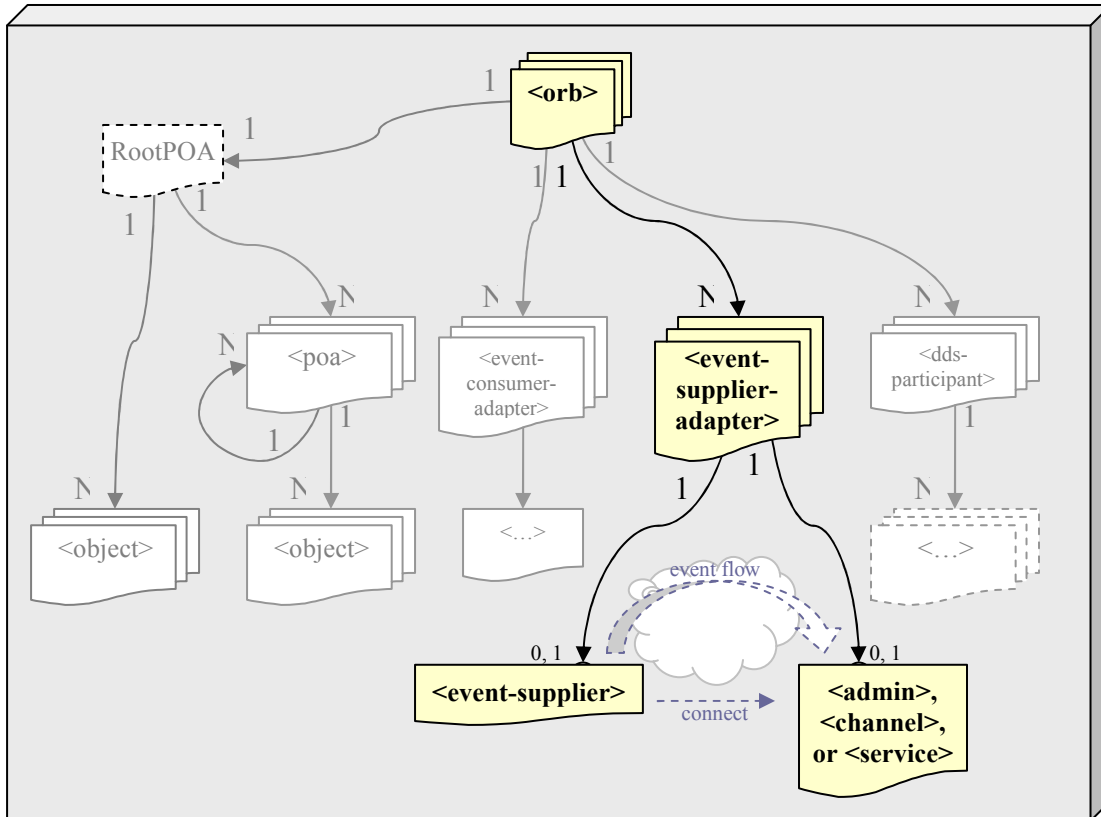
<!ELEMENT event-supplier-adapter ((%conn-ports;)?, event-consumer?, props?)>
<!ATTLIST event-supplier-adapter id ID #IMPLIED>
<!ATTLIST event-supplier-adapter type (%event-types;) #REQUIRED>
<!ATTLIST event-supplier-adapter topic CDATA #IMPLIED>

<!ELEMENT event-supplier (bean|ref)>
<!ATTLIST event-supplier impl-ref IDREF #IMPLIED>
<!ATTLIST event-supplier impl-type (%impl-kinds;) #REQUIRED>

```

The following diagram illustrates the `<event-supplier-adapter>` element as well as some of its child elements as part of the PocoCapsule/CORBA core schema hierarchy, the described supplier to the service connection, and event flow.

³² Similar to `<event-consumer-adapter>`, if the `<service>`, `<channel>`, or `<admin>` is not specified as child element of `<event-supplier-adapter>`, the adapter will connect to the service resolved from the ORB using the name "NotificationService".



Attributes of the `<event-supplier-adapter>` are:

- **id** attribute: An `<event-supplier-adapter>` element is actually transformed to an eager singleton `<bean>` instance³³ of type `CORBA::Object` with this id. This bean is actually the object reference of the proxy consumer created in the event service. As discussed in the previous section, the type narrowed version of this reference is injected to the event supplier by its proxy injection method.
- **type** attribute: specifies the type of the event supplier. In this release, the valid values of this attribute are “*typed*”, “*untyped*”, “*structured*” or “*sequence*”.
- **topic** attribute: This attribute can be used for typed suppliers and is ignored by others. It provides a primitive event channel multiplex mechanism. The value of this attribute is the “*key*” or repository id used to connect to the channel. Only consumers subscribed with the compatible key will be able to receive events from this supplier.

Child elements of `<event-supplier-adapter>` are described below.

5.5.1 `<event-supplier>` element

This child element of `<event-supplier-adapter>` specifies a supplier implementation that accepts the injection of a proxy consumer reference representing the connection

³³ The `<bean>` element is defined in the `poco-application-context.dtd`. The lifecycle management of eager singleton beans is described in the *PocoCapsule/C++ IoC&DSM Developer Guide* document.

established by the event supplier adapter to the intended event service. Attributes of this element are similar to those of the `<event-consumer>` element discussed previously:

- ***impl-ref***: This attribute specifies the implementation bean. This is a form of shortcut of the `<ref>` child element.
- ***impl-type***: This attribute specifies the category of the implementation bean referred by the *impl-ref* attribute, child element `<bean>`, or `<ref>`. It could take values of “*native*”, “*servant*”, or “*object*” with the following meanings:
 - “*native*”: The implementation is a “*native*” consumer (listener) bean that implements the type narrowed proxy consumer reference injection operation.
 - “*servant*”: This option is reserved in this release.
 - “*object*”: The implementations is a proxy supplier reference bean (of C++ class type able to be implicitly casted to `CORBA::Object`) which supports the required proxy consumer injection method. For instance, the *impl-ref* attribute or the `<ref>` child element is the *id* attribute of an `<object>` or `<event-consumer-adapter>` element.

5.5.2 `<admin>`, `<channel>`, `<service>` elements

By default, the proxy consumer is created from the default supplier admin of the default channel (or default typed channel)³⁴ of the service resolved from the ORB using the service name “*NotificationService*”.

If an alternative admin, channel, or service is to be used, then the `<event-consumer-adapter>` should be declared with an `<admin>`, `<channel>`, or `<service>` child element. The rules here are:

- If the `<admin>` element is declared, the proxy consumer will be created from the specified supplier admin.
- If the `<channel>` element is declared, the proxy consumer will be subscribed from the default supplier admin of the specified channel.
- If the `<service>` element is declared, the proxy consumer will be created from the default supplier admin of the default channel³⁵ of the specified service.

These three elements are defined as follows:

³⁴ A default channel is the first channel in the channel sequence returned from the `get_all_channels()` or `get_all_typed_channels()` operations on the service (namely the channel or typed channel factory). If this channel does not exist, PocoCapsule/CORBA will create one.

³⁵ A default channel is the first channel in the channel sequence returned from the `get_all_channels()` or `get_all_typed_channels()` operations on the service (namely the channel or typed channel factory). If this channel does not exist, PocoCapsule/CORBA will create one.

```

<!ELEMENT admin      ((bean|ref)?)>
<!ATTLIST admin     ior      CDATA  #IMPLIED>
<!ATTLIST admin     ref      IDREF  #IMPLIED>

<!ELEMENT channel   ((bean|ref)?)>
<!ATTLIST channel   ior      CDATA  #IMPLIED>
<!ATTLIST channel   ref      IDREF  #IMPLIED>

<!ELEMENT service   ((bean|ref)?)>
<!ATTLIST service   ior      CDATA  #IMPLIED>
<!ATTLIST service   ref      IDREF  #IMPLIED>

```

Its attributes and child elements are defined as:

- ***ior*** attribute: The IOR (or URL) of the correspondent admin, channel, or service reference.
- ***ref*** attribute and ***<bean>*** or ***<ref>*** child elements: Specifies the client stub bean of the respective CORBA object (namely, supplier admin, channel, or service).

For instance, to connect to a channel with a given corbaloc URL, the deployment descriptor could be expressed as follows

```

<!-- connect a listener to typed event service -->
<event-supplier-adapter type="typed">
  <channel ior="corbaloc::192.168.2.3:12345/my-channel"/>
  <event-supplier impl-type="native"
    interface="sample::MyTypedEvent">
    <bean class="MyTypedEventEmitterNativeImpl"/>
  </event-supplier>
</event-supplier-adapter>

```

5.5.3 ***<props>*** element

The ***<props>*** child element of ***<event-supplier-adapter>*** is similar to the ***<policies>*** of the ***<poa>*** element. Like ***<poa>***'s ***<policies>*** is to configure the QoS policies supported by the POA (namely all objects from that POA), the ***<props>*** is mostly to support QoS configuration of the given ***<event-consumer-adapter>***. This element is reserved for future enhancement and is ignored in the current release.

6. DDS Applications

6.1 OMG Data Distribution Service (DDS)

In OMG Event/Notification Service, the untyped/structured/sequence events heavily rely on the *CORBA::Any* and therefore incur considerable performance and network bandwidth overhead³⁶. The straightforward solution is simply the typed event service. In fact, OMG should not even specify these pre-defined event types but only the typed one.

Unfortunately, most vendors and the specification committee themselves do not have the expertise to build an interoperable³⁷ and decently performed typed event service³⁸. Therefore, OMG DDS naturally came as a secondary alternative³⁹.

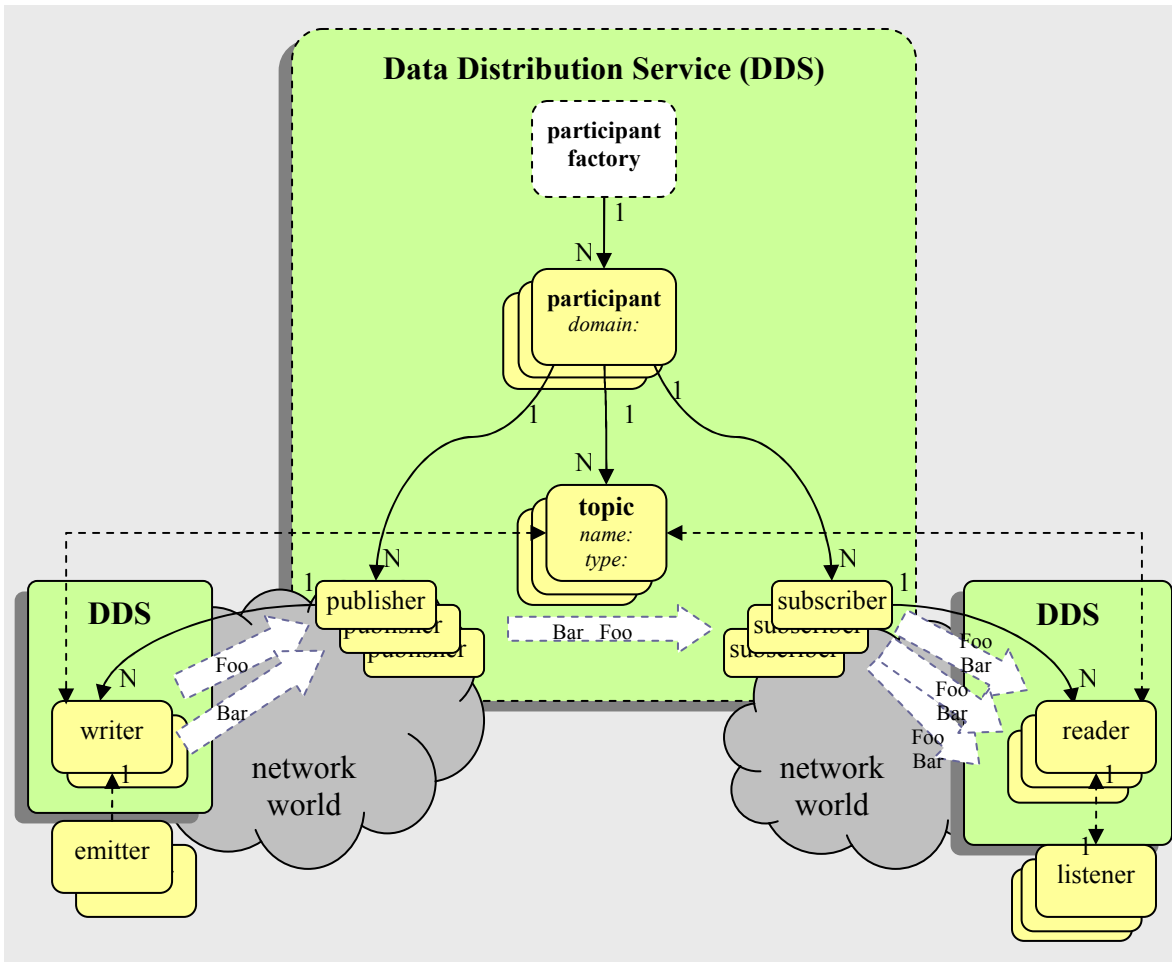
Comparing to the OMG Event/Notification Service, the connection model of DDS involves more infrastructure level plumbing. The PocoCapsule/CORBA+DDS hides most of these complexities behind a highly intuitive, expressive, declarative, and self-documented deployment model. This deployment model only requires application developers to have conceptual level knowledge of DDS and allows them to be able to focus on domain/business logic rather than low level framework connection model. This concept level view is summarized in the following diagram:

³⁶ For instance, an IDL enum only costs four bytes on the wire in CDR encoding. However, it would cost tens or even hundreds bytes on the wire if the same enum was inserted in a CORBA Any. Real cases of such applications are seen in JTRS-SCA from US Navy and CORBA/TMN specifications from 3GPP, TMF and ITU-T. These specifications all heavily use IDL enums and/or structs in CORBA Anys. A simple SCA event or TMN Alarm or Attribute Value Change event therefore would cost the CDR stream 2kbytes to 8Kbytes with less than 5% for real user data, 5% for IIOP request header, and rest 90% for metadata from CORBA Any CDR encoding! A defense user evaluated structured event service from a claimed leading vendor with a typed event service from a third party. The result showed the typed event service is 40 times (namely 4,000%) faster in their application.

³⁷ This interoperability includes the support of applications that have mixed GIOP 1.0, 1.1, and 1.2 participants.

³⁸ Many vendors were misguided to use the DSI/DII based implementation scenario suggested in the early Typed Event Service specification. This scenario is not only complicated to implement but also poorly performs.

³⁹ Similarly, the structured and sequence event services also intended to be a secondary alternative to typed event service.



This diagram illustrates the following characteristics of the DDS service:

- *Local engine*: Unlike OMG Event/Notification Service which requires only a vanilla ORB in event consumer and supplier applications, the OMG DDS assumes a local DDS engine as well as (optionally) a remote middleware service. Therefore, the diagram above is only conceptual. A given DDS implementation may not have the remote middleware service and/or may place some or all of its service objects inside data emitter or listener side local DDS engine⁴⁰.
- *Conceptual similarity to OMG Event/Notification Service*: Although the above and other difference, many concepts of DDS and Event/Notification Service are comparable (not exactly the same), as summarized in the following table:

<i>DDS concepts</i>	<i>Event/Notification Service concepts</i>
DomainParticipantFactory	ChannelFactory
DomainParticipant	Channel and FilterFactory
Topic	Filter

⁴⁰ For instance, in the DDS example comes with the PocoCapsule/CORBA, the demo DDS implementation have all DDS objects, including participants, subscribers, publishers and topics simplified as local objects.

Publisher	SupplierAdmin
Subscriber	ConsumerAdmin
DataWriter	ProxyPushConsumer
DataReader	ProxyPullSupplier

- *Publish/subscribe*: User implemented data emitters and listeners. The emitters are connected to the DDS service and push data into the service through *data writer* objects created from DDS. The data listeners are subscribed to the service and pull data back from the service through *data read* objects allocated by DDS.
- *Connections*: Each publisher can accept connections from multiple data emitters (by creating multiple writers).
- *Subscriptions*: Each subscriber can accept subscriptions from multiple data listeners (by allocating multiple separate data readers).
- *Message flow*: Unless dropped off in topics filtering or QoS control, data pushed into a data domain (with given id) from a data writers will all be forwarded and replicated to all data listeners in the domain that subscribed to the same topic.

The actual OMG DDS is more complicated than this conceptual description. It consists of additional concepts and artifacts not mentioned in the discussion above.

6.2 DDS Data Model, Typed Reader and Writer

DDS data types are simply IDL structures defined by users⁴¹. For each such data type, a typed DataReader and DataWriter IDL interfaces and necessary implementations will be generated from DDS vendor provided tools. For instance, for the following DDS type *Stock*:

```
struct Stock {
    string symbol;
    float quote;
};
```

The corresponding *typed data reader* StockDataReader and *typed data writer* StockDataWriter generated from a compliant DDS implementation will be⁴²:

⁴¹ To avoid a problem introduced by a flaw in DDS specification, DDS structures should only be defined in IDL global space, not within a scope of module or interface.

⁴² Instead of generating an IDL interface of a given DDS type, a DDS implementation may explicitly generate the stub classes of its typed reader/writer, and wrap their server skeletons within the typed “type support” class.

```

interface StockDataReader : DDS::DataReader {
    DDS::ReturnCode_t take_next_sample(
        inout Stock          data,
        inout DDS::SampleInfo info);
    ...
};

interface StockDataWriter : DDS::DataWriter {
    DDS::ReturnCode_t write(
        in Stock          data,
        in DDS::InstanceHandle_t handle);
    ...
};

```

Typed data readers and writers are used by data emitters and data listeners to send and pull data into and out of the DDS service.

Note: Besides generating the typed reader and the typed writer for a given user defined DDS data type, the tool from a DDS vendor should also generate the typed “type support implementation” class for each given DDS type. This class encapsulates the typed reader/writer implementations. In conventional programmatic deployment, application developers should be registered “type support implementation” classes to the DDS participant. The registered names of these types can be used to refer them in creating DDS topics. This complexity, however, is completely milted away in the declarative deployment model of PocoCapsule/CORBA for DDS to be discussed in the following sections.

6.3 DDS Data Emitter Application

DDS data emitter applications are similar to event suppliers of Event/Notification Service. An event supplier needs a proxy consumer reference from the service and pushes event to this proxy. Similarly, a DDS data emitter application requires a data writer from a DDS service as a proxy to write data into the service as in the following example:

```

...
DDS::DataWriter_ptr writer = ... // created from a DDS publisher

// narrow the writer to typed writer
StockDataWriter_var stock_writer = StockDataWriter::_narrow(writer);

Stock data;
DDS::InstanceHandle_t handle;

// write the data out
for(;;) {
    data = ...; // the data to be sent
    r = stock_writer->write(data, handle);
    ...
}
...

```

The detail of how to create a data writer from a DDS publisher with a desired topic and type support is hidden behind the scene in the DDS application deployment model of

PocoCapsule/CORBA and therefore can be ignored by domain application developers. Same as an event supplier application, to decouple the application from the PocoCapsule/CORBA container, a data writer injection method with user specified name can be implemented for “dependency injection”. With this design, data emitter applications are usually implemented as component, namely a plain old C++ object, as in the following example:

```
class MyStockDataEmitterImpl
{
    StockDataWriter_var stock_writer;

public:
    // data writer injection method.
    void connect_stock_writer(DDS::DataWriter_ptr writer) {
        stock_writer = StockDataWriter::_narrow(writer);
    }

    void blabla() {
        Stock data = ...; // the data to be sent
        DDS::InstanceHandle_t handle;
        r = stock_writer->write(data, handle);
        ...
    }
    ...
};
```

In a deployment descriptor, as will be illustrated in a section next, PocoCapsule is able to create a data writer from a DDS publisher and inject it to the component above through the user specified injection method.

6.4 DDS Data Reader Listener Implementation

The DDS data receiving side applications use the so-called “*double dispatch*” pattern⁴³. Namely, a data receiving application component is actually implemented as a “*data reader listener*” that supports the *DDS::DataReaderListener* interface. This listener is subscribed to a DDS subscriber with the specified topic. Upon data available, the *on_data_available()* callback method of this listener will be triggered (forward dispatch) with a data reader reference (in type of *DDS::DataReader_ptr*) as input parameter. Listener can simply narrow this pointer to the typed data reader of the subscribed topic and pull data back from the service (reverse dispatch).

The following example illustrates an implementation of this kind of double dispatch data pulling listener component⁴⁴:

⁴³ It is also known as “visitor” pattern.

⁴⁴ This example assumes *DDS::DataReaderListener* is simplified as an IDL local interface. If it is not, the implementation should inherit from POA skeleton or delegated through TIE skeleton.

```

class MyStockDataReaderListenerImpl : public DDS::DataReaderListener
{
public:
    ...
    void on_data_available(DDS::DataReader_ptr reader) {
        // narrow the reader to typed reader
        StockDataReader_var stock_reader =
            StockDataReader::_narrow(reader);

        Stock data;
        DDS::SampleInfo info;

        // pull data until failure or data queue is empty
        for(;;) {
            r = stock_reader->take_next_sample(data, info);

            if( r !=DDS::RETURN_OK;) {
                return;
            }

            cout << "Stock: " << data.symbol.in() << endl;
            cout << "Quote: " << data.quote.in() << endl;
        }
    }
};

```

6.5 PocoCapsule DDS Application Deployment Model

The PocoCapsule DDS application deployment model is a DSM extension of the PocoCapsule/CORBA core schema and is defined in the *corba-dds-application.dtd* DTD file. The transformation templates from this DSM to the PocoCapsule/CORBA core schema are declared in the *dds2corba.xml* style sheet file. A DDS application deployment descriptor should:

- declare DOCTYPE *corba-dds-application* with the system id:
"http://www.pocomatic.com/corba-dds-application.dtd"
- declare transform process instruction with href equals to:
"http://www.pocomatic.com/dds2corba.xml"
- use `<corba-dds-application>` as the doc node.

This is illustrated in the following XML deployment descriptor skeleton:

```

<?xml ...?>
<!DOCTYPE corba-dds-application
  SYSTEM http://www.pocomatic.com/corba-dds-application.dtd>

<?xml-transform
  type="text/xsl"
  href="http://www.pocomatic.com/dds2corba.xsl"?>

<corba-dds-application>

...

</corba-dds-application>

```

Note: The *corba-dds-application* schema is an extension of *corba-application-context* schema, which in turn is an extension of the *poco-application-context* schema. Therefore, elements defined in these two schemas, such as *<bean>*, *<object>*, *<poa>*, *<event-consumer-adapter>*, etc., can all be used in the *corba-dds-application* descriptor explicitly.

In the PocoCapsule/CORBA core schema (namely the *<corba-application-context>*), the *<orb>* element is defined as (see section 3.3 of this document):

```

<!ELEMENT orb (args?,
  (object|poa|
  event-consumer-adapter|event-supplier-adapter|
  dds-participant)*)>

```

There, in the core schema, the *<dds-participant>* was declared as in the DTD segment above but was not defined. It is reserved for the extension *<corba-dds-application>* schema introduced in this chapter.

The *<dds-participant>* element is defined in the *corba-dds-application* schema as:

```

<!ELEMENT dds-participant (dds-participant-qos?,
  (bean|ioc|dds-topic
  |dds-data-reader|dds-data-writer
  |dds-subscriber|dds-publisher)*)>

<!ATTLIST dds-participant id ID #IMPLIED>
<!ATTLIST dds-participant domain CDATA #REQUIRED>

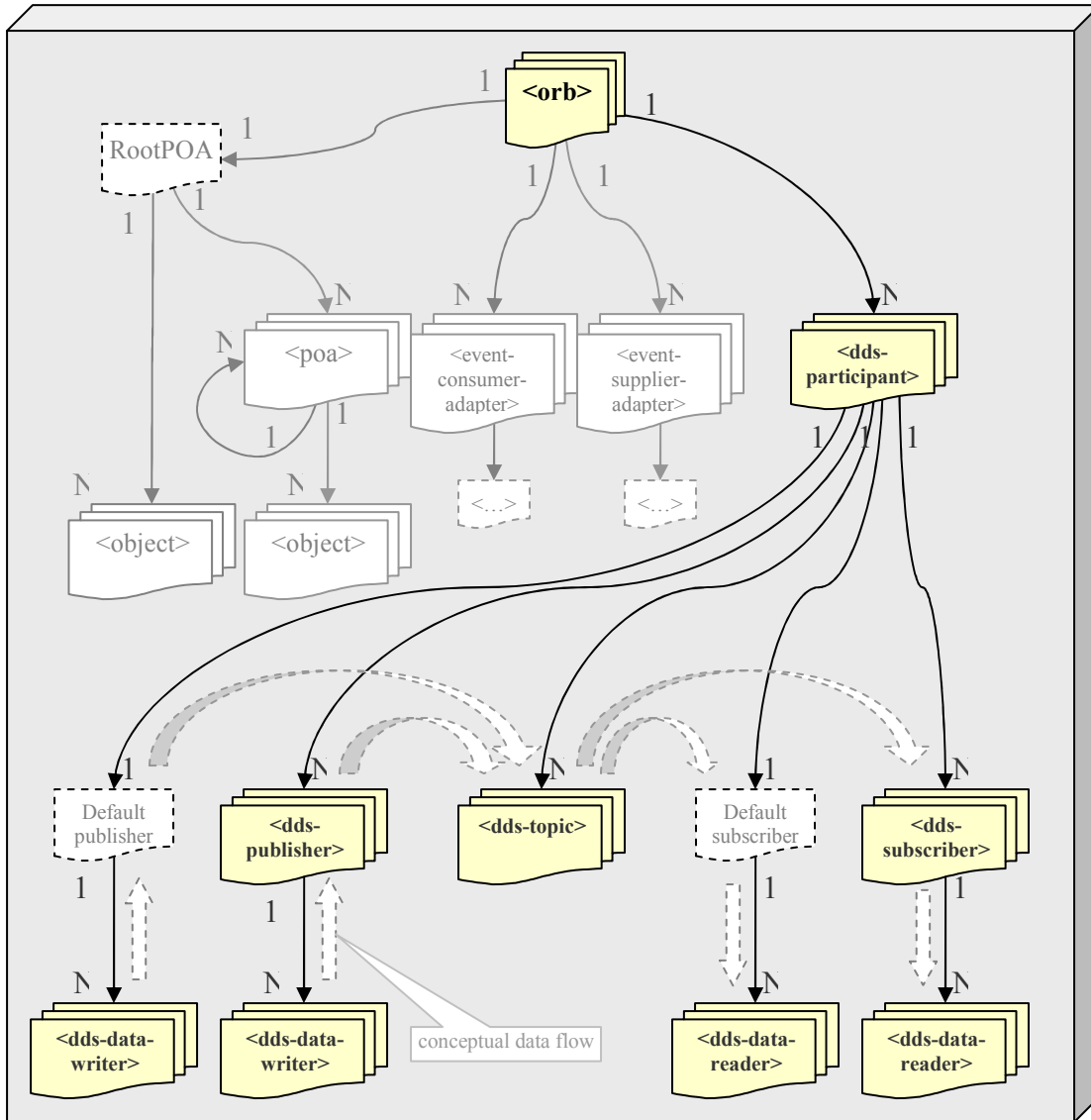
```

The DDS participant factory is hidden inside the framework. Therefore, the *<dds-participant>* is defined as a direct child of *<orb>* element. It is assumed that PocoCapsule/CORBA can resolve the DDS domain participant factory reference from the ORB with service id “*DataDistributionService*”⁴⁵.

⁴⁵ Therefore, if the DDS factory is embedded locally, it should be registered to the ORB using *orb’s register_initial_reference()* or *PortableInterceptor::ORBInitializer*. If the DDS is a remote service, its reference should be registered to the local ORB using *register_initial_reference()* or *-ORBInitRef* to

The `<dds-participant>` element declares a DDS domain participant to be instantiated on demand. The `id` attribute can be used to retrieve this instance, as `DDS::DomainParticipant_ptr` from the application context. The `domain` attribute specifies the domain id (a positive integer) of the participated data domain.

The `<dds-participant>` element introduces five new DDS specific elements, namely `<dds-topic>`, `<dds-data-reader>`, `<dds-data-writer>`, `<dds-subscriber>`, and `<dds-publisher>`. These elements and implied conceptual data flow are illustrated in the following schema element tree diagram.



The underlying DDS objects declared by these child elements were mentioned previously. Their usages are going to be described in the rest part of this chapter.

ORB_init(). With PocoCapsule/C++, it can also be done using `initDefaultAppEnv` on the application context before the deployment (see the examples/corba/dds example in PocoCapsule/CORBA distribution).

6.6 The `<dds-topic>` element

The `<dds-topic>` is a child element of `<dds-participant>`. Its schema is defined as follows:

```
<!ELEMENT dds-topic (dds-topic-qos?)>
<!ATTLIST dds-topic id          ID #IMPLIED>
<!ATTLIST dds-topic name       CDATA #REQUIRED>
<!ATTLIST dds-topic type       CDATA #REQUIRED
```

This element declares a DDS topic bean, of C++ type `DDS::Topic`, can be referred within the same descriptor or retrieved from the application context using the declared *id*. The topic name, specified as the *name* attribute, serves as a fine-grained data marker and discriminator, and will become clear in the next two sections. The *type* attribute specifies the DDS type name of this topic, namely the name of the IDL structure.

A topic declaration implicitly instantiates the DDS type support implementation⁴⁶ generated from DDS implementation, and registers it to the domain participant.

For example, the following description declares a DDS topic with user assigned name “*my-stock-topic*” of DDS type *Stock*. This topic and its type support will both be create/registered from/to the participant of domain 123.

```
<dds-participant domain="123">
  <dds-topic id="my-topic" name="my-stock-topic" type="Stock"/>
  ...
</dds-participant>
```

The `<dds-topic-qos>` child element of a `<dds-topic>` is declared but not defined. It is reserved for supporting QoS in later enhancement.

6.7 The `<dds-publisher>` and `<dds-writer>` elements

These two elements are subnodes of `<dds-participant>` and are used for setting up and deploying DDS data emitter applications. They are defined as:

⁴⁶ This detail can safely be ignored for domain developers. As specified by DDS specification, the DDS type support implementation for IDL structure *Stock* is the class *StockTypeSupportImpl*. A compliant DDS implementation will generate these classes for specified DDS types.

```

<!ELEMENT dds-publisher (dds-publisher-qos?, dds-listener?,
dds-data-writer*)>
<!ATTLIST dds-publisher id ID #IMPLIED>
<!ATTLIST dds-publisher listener IDREF #IMPLIED>

<!ELEMENT dds-data-writer (dds-writer-qos?, dds-listener?, ioc*)>
<!ATTLIST dds-data-writer id ID #IMPLIED>
<!ATTLIST dds-data-writer topic IDREF #REQUIRED>
<!ATTLIST dds-data-writer listener IDREF #IMPLIED>

```

- A **<dds-publisher>** element declares a DDS publisher which can be retrieved from the application context or referred within the same deployment descriptor by its id. The corresponded C++ type of this bean is *DDS::Publisher*.

A listener bean of type of CORBA object or POA servant can be declared for this publisher as a **<dds-listener>** subnode or referred by the *listener* attribute.

The primary purpose of this element is to support creating a DDS publisher with alternative user specified QoSs. The QoSs support will be added as an enhancement in later release.

- A **<dds-data-writer>** element declares a DDS data writer which can be retrieved from the application context or referred within the same deployment descriptor by its id. The corresponded C++ type of this bean is *DDS::DataWriter*.

A listener bean of type of *DDS::DataWriterListener* reference or *POA_DDS::DataWriterListener* servant can be declared under this data writer as a **<dds-listener>** subnode or referred by the *listener* attribute.

Arbitrary number of **<dds-data-writer>** elements can be declared as subnodes of **<dds-participant>** or **<dds-publisher>**. In the first case, an implicit **<dds-publisher>** will be created and used as parent node of declared **<dds-data-writer>** elements. This implicitly created DDS publisher will use default QoSs.

A **<dds-data-writer>** can also be declared with its alternative user specified QoSs. However, QoSs support will be an enhancement for the next release.

To have data emitter applications decoupled from the PocoCapsule framework, the “dependency injection” pattern should be used. Namely, while using PocoCapsule to create DDS data writers, the created data writers should be injected into data emitter applications through calling back setter methods or receptacle ports, as illustrated in the following example:

- A data emitter application is implemented as an arbitrary C++ class with an user defined data writer injection operation *connect_stock_writer()*:

```

class MyStockDataEmitterImpl
{
    StockDataWriter_var stock_writer;

public:
    // data writer injection method.
    void connect_stock_writer(DDS::DataWriter_ptr writer) {
        stock_writer = StockDataWriter::_narrow(writer);
    }

    void blabla() {
        Stock data = ...; // the data to be sent
        DDS::InstanceHandle_t handle;
        r = stock_writer->write(data, handle);
        ...
    }
    ...
};

```

- The deployment descriptor declares a data writer and an instance of emitter implementation class, with the data writer injected as a dependency:

```

...
<dds-topic id="my-topic" name="my stock topic" type="Stock"/>
...

<!-- declares a data writer -->
<dds-data-writer id="my-writer" topic="my-topic"/>
...

<!-- a data emitter -->
<bean class="MyStockDataEmitterImpl" lazy-init="false">
    <!-- inject the data writer declared above -->
    <ioc method="connect_stock_writer">
        <method-arg ref="my-writer"/>
    </ioc>
</bean>

```

6.8 The `<dds-subscriber>` and `<dds-reader>` elements

These two elements are used for deploying DDS data listener applications. They are defined as:

<!ELEMENT dds-subscriber	(dds-subscriber-qos?, dds-listener?, ioc*, dds-data-reader *)>
<!ATTLIST dds-subscriber	id ID #IMPLIED>
<!ATTLIST dds-subscriber	listener IDREF #IMPLIED>
<!ELEMENT dds-data-reader	(dds-reader-qos?, dds-listener ?, ioc*)>
<!ATTLIST dds-data-reader	id ID #IMPLIED>
<!ATTLIST dds-data-reader	topic IDREF #REQUIRED>
<!ATTLIST dds-data-reader	listener IDREF #IMPLIED>
<!ELEMENT dds-listener	(bean ref)?>
<!ATTLIST dds-listener	ref IDREF #IMPLIED>

- A `<dds-subscriber>` element declares a DDS subscriber which can be retrieved from the application context or referred within the same deployment descriptor by its id. The corresponding C++ type of this bean is `DDS::Subscriber`.

A listener bean of type `CORBA::Object` or POA servant can be declared for this subscriber as a `<dds-listener>` subnode or referred by the `listener` attribute.

The primary purpose of this element is to support creating a DDS subscriber with alternative user specified QoSs. The QoSs support will be added as an enhancement in later release.

- A `<dds-data-reader>` element declares a DDS data reader which can be retrieved from the application context or referred within the same deployment descriptor by its id. The corresponding C++ type of this bean is `DDS::DataReader`.

A listener bean of type `DDS::DataReaderListener` reference or `POA_DDS::DataReaderListener` servant can be declared under this data reader as a `<dds-listener>` subnode or referred by the `listener` attribute. This listener will be subscribed to the DDS service with the specified topic under the specified domain. The listener's `on_data_available()` method will be invoked by the service with a DDS data reader reference as input callback handler upon data available. The listener can narrow this data reader handler into the type specific data reader that is generated from DDS tool, and retrieves data from the typed reader using the data specific method `take_next_sample()`.

Arbitrary number of `<dds-data-reader>` elements can be declared as subnodes of `<dds-participant>` or `<dds-subscriber>`. In the first case, an implicit `<dds-subscriber>` will be created and used as parent node of the declared `<dds-data-reader>` elements. This implicitly created DDS subscriber will use default QoSs.

A `<dds-data-reader>` can also be declared with its alternative user specified QoSs. However, QoSs support will be an enhancement for next release.

To have data listener applications decoupled from the PocoCapsule framework, the “dependency injection” pattern should be used. Namely, while using PocoCapsule to subscribe DDS data writers, the created data writers should be injected into data emitter applications through calling back setter methods or receptacle ports, as illustrated in the following example:

- A data listener application is implemented DDS data reader listener:

```

class MyStockDataReaderListenerImpl
    : public DDS::DataReaderListener
{
public:
    ...
    void on_data_available(DDS::DataReader_ptr reader) {
        // narrow the reader to typed reader
        StockDataReader_var stock_reader =
            StockDataReader::_narrow(reader);

        Stock data;
        DDS::SampleInfo info;

        // pull data until failure or data queue is empty
        for(;;) {
            r = stock_reader->take_next_sample(data, info);

            if( r !=DDS::RETURN_OK;) {
                return;
            }

            cout << "Stock: " << data.symbol.in() << endl;
            cout << "Quote: " << data.quote.in() << endl;
        }
    }
};

```

- The deployment descriptor declares a data reader with an data reader listener as a subnode:

```

...
<dds-topic id="my-topic" name="my stock topic" type="Stock"/>
...

<!-- declares a data reader with a listener -->
<dds-data-reader id="my-writer" topic="my-topic"/>
  <!-- listener of this reader -->
  <dds-listener>
    <bean class="MyStockDataReaderListenerImpl"/>
  </dds-listener>
</dds-data-reader>

```

This data reader listener will be subscribed to the DDS service with the specified topic under the specified domain. The listener's *on_data_available()* method will be invoked by the service with a DDS data reader reference as input callback handler upon data available. As shown previously, the implementation of this listener narrows this data reader handler into the type specific data reader (the *StockDataReader*) generated from DDS tool and retrieves data from this typed reader using the method *take_next_sample()*.

A complete example of DDS data reader and writer application can be found in the `examples/corba/dds` directory of the PocoCapsule/C++ installation.

<this is an empty page>

Appendix A. SDR and JTRS-SCA

In a simplified and conceptual description, a software defined radio (SDR)⁴⁷ is a wireless communication device or equipment that, in order to quickly and easily support multiple and/or different radio bands/modes within a wide spectrum, uses reconfigurable software programs running on generic hardware/OSS⁴⁸ to perform certain or all radio signal processing.

To facilitate the quick composable and reconfigurable requirements, SDR waveform applications are commonly architected as composites wired out of software functional components. The de facto standard of such architecture is the Software Communications Architecture (SCA)⁴⁹ from US Navy Joint Tactical Radio System (JTRS), or its commercial adaptation the Software Radio Architecture (SRA)⁵⁰ from the SDR forum (SDRF). The core framework (CF) of this architecture defines a CORBA based component framework for SDR applications.

By the SCA component model, components of a single SDR waveform application are assumed to be distributed in multiple POSIX address spaces across processes or host boundaries. Therefore, SDR components are simply defined as conventional CORBA objects that can be invoked remotely. The OMG Name Service and Event Service are defined as the distributed component registry used by the SCA assembly controller and the SCA distributed publish/subscribe service respectively.

Although modeling a single SDR waveform application as a composite wired up from distributed components is claimed to be necessary, it is one of the major obstacles for mainstream adoption of SDR. For radio wireless devices that have all their components resided in a single address space, a CORBA middleware layer is not a silver bullet but only an unnecessary burden of footprint, power consumption, learning curves, and excessive engineering complexities, product vulnerabilities, costs, and risks. This observation has prompted some practitioners to seek alternative component architectures for SDR waveforms without CORBA, for instance by directly using conventional non-CORBA C++ classes developed manually or generated automatically from the middleware platform independent model (PIM) UML diagrams of SCA specification⁵¹.

Therefore, to receive acceptance of mainstream defense and commercial users, the component framework for SDR waveform applications has to take a significantly step

⁴⁷ http://en.wikipedia.org/wiki/Software-defined_radio

⁴⁸ Such as POSIX platforms.

⁴⁹ <http://jtrs.spawar.navy.mil/sca>

⁵⁰ <http://www.sdrforum.org>, <http://www.omg.org/docs/sdo/00-12-05.pdf>

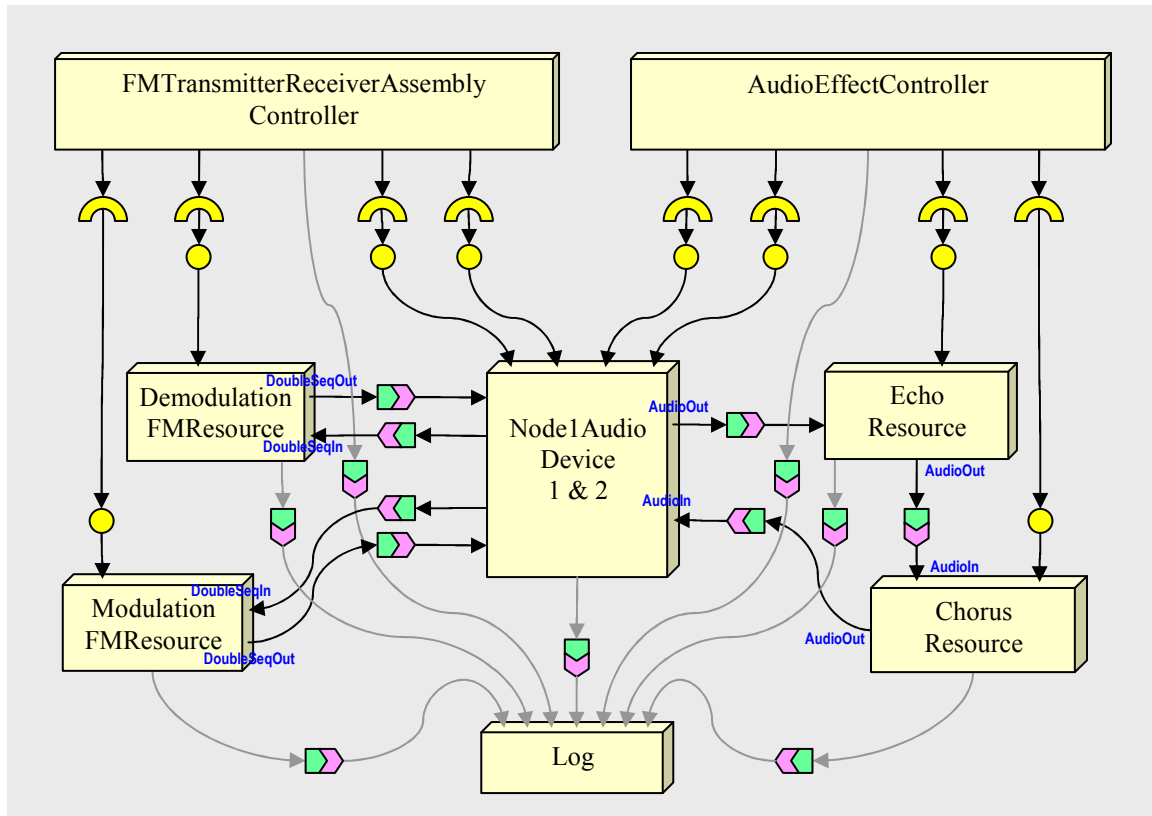
⁵¹ This transition also appeared in the Parlay/OSA (<http://www.parlay.org/en/specifications/>). Early CORBA based Parlay specifications were CORBA based and has been recognized to have unnecessary overhead and complexities. Later Parlay/OSA is changed to UML based, with the CORBA/IDL as one of its mappings.

forward beyond JTRS-SCA and CORBA. The next generation framework should meet the following four criteria:

- ***Neutral to heterogeneous component models***: The new component framework should not only be orthogonal and independent to middleware platforms, but more importantly not enforce a particular component model. For instance, it should seamlessly support plain old user defined C/C++ objects as SDR components as well as JTRS-SCA components (e.g. *CF::Resource* objects) based on CORBA (or other PSM mappings of JTRS-SCA).
- ***Simple core assembly model/schema***: The new component framework should not incur steep learning curve, long ramp up time, and high development and maintenance cost. The core model/schema itself should be substantially simple, intuitive, and straightforward to application developers and novices, without relying on any UI and/or other massive code generating tools.
- ***Easily extensible and customizable assembly schema***: Instead of mandating a single rigid “one-size-fit-all” model/schema at a predefined abstraction level or scheme, the core assembly model/schema should be able to be extended and customized not only by framework vendors but more importantly by domain application developers. Domain specific assembly models/schemas, including the JTRS-SCA assembly schema, should be merely domain specific languages (DSMs) realized on top of the core model/schema in a matter of few hours with few hundreds lines of declarative code by domain experts instead of weeks or even months of efforts and thousands lines of code by framework vendors.
- ***Lightweight and zero overhead***: The extra power consumption and IO overhead should be significantly reduced to near zero. The memory footprint should be one or two orders of magnitude smaller than the first generation SCA CF implementations available today. Namely, the footprint should be reduced from one or several megabytes to, for instance, less than 100Kbytes.

PocoCapsule/C++ is able to meet all these criteria. A classic SDR example from the SCA reference implementation (SCARI)⁵² by CRC is used to demonstrate these characteristics and methods. In this example, a SDR waveform is wired up from 8 components using 22 connections, as illustrated in the following UML 2.0 and CCM component diagram:

⁵² <http://www.crc.ca/scari>



This waveform is implemented in different component models and assembled in various schemas, including the component and assembly models of JTRS-SCA, as illustrated in the following three schemes:

- ***Plain old C++ components, and the PocoCapsule/C++ IoC core assembly schema***⁵³:
 - In this non-CORBA waveform application scheme, services of a given plain old C++ component are exported as pointers of callback C++ objects returned from getter methods of the given provider component.
 - Then, to wire up components, these pointers are injected into their user components through their setter methods.
 - These wirings are declared in the IoC schema and are set up by the PocoCapsule/C++ container accordingly.

Using the PocoCapsule/C++ core IoC schema:

- A SDR component instance is declared by a *<bean>* element with the desired C++ class name.
- Retrieving a service object pointer (of type *SDR::PushPort**) from a provider component is declared as a *<bean>* element with the name of getter method as the value of the *factory-method* and the *id* of the provider component bean as *factory-bean*.

⁵³ See descriptions in the “PocoCapsule/C++ IoC&DSM Developer Guide”.

- Injecting a service object pointer (of type *SDR::PushPort**) into a user component is declared as a *<ioc>* child element of the user component, with the setter name as the value of the *method* attribute.

An example in this scheme can be found in the examples/basic-ioc/sdr directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

- ***Plain old C++ components, and a DSM assembly schema:***

- Similar to the previous scheme, the services provided by SDR components are exported (of type *SDR::PushPort**) through getter methods of provider components.
- Similarly, these services (of type *SDR::PushPort**) are injected to user components through their setter methods.
- These wirings are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only *<component>* and *<wire>* elements in the see the DTD file⁵⁴:

- A *<component>* element declares a SDR component instance of the specified C++ class.
- A *<wire>* child element of a *<component>* declares a wiring between the specified *receptacle* port of this component and the specified *facet* port of a specified service *provider* component.

An example in this scheme can be found in the examples/basic-ioc/dsm-sdr directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

- ***JTRS-SCA CF CORBA components model and JTRS-SCA software assembly schema:***

- In this scheme, the JTRS-SCA core framework component model is applied. Components are simply CORBA objects supporting the *CF::Resource* IDL interface.
- Services (of type *CF::Port*) provided by these components are also CORBA objects and their references can be retrieved from the JTRS-SCA standard *CF::Resource::getPort(in string service_name)* IDL method on these components.
- These service object references are connected to their user components through the *CF::Port::connectPort(in Object service, in string connection_id)* IDL method of these user components.
- These connections are declared in a JTRS-SCA software assembly descriptor (SAD) and are set up by the PocoCapsule/C++ container accordingly. In fact, the SAD schema is handled also as a DSM schema in PocoCapsule/C++ in less than 150 lines of declarative code (XSLT style sheet).

⁵⁴ The examples/basic-ioc/dsm-sdr/sdr-device.dtd in the PocoCapsule/C++ installation directory

An example in this scheme can be found in the `examples/corba/jtrs-sca` directory of the PocoCapsule/C++ installation. This example is originated from a SCARI example, with mocked service implementations.

Similarly, other SDR component framework schemes (namely combinations of component model and assembly schema) can easily be supported in PocoCapsule/C++ as user or third party defined DSMs. These DSMs can be defined directly on top of the PocoCapsule/C++ core IoC scheme, or progressively on top of other high level DSM schemes.

<this is an empty page>

Appendix B. Robotic Component and OMG-RTC

Driven by the ever increasing complexities against the constant requirements of quick development, flexible integration, and low total lifecycle costs, it becomes more and more desirable that robot software systems to be built from reusable components. Robotic systems are inherently component-based at the hardware layer. Therefore, there are far less hesitations and far more practices on pursuing “component-based development” at the software layer in the robot community than in others, indicated by the overwhelming number of robotic component collections, toolkits, and frameworks, such as Player/Stage⁵⁵, CARMEN⁵⁶, OROCOS⁵⁷, ORCA⁵⁸, MARIE⁵⁹, CLARAty⁶⁰, MIRO⁶¹, Microsoft Robotics Studio⁶², JAUS⁶³ implementations⁶⁴, as well as the OMG robotic technology component (RTC) specification⁶⁵ and its experimental implementation the OpenRTM-aist⁶⁶.

The productivity in developing a given robotic application and the customizability and extendibility in deploying and maintaining it are not solely decided by how well robot monitoring and controlling functions are factored in separate components achieved in the component collections or toolkits above, but also heavily rely on how effective and flexible these components are able to be assembled back together into a seamlessly collaborated system and deployed in a service adapting environment. Although some frameworks listed above do offer assembly solutions⁶⁷ applicable to their particular component models, robotic applications are mostly assembled, deployed, and configured manually (using APIs and/or proprietary configuration files). Therefore, it is desirable to have a common deployment framework that meets the following criteria:

- ***Neutral to heterogeneous component models***: The diverse needs of heterogeneous robotic systems and the assets of the vast existing robotic component collections, toolkits, and frameworks rule out any attempt of reinventing a single dominated robotic component standard model. Therefore, the new component framework should

⁵⁵ <http://playerstage.sourceforge.net>

⁵⁶ <http://carmen.sourceforge.net>

⁵⁷ <http://www.orocos.org>

⁵⁸ <http://orca-robotics.sourceforge.net>

⁵⁹ <http://marie.sourceforge.net>

⁶⁰ <http://claraty.jpl.nasa.gov>

⁶¹ <http://www.informatik.uni-ulm.de/neuro/310.html>

⁶² <http://msdn.microsoft.com/robotics>

⁶³ <http://www.jauswg.org>. Instead of as a component framework, JAUS should be more appropriately categorized as a message framework.

⁶⁴ <http://openjaus.com> and <http://www.resquared.com/JAUS-SDK.html>

⁶⁵ <http://www.omg.org/docs/ptc/06-11-07>

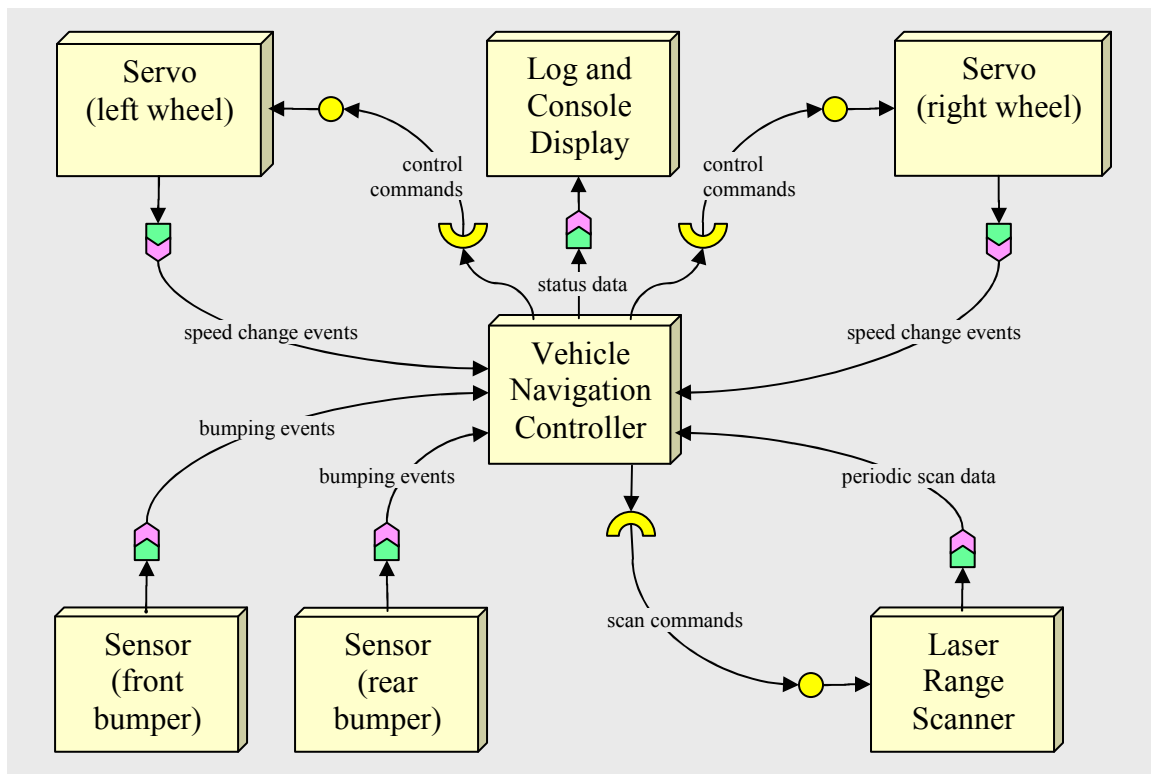
⁶⁶ <http://www.is.aist.go.jp/rt/OpenRTM-aist>

⁶⁷ Such as the XML deployment of OROCOS, and the OMG-D&C for OMG-RTC.

not only be orthogonal to middleware platforms, but more importantly be neutral to component models without enforcing a particular one.

- **Simple core assembly model/schema:** The new component framework should not incur steep learning curve and high development and maintenance cost. The core model/schema itself should be simple, intuitive, and straightforward to application developers and even novices, without relying on additional UI and/or other massive code generating tools.
- **Easily extensible and customizable assembly schema:** Instead of mandating a single rigid “one shoe fit all” predefined abstraction level, the core assembly model/schema should be able to be extended and customized not only by framework vendors but more importantly by domain application developers. Domain specific assembly models/schemas should be merely domain specific languages (DSMs) realized on top of the core model/schema in a matter of few hours with few hundreds lines of declarative code by domain experts instead of weeks or even months of effort and thousands lines of code by framework vendors.
- **Lightweight and zero overhead:** The component framework should be able to fit in the most rigorous real-time embedded environments and should not introduce performance overhead or real-time latency and jitter.

PocoCapsule/C++ is able to meet all these criteria. A robotic vehicle example from Microsoft Robotics Studio is used to demonstrate these characteristics and methods. In this example, a robotic vehicle motion control application is wired up from 7 components using 9 connections, as illustrated in the following diagram:



This robotic vehicle control system is implemented in two different component models and assembled in two different schemas, including the OMG-RTC model, as illustrated in the following two schemes:

- **Plain old C++ components and a DSM assembly schema⁶⁸:**
 - In this non-CORBA scheme, controllable ports and event sinks of a given plain old C++ component are exported as pointers of callback C++ objects returned from getter methods of the given provider component.
 - Then, to wire up components, these pointers are injected into controller or event emitter ports of the paired components through their setter methods (namely, the *receptacles*).
 - These wirings are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only `<component>`, `<controls>`, and `<listens>` elements in the DTD file⁶⁹:

- A `<component>` element declares a robotic component instance of the specified C++ *class*.
- A `<controls>` child element of a `<component>` declares a wire from the specified controller port (*receptacle*) of this component to the specified controllable port (*facet*) (of the specified *type*) of a *device* component.
- A `<listens>` child element of a `<component>` declares a wire from the specified event *sink* port of this component to the specified *receptacle* port of an event *emitter* component.

An example in this scheme can be found in the `examples/basic-ioc/robot-vehicle` directory of the PocoCapsule/C++ installation.

- **OMG-RTC CORBA components model and a DSM assembly schema:**
 - In this scheme, the OMG-RTC component model is applied. Components are simply CORBA objects with IDL interfaces extended from `RTC::LightweightRTOObject`.
 - By OMG-RTC, services (*facets*) provided by RTC components are also CORBA objects. A service *facet* reference can be retrieved from the given component using the `provide_<facet_name>()` IDL method.
 - By OMG-RTC, event *sinks* of RTC components are also CORBA objects. A *sink* reference can be retrieved from the given component using the `get_consumer_<sink_name>()` IDL method.
 - By OMG-RTC, a service or event sink object reference can be connected/subscribe to its user/emitter components through the `connect_<receptacle_name>(in Type ref)` IDL method of its user components.
 - These connections are declared in a quite descriptive DSM schema and set up by the PocoCapsule/C++ container accordingly.

⁶⁸ See descriptions in the “*PocoCapsule/C++ IoC&DSM Developer Guide*”.

⁶⁹ The `examples/basic-ioc/robot-vehicle/robotic-application.dtd` file in the PocoCapsule/C++ directory.

This DSM schema is defined on top of the PocoCapsule/C++ core IoC schema with only `<component>`, `<uses>`, and `<listens>` elements in the DTD file⁷⁰:

- A `<component>` element declares a robotic component instance of the specified C++ *class*.
- A `<uses>` child element of a `<component>` declares a wire from the specified *receptacle* of this component to the specified *facet* port (of the specified *type*) of a service *provider* component.
- A `<listens>` child element of a `<component>` declares a wire from the specified event *sink* port of this component to the specified *receptacle* port of an event *emitter* component.

Either of the two deployment schemes above only requires a DSM schema definition and its transformation XSLT style sheet. This accounts for less than 150 lines of declarative code that can be created in less than an hour by a domain expert, comparing to potentially hundreds or thousands lines of code and days or weeks effort of a framework experts if the assembly/deployment engine was developed manually from scratch.

Other robotic component framework schemes (namely combinations of component model and assembly schema) can easily be supported in PocoCapsule/C++ as user or third party defined DSMs. These DSMs can be defined directly on top of the PocoCapsule/C++ core IoC scheme, or progressively on top of other high level DSM schemes.

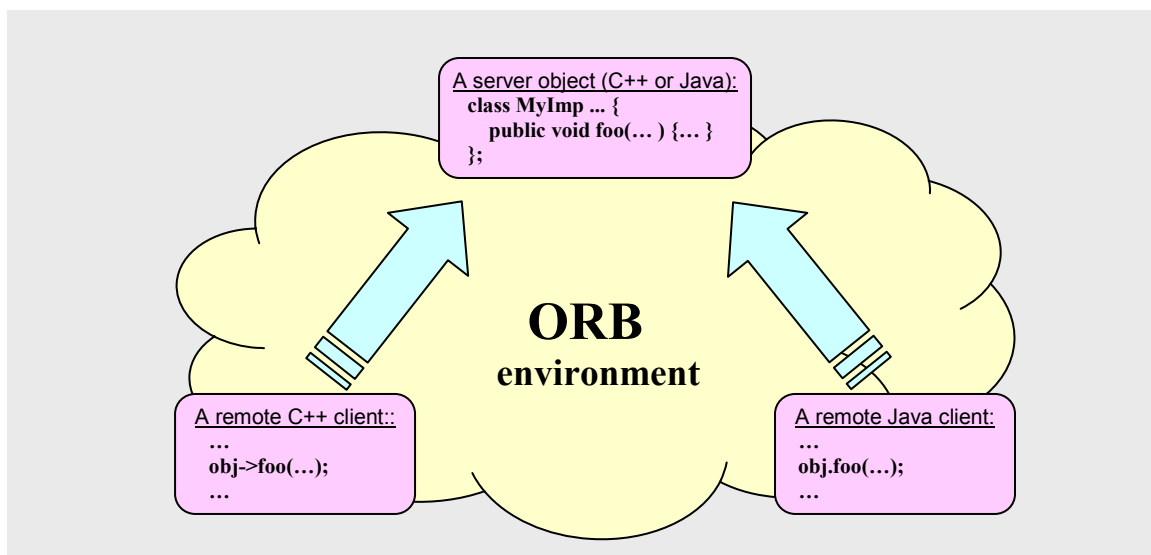
⁷⁰ The examples/corba/rtc/rtc-application.dtd file in the PocoCapsule/C++ directory.

Appendix C. CORBA Overview

C.1 CORBA Architecture

CORBA is not fundamentally different from other client/server Remote Procedure Call (RPC) middleware architectures, such as DCE RPC, JavaRMI, and SOAP based RPC (such as JAX-RPC). The core of CORBA is the Object Request Broker (ORB). An ORB is a software engine to present a virtual location transparent environment for distributed applications. In this ORB virtual environment, distributed applications can directly invoke methods on remote objects as if they were in the local program addressing space. In the view of the application level object method invocation, an ORB runtime environment can be considered as a virtual object addressing space that spans across traditional POSIX process addressing spaces and network nodes.

This vision of a CORBA ORB runtime environment is illustrated in the following diagram:



In this picture, an instance of the CORBA object implementation class MyImpl is located in a server node. Clients written in C++ or Java are able to invoke the remote object server's foo() operation using a local invocation syntax, as illustrated below:

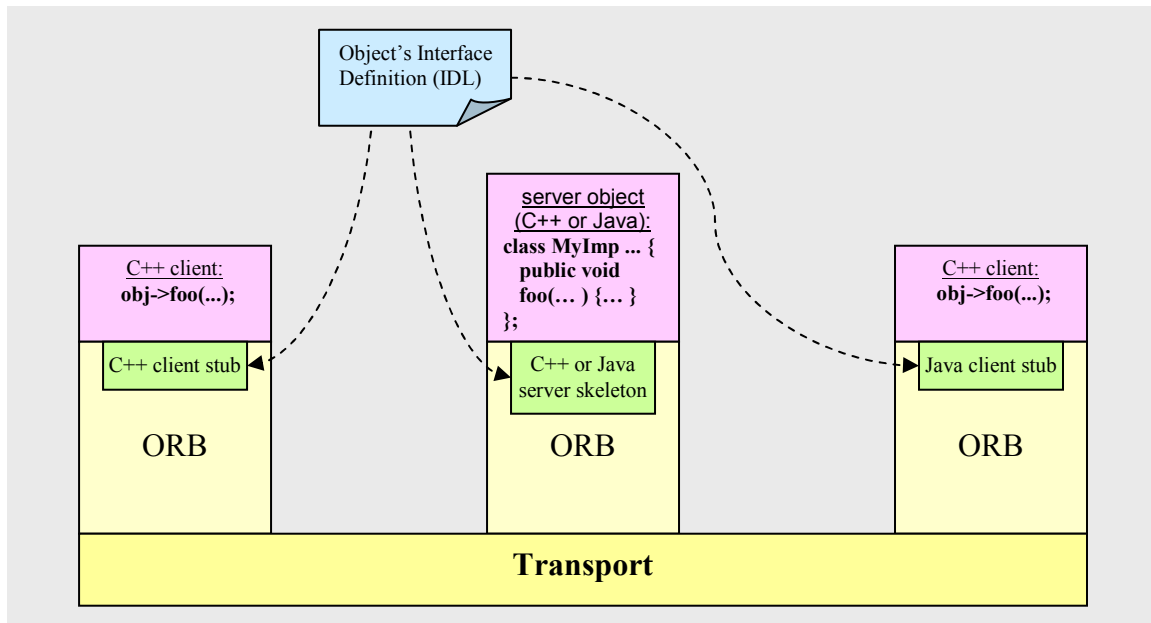
```
obj->foo(...); // in a C++ client
```

or, in java:

```
obj.foo(...); // in a Java client
```

The variable *obj* in the example above is a client side local delegate of the remote object, and is referred to as a *stub* in CORBA terminology.

In the view of the Client/Server RPC, the CORBA architecture is illustrated in the following figure:



The figure above illustrates that the invocations made on client stubs are forwarded to the server by the ORB runtime engine. It also shows that the client stubs and server skeletons are generated from a user designed interface definitions (IDL).

C.2 Characteristics of CORBA Architecture

Some characteristics of CORBA architectures are summarized below:

- **OS neutral:** CORBA is an application level framework. Therefore, it supports application interconnection across heterogeneous operating system environments (e.g. Unix/Linux, various RTOSs, Windows, JVM, .NET, and mainframes).
- **Programming Language neutral:** A client application can transparently invoke methods on a remote CORBA object that is implemented in a different programming language.
- **Vendor neutral and transparent interoperability:** A client application can transparently invoke methods on a remote CORBA object that is implemented using a third party ORB runtime.

- **Portable client and server programming models:** Application clients or servers are either source code level portable (for C++) or binary level portable (for Java) to an ORB runtime from a third party.
- **Transparent connection management:** CORBA client and server applications are shielded from the system logic of establishing, repairing, and managing client/server transport connections and server listener ports. The underlying ORB runtime engines are able to use suitable transports (e.g. TCP, shared memory), connection sharing strategies, and connection open/recycling strategies, all behind the scene.
- **Transparent and customizable server thread management:** CORBA server applications are shielded from thread management. The ORB runtime can transparently dispatch and process requests concurrently.

Typical CORBA implementations also support the following features:

- **Customizable server QoS models:** CORBA server can be configured to support different request processing scenarios, object lifespan policies, thread models, etc..
- **Distributed publish/subscribe event/notification:** CORBA supports distributed publish/subscribe applications, via OMG Event/Notification service and/or OMG Data Distribution Service (DDS).
- **Distributed transaction management:** CORBA supports distributed transactions, via OMG Transaction Service.
- **Name Service:** A distributed directory service, for application to resolve programming meaningful object references from business meaningful names.

C.3 CORBA Object Interfaces

The concept of “*interface*” in CORBA is same as that in other common object oriented environments. An *interface* of an object specifies a set of method signatures supported by this object. A method (operation) signature specifies its operation name, return value and parameter types, directions, and exceptions.

In CORBA, business logic object interfaces are specified by developers using the programming language neutral *Interface Define Language*, namely the *OMG IDL*. The client stub and the server skeleton of an object interface can be generated for different programming language using ORB development tools.

Instead of saying “*interface of a server*”, we used the term “*interface of an object*”. This is because that CORBA objects are mostly defined by application developers as business logic objects. A CORBA server can host multiple objects (could be a large number of). This is similar to that a web server can host multiple (could be a large number of) different user provided web services/pages.

C.4 CORBA Object References

The concept of “*object reference*” in CORBA extends the same concept of traditional Object oriented programming environments. CORBA object references (of nonpseudo, non-local objects) have the following characteristics:

- **Object references are local delegates of their remote objects:** a client invokes a method of a remote object by invoking the same method on its local reference, as if the object were located locally. This invocation will be forwarded to the remote (or local) target object implementation transparently.
- **Object references are serializable:** Object references can be used as parameters or return values of CORBA object method invocations, and therefore, can be transferred from one programming addressing space and environment (OS and programming language) to another remotely.
- **Object references are externalizable:** Object references can be converted to normal text string. Object references can be restored from these text strings at another time, location, and environment programming environment (OS, programming language). For instance, one can stringify an object reference in a Java application as:

```
String ior = orb.object_to_string(ref);
```

Then, a C++ client application can de-stringify this string back into a reference stub using the ORB's `string_to_object()` method.

- **Object references have no relevance to the activeness of their target object implementations:** Object references can be created, serialized (transferred), and externalized (to text strings), without instantiating their object implementations and even without starting their implementation server processes. ORB has mechanisms (e.g. servant manager and implementation repository) to incarnate implementation instances or even to ignite the server processes on arriving of client requests.
- **Object references have no relevance to existence of the target object implementations, and vice versa:** An invocation made on an object reference, whose object implementation is no longer in existence, will raise a runtime exception. Object references do not monitor the aliveness of their objects. Server objects do not keep count of their referencing clients either.

Besides:

- **Object references are not transport connection endpoints:** even through programming entities (stubs) of object references may encapsulate transport connection end points (such as sockets) transiently.
- **Object references are not pointers pointing to their object implementations:** even through programming entities (stubs) of object references may encapsulate such kinds of pointers in case of co-located implementations.