Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

DIMMA Design and Implementation

DIMMA Team

Abstract

This document describes the design and implementation of DIMMA, a real time, multi-media enabled ORB developed by APM Ltd. under the ANSA Phase III research program.

The document is intended to act as an introduction to the DIMMA code by providing a high level overview of the design, and by explaining the principle behind the key design decisions.

APM.2063.01	**Approved**	30th September 1997
	Technical Report	

Distribution:
Supersedes:
Superseded by:

Copyright © 1997 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

DIMMA Design and Implementation

DIMMA Design and Implementation

DIMMA Team

APM.2063.01

30th September 1997

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright „ 1997 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

APM Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

5 1 Introduction
5 1.1 Scope
5 1.2 Audience
5 1.3 Related documents

7 2 Overview
8 2.1 Structure of document

9 3 The CORBA Personality
9 3.1 Overview
9 3.2 Jet Computational API
9 3.2.1 The CORBA class and related functions
10 3.2.2 Constructed Types
13 3.3 Trader Interface
14 3.3.1 Bind
14 3.3.2 Export
15 3.4 IDL Compiler
15 3.4.1 Overview
15 3.4.2 Support for Flows
15 3.4.3 Code Generation
17 3.5 Generated Code
17 3.5.1 Classes
18 3.5.2 Files
19 3.5.3 Content
22 3.6 Engineering
23 3.6.1 Generic Object References
25 3.6.2 Marshalling

27 4 The ODP Library
27 4.1 Computational API
27 4.1.1 C++ Language mapping
28 4.1.2 Signatures
28 4.1.3 Invocation references
29 4.1.4 Objects
30 4.1.5 Interfaces
31 4.1.6 Arguments
31 4.1.7 Terminations
32 4.1.8 Results
32 4.1.9 Operations
32 4.1.10 Invocations
33 4.1.11 Constructed Types
33 4.1.12 Class relationships
34 4.1.13 IDL Compiler and Stubs

36 4.1.14 Class hierarchy of an application
38 4.2 Engineering
38 4.2.1 Invocation References
38 4.2.2 Exceptions

41 5 Engineering API
41 5.1 Overview
41 5.2 Generic Stubs
42 5.3 Marshalling
42 5.4 Binders and Bindings
43 5.5 Interface References

45 6 Binding
45 6.1 Overview
45 6.1.1 Creating an interface reference
45 6.1.2 Establishing a binding
45 6.1.3 Components of a binding
45 6.2 Implicit Binding
46 6.2.1 Server side
46 6.2.2 Client side
47 6.3 Explicit Binding
47 6.3.1 Endpoints
48 6.3.2 Explicit binders
49 6.4 IIOP Binder
49 6.4.1 IIOP QoS
51 6.4.2 IIOP Binder Operation
52 6.5 AnsaFlow Binder
52 6.5.1 AnsaFlowQoS
53 6.5.2 AnsaFlow Binder Operation

55 7 Buffers and Marshalling
55 7.1 Overview
55 7.2 Buffers
56 7.3 Marshallers
56 7.4 Design notes

59 8 Resource Framework
59 8.1 Overview
59 8.2 Factories
59 8.3 Pools

61 9 Communications Framework
61 9.1 Purpose
61 9.2 Overview
62 9.2.1 Message processing
62 9.2.2 Network interface
62 9.2.3 Threading model
62 9.2.4 Resourcing
62 9.3 Protocol Construction
64 9.3.1 Protocol
65 9.3.2 Module
65 9.3.3 Channel

66 9.3.4 Protocol Binding
66 9.4 Invocation Concurrency
66 9.5 Message Concurrency
67 9.6 Dynamic Protocol Loading
67 9.6.1 Native Protocols
67 9.6.2 Adding new Protocols
67 9.6.3 Protocol Loading
68 9.6.4 Protocol selection
68 9.7 IIOP Implementation
68 9.7.1 TCP Layer
69 9.7.2 IIOP layer
70 9.7.3 Client Session Layer
71 9.7.4 Complete Client-side Protocol Stack
72 9.7.5 Server-side Protocol Stack
73 9.8 Flow Implementation
73 9.8.1 UDP Layer
74 9.8.2 RTP Layer
75 9.8.3 Binding Layer
76 9.8.4 Complete Protocol Stack

79 10 Threading
79 10.1 Overview
79 10.2 Tasks
80 10.2.1 Tasks and Resource Pools
80 10.3 Lightweight Threads
81 10.4 Null Threads
81 10.5 Single-threaded DIMMA

83 11 Locking
83 11.1 Locking model
83 11.1.1 Class Mutex
83 11.1.2 Class Condition
83 11.1.3 Class RecursiveLock
83 11.2 Synchronisation model
84 11.2.1 Class Synchronisable
84 11.2.2 SynchronisedObject macro
84 11.2.3 SynchronisedClass macro

85 12 Trader

1 Introduction

This is the DIMMA Design and Implementation document which is intended to provide a description of the design and implementation of the **D**istributed **I**nteractive **M**ulti**M**edia **A**rchitecture (DIMMA) ORB.

1.1 Scope

The document provides an overview of the structure of the DIMMA nucleus, describes the interfaces that DIMMA offers to the external world, and provides high level design descriptions of the components that make up the DIMMA nucleus.

The document reflects DIMMA release 2.01.

1.2 Audience

The document is targeted at a technical audience. who wish to understand the internal design, and design rationale for the DIMMA implementation.

It is assumed the reader is familiar with:

- the RM-ODP standard for Open Distributed Processing [ISO/IEC 95]
- . Many of the concepts and terminology used within DIMMA originate from these standards;
- the CORBA 2.0 standard [OMG 95]. The principle interface offered by this version of DIMMA is an extended subset of the CORBA 2.0 standard.

1.3 Related documents

It is recommended that the document “An Introduction to DIMMA” [APM.1995] is read *before* this document.

Application programmers wishing to make use of the facilities offered by DIMMA should make use of:

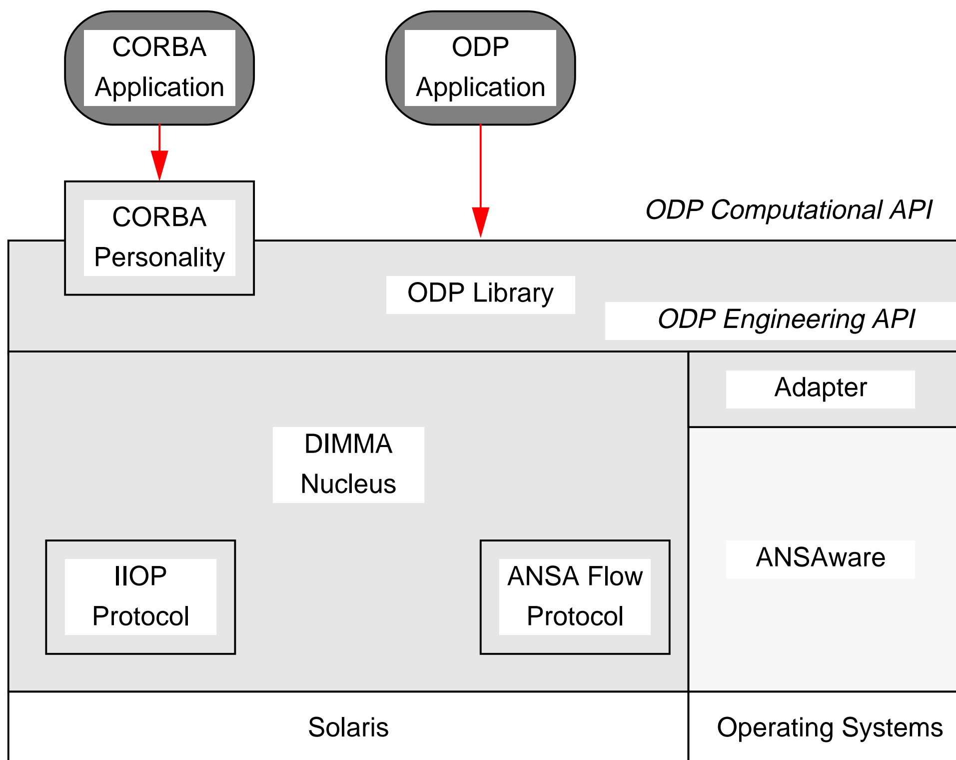
- Writing a DIMMA Application [APM.2037]
- DIMMA Tracing [APM.1980]
- Building DIMMA 2.0 [APM.2036]

2 Overview

DIMMA is constructed as a set of small components which may be combined in many different ways to suite the diverse needs of applications and in recognition of that fact that the real world needs Distributed Processing Environments (DPEs) that are high performance, downsizeable and scalable. In this sense, DIMMA may be regarded as a microkernel DPE.

The set of components are considered to be collected into groups and arranged in layers; this is also reflected in the source structure. The layering is depicted in figure 2.1

Figure 2.1: DIMMA Layered Architecture



DIMMA provides two application programming interfaces (APIs): a CORBA extended subset called Jet and a proprietary one based on the ODP-RM concepts ([ISO/IEC 95]) called the ODP Computational API. Jet is implemented as a *personality* built on top of the ODP facilities and hence shares a number of the former's features.

Both APIs are mapped onto a common ODP Engineering API by the ODP Library in order to facilitate hosting on different implementations of the Nucleus. The DIMMA nucleus supports the ODP Engineering API directly, whilst adapters must be provided for other Nuclei.

To date only one adapter has been produced and this was to allow applications written using Jet or ODP to communicate over ANSAware. The ANSAware adaptor was used to support the ODP Engineering API, during development of the DIMMA nucleus. It is no longer supported in DIMMA release 2[1]

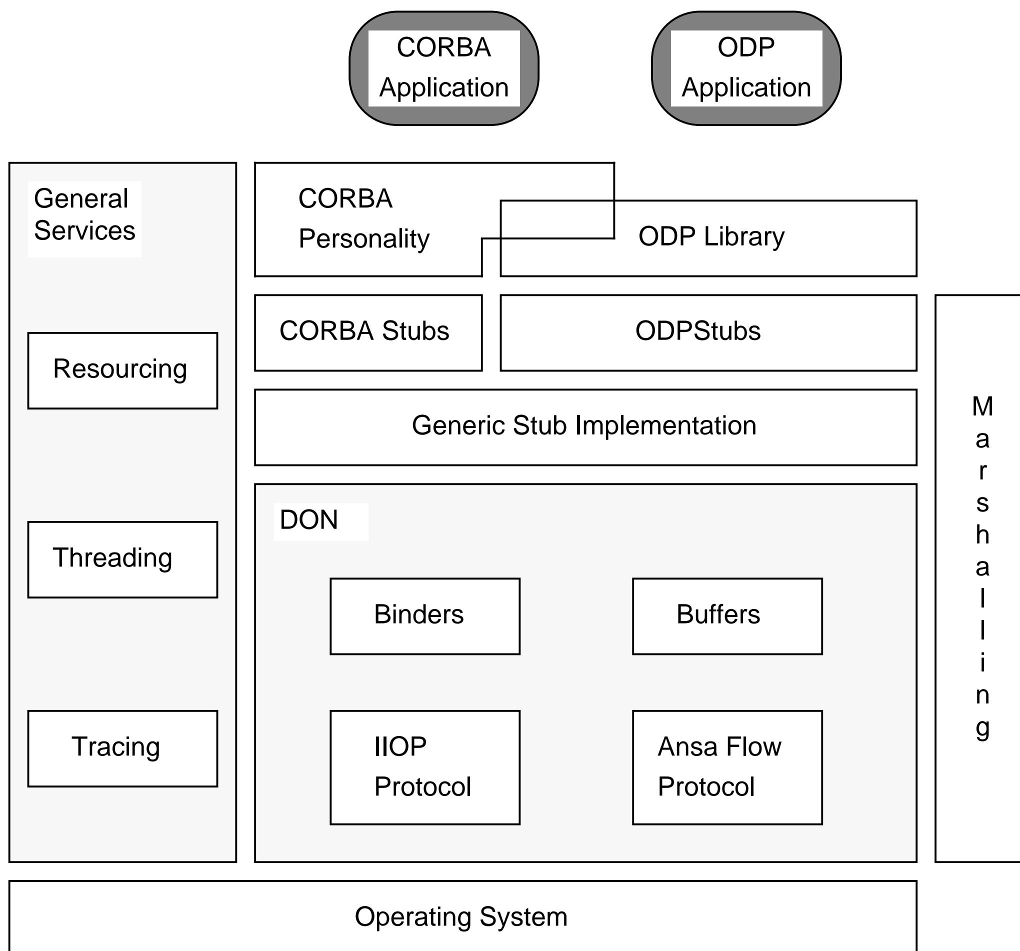
The DIMMA nucleus provides the distribution engineering apparatus such as binders and communications protocols. Currently two protocols are provided: the CORBA internet interoperability protocol (IIOP) for interoperability with other vendors ORBs and a proprietary protocol (ANSA Flow), optimised for transporting multi-media flows.

The DIMMA nucleus must be hosted on an operating system able to provide the necessary soft real-time facilities, for example Posix threads [POSIX]. The current release of DIMMA runs over Solaris 2.5.

2.1 Structure of document

This document is organised as chapters each of which describe the major components of DIMMA: these components are illustrated in figure 2.2

Figure 2.2: DIMMA Architecture - Components



1. Support for the ANSAware adaptor was not seen to be a critical requirement for DIMMA 2.0. It would be feasible to re-introduce support for this adaptor if required.

3 The CORBA Personality

3.1 Overview

The CORBA personality of DIMMA ([OMG 95]), called Jet, consists of three main parts:

- The computational API.
- The CORBA IDL compiler.
- Engineering - interaction with the underlying DIMMA nucleus.

The API comprises support for CORBA that is not automatically generated by the IDL compiler. This includes the CORBA class, with associated types and methods, plus support for CORBA types and exceptions.

The IDL compiler uses SunSoft's implementation of the compiler front end (CFE) for the OMG Interface Definition Language, release 1.3. This implementation as supplied by SunSoft consists of:

- A main program for driving the compilation process
- A parser and attendant utilities
- An empty code generator or back end, to be supplied by the user. This back end takes the processed input and produces output appropriate to the target DPE engineering and in the required programming language.

The major part of the DIMMA work on the IDL compiler has been in producing a back end to support all the features required by DIMMA. This primarily includes client and server stubs, and support for types. Some changes have also been made to the front end to support the addition of flows.

The API engineering cannot be entirely separated from the other two areas, and comprises mainly the distribution aspects of Jet.

Jet comprises a subset of CORBA functionality, sufficient to exploit the various components of DIMMA. For a description of which aspects of CORBA are supported, refer to [APM.2037].

3.2 Jet Computational API

3.2.1 The CORBA class and related functions

Jet is not a complete CORBA implementation, but consists only of those parts of CORBA which were deemed essential to the purpose of exploiting DIMMA. Hence the CORBA class does not support all the elements of that class from the CORBA specification.

The CORBA class is defined in `$DIMMA/dpe/Jet-CORBA/corba.hh`. This defines the basic CORBA types, mapped to the equivalent underlying ODP types; with the exception of Boolean, which is defined to be an octet, as required by the CORBA specification. Object is defined as an `odp_Reference`,

which is an interface for manipulating the 'on-the-wire' format of an interface reference. The Object_var class is implemented as an ODP untyped invocation reference (`odp_GenInvocationRef`).

Also defined in corba.hh are the string operations:

- `string_alloc`
- `string_free`
- `string_dup`

The implementation for the Object marshalling operators is in `$DIMMA/dpe/Jet-CORBA/JetDistributed/jet.cc`. The implementation for the other methods described here is in the form of in-line code.

3.2.2 Constructed Types

The corba_xxx.hh files in the `$DIMMA/dpe/Jet-CORBA` directory provide support for constructed types, including sequences and various _var and similar types. Some of these are implemented as C++ templates.

3.2.2.1 String_var, String_mgr and Struct_var

Full support is provided for the CORBA::String_var type in `corba_String_var.hh`, and for the Struct_var types in `corba_Struct_var.hh`.

An additional string type is provided for strings within a struct. In order to be able to use aggregate initialisation of a fixed-length CORBA struct, i.e.

```
FLstruct buff1 = {5,6};
```

as opposed to using a class constructor, the class which is used to represent the struct may not have any user-defined constructors, destructors or assignment operators; so each struct member must be of a self-managed type. This would not be the case if a String were used directly, since this is represented as a char*, which is not self-managed. To avoid this problem, all strings within struct classes are represented as a `corba_String_mgr`, which manages its own memory correctly. This is located in `corba_String_mgr.hh`.

File `corba_Struct_var.hh` provides support for struct_vars. This includes constructors, destructor, assignment operators and so on. The methods make use of the fact that all struct members are of the self-managed variety. This means that the default assignment operator can be used to copy structs, and each struct member carry out the appropriate actions.

3.2.2.2 Sequences

Sequences are either bounded or unbounded, and contain or do not contain strings (sequences containing strings need to be treated specially). Sequences hence fall into four distinct categories with associated header files:

- `corba_Sequence.hh` - unbounded sequences
- `corba_BoundedSequence.hh` - bounded sequences
- `corba_SeqStr.hh` - unbounded sequences of strings
- `corba_BoundedSeqStr` - bounded sequences of strings

All of these hold templates for sequences. In addition the class `corba_ManagedSequence` converts a sequence into a class that can be memory managed by a smart pointer. This inherits from `corba_RefCounter`, which provides support for the smart pointer. Finally the `corba_Pointer` class

provides support for the sequence_var. These are found in files `corba_ManagedSequence.hh`, `corba_RefCounter.hh` and `corba_Pointer.hh` respectively.

In the header file generated by the IDL compiler, a sequence defined in the IDL as:

```
typedef sequence<long> BuffType;
```

is represented as:

```
typedef
corba_ManagedSequence<corba_Sequence<CORBA::Long>,CORBA::Long>
                                        BuffType ;
typedef corba_Pointer<BuffType, CORBA::Long> BuffType_var ;
```

This is a simple unbounded sequence, hence the use of the `corba_Sequence` template, converted to a memory managed sequence with the `corba_ManagedSequence` template, with _var support provided by the `corba_Pointer` template.

The `corba_RefCounter` class contains a reference count, and provides methods to increment and decrement the reference count, as described earlier. The `corba_ManagedSequence` class inherits from this and from the appropriate base sequence class. Its methods are implemented by delegating to its super class.

The `corba_ManagedSequence` class has a friend which is the `corba_Pointer` class (supporting sequence_var), so that the latter has access to the reference counter. Operations on a sequence made through the `corba_Pointer` class result in the associated memory being managed by transparent manipulation of the reference counter.

Each of the four sequence base class templates provides the following functions:

- constructors, destructor, assignment operators as in CORBA specification
- allocbuf
- freebuf
- maximum
- length (both set and return)
- [] operators
- marshalling functions

3.2.2.3 *Object references*

The typed object reference _var, e.g. Echo_var for interface Echo, is defined as an instantiation of the C++ template `corba_InvocationRef`, in the general header file created by the IDL compiler:

```
typedef corba_InvocationRef<Echo> Echo_var;
```

Thus `corba_InvocationRef`, which inherits from `CORBA::Object_var`, provides the base class template for all CORBA object references. The primary purpose of the object reference is to be dereferenced (->) in order to invoke operations on local or remote interfaces. Object references can also be

initialised, copied, assigned, deleted and passed as arguments or results of an invocation.

The untyped or generic `CORBA::Object_var`, from which `corba_InvocationRef` inherits is itself derived from the ODP generic invocation reference `odp_GenInvocationRef`.

A `corba_InvocationRef` supports the usual range of constructors, destructor, assignment operators and so on. All memory management within the `corba_InvocationRef` is carried out by reference counting. Both `odp_Signature` and `odp_Reference` classes inherit from an `odp_RefCounter` class, which provides a reference counter and methods for incrementing and decrementing the reference counter. The methods within the `corba_InvocationRef` class use these reference counters to control memory management.

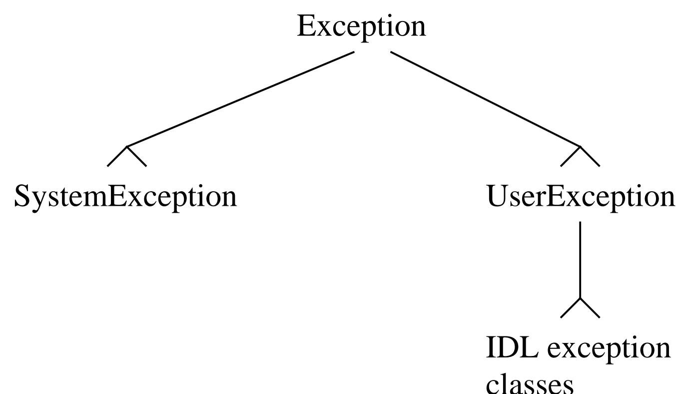
3.2.2.4 *Exceptions*

The structure of exception classes is shown in Figure 3.1. All support for base class exceptions in Jet is located in `$DIMMA/dpe/Jet-CORBA`. The base class for all exceptions may be found in `Exception.hh/cc`. This Exception class includes some additional non-CORBA diagnostic support. It provides each exception with:

- a name
- a method to extract the name
- marshalling operators << and >>
- constructors, destructor and assignment operator.

All exceptions in Jet inherit from this class.

Figure 3.1: Exception class hierarchy



Files `SystemException.hh/cc` provide support for CORBA system exceptions. Similarly files `UserException.hh/cc` provide support for a base `UserException` class which is inherited by all exception classes defined in IDL files. An example of such an exception might be:

```
exception Reject
{
     string str;
     short s;
};
```

This would result in the code shown in Figure 3.2. Code similar to this is generated for each exception declared in the IDL. The constructors pass the name of the exception to the constructor for the inherited `UserException`. Any memory required within the exception is allocated in the constructor (here in `string_dup`) and freed in the destructor. Finally the data itself is declared.

Figure 3.2: Implementation of IDL exception

```
#ifndef Echo_Reject__Exception
#define Echo_Reject__Exception
   class Reject : public CORBA::UserException
   {
   public:
      Reject( CORBA::String _str, CORBA::Short _s )
         : UserException("Echo::Reject")
      {
         if( _str != 0) {
            str = CORBA::string_dup( _str );
         }
         s = _s;
      }
      Reject( const Reject & other )
         : UserException("Echo::Reject"), str(0)
      {
         if( other.str != 0) {
            str = CORBA::string_dup( other.str );
         }
         s = other.s;
      }
      ~Reject() { CORBA::string_free( str ); }
      CORBA::String str ;
      CORBA::Short s ;
   } ;
#endif // Echo_Reject__Exception
```

3.3 Trader Interface

A description of the trader service itself is given in Chapter 12. This section describes the methods of interfacing to the trader from the Jet API. Since DIMMA provides no CORBA repositories, an attempt was made to provide consistency with another commercial ORB. The `_bind` operation is modelled on that of Orbix. Orbix, however, does not have any equivalent to `_export`, so this is modelled on ANSAware.

3.3.1 Bind

The _bind method is modelled on that of Orbix for application portability. For interface Echo, we have:

```
static Echo_var _bind(
     const char* IT_markerServer = "",
     const char* host = "");
```

The two arguments default to null strings and may be omitted. If used, they have no effect within the DIMMA implementation of `_bind`. Originally, when the ANSAware trader was used, they were used to specify properties to the trader.

The `_bind` method may thus be invoked as:

```
Echo_var ev = Echo::_bind();
```

The `_bind` method is invoked on the type of object for which a `_var` is required. The `_bind` method is therefore generated by the IDL compiler and appears in the _C.cc file for the interface.

The `_bind` method obtains a reference to the local trader and invokes its import method to obtain the associated interface reference, returning it as a generic invocation reference. This is then narrowed to the specific type of interface reference (here Echo) using the >>= operator:

```
Echo_var objRef;
...
if (!(trader->import(type_name, ctxt_name, prop) >>= objRef)) {
 throw CORBA::NO_IMPLEMENT ();
 // Cannot find implementation of interface
  }
```

3.3.2 Export

The definition of the `_export` method is:

```
int _export(CORBA::Object_var& iref, const char *name);
```

It is invoked as:

```
Echo_var my_Echo = new Echo_i;
if (_export(my_Echo) == -1) {
     cout << "failed to export Echo interface" << endl;
     exit(1);
}
```

Here an object `Echo_i` (Echo implementation) is created and then exported. A second `char *` argument to `_export` is permitted for consistency with ANSAware, but is not required.

The object reference, here `my_Echo`, is passed to `_export` as an `Echo_var`, while `_export` expects a `CORBA::Object_var`. Since `Echo_var` inherits from `CORBA::Object_var`, this is widened automatically. This feature enables a single `_export` method to be used for all interfaces and so the method is located in the Jet library rather than in the stubs generated by the IDL compiler. The source for `_export` is found in `$DIMMA/dpe/Jet-CORBA/JetDistributed/jet.cc`.

The `_export` method obtains a reference to the local trader and invokes the export method.

3.4 IDL Compiler

3.4.1 Overview

The SunSoft compiler operates in two passes. The first pass, which is provided by SunSoft, parses the IDL input and produces an internal representation known as an Abstract Syntax Tree (AST). This pass also performs a complete syntax and semantic check of the input to ensure that only legal IDL is accepted. If an error is discovered, the second pass is not invoked.

The second pass takes the AST and produces output in the programming language and format required. The demonstration back end provided by SunSoft is effectively a dummy, since it generates no code and merely provides a mechanism by which the parser can be run.

The main part of the work in implementing Jet has been to write a back end (code generator) which implements the CORBA C++ mapping and provides an interface into the DIMMA nucleus. Minor changes have also been made to the front end to provide support for flows.

3.4.2 Support for Flows

To provide support for flows, an additional keyword has been added to the CORBA IDL syntax, this being the word *flow*. The syntax of a flow is identical to that of an interface but with some semantic restrictions. Operations, corresponding to types of frame within a flow, are effectively oneway. They may take *in* arguments, but not *out* or *inout* arguments, and may not return any results. As a flow is so similar to an interface, no additional classes have been added to the AST. Instead, an attribute has been added to both interface and operation classes, to indicate whether or not the interface is a flow, and whether the operation occurs within a flow or an interface.

There are therefore two aspects of adding flows; adding a keyword to the syntax, and distinguishing between interface and flow in the AST for both interface and operation.

3.4.3 Code Generation

The code generation is carried out by the back end of the IDL compiler located in directory `be`. Each major component of the IDL has its own class in the AST, and these are inherited by a class for each component in the back end. For example, class `be_interface` inherits from class `AST_Interface`. The main classes within the backend are defined in the following header files:

- be_argument.hh - class be_argument handles operation arguments
- be_classes.hh - classes which do not have their own file, e.g. predefined types, expressions
- be_constant.hh - class be_constant handles constants
- be_enum.hh - class be_enum handles enumerated types
- be_exception.hh - class be_exception handles exception types
- be_interface.hh - class be_interface handles interfaces

- be_module.hh - class be_module handles modules
- be_operation.hh - class be_operation handles operations
- be_sequence.hh - class be_sequence handles sequences
- be_structure.hh - class be_structure handles structures
- be_typedef.hh - class be_typedef handles typedefs
- be_util.hh - utility functions generic to several classes

In addition to inheriting from the appropriate AST class, most classes inherit from `AST_Decl`, which is a generic class.

Control of code generation is carried out in file `be_produce.cc`. The function `be_produce`, which is called by the main compiler driver, determines which files to generate. For each file, it opens the file and invokes a top-level function for that file. This function (also in `be_produce.cc`) writes a header to the file, along with any code which is file specific, but not IDL specific, e.g. some include statements. It then invokes an iterator to loop through the AST for the outer level of scope, and invokes a generic function for the particular file.

There is generally an iteration function for each file generated. These are found in `be_util.cc,` and they iterate through each AST entry at the top level of scope, identify the type of item (e.g. interface, module), and invoke the appropriate method on that item.

For each class requiring output for any generated file, there will be an appropriate method. As an example, the `be_interface` class supports the following methods (the target file is given in parenthesis):

- dumpTable - set up the dispatch table for the interface (_N.cc)
- dumpMarshall - dump marshalling operators for iref_var (_N.cc)
- dumpServerStub - dump the interface-dependent server stubs (_S.cc)
- dumpClientStub - dump the interface-dependent client stubs (_C.cc)
- dumpIncludeFile - dump the interface-dependent parts of the main header file
- dumpSimpl - dump the interface-dependent parts of the server implementation file (_i.tc)
- dumpSHimpl - dump the interface-dependent parts of the server implementation header file (_I.th)
- dumpServMain - dump the construction and export part of the server implementation file (_i.tc)
- countInherits - utility for interface: counts number of interfaces this one inherits from
- dumpInherits - utility for interface: dumps code for inherited interfaces

Similar methods exist for the other AST types, e.g. modules, operations and so on.

Continuing the example for an interface, we now invoke the `dumpIncludeFile` method on the particular instance of the interface within the AST. This method writes to the header file all code directly related to the interface, as well as invoking other methods to handle items declared within the interface. It will normally do this by invoking the `BE_dumpIncludeFile` function, which identifies the type of the item which it is passed and invokes the appropriate operation on that item. In this way the AST is parsed iteratively. For

interfaces, any code for type declarations needs to be inserted prior to the declaration of operations, since these are likely to use the type declarations. Consequently the iterative procedure is used to produce code for all of these, but the `BE_dumpIncludeFile` function ignores operations. The `dumpIncludeFile` method within the interface class iterates through the operations, invoking the `dumpIncludeFile` method on the operation class directly.

Additional iteration methods are called for different sections of code. In general, the BE methods are used when all or most types are required, and the direct invocation method of iteration is used only when a single type is required. These iteration methods form the major part of the back end of the IDL compiler.

In addition to the iteration methods, there are many utility functions. These include:

- is_pre_defined: returns true for a predefined type (integer, float etc.)
- is_fixed: returns true for fixed length sequences and structs
- is_string: returns true for a string
- BE_is_variable: returns true for any variable length type

For a full list of these types of function, refer to `be_util.cc`. This file also includes generic functions for dumping scoped names in various forms.

3.5 Generated Code

3.5.1 Classes

The class hierarchy for an interface is given in Figure 3.3. This shows the simple hierarchy for a single interface together with the classes generated for that interface. The IDL for this interface, called `account`, is simply:

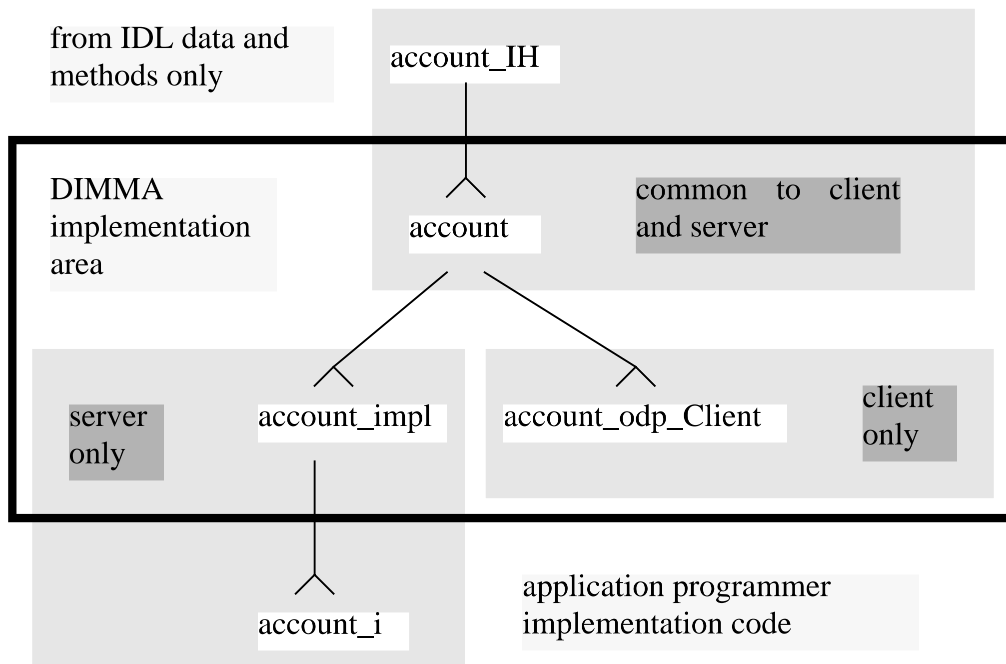
```
interface account {
// type and operation definitions
};
```

The classes hold the following methods:

- account_IH: typedefs and virtual methods declared in the IDL.
- account: additional implicit methods for the class (which are not declared in the IDL), i.e. _bind, conversion operators for odp_GenInvocationRef and the generic DIMMA invocation reference type.
- account_impl: the implementation class including the DIMMA-related implementation methods, i.e. odp_stub which generates DIMMA server stubs.
- account_i: the implementation class as seen by the application programmer. This has constructor, destructor and the methods declared in the IDL.
- account_odp_Client: the class for the client stubs.

These classes fall into three groups, as shown in Figure 3.3. Two classes are common to client and server, two classes relate only to the server, and one class is specific to the client. The reason for two implementation classes is to separate DIMMA implementation from that produced by the application

Figure 3.3: Class hierarchy for a single interface



programmer. The _i class that the programmer completes has no extraneous methods or data.

The distinction between the two generic classes is similar: the _IH class contains only data and methods taken from the IDL, while the main interface class contains additional methods. The heavy box in Figure 3.3 encompasses all those classes which interface to DIMMA. The classes outside the box relate purely to the IDL. The distinction between the _IH class and the main interface class is important when interfaces within the IDL inherit, as shown in Figure 3.4. A simple IDL might be:

```
interface account {
// methods and data for account
};
interface currentAccount : account {
// methods and data for currentAccount
};
```

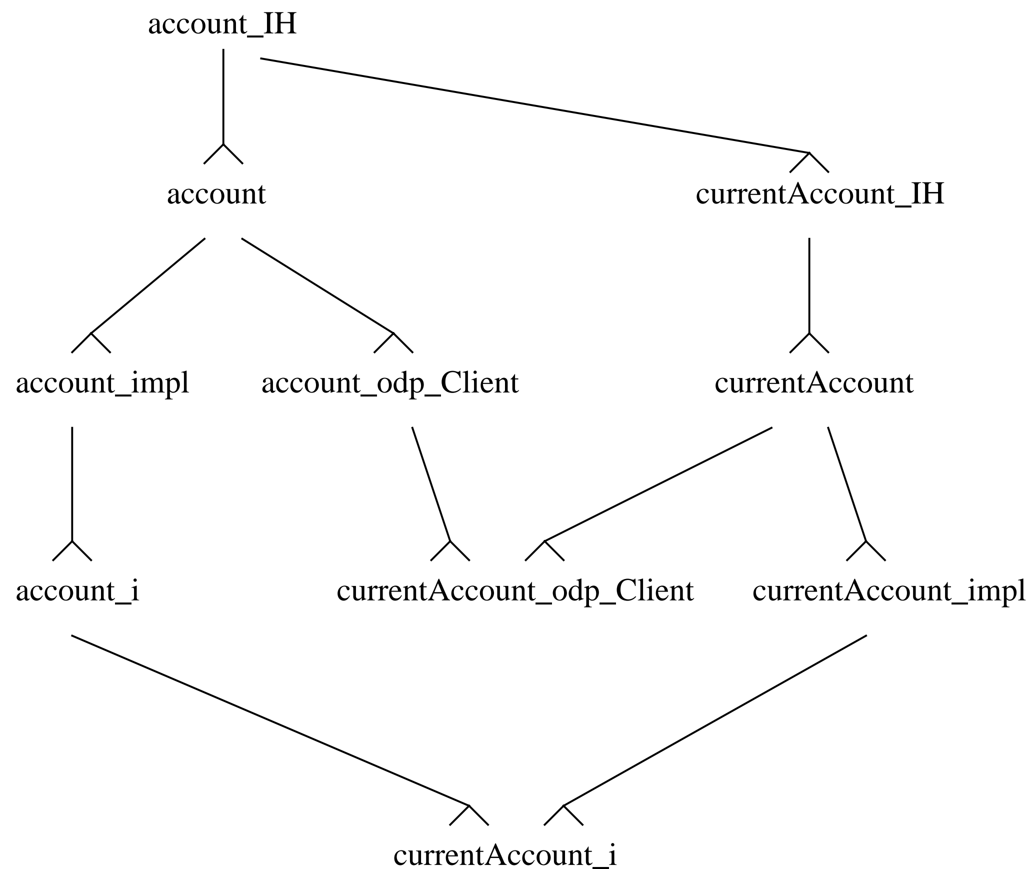
Here it can be seen that the `currentAccount` classes inherit the types and methods which they may need to define their own types and methods early in the hierarchy. Subsequently the client and server classes inherit only what is necessary.

3.5.2 Files

There are six files generated by the IDL compiler, as shown in Figure 3.5. These are listed below for an IDL source file xxxx.idl. A more detailed discussion of the contents of the files is given in section 3.5.3.

- xxxx.hh - header file, included by both client and server
- xxxx_N.cc - code common to both client and server, mostly related to distribution aspects
- xxxx_C.cc - client stubs and related code

Figure 3.4: Inheritance of interface classes



- xxxx_S.cc - server stubs and related code
- xxxx_i.th - header file for implementation
- xxxx_i.tc - implementation file

The latter two files provide a basic template for the server implementation, with a declaration of the _i class and dummy methods. The application programmer is required to rename these files to _i.hh and _i.cc, and to add the implementation of the methods. The class declaration has been separated out into the implementation header file so that the class can be extended (with non_IDL data/methods) if required.

3.5.3 Content

For specific examples of each of the files described below, refer to the files generated from the IDL in the examples directory tree.

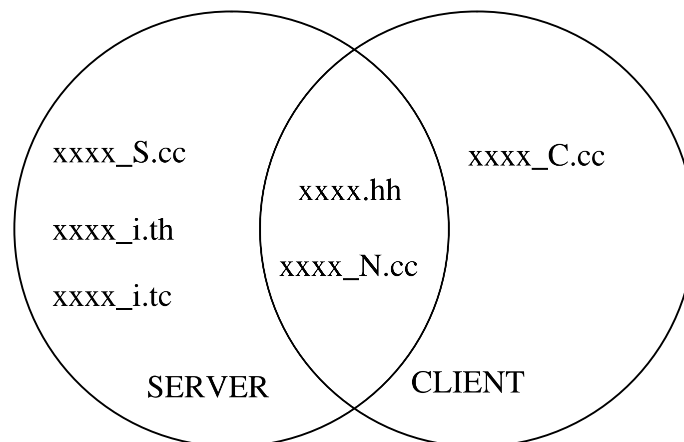
3.5.3.1 xxxx.hh

This is the header file which will be included by all the other generated files, and by client and server sources. It therefore contains statements to include all other header files necessary for Jet.

For each interface in the IDL, the following code is generated:

- Definition of the `iref_var` class - this is typedef'ed to the template `corba_InvocationRef<xxxx>` (see section 3.2.2.3)

Figure 3.5: The files generated from xxxx.idl



- Definition of the `_IH` class including all types and methods declared in the IDL file
- Definition of the main interface class which inherits from the `_IH` class and from `odp_Signature`, which is the DIMMA signature class. This class includes:
  - — an enumerated list of exceptions declared within the interface
  - — an enumerated list of operations declared within the interface
  - — a declaration of the name table used for distribution
  - — a declaration of the `_bind` method
  - — a declaration of the conversion operator used by `_bind`, which converts a generic interface reference to the reference for the current interface
  - — declarations for marshalling and unmarshalling operators for the `xxxx_var` interface reference
  - — declarations for methods on the interface: `_narrow`, `_duplicate`, and `_object_to_string`
- Definition of the `xxxx_odp_Client` class which inherits from the main interface class and from the `odp_ClientStub` class which holds the DIMMA client stubs. This definition includes virtuals for all methods within the interface.
- Definition of the `xxxx_impl` class which inherits from the main interface class and which contains the server stubs. This definition includes virtuals for all the methods within the interface, and a declaration of the `odp_stub` method which creates DIMMA server stubs.

3.5.3.2 *xxxx_N.cc*

This file holds code common to both client and server, relating mainly to distribution aspects. For each interface in the IDL, the following code is generated:

- The dispatch table. This is used by the underlying DIMMA code to locate remote interfaces and operations. The data used to set up the table consists of a string holding the following:

- — the name of the interface, preceded by its length in octal embedded at the start of the string
- — the number of methods, and the name of each method, each preceded by its length in octal embedded at the start of the string
- — the number of exceptions, and the name of each exception, each preceded by its length in octal embedded at the start of the string

- The _duplicate method on the interface, which uses reference counting to duplicate the `xxxx_var`.
- Marshalling and unmarshalling operators for the `xxxx_var` (see section 3.5.3.2).

3.5.3.3 *xxxx_C.cc*

This file holds the implementation of the client stubs, and of other client-specific functions. For each interface in the IDL, the following code is generated:

- `_bind` method for binding to an instance of the server (see section 3.3).
- `_narrow` method for converting from a `CORBA::Object_var` to `xxxx_var` (again see section 3.5.2). This is client-specific since it is assumed that, given an `Object_var`, the only reason for converting it to an `xxxx_var` is to make an invocation on `xxxx`, in which case client stubs must be generated.
- `_object_to_string` method where an ASCII representation of an interface reference is required. This method returns the same string as that which is written to file during trading (see section 3.3).
- Conversion operator >>= converts a generic invocation reference to an `xxxx_var`, creating client stubs if necessary.
- Constant values and strings, if any
- Client stubs for each operation declared within the interface. A client stub will typically include the following:
  - — creation of buffer for marshalling ins and inouts (arguments sent out to the server)
  - — the marshalling of ins and inouts
  - — the invocation, using the DIMMA call method, which returns a buffer with the results, including any exceptions
  - — a switch statement which switches according to the success or otherwise of the invocation
  - — a case statement for successful completion (`no_exception`) which unmarshalls results, outs and inouts, and returns the method type if any
  - — a case statement for system exceptions, which unmarshalls the particular exception and throws it
  - — a case statement for each exception declared in the IDL for this operation, which unmarshalls any arguments to the exception, creates an instance of the exception class, frees any memory created during unmarshalling (e.g. for strings), and throws the newly created exception
  - — a default case for any unknown exceptions, which throws a `CORBA::UNKNOWN` exception

Where there are inout arguments which require memory management, e.g. strings, then these are freed prior to unmarshalling the results.

3.5.3.4 *xxxx_S.cc*

This file holds the implementation of the server stubs, and of other server-specific functions. For each interface in the IDL, the following code is generated:

- `xxxx_impl::odp_stub()` method which creates a new instance of DIMMA server stubs using a template:

```
stub = new odp_ServerStub<xxxx>(this);
```

- The constructor which is invoked when the statement above is executed, i.e. the odp_ServerStub constructor. This invokes a constructor in an inherited class, passing the name table for the current interface.
- The server stubs, in the `upcall` method on `odp_ServerStub<xxxx>`. This method is passed a buffer with the operation code and any in and inout arguments. It executes a switch statement to switch to the stub for the required operation. A default case traps out invalid operations.
- The server stub for each operation. This includes:
  - — declaration of arguments
  - — unmarshalling of in and inout arguments
  - — invocation of the method's implementation
  - — creation of a buffer for marshalling the results, initialised with the success code or otherwise of the invocation
  - — marshalling of results
  - — freeing of any allocated memory (e.g. strings)

3.5.3.5 *xxxx_i.th*

This is the implementation file which the application programmer can modify to take data and additional methods which are not in the IDL file. It contains a declaration of a default constructor and destructor for the xxxx_i class, and declarations of each operation declared in the IDL file. This file should be renamed to `xxxx_i.hh` prior to use. It is created with a different name so that the modified version will not be accidentally overwritten.

3.5.3.6 *xxxx_i.tc*

This is the main implementation file. It should be renamed by the application programmer and the methods completed. It is created with a different name so that the completed version will not be accidentally overwritten. It contains a "main" function, and empty methods for each operation specified in the IDL. The "main" function by default creates a new instance of the _i class for each interface in the IDL, and exports each of these to the trader.

3.6 Engineering

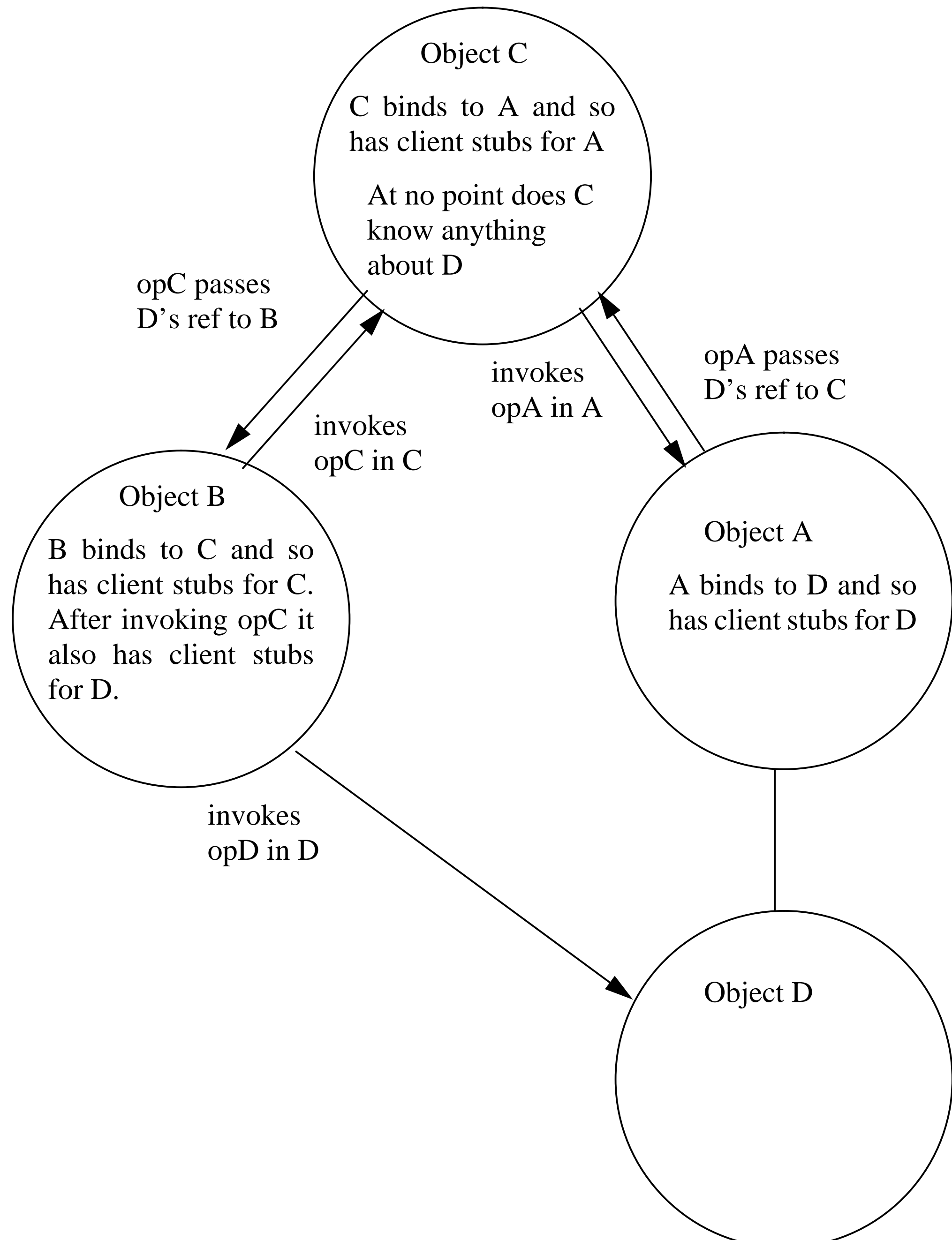
The Jet library implements the Jet computational API, i.e interfaces this to the underlying DIMMA nucleus.

3.6.1 Generic Object References

The `CORBA::Object_var` type is used to hold generic interfaces with no associated stubs. Thus a third party C can pass a generic object reference from server A to client B, such that B can subsequently invoke operations on A, even though C may be unable to invoke such operations.

In Figure 3.6, object A imports a reference to object D. A passes this on to C as

Figure 3.6: Using CORBA::Object



an untyped `CORBA::Object`. C sees only a generic reference and so it is unable

to invoke any operations on interface D. Consequently it needs no client or server stubs for D. In this example, A is behaving in a manner similar to that of a trader.

B invokes an operation on C and receives the generic object reference, but knowing about D, it is able to explicitly narrow this to a reference for D and hence invoke operations upon it. Figure 3.7 shows an example IDL description for such a situation. In practice each interface would reside in a different IDL

Figure 3.7: IDL for using CORBA::Objects

```
interface A {
    Object opA(); // invokes opC and returns the result
};

interface B {
    // B is the client
};

interface C {
    Object opC(); // returns D's ref
};

interface D {
    void opD(); // test op
};
```

file, so that the stubs would be generated in separate files. Otherwise all client stubs would be located in a single file and hence all servers would link in all the client stubs.

The point of this example now becomes clear. C now has a reference for D and is able to invoke D. But C did not import a typed reference to D and has no client stubs for D. Now, opA returns a `CORBA::Object`, but C requires a reference for D (a D_var) and hence must narrow the `CORBA::Object`:

```
C_var myc = c::_bind();           // bind to C
CORBA::Object obj = myc->opC();   // invoke opC and get an Object
D_var myd = D::_narrow( obj );    // narrow to a D_var
myd->opD();                       // now invoke on D
```

The `_narrow` function is created by the stub compiler for each interface, and is of the form (for object D):

```
D_var D::_narrow( CORBA::Object_var ref ) {
     return new D_odp_Client( ref.reference() ); }
```

which creates the client stubs.

See also section 3.6.1 which describes generation of client server stubs when marshalling interface references.

3.6.2 Marshalling

The implementation of the marshalling operators << and >> for basic types is the same as for the ODP basic types. Marshalling operators for sequences are carried out by templates which are described more fully in section 3.2.2.

Marshalling operators for interface reference _var's and for structures are generated by the IDL compiler and written to the _N.cc file.

There are three marshalling operators for each interface reference, two for marshalling an interface reference (one taking a const), the other for unmarshalling. The marshalling operators invoke the reference method (inherited from odp_GenInvocationRef class). In addition to providing a generic reference in a form suitable for marshalling, this method also generates underlying server stubs if these do not already exist.

4 The ODP Library

The ODP library provides a portable computational API based on the ODP-RM [ISO/IEC 95] concepts, implemented in C++ and providing a common basis for a range of distributed programming APIs (see [APM.1392] for additional detail). The engineering is also portable, in that it is independent of computational API, protocol and platform.

4.1 Computational API

The ODP-RM [ISO/IEC 95] concepts supported by the ODP computational API are:

- Objects;
- Interfaces;
- Terminations;
- Invocations;
- Operations;

In addition, the concepts of Signature and Invocation Reference are introduced:

- A Signature defines the operations and terminations of an interface;
- An Invocation Reference identifies an interface in the computational viewpoint;

4.1.1 C++ Language mapping

The C++ language has all the features needed to implement distributed programs, but it also it has many features that are impossible, difficult or dangerous to distribute. Accordingly the ODP C++ API is mainly concerned with placing restrictions on the way C++ is used to implement distributed programs.

Objects, interfaces and signatures are all mapped into C++ classes, but with different and quite stringent restrictions.

Operations map quite naturally into methods. Named terminations map into exceptions which naturally cater for multiple results. The last result of an anonymous termination maps into the method result and any preceding results are mapped into additional reference arguments.

Invocation references are mapped into typed smart pointers to the (abstract) signature. The (concrete) class that the smart pointer actually points to can be either a local interface or a client stub for a remote interface. The C++ virtual function calling mechanism then provides complete access transparency between local and remote invocations.

Primitive types are mapped onto the C++ their equivalents. they are accessed directly and passed as arguments by value. Constructed types and interfaces are always accessed and passed as arguments via smart pointers.

Local garbage collection of interfaces, objects and constructed types is done by reference counting and is implemented via the smart pointers.

4.1.2 Signatures

A signature is represented by a C++ abstract class definition which publicly inherits the base `odp_Signature` class.

The operations are defined as pure virtual methods.

The named terminations all inherit from the base termination of the signature so that they can be caught collectively if desired.

```
class BankManager_Sig : public odp_Signature
{
public:
    class Termination : public odp_NamedTermination {} ;
    class invalidPin  : public Termination {} ;
    class notOwner    : public Termination {} ;
    virtual BankPin newBranch (BankPin mpin, BankBranch & odp_r1)
                        throw (invalidPin,
                              odp_EngineeringTermination) = 0 ;
    virtual Pence balance (BankPin mpin)
                        throw (invalidPin,
                              odp_EngineeringTermination) = 0 ;
    virtual BankPin getPin ()
                        throw (notOwner,
                              odp_EngineeringTermination) = 0 ;
} ;
```

4.1.3 Invocation references

An invocation reference is represented by a smart pointer to a signature. This pointer is generated by the `odp_InvocationRef` template parameterised by the signature. It is recommended that a typedef for the invocation reference be inserted in the same header file as the signature.

```
typedef odp_InvocationRef<BankManager_Sig> BankManager ;
```

In the example above, the type of an invocation reference to (a local or remote instance of) a `Manager` interface defined in the `Bank` module is `BankManager`.

Invocation references are defined by the template:

```
template <class SIGNATURE> class odp_InvocationRef
{
private:
     SIGNATURE * ir ;
public:
     odp_InvocationRef () ;
     odp_InvocationRef (SIGNATURE * sp) ;
     odp_InvocationRef
             (const odp_InvocationRef<SIGNATURE> & other) ;
     ~odp_InvocationRef () ;
     odp_InvocationRef<SIGNATURE> & operator= (SIGNATURE * sp) ;
     odp_InvocationRef<SIGNATURE> & operator=
             (const odp_InvocationRef<SIGNATURE> & other) ;
     int operator== (int nil) const ;
     int operator!= (int nil) const ;
     SIGNATURE * operator-> () const ;
} ;
```

This generates a smart container for the signature pointer. The default constructor initialises an invocation reference with a nil pointer. The other two constructors initialise an invocation reference from a signature pointer and an existing invocation reference.

There are two assignment operators which overwrite an invocation reference with a signature pointer and an existing invocation reference.

A nil invocation reference is one which contains a signature pointer equal to zero. The `==` and `!=` operators will test if the signature pointer is equal or not equal to zero. Testing against any integer other than zero will always give an unequal outcome. It is not possible or meaningful to compare two invocation references as different invocation references may refer to the same interface.

The `->` operator is the main purpose of an invocation reference and is used to invoke operations on the interface pointed to by the enclosed signature pointer.

4.1.4 Objects

An ODP object is represented by a C++ object defined by a class which publicly inherits from the `odp_Object` base class.

```
class Bank : public odp_Object
{
     friend class Account ;
     friend class Customer ;
     friend class Branch ;
     friend class Manager ;
     friend BankManager Bank_factory () ;
private:
     Pence  cash ;
     Bank () : cash(0) {}
     BankManager body () ;
};
```

The object class must declare all its interface classes as friends, to allow the interface classes to access the private object data. It must also declare its factory function as a friend in order that the factory function may access the private factory method `body`. A factory may be defined to return zero, one or many invocation references to interfaces of the newly created object. A factory

must never return a pointer to the object itself as this must remain anonymous and only be accessible via its interfaces.

Any data which is to be shared between the object's interfaces is declared as private, a constructor defined to initialise it and a body method defined to create the initial interface(s).

An object should have no public methods or data, and the only private methods allowed are a constructor, body method and a destructor. Only the factory function should call the constructor and body methods. Any destructor will only be invoked when all the object's interfaces have been deleted.

4.1.5 Interfaces

An ODP interface is characterised by a signature and some behaviour [ISO/IEC 95]. That is, a signature corresponds to the interface *type,* the ODP interface itself provides a particular implementation. This construct is provided to support the ODP computational model, whereby an object may export multiple interfaces, and also provide multiple implementations of the same interface.

An ODP interface is represented by a C++ object defined by a class which publicly inherits from both the interface's signature and an `odp_Interface` class template parameterised by the interface's object class. The `odp_Interface` class provides implementation that is shared amongst all interfaces.

The interface class must define as friends any interfaces that access its private data and any interfaces (or its object) that call its constructor.

All interface data must be private as must its constructor.

The interface's public methods must implement the virtual methods defined by its signature.

```
class Manager : public BankManager_Sig
              , public odp_Interface<Bank>
{
     friend class Bank ;
private:
     BankPin  pin ;
     Manager (Bank * obj)
          : odp_Interface<Bank>(obj) , pin(0) {}
public:
     BankPin  newBranch (BankPin mpin, BankBranch & r1) ;
     Pence    balance   (BankPin mpin) ;
     BankPin  getPin    () ;
} ;
```

The `Manager` interface inherits from `odp_Interface<Bank>` a const pointer to its `Bank` object.

```
Bank * const object ;
```

This pointer can be used by operations in the interface to access the shared data in the object and is initialised by the `odp_Interface` constructor. The interface constructor must therefore have an object pointer argument that can be passed onto the `odp_Interface` constructor.

An interface can access the private data of another interface if it has been declared a friend and passed a pointer.

```
class Customer: public BankCustomer_Sig
              , public Interface<Bank>
{
     friend class Account ;
     friend class Branch ;
private:
     Branch * const  branch ;
     const BankPin  pin ;
     Customer (Bank * obj, Branch * bp)
          : odp_Interface<Bank>(obj), branch(bp), pin(rand()) {}
public:
     BankAccount  newAccount (BankPin cpin,
                              BankAccountNo & odp_r1) ;
     BankAccount  getAccount (BankPin cpin, BankAccountNo no) ;
} ;
```

In the example above, the `Customer` interface allows the `Branch` interface to create it and it can access the `Branch` interface's data because it is passed a pointer to the `Branch` interface in its constructor; and presumably the `Branch` interface makes it a friend.

Also, the `Customer` interface has allowed the `Account` interface to access its data; and presumably the `Account` interface's constructor will be passed a pointer to the `Customer` interface.

4.1.6 Arguments

An argument with a primitive type is directly passed by value. Both client and server end up with their own copy of the argument.

An argument with a constructed type is referred to by a smart pointer and passed by value. Both client and server end up with their own copy of the argument and smart pointer.

An argument which is an interface is referred to by an `odp_InvocationRef` and passed by reference. Both client and server end up with their own `odp_InvocationRef` which refers to the same instance of the interface. The interface referred to may reside in the client object, the server object or a third party object (i.e. anywhere). Using an interface as an argument does not change its location.

4.1.7 Terminations

The anonymous termination is handled differently from named terminations.

4.1.7.1 The anonymous termination

The last result of an operation's anonymous termination is passed as the result of the C++ method implementing the operation.

Any preceding results are returned via C++ reference (&) arguments added to the end of the method's argument list.

4.1.7.2 Named terminations

A named termination is represented by a C++ exception of the same name. The exception name is declared in the scope of the signature declaration and must be qualified by the signature name when used outside of the corresponding interface definition (e.g. when it is caught by a client).

All of a named termination's results are passed as arguments of its exception.

4.1.8 Results

Results are passed with the same semantics as arguments.

But remember that the additional arguments added to a method's argument list to specify where to return all but the last result of an anonymous termination must be C++ references (&) to the result types being returned.

4.1.9 Operations

An operation is represented by a C++ method in an interface class.

```
BankAccount Customer::newAccount(BankPin cpin,
                              BankAccountNo & odp_r1)
{
     if (cpin == pin)
     {
          int  accNo = branch->nextAccNo++ ;
          branch->accounts[accNo].acc =
                              new Account(object,branch,this) ;
          branch->accounts[accNo].pin = pin ;
          odp_r1 = accNo ;
          return branch->accounts[accNo].acc ;
     }
     else throw invalidPin() ;
}
```

In the example above, the first result of the operation's anonymous termination is returned via the `odp_r1` argument and the last result is returned as the result of the method. The named termination is thrown as an exception.

4.1.10 Invocations

An operation in a (local or remote) interface is always invoked via an invocation reference. The `->` operator provided by the `odp_InvocationRef` template returns a signature pointer which points to a local interface or client stub on which the method representing the operation is invoked.

```
BankPin  mpin = manager->getPin() ;
```

If the anonymous termination has more than one result then all but the last must be declared prior to the invocation and added to the argument list.

```
BankAccountNo acc1no ;
BankAccount   acc1 = customer1->newAccount(cpin1,acc1no) ;
```

Named terminations can be caught by using the exception handlers of a C++ try statement.

```
try  {
     BankAccountNo acc1no ;
     BankAccount   acc1 = customer1->newAccount(cpin1,acc1no) ;
     }
catch ( BankAccount::invalidPin )
{
     cout << "\nEXCEPTION: invalid pin\n\n" ; exit() ;
}
```

4.1.11 Constructed Types

`odp_basic_types.hh` defines the ODP types in terms of C++ types. These definitions are given in Table 4.1. Some of these types are platform-dependent.

Table 4.1: Basic odp types

odp type	C++ type
odp_Boolean	int
odp_Char8	char
odp_Octet	unsigned char
odp_Int16	short
odp_Nat16	unsigned short
odp_Int32	int/long
odp_Nat32	unsigned int/unsigned long
odp_Int64	long
odp_Nat64	long
odp_Real32	float
odp_Real64	float

`odp_StringPtr.hh` provides more sophisticated support for strings than the normal char*, discriminating between constant and variable strings.

`odp_Struct_var.hh` provides a smart pointer for memory managed structure and union classes. The pointer maintains a reference count in the managed object. When this reference count returns to zero, it will delete the managed object.

`odp_Sequence.hh` and `odp_BoundedSequence.hh` provide C++ templates for generating sequences. All element vectors must be unshared and allocated from the heap. They are deep copied and garbage collected.

`odp_ManagedSequence.hh` turns a sequence or bounded sequence into a class that can be memory managed by a smart pointer. `odp_Pointer.hh` contains the small pointer for memory managed sequence classes. This pointer maintains a reference count to the managed object. When this returns to zero, it will delete the managed object. A managed sequence (here `BuffType`) is declared as follows:

```
typedef
odp_ManagedSequence<odp_Sequence<odp_Int32>,odp_Int32> BuffType;
typedef odp_Pointer<BuffType, odp_Int32> BuffType_var ;
```

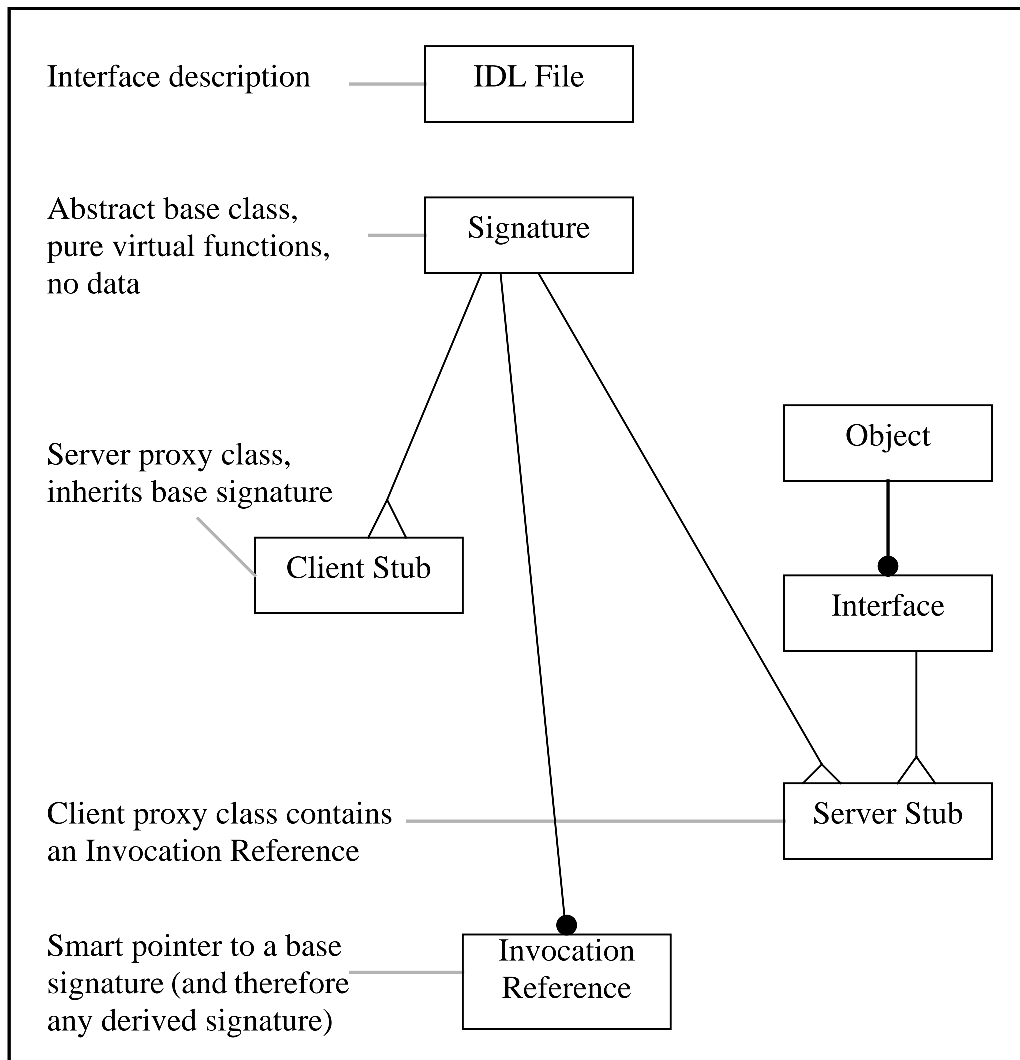
In a similar manner, `odp_ManagedArray.hh` turns an array into a class that can be memory managed by a smart pointer. This pointer is provided by `odp_Array_var.hh`.

The reference counter in all the managed types is provided in `odp_RefCounter.hh`. This provides a constructor and destructor, and facilities for incrementing and decrementing the counter. The managed classes inherit from the `odp_RefCounter` class.

4.1.12 Class relationships

The relationship between the main classes described above is illustrated in figure 4.1.

Figure 4.1: Class relationships



4.1.13 IDL Compiler and Stubs

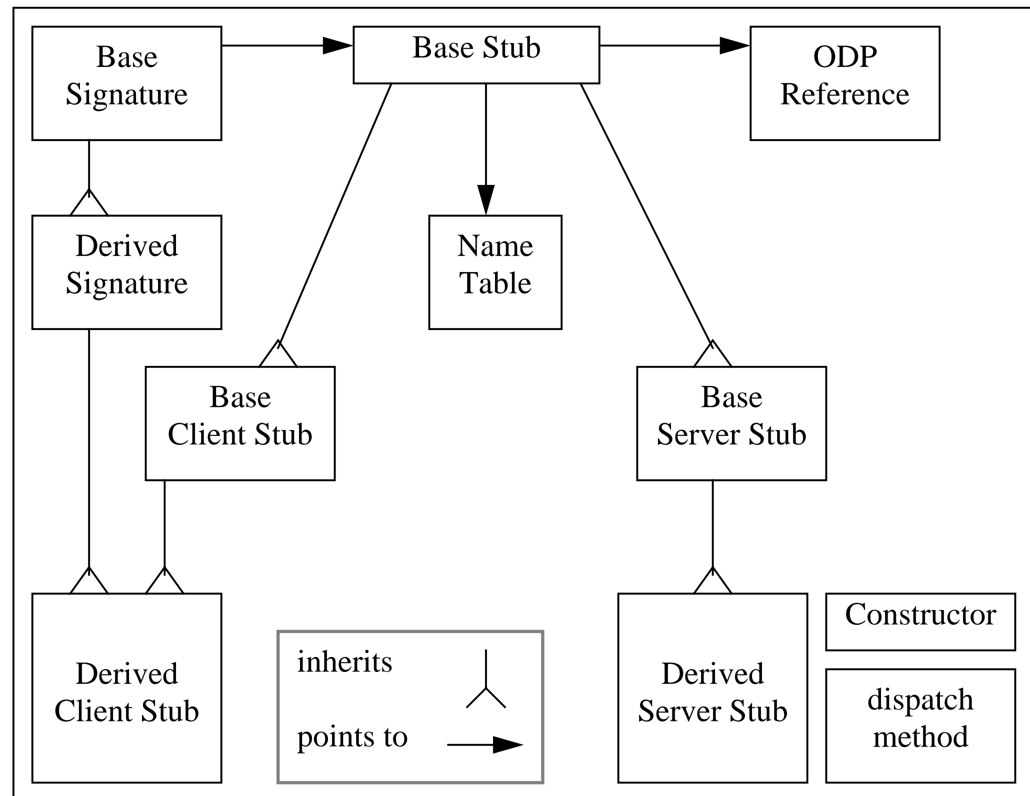
There is no IDL compiler currently available for the ODP API and stubs must be constructed by hand.

The relationship between the various components of the stubs is given in Figure 4.2. Here the term “derived”, as in “derived signature”, refers to the component for a particular interface. For example, an interface Echo will have a base signature class `Echo` which inherits from the base signature class. Similarly it will have a client stub `odp_Echo_Client`. These classes which are specific to the interface are known as “derived” classes.

The various components comprise:

- Base signature: this is the base class for all interface signatures. It inherits from the reference counter class for memory management. It is a separate class from the interface, both so that an invocation reference can be declared before the derived interface is defined, and so that either a derived interface or a derived client stub can inherit a derived signature with its counter and base stub pointer. It contains methods to initialise and to return the stub pointer.

Figure 4.2: Stub component relationships



- Derived signature: this inherits from the base signature class, and also contains methods declared in the IDL.
- Base stub: this is the common base class for client and server stubs. It provides methods to obtain the two items it points to, the interface reference (as a CORBA IOR see [OMG 95]) and the name table.
- Base client stub: this inherits from the base stub class and provides marshalling functions.
- Derived client stub: this must inherit from both the base client stub class and the derived signature class, thus providing a complete client stub for use in applications.
- Base server stub: this is a template for generating dispatchers and derived server stubs. It requires a constructor and a dispatch procedure for each type of ODP server stub.
- Derived server stub: this provides the constructor and dispatch procedure for the base server stub. The constructor initialises the base server stub's pointer to the signature. The dispatcher unmarshalls the operation code, then switches to the required operation and carries out the appropriate unmarshalling of arguments, invokes the operation, and finally marshals and transmits the results.
- Name table: this provides a list of interfaces and operations which are used to associate client invocations to server operations.
- ODP reference: the interface reference which is used to communicate between client and server.

Examples of client and server stubs can be found in the examples directories. The client stubs must provide the following:

- One stub method per server method
- A request function that binds if necessary and generates a protocol dependent transmitter buffer
- Marshalling of arguments into the buffer using the << operator
- An invoke function which carries out the RPC and returns a receiver buffer
- A switch statement to decode the response:
  - 0: normal return
  - 1: system exception
  - others: application exceptions
- Unmarshalling of results from the buffer using the >> operator
- Exceptions thrown by client stub
- Buffer released by a smart pointer

The server stubs must provide the following:

- A single dispatch method called with a protocol dependent receiver buffer
- A switch statement to decode the operation, with one branch per server method. Each branch should:
  - unmarshalls arguments using the >> operator
  - call the corresponding server method
  - catch any exceptions
  - get a transmitter buffer using the response function and optionally signal an exception
  - marshall results using the << operator
  - return the transmitter buffer

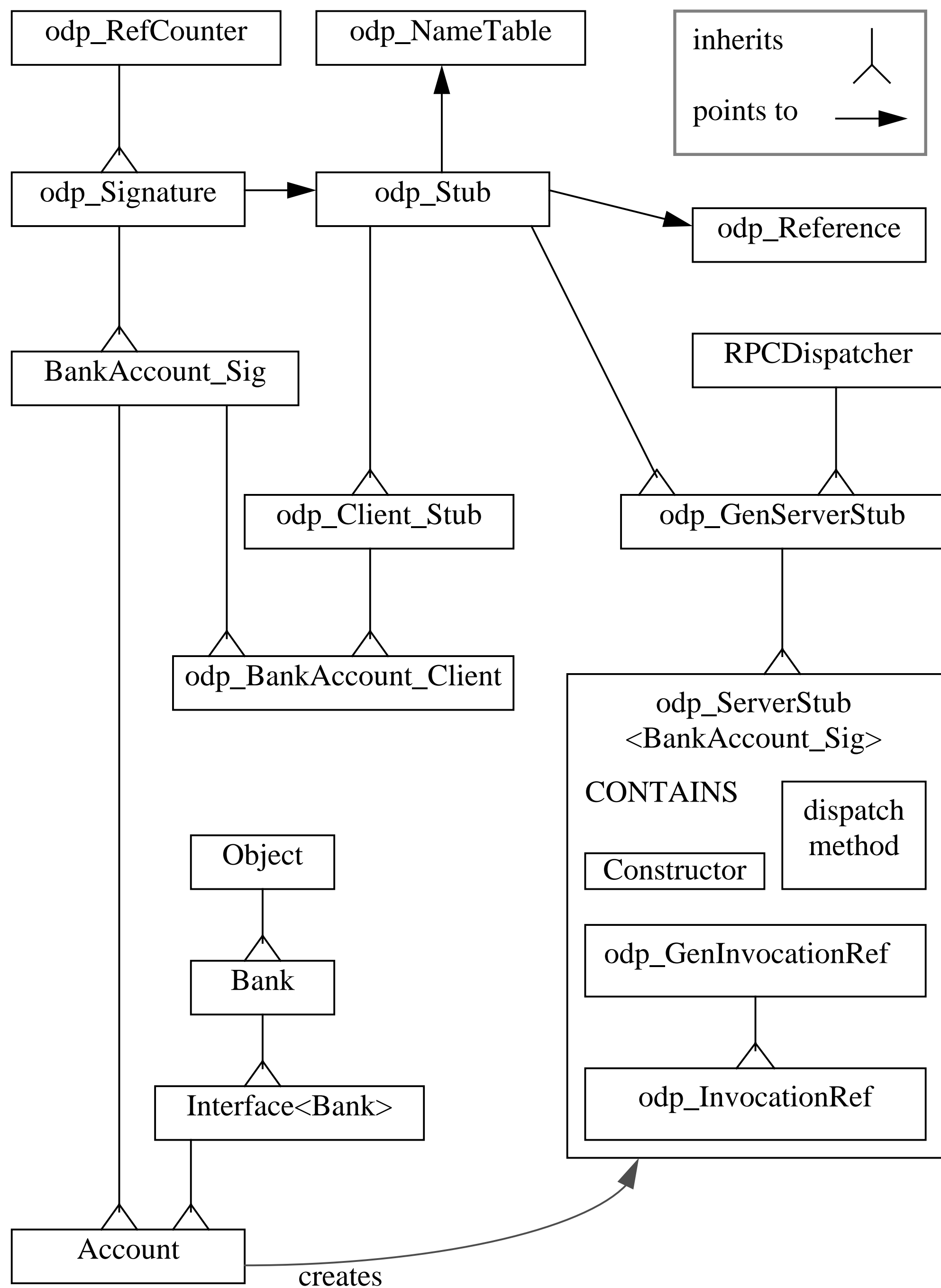
4.1.14 Class hierarchy of an application

This section demonstrates the class hierarchy of an example. The examples/ODP/tiny_bank example is used, since it is the only example which uses an Object. A simplified version of the IDL, with no operations shown, is given below:

```
Bank : module
[
     Account : interface ( ... );
     Customer : interface ( ... );
     Branch : interface ( ... );
     Manager : interface ( ... );
]
```

Here we have an object, `Bank`, with four interfaces, `Account`, `Customer`, `Branch`, and `Manager`. Each of these interfaces has several operations, but they are not shown here as they are not relevant to this discussion. The various classes involved here are shown in Figure 4.3.

Figure 4.3: Application class hierarchy



In addition to the classes specific to the application, this figure also shows the underlying DIMMA classes.

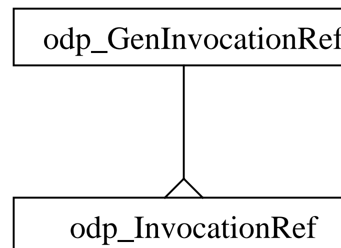
To simplify the diagram of the class hierarchy, only one interface is shown, the Account interface. The others would demonstrate a similar hierarchy.

4.2 Engineering

4.2.1 Invocation References

The class hierarchy for invocation references is shown in Figure 4.4.

Figure 4.4: Invocation reference classes



odp_GenInvocationRef is a “generic” invocation reference and all typed invocation references are subclasses of this. It is also used to hold untyped invocation references.

All copies of an InvocationRef which refer to a particular instance of an interface or client stub update a count in the instance. This reference count is kept by a counter inherited by the base signature from which each interface signature is derived. This is then inherited by the interface or client stub. When the reference count returns to zero, the counter is deleted, as well as its derived interface or client stub.

The odp_InvocationRef class is the base class template for all invocation references.

Invocation references contain two pointers, one to an interface reference (odp_Reference), and one to a signature. Either or both may be null.

For local interfaces, the signature points directly to the target interface class. Things are a little more complicated for the remote case.

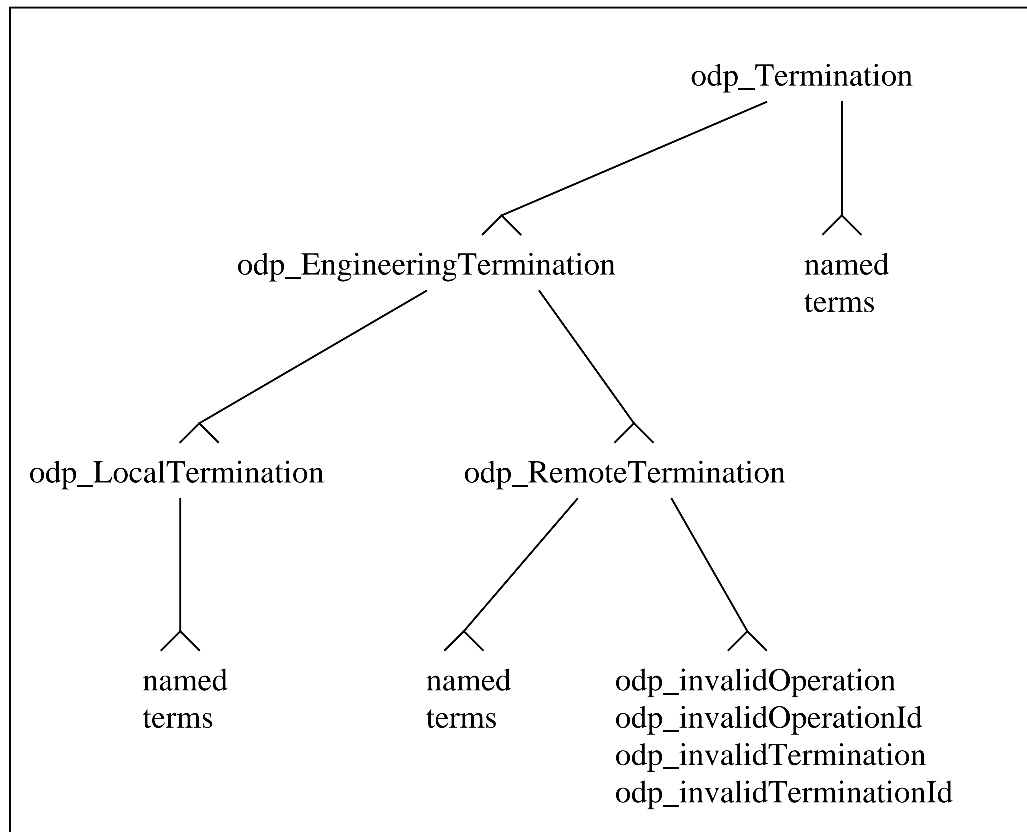
An invocation reference is initialised by an application with a pointer to an instance of an interface. This is stored in the signature pointer. When the invocation reference is used to export the interface outside the capsule, an interface reference (IOR) is constructed and stored in the interface reference pointer. Subsequent exports will use the latter rather than constructing a new IOR.

An interface is transferred between capsules as an IOR. When a client imports this, an invocation reference is created and the IOR stored in its interface reference pointer. If the client has a typed invocation reference, a client stub will be constructed and stored in the signature pointer. Otherwise this will remain null.

4.2.2 Exceptions

The structure of the classes for exception handling is shown in Figure 4.5. The base class for terminations is the odp_Termination class. Within the ODP engineering, all terminations inherit from the odp_EngineeringTermination class, which itself inherits from the odp_Termination class. There are two

Figure 4.5: odp terminations



primary types of odp_EngineeringTerminations classes: those for local terminations, and those for remote terminations.

All terminations have a name, conventionally the (sub)class name, and an identifier. The identifier is provided for use of “personality” adapters to aid in converting the native termination into a personality specific exception. Anonymous terminations can be created by using the constructor argument defaults, but the adaptor will find it hard to convert such terminations into accurate representations for its personality.

Instances of terminations are generally constructed using a template. These exist for the base `odp_Termination` class, and for local and remote terminations. For example, the template for remote terminations is:

```
#define REMOTE_TERMINATION(NAME,ID)
class NAME : public odp_RemoteTermination {
public:
     NAME () : odp_RemoteTermination( #NAME, ID ) {}
protected:
     NAME (const char* s, odp_termination_id_t id)
          : odp_RemoteTermination(s, id) {}
};
```

and an example of an instantiation of a remote template is:

```
REMOTE_TERMINATION (odp_bindFailure, odp_bindFailure_Id)
```

The template then becomes instantiated as:

```
#define REMOTE_TERMINATION(odp_bindFailure, odp_bindFailure_Id)
class odp_bindFailure : public odp_RemoteTermination {
public:
     odp_bindFailure() : odp_RemoteTermination(
          "odp_bindFailure", odp_bindFailure_Id ) {}
protected:
     odp_bindFailure(const char *s, odp_termination_id_t id)
          : odp_RemoteTermination(s, id) {}
};
```

Similar templates exist for the base termination class and for local terminations. The base termination class can be used for user-defined terminations, or preferably a new sub-class should be generated for this.

Some terminations require additional code, and these are defined separately, rather than using the templates. An example is the termination `odp_invalidOperation`, which has an additional name:

```
class odp_invalidOperation : public odp_RemoteTermination {
public:
     odp_invalidOperation(char* n) :
          odp_RemoteTermination("odp_invalidOperation",
          odp_invalidOperation_Id)
     {
          name = new char[strlen(n)+1];
          strcpy (name, n);
     }
     ~odp_invalidOperation() { delete [] name; }
     char * name;            // additional name
};
```

The id codes for the pre-defined terminations are specified in `odp_TerminationEnum.hh`.

5 Engineering API

5.1 Overview

The DIMMA engineering API represents the interface that the ODP computational API is written to and effectively defines the interface to the DIMMA nucleus. Other personalities, such as Jet, are largely hosted by the ODP computational API but may also directly use some of the facilities provided by the engineering API.

The engineering API also defines the underlying format of stubs and provides generic implementations of these. Hence, code generated by an IDL compiler must conform to the engineering API.

The engineering API encompasses:

- Generic Stubs
- Marshalling of primitive types
- Binders and Bindings
- Interface references

The following sections discuss these in more detail.

5.2 Generic Stubs

The purpose of the generic stubs is to provide a convenient interface between the personality specific stubs and the DIMMA nucleus. They also serve to collect together the code that is common to all stubs.

The generic stub code is located in the `ODPDistributed` directory of the ODP Library and a class exists for each type of stub:

- Client
- Server
- Transmitter
- Receiver

The first two being associated with operational interfaces, whilst the last two are relevant to flows. Each of these provides a thin layer on top of the DIMMA nucleus.

In general, stubs are created when a reference to an object crosses the capsule boundary. A server stub is created when a reference to an object is exported, either explicitly or returned as an argument of an operation. Likewise, client stubs are created when a reference to an object is imported, either explicitly or as the result of an invocation.

However, there is an exception to the above behaviour concerning the use of generic invocation references. These are provided for applications wishing to

process interface references without ever invoking them. An example might be a trader that stores and retrieves interface references on behalf of its clients.

In this situation, the import of a generic interface reference does not result in conversion to an invocation reference, nor does it result in the creation of stubs. An application subsequently wishing to invoke an operation on the interface must first explicitly convert the generic reference to a type specific one.

The details of how this is accomplished depends on whether the CORBA or ODP personality is being used and is covered in the associated chapters.

5.3 Marshalling

The engineering API defines [un]marshalling operators (<< and >>) for each of the primitive ODP types. These exist as two abstract interfaces, odp_Transmitter for marshalling and odp_Receiver for unmarshalling. Concrete implementations of these are provided by each protocol. The separation between interface and implementation exists to enable stubs to be made protocol independent.

Details of the marshalling implementations can be found in chapter 7.

5.4 Binders and Bindings

The engineering API defines four types of implicit binder:

- Client
- Server
- Transmitter
- Receiver

The first two are for binding operational interfaces and the last two are for flows.

Implicit binders are implemented as part of an engineering binding, i.e they are configured as part of the communications path, in-line between a stub and protocol channel. This arrangement allows maximum flexibility in terms of establishing and tearing down local bindings in a manner that is transparent to the stubs. As a consequence, stubs view implicit binders as implementing a *binding* rather than a *binder* interface.

The engineering API recognises a single explicit binder `ExplicitBinder` which provides the interface to the protocol specific binders. This binder is a static object which is located outside the communications path.

Bindings are objects which link stubs to communication channels created by the nucleus. They offer the following facilities:

- Marshaller creation
- Get methods to return interface references and protocol tags for the binding which they represent
- Get and set methods for interface type (as strings)
- I/O methods which interface with the underlying channel

More detail on binding can be found in chapter 6.

5.5 Interface References

Interface references correspond to the “on-the-wire” representation of a reference to an interface. These hold addressing information such as network address for locating the capsule and additional details to identify the interface within the capsule.

DIMMA adopts the format defined by CORBA for Interoperable References (IOR) for its interface references. This consists of a sequence of protocol profiles, each containing addressing information for a specific protocol.

DIMMA hides the implementation details of interface references behind the interface provided by the `odp_Reference` class. This offers methods to:

- Get a string representation of the interface reference
- Construct an interface reference from a string
- Get the type of interface (as a string)
- Marshall and unmarshall an interface reference

Interface references are created from the computational invocation reference type whenever an interface is exported outside a capsule. They are converted back to an invocation reference at a client in an analogous manner.

6 Binding

6.1 Overview

Binding is the making of an association between a client object and a specific instance of an interface. In engineering terms, this equates to:

1. creating an interface reference to the interface instance
2. passing the reference to the client
3. establishing a communications channel between client and interface

Items one and three are the subject of this chapter, whilst number two is the subject of trading and is covered by chapter 12.

6.1.1 Creating an interface reference

Creation of an interface reference is preceded by the creation of a local server binding, i.e. the construction of at least one communications channel between stub and network interface. This provides a network address, or endpoint, which a client can subsequently use to establish the binding. It is this addressing information which is incorporated in an interface reference.

6.1.2 Establishing a binding

The establishment of a binding normally begins at the client after it has obtained an interface reference for the desired interface. The process is analogous to that performed at a server when creating an interface, i.e. a local communications channel is established between the stub and network interface. The asymmetry lies in the association of the network endpoint with that of the server's interface, rather than creating an address to export.

6.1.3 Components of a binding

The major components which comprise a DIMMA engineering binding appear in figure 6.1.

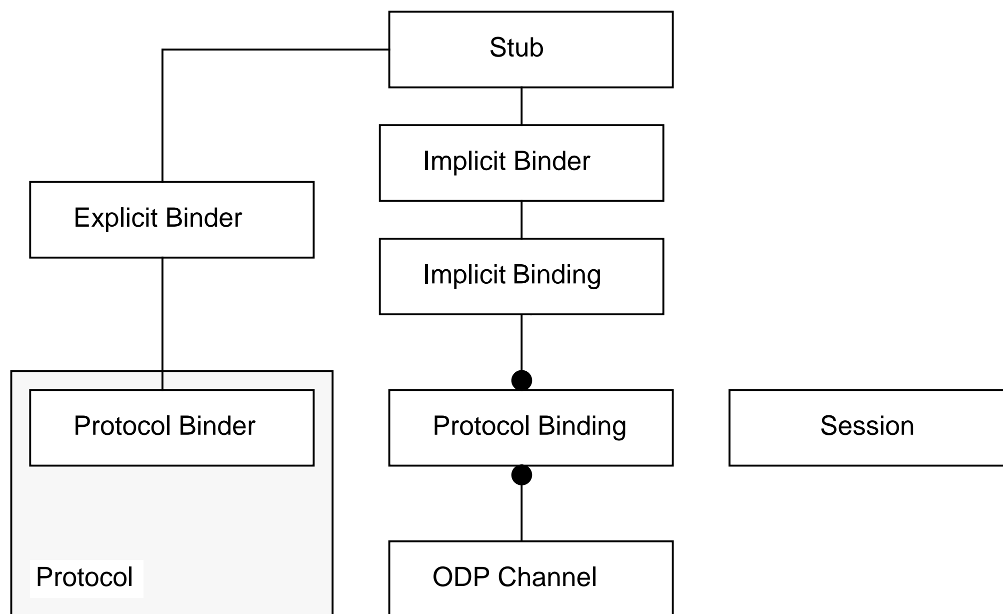
An engineering binding can be broadly divided into two: components above the *protocol binding* are protocol independent whilst those below are protocol specific. The former are the subject of this chapter whilst the latter are the subject of the communications framework and are covered in chapter 9.

6.2 Implicit Binding

Implicit binding is the default behaviour used when the application has no particular interest in the properties of the binding, i.e. has no particular QoS requirements.

At the server, implicit binding occurs when an interface is exported. This may occur either through an explicit export operation performed on the trader, or

Figure 6.1: Engineering Binding



as the result of an interface being passed out of the capsule as the result of an invocation.

The implicit binder is configured in-line with the engineering binding to make the process of binding transparent to the stub. This also enables the client's implicit binder to make its own decisions on when to establish or tear down the communications channel.

6.2.1 Server side

A typical server side implicit binder will create a communications channel for each protocol that it knows about. This is to maximise the chances that a client will subsequently be able to find a matching protocol when looking to establish a binding to the interface.

It follows that a server stub, via its implicit binder, will be potentially associated with several communications channels. It is the job of the *Implict Binding* in figure 6.1 to record each of these channels.

An interface reference constructed from a local implicit binding will consist of addressing information, or profiles, for each of the communication channels. It follows that it is the *Implicit Binding* that is responsible for creating such an interface reference.

6.2.2 Client side

A typical client side implicit binder establishes a binding to an interface by creating a local communications channel that is associated with the interface's network address. This is normally performed on the first invocation, although a client side implicit binder is free to choose any policy that it deems suitable.

The client binder chooses a protocol from the set supported by the interface (and hence contained in the interface reference) according to a particular policy. In DIMMA this is a first match policy.

A client side implicit binding is associated with only one protocol, in contrast to that of a server binding. It would therefore seem reasonable to directly associate a client stub with a communications channel. However, this is not the case in DIMMA, an intervening client side implicit binding object being inserted which performs some useful supplementary functions.

6.3 Explicit Binding

The explicit binding model exists primarily for real-time and multi-media applications to control the QoS of their bindings. Bindings may be established or torn down under application control and resources may be reserved at the time of binding in order to provide a specific QoS.

The key differences between explicit and implicit binding are:

- explicit binding is always initiated by an application (as opposed to the infrastructure)
- explicit binding is always associated with a specification of QoS at some level
- explicit binding will always result in specific resourcing of the channel
- an explicit binding is only ever associated with one communications channel and one network endpoint
- explicit binding may be initiated by either client, server, or a third party;

The process of explicit binding requires at least:

- the creation of local endpoints
- the exchange of the endpoints between both parties
- the establishment of the binding (the bind operation)

6.3.1 Endpoints

An explicit binding represents an end to end application connection, i.e. a single, specific communications path between a client and an interface. This poses no problem for a client since it always views a binding as a single association between it and a specific interface. However, the situation is less straightforward for a server.

6.3.1.1 Server Endpoints

In general, many clients may make concurrent use of a specific server interface and these may be multiplexed over a smaller number of communications channels. This is clearly unacceptable for an explicit binding which is associated with a specific QoS. A server needs a means of creating local bindings with associated QoS, independent of interface creation, i.e. multiple local bindings may be associated with a single interface.

A server therefore needs a means of constructing a local binding or network endpoint and this must be typed to preserve type safety. DIMMA provides this from the CORBA personality as an operation on an interface var. The operation `var.CreateEndpoint(QoS)` constructs a new local binding for the interface corresponding to `var`.

6.3.1.2 *Flow Endpoints*

In addition to operational interfaces, DIMMA also supports multi-media flows. A flow corresponds to a unidirectional flow of continuous data from source to sink. A flow source is known as a *Transmitter*, whilst a flow sink is known as a *Receiver*.

In general, a Transmitter is analogous to a client in that it typically sends to a single specific Receiver, whilst a Receiver is analogous to a server in that it may potentially receive frames from a variety of Transmitters. It is fairly evident that a Receiver needs the ability to construct endpoints in the same way as a server. What is less obvious, is that under certain conditions, a Transmitter may also need this ability.

Multicast transmission is a case in point. In this scenario, a Transmitter transmits to a multicast address as opposed to a specific Receiver. It therefore has no Receiver endpoint to bind to and must instead bind locally to a specific multicast address.

DIMMA provides the multicast facility through two operations on an interface var:

- `var.CreateEndpoint();`
- `var.Bind (QoS, address);`

The first constructs a new stub of the type associated with `var` (this normally occurs implicitly when an interface reference is imported), whilst the second creates a local binding to the specified multicast address.

6.3.1.3 *Application Endpoint Implementation*

What exactly gets created using a *CreateEndpoint* operation, or in other words, to what does an Endpoint correspond? The requirements include the ability to:

- identify an interface and its operations
- export an interface to allow exchange of endpoints
- identify the underlying communications channel

These requirements are a superset of those for any other engineering binding and in DIMMA, are supported by the same components. hence an endpoint:

- corresponds to an instance of a stub
- is exported as an interface reference containing a single protocol profile

6.3.2 Explicit binders

Explicit binders bind specific endpoints, explicitly created by application objects and take a QoS specification for the required binding.

In principle, a DPE may support many explicit binders, each with their own notion of QoS specification. These would normally be identified from applications by their name, e.g. `Binder.Bind(endpoint, QoS)`. However, DIMMA currently supports only a single explicit binder for each interface type[1] and hence its name need not be specified.

1. An explicit binder for each interface type is necessary to preserve type safety.

Explicit binders have no need to perform channel management outside of an applications control and hence are not configured as part of the engineering binding (unlike implicit binders).

The DIMMA explicit binder interacts with protocol specific binders to perform the business of binding proper, i.e. configuring the local communications channels and resourcing them appropriately. The (protocol independent) explicit binder accepts a QoS specification that is oriented towards engineering concepts but that is otherwise protocol independent. This specification, called `EngineeringQoS`, is passed on to the protocol specific binder.

The protocol specific binder performs the mapping between `EngineeringQoS` and the protocol specific resourcing mechanisms (see §9). It creates the communications channel, reserves any resources required by the QoS specification and passes a protocol binding back to the explicit binder.

6.4 IIOP Binder

This section provides a description of the QoS parameters supported by the Binder, and a description of the operation of the IIOP protocol binder.

The section can be regarded as a commentary on the construction of an IIOP ODP channel stack, the structure which is described in section 9.7.

6.4.1 IIOP QoS

`IIOPQoS` objects describing the required configuration of an ODP channel are passed into `ProtocolIIOP` when the protocol is asked to create a Binding and a Channel stack. If no QoS object is specified, default QoS objects are created.

The following sub-sections describe the QoS parameters supported by the IIOP binder:

6.4.1.1 Buffer Management

The `IIOPQoS` object contains two implementations of the `Buffer::Allocator` interface, one for Transmit buffers, one for Receive buffers. The `Buffer::Allocator` provides an allocator for objects of type `BufferIIOP`.

The implementation of this interface may be provided by a client (of the binder). DIMMA 2.0 provides two standard implementations which are:

- a `BufferIIOPFactory`, which creates `BufferIIOP` instances on demand
- a `BufferIIOP` *pool* of a number of pre-allocated, re-usable buffers.

6.4.1.2 Client Session Management

A `ClientIIOPQoS` object contains an implementation of the `ClientSession::Allocator` interface, which the binder uses to provide a client session layer for the protocol stack.

Different session management behaviour can be achieved by changing the `ClientSession::Allocator` in the QoS. The following implementations of `Client::Allocator` are provided with DIMMA 2.0:

- A `MultiplexClientSessionFactory` (the default) allocates `MultiplexClientSessions`, giving the ODP channel a full multi-threading capability;

- A `ClientSessionFactory` allocates `ClientSessions`, which provide a null session layer, intended for use in ODP channels that do not need to support thread multiplexing.
- Pools of `MulitplexClientSessions`, or `ClientSessions` may also be constructed to implement this interface.

6.4.1.3 *Server Session Management*

Different session behaviour can be achieved on the server side by changing the `Threading::Allocator` in the `ServerIIOPQoS`. The following implementations of `Threading::Allocator` are provided with DIMMA 2.0:

- A `TaskFactory` (the default) allocates `Tasks`, causing sessions to be dispatched in new Tasks;
- A `NullThreadFactory` allocates Null Threads, causing the sessions to be dispatched in the context of the current thread;
- A `ThreadScheduler` which can schedule *n* Threads amongst *m* Tasks.

6.4.1.4 *Listener Task Policy*

A listener task is potentially attached to a TCP channel to read and dispatch incoming messages. The QoS supplied to the binder contains a *ListenerTaskPolicy* which may be:

- **`none`** indicates that no Listener should be attached.
- **`perCapsule`** indicates that the Capsule wide Listener will be attached to the Channel.
- **`perProtocol`** indicates that the Protocol wide Listener will be attached to the Channel;
- **`perChannel`** indicates the (ODP) Channel will get a dedicated Listener task.
- **`perSocket`** indicates that each socket will get a dedicated Listener task.

6.4.1.5 *Socket Management*

Policy for socket management on the server side may be configured by providing an `IOChannelTCP::Allocator` in the `ServerIIOPQoS` (class `IOChannelTCP` models a single connected socket). The following implementations of `IOChannelTCP::Allocator` are provided with DIMMA 2.0:

- An `IOChannelTCPFactory` (the default) allocates a new `IOChannelTCP` on demand;
- A pool of pre-allocated `IOChannelTCPs`, allows the reservation of a set of sockets (file descriptors) for the server channel to ensure that incoming connection requests can be honoured.

6.4.1.6 *TCP Socket configuration*

When a binder creates a TCP channel, it extracts relevant TCP Socket configuration parameters from the overall IIOP QoS supplied. This is the class `TCPQoS` which specifies such things as transmit and receive buffer size for the underlying TCP implementation.

6.4.2 IIOP Binder Operation

The IIOP binder is implemented as a class, one instance of which is logically contained within the IIOP protocol implementation. It is accessed via `ProtocolIIOP` by external clients (usually this means the protocol independent binders that are supplied with DIMMA to perform implict or explicit binds).

The binder provides operations to:

- create a server ODP channel, topped by a `ServerBindingIIOP`. Arguments are required to specify the address of the implementation, and the required QoS configuration for the channel;
- open a client ODP channel, topped by a `ClientBindingIIOP`. Arguments are required to identify the interface reference of the remote server, and the required QoS configuration for the channel.

6.4.2.1 Client-side channel construction

The `ModuleTCP` within `ProtocolIIOP` is used to create a basic `IOChannelTCP`. QoS parameters determine:

- the Listen task policy (see §6.4.1.4);
- the basic TCP socket configuration (see §6.4.1.6)
- the read buffer allocator (see §6.4.1.1)

Construction of the IIOP layer is not influenced by any current QoS parameters.

The session layer is configured according to the type of `ClientSession::Allocator` provided by the QoS object.

If a simple `ClientSession::Allocator` has been configured, then this is used directly as the (null) session layer.

If an allocator of `MultiplexClientSessions` has been provided by the QoS, then a session layer is constructed that consists of an instance of `ModuleClientSession` (constructed with the QoS specified allocator as an argument) to provide the session multiplex/de-multiplex services.

In this configuration `ModuleClientSession` becomes a proxy for the QoS specified `MultiplexClientSessions` allocator, as it needs to manage the set of waiting sessions in order to be able to de-multiplex upcalls to the correct session.

`ClientBindingIIOP` is built with the QoS specified write buffer allocator to provide a service to the stub to allocate buffers in which to construct requests.

6.4.2.2 Server-side channel construction

At the server side an instance of *ListenChannelTCP* rather than *IOChannelTCP* is created by the binder to listen for connect requests from clients. Essentially the same QoS parameters apply as for the client side, plus the following:

- an `IOChannelTCP::Allocator` is provided to the `ListenChannelTCP`, to be used to allocate a `IOChannelTCP` instance to handle a newly opened connection;
- a `Threading::Allocator` is provided for use by `ListenChannelTCP` when the Listen Task policy is perSocket, to allow resource reservation of a pool of tasks to service up to *n* client connections.

Construction of the IIOP layer is not influenced by any current QoS parameters.

The server session layer is constructed with a QoS specified `Threading::Allocator`. This provides for the control of concurrency of invocation of the server implementation.

Each client invocation runs in the context of a separate server session. The `Threading::Allocator` implementation provides the required unit of concurrency within which to schedule this server session (null Task, Task, Thread *etc*.).

6.5 AnsaFlow Binder

This section provides a description of the QoS parameters supported by the Binder, and a description of the operation of the AnsaFlow protocol binder.

The section can be regarded as a commentary on the construction of an AnsaFlow ODP channel stack, the structure which is described in section 9.8

6.5.1 AnsaFlowQoS

`AnsaFlowQoS` objects describing the required configuration of an ODP channel are passed into `ProtocolAnsaFlow` when the protocol is asked to create a Binding and a Channel stack. If no QoS object is specified, default QoS objects are created.

The following sub-sections describe the QoS parameters supported by the AnsaFlow binder:

6.5.1.1 Buffer Management

There are two aspects to Buffer management, with respect to QoS: Buffer Allocation; and Buffer overflow policy.

The `AnsaFlowQoS` object contains an implementation of the `Buffer::Allocator` interface, which provides an allocator for protocol specific buffers. The implementation of this interface may be provided by a client (of the binder). DIMMA 2.0 provides two standard implementations which are:

- a `BufferRTPFactory`, which creates Buffers on demand (this is the default); or
- a pool of `BufferRTPs` of a limited size.

The Buffer overflow policy is needed when marshallers and unmarshallers are attached to the Buffer, to specify whether the Buffer is to be static or dynamically extensible.

6.5.1.2 Session Management

The `ReceiverBindingAnsaFlowQoS` (supplied to the binding on construction) contains an implementation of the `Threading::Allocator` interface, which the binding uses to allocate Threading objects for execution of receiver sessions.

The `Threading::Allocator` implementations provided with DIMMA are described in section 6.4.1.3.

6.5.1.3 ListenerTaskPolicy

When the binder creates or opens a `ChannelUDPReceiver`, it potentially attaches a Listener (task) to it. The QoS supplied to the binder contains a ListenerTaskPolicy which is as described in section 6.4.1.4

6.5.1.4 UDP Socket configuration

When a binder creates a `ChannelUDPTransmitter` or `ChannelUDPReceiver`, it extracts relevant UDP Socket configuration parameters from the QoS supplied.

UDP QoS specifies such things as buffer size for the underlying UDP implementation.

6.5.2 AnsaFlow Binder Operation

The following events are a typical sequence which would occur as the AnsaFlow Protocol is constructed, asked to create a receiver channel and create a transmitter binding to it.

When a `ProtocolAnsaFlow` object is created it constructs one `ModuleRTP` object and one `ModuleUDP` object. The `ModuleUDP` object is given a pointer to the `ModuleRTP` object, as an implementation of the abstract framework class `ConnectionlessReadyModule`. This abstraction decouples `ModuleUDP` from `ModuleRTP`, allowing `ModuleUDP` to be deployed underneath any implementation of `ConnectionlessReadyModule`.

6.5.2.1 Receiver Binding

An external client (for example a protocol independent implicit or explicit Binder), would invoke CreateReceiver on `ProtocolAnsaFlow`. It would supply a `FlowDispatcher` object to dispatch messages to, optionally a Quality of Service object and possibly an address to listen for messages on. If no QoS is supplied, defaults are chosen. If no address is supplied, one is constructed which will later be published for transmitters to bind to.

In response to the CreateReceiver request, `ProtocolAnsaFlow` (which implements the AnsaFlow binder) will construct an ODP channel (see §9.8), by performing the following actions:

- invokes `CreateChannel` on `ModuleUDP` to create a `ReceiverChannelUDP`. This is an implementation of the framework interface `ConnectionlessInputChannel` and is configured with the UDP QoS described in §6.5.1.4. The `ReceiverChannelUDP` is connected to `ModuleUDP` (an implementation of `ConnectionlessInputModule`*)* to provide the required message de-multiplexing;
- invokes `CreateChannel` on `ModuleRTP`, to create a `ReceiverChannelRTP` connected to the lower `ConnectionlessInputChannel` implementation*;*
- creates a `ReceiverBindingAnsaFlow`, connected to the lower `ReceiverChannelRTP`;

6.5.2.2 Transmitter Binding

An external client (for example a protocol independent implicit or explicit Binder), would invoke `BindTransmitter` on `ProtocolAnsaFlow`. It would supply an address and, optionally, a Quality of Service object. In response to the `BindTransmitter` request, `ProtocolAnsaFlow` performs the following actions:

- invokes `CreateChannel` on `ModuleUDP` to create a `TransmitterChannelUDP`. This is an implementation of the framework interface `ConnectionlessOutputChannel` and is configured with the UDP QoS described in §6.5.1.4.;
- use `CreateChannel` on `ModuleRTP`, to create a `TransmitterChannelRTP` connected to the lower `ConnectionlessOutputChannel` implementation;
- creates a `TransmitterBindingAnsaFlow`, connected to the lower `TransmitterChannelRTP`;

7 Buffers and Marshalling

7.1 Overview

Buffers are containers for messages that are exchanged between capsules. Messages typically consist of user data and protocol specific information. The user data corresponds to a serialised form of the arguments supplied as part of an invocation. Protocol specific information typically comprises headers containing addressing information.

Marshalling is the term used to describe the serialisation of data such as invocation arguments, which may be presented as a variety of types: strings, structures and arrays for example. The inverse process of converting the serialised stream back to the original types is called unmarshalling.

Concrete buffers are manufactured by protocols which are responsible for the physical representation of the buffer. This representation may be optimised for the specific protocol, in terms of efficiently storing headers for example.

Apart from protocols, the main users of buffers are stubs. However, in order to shield the stubs from protocol specific details, stubs always access buffers through marshallers.

7.2 Buffers

Buffers are responsible for:

- Containing serialised messages
- Manipulating protocol specific header information

The physical layout of a buffer is determined by the protocol, hence protocols implement the concrete buffers. However, in order to simplify the handling of buffers within the protocol independent framework, all buffers conform to the generic `Buffer` type.

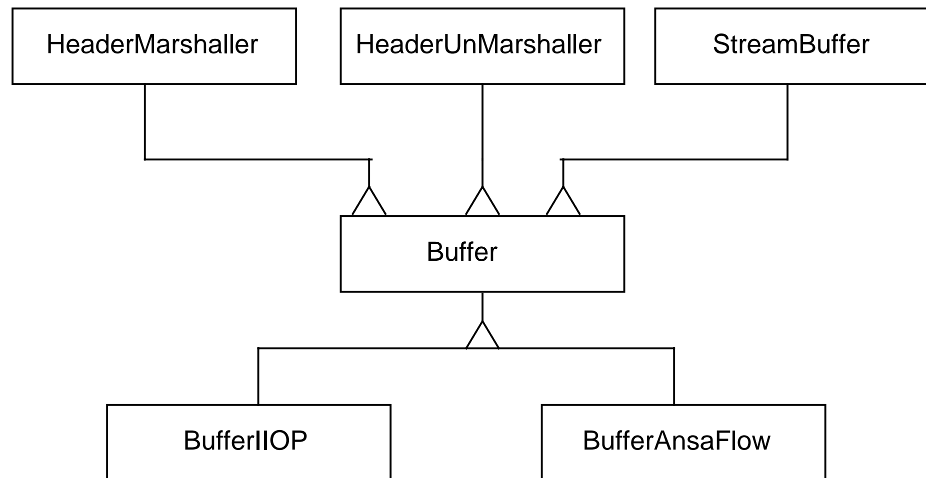
The manipulation of header information is expressed as interfaces (abstract classes) principally for the use of stubs. These provide methods to marshall and unmarshall generic request and response codes. Request codes are used by protocols to identify the operation to be invoked whilst response codes are used to deliver terminations to the client.

The class hierarchy is shown in figure 7.1.

`HeaderMarshaller` and `HeaderUnmarshaller` are interfaces, `StreamBuffer` is an abstract class defining the marshalling pointers and `Buffer` defines some generic manipulation methods for the use of protocols.

The marshalling pointers are held in `StreamBuffer` rather than in the marshallers to enable several marshallers to be used sequentially, each marshalling data according to different policies. For example, a protocol might wish to marshall its header in a different format from that of the payload, the format of the latter being defined by the stubs.

Figure 7.1: Buffer Class Hierarchy



7.3 Marshallers

Marshallers are responsible for serialising the values of primitive data types, e.g. the types directly supported by a particular IDL. Constructed types such as user defined structures are serialised by marshalling their constituent primitive types. In practice, code for the latter is normally generated by an IDL compiler.

A specific marshaller implements a single marshalling policy, e.g. little or big endian encoding. In practice, the number of variables influencing marshalling policy is small and this is implemented as a template class (`Marshalling`). The template class handles both marshalling and unmarshalling, and consequently implements two interfaces, `odp_Transmitter` and `odp_Receiver`. These interfaces define the marshalling and unmarshalling methods respectively for the primitive data types and do so in terms of the streaming operators >> and <<.

A marshaller is always associated with a buffer into which the serialised data is written using methods corresponding to primitive machine types e.g. byte, int, etc. A lowest common denominator approach such as writing everything as bytes is specifically *not* used as it results in poor performance.

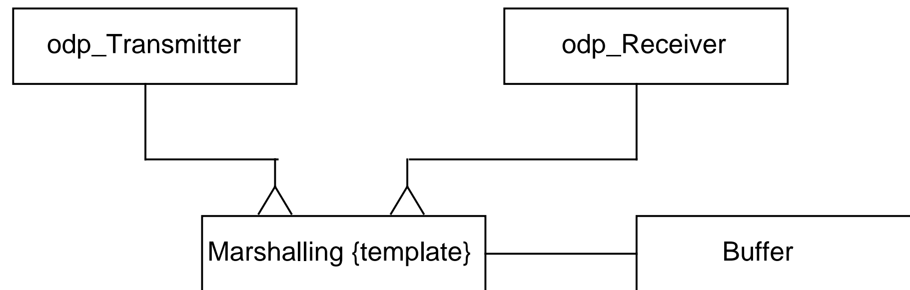
The class hierarchy for marshallers appears in figure 7.2.

7.4 Design notes

Buffer and marshallers in DIMMA are modelled after the C++ iostream classes and were originally implemented in terms of these classes. Subsequent performance analysis indicated that this was causing a significant performance problem and the implementation was reworked. Specifically the following problems were addressed:

- Too many calls on lowest common denominator routines such as writing a byte

Figure 7.2: Marshaller Class Hierarchy



- Dynamic marshalling policy caused excessive run-time overhead and was replaced with a static marshalling template
- Alignment method was found to be among the top five contributors to processor usage and was carefully optimised
- Small heavily used methods were made inline methods

A detailed description of the work carried out to analyse and increase the performance of DIMMA can be found in [APM.2046]

8 Resource Framework

8.1 Overview

The DIMMA resource framework is designed to support quality of service (QoS) specified configuration of resources, based on a model of resource reservation. In particular, the design aims to resource communication channels when the channel is established. Normally this is accomplished by an application making an explicit binding call and specifying a particular QoS. The QoS is mapped onto suitable engineering mechanisms and the resources for these are reserved and allocated to the channel.

This model of resource reservation, based on configuring a channel at bind time, seeks to maximise the performance of subsequent invocations by avoiding the need for run time checks in the communications path. In addition, by reserving resources for the sole use of a channel, crosstalk between channels is minimised and there is a bounded latency[1] in terms of allocating resources.

DIMMA employs two concepts to manage resources such that the communications path is unaware of whether resource reservation is being used or not. In other words, the configuration of a channel based on QoS specified resource reservation, or that using implicit binding with no reservation, is transparent to the communications path. The two concepts are `Factories` and `Pools`. Both implement the same `ResourceAllocator` interface which is used by the communications channels.

8.2 Factories

A factory manufactures resources on demand, i.e. it supports a model where no resource reservation is required. For example, allocation of a protocol buffer would result in a call to the C++ operator `new`.

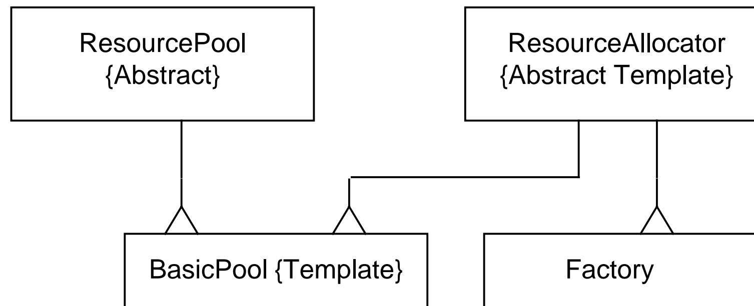
8.3 Pools

A pool is a container or cache for pre-allocated resources of a particular type and configuration, i.e. all resources in a pool are identical instances of a specific type. A pool is instantiated with a size and a factory from which to obtain the resources. The constructor allocates the required number of resources from the factory and places then on an internal list.

A resource is obtained from a `Pool` using the `Allocate`() method. Since all resources are equal, the first resource from the internal list is returned. The object model for pools is given in figure 8.1.

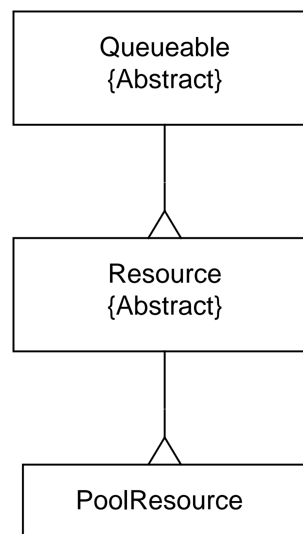
1. There will always be some latency since even a reserved resource must be allocated (removed) from a pool or cache.

Figure 8.1: Pool object model



Resources are returned to the pool by calling `Free`() on the resource rather than on the pool. This avoids the need for the caller to keep track of from where the resource was allocated and helps prevent errors arising from attempts to return a resource to the wrong pool. All resources capable of being placed in pools must inherit from the `PoolResource` class which keeps track of the pool to which they must be returned and implements the `Free`() method. The object model for resources appears as figure 8.2.

Figure 8.2: Resource object model



9 Communications Framework

9.1 Purpose

The purpose of the communications framework is to simplify the production of new protocols:

- by providing general components that may be re-used
- by layering functionality to:
  - facilitate code re-use across multiple protocols
  - provide configurable functionality through composition of layers
- using a set of general interfaces together with construction guidelines
- using a uniform approach to resourcing for QoS

9.2 Overview

The communications framework comprises a set of generic interfaces and components, and informal guidelines pertaining to their use.

Due to the requirement to support many diverse protocols, e.g connection oriented, connectionless, RPC, flow, etc., the protocol framework cannot be too prescriptive and aims instead to provide a minimal number of generally useful 'building blocks'. Likewise, many of the compositional constructs are too informal to express in a strongly typed language like C++ and instead are presented as a 'cook book' of guidelines.

The framework considers a protocol to comprise a set of *modules*, each supporting a layer of protocol. The definition of what constitutes a protocol layer is not a formal one: it may be anything that is reasonably self-contained in terms of the framework interfaces. That said, a protocol layer typically has at least one of the following characteristics:

- performs multiplexing of demultiplexing of messages
- dispatches messages to stubs
- interfaces with the operating system network facilities

To this end, a *module* (providing a layer of protocol) will normally be associated with specific message header information, i.e. it will typically add a header on message transmission and remove it on receipt. This header is used to hold addressing information for message multiplexing.

Modules create *channels* in response to requests from their associated protocol and these act as conduits for message transmission and reception. Like the modules of a protocol, *channels* are also layered, forming a channel 'stack' or *ODP Channel*.

9.2.1 Message processing

Messages arriving from a stub are presented to the top channel and make their way down the channel stack until they reach the lowest level channel (called anchor channels) which passes them to the operating system network interface.

The processing of incoming messages is not necessarily symmetric with that of transmission. Messages arriving from the network are processed by low level channels but cannot necessarily be passed directly to the next higher level channel, e.g. when there is channel multiplexing. In this case, the message is passed by the channel to the next higher level module, which interprets the associated header information to identify the next higher level channel to which the message is then passed. In this way, the message makes it way up through the protocol, alternating between channel and module, until it reaches the highest channel where it is dispatched to the stub.

The framework allows for optional message concurrency on a channel through the concept of a session layer. This may be regarded as a layer of multiplexing between the highest level channel and the stub.

9.2.2 Network interface

The current DIMMA protocols both interface to the network through the operating system provided socket abstraction. In each case, the anchor channels encapsulate a socket and provide the I/O interface to the protocol.

9.2.3 Threading model

The DIMMA communications framework is intentionally independent of the model of threading used to drive the protocol. Messages may be sent and received synchronously, asynchronously or a mixture of both, e.g. synchronous transmit and asynchronous receive.

9.2.4 Resourcing

A key objective of the communication framework is to separate the concerns of resource reservation from the functional aspects of the protocol. This allows binders to achieve a wide variety of QoS by attaching suitable resource allocators to specific channels.

Resources are allocated to individual *ODP Channels* by the protocol specific binder. The binder constructs suitable pools and factories for the key resources, and attaches these to the objects in the ODP Channel which need them.

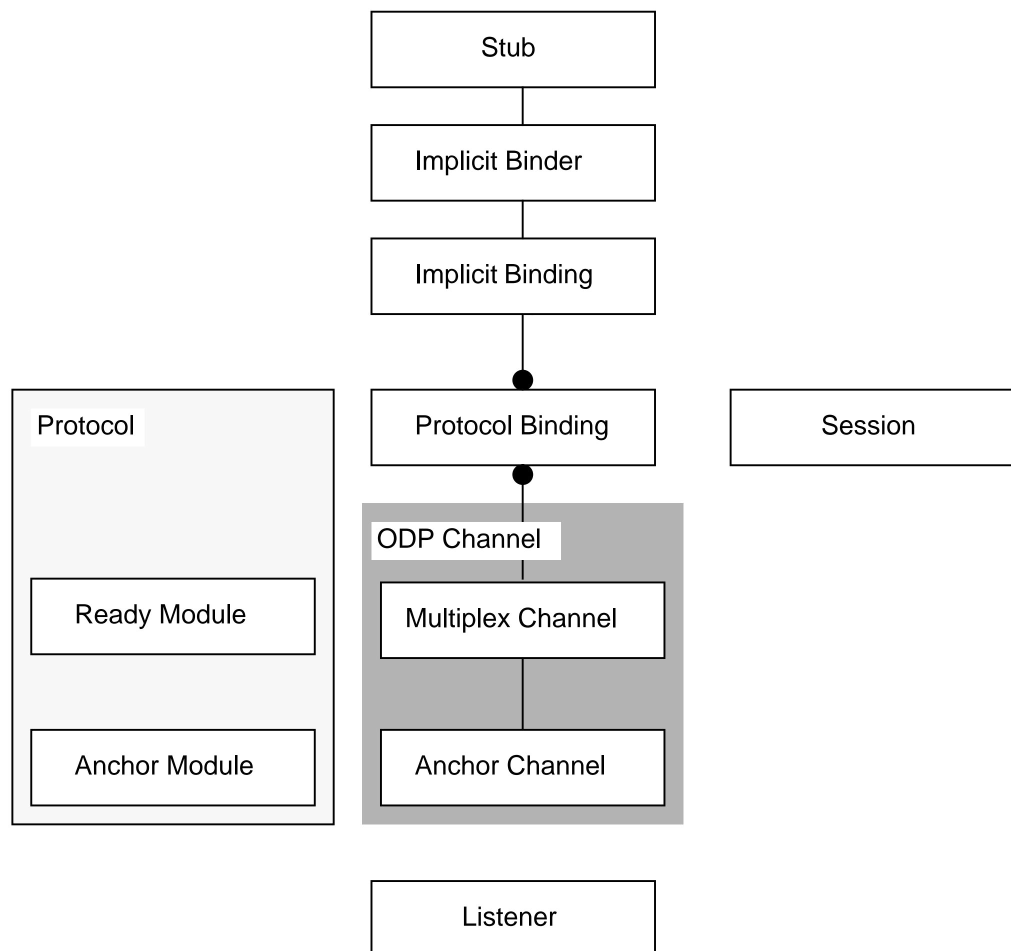
For example, a threading allocator (pool or factory) might be assigned to a protocol binding. The latter would use this to obtain threads to process each invocation.

9.3 Protocol Construction

The major components of the communications framework appear in figure 9.1

Whilst these are not all directly associated with the core protocol framework, it is helpful to set the framework within the context of a complete communications path between stub and network (a local engineering binding).

Figure 9.1: A local engineering binding



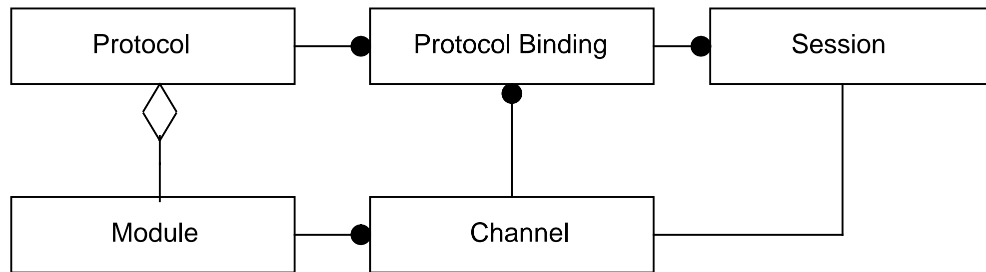
A local engineering binding can be broadly divided horizontally: components below the *protocol binding* are protocol specific and those above are protocol independent. The former are the subject of this chapter whilst the latter are related to the process of binding and are described in chapter 6.

The primary components of the framework are:

- protocol
- module
- channel
- protocol binding
- session
- listener

The relationship between these components is captured more formally in an OMT object diagram appearing as figure 9.2. These components are described in more detail in the next sections of the document.

Figure 9.2: Framework classes



9.3.1 Protocol

A *protocol* supports message exchange between capsules in a specific format and according to a given set of rules. For example, in DIMMA, the CORBA IIOP RPC protocol is implemented by the class `ProtocolIIOP`.

A protocol is viewed primarily as a factory for ODP channels by the protocol independent part of the DPE. A protocol is able to construct a server channel for an interface, or a client channel associated with a given interface.

ODP channels represent peer to peer connections that provide invocation (RPC) or flow semantics according to the type of protocol.

9.3.1.1 Interfaces

A *Protocol* implements three interfaces:

- Interface Reference profile support
- A factory for ODP Channels offering a `ProtocolBinding` interface
- A factory for ODP Channels offering a `Channel` interface

All of the above interfaces are used by Binders.

The interface reference profile support interface provides methods to retrieve the protocol tag associated with the protocol and the profile corresponding to a given ODP Channel. These are used by the binder to construct an interface reference for a specific ODP channel when a reference to an interface is to be passed out of the capsule.

The `ProtocolBinding` factory interface manufactures complete communications channels, or ODP channels that offer a (specialisation of) the `ProtocolBinding` interface (see §9.3.4 for a description of the services offered by this type of interface).

The final interface is similar to the above except that it creates a more primitive ODP channel that offers a specialisation of the `Channel` interface rather than a *Protocol Binding* interface (see §9.3.3 for a description of the services offered by this type of interface).

9.3.1.2 Implementation

A *protocol* is implemented as a ‘stack’ of protocol layers each managed by an associated *module*. It provides a *binder* that interfaces with the *modules* to construct an ODP channel, which itself consists of a ‘stack’ of *channels*.

9.3.2 Module

Modules are responsible for managing the details of a particular protocol layer. They:

- act as a factory for *Channels* for the associated *Protocol*
- implement the message semantics of the protocol layer
- assist lower level *Channels* in passing an incoming message from the network up through the protocol

It is the module that actually implements the protocol layer, i.e. it understands the format and semantics of messages. Modules typically add protocol headers as messages pass down through the protocol and remove them on the way up. The headers are used to hold addressing information to enable *Channel* multiplexing.

Modules multiplex a set of channels onto a lower layer one by recording an identifier for the channel in the protocol header as the message passes down the protocol from channel to module.

Demultiplexing is performed when a lower level channel passes an incoming message up to the next higher level module. The module uses the channel identifier in the protocol header to locate the appropriate channel and calls the latter's upcall method.

9.3.2.1 Interface

A module has interfaces conceptually located:

- at the top - providing *Channel* factory methods
- at the bottom - providing upcall messaging methods

There are two variants of the latter. Which of these a module implements depends on the location of the module in the protocol.

A module at the bottom of a protocol is called an anchor module and often has no upcall interface. This is because its channels directly encapsulate the network message I/O interface (e.g. sockets) and demultiplexing of sockets is not performed by the module[1].

Modules at the next higher level will usually implement the Ready method which is called by anchor channels when a message arrives. A module will typically respond to this by using the anchor channel to read the message which is then passed to the appropriate channel.

Other higher level modules are normally demultiplexing modules and implement the Upcallable interface, which takes the message as a parameter and uses the header it contains to perform the demultiplexing.

9.3.3 Channel

Channels provide the interface to the associated protocol layer for real I/O, i.e transmitting and receiving messages. The messaging model consists of simple transmit and receive methods.

Like *Modules*, *Channels* are associated with interfaces conceptually located:

- at the top - providing message I/O to upper level channels

1. In the existing DIMMA protocols, the demultiplexing function is performed by the Listener.

- at the bottom - providing upcall methods for messages arriving from the network via their respective *module*.

The similarity with modules extends to the type of channel which is dependent on its location in the protocol. The lowest level channel, or anchor channel, encapsulates the network messaging interface and uses this to provide real I/O.

Channels at other positions in the protocol assist their modules in providing multiplexing. A multiplexed channel contains a unique identifier which is used by the module to identify the channel when demultiplexing an incoming message.

9.3.4 Protocol Binding

The *protocol binding* interfaces the ODP channel with the protocol independent part of the local binding and provides semantics appropriate to the type of stub, e.g. an operational stub would require an RPC protocol binding with methods such as *call*.

Protocol binding is a somewhat overloaded class and it is responsible for:

- mapping stub semantics, e.g. RPC, onto those of a simple I/O channel
- providing message concurrency on a channel by performing session management
- manufacturing marshallers for stubs
- constructing interface reference profiles on behalf of protocol independent binders

9.4 Invocation Concurrency

Sessions represent invocation context and are used to manage the multiplexing of invocations over a single channel. *Sessions* are considered protocol specific and have no generic framework interfaces.

In the existing DIMMA protocol implementations, *sessions* are managed by the *protocol binding*.

9.5 Message Concurrency

Protocol listeners provide an implementation for driving protocols using the upcall messaging model. A *listener* listens for activity on an operating system socket and upcalls the protocol's anchor module on receipt of an event. Listeners are intended to be run in their own thread.

DIMMA has two implementations of Listener:

- `Listener`
- `SharedListener`

The first is intended for protocols using a dedicated threading model, i.e one thread for each anchor channel and hence only one underlying socket.

The `SharedListener` provides multiplexes its thread over a set of anchor channels and provides the module with the identity of the channel on which the event arrived as part of the upcall.

Both implementations offer an interface to add and remove channels from the set on which to listen. Although a `Listener` can only listen on a single channel, a common interface makes this transparent to the rest of the protocol.

Both implement the `Runnable` interface since they are intended to be executed by a dedicated thread.

9.6 Dynamic Protocol Loading

The DIMMA communication framework is designed to support multiple RPC and/or Flow protocols.

In order to provide a microkernel DPE with minimum footprint, all DIMMA protocol implementations are constructed as separate, dynamically loadable, shared objects that are only loaded into a capsule address space if required.

Typically a DIMMA application is linked without reference to *any* protocol implementations: the protocol selection and dynamic link/loading is performed at run time by a binder.

9.6.1 Native Protocols

The base DIMMA 2.0 provides two ‘native’ protocol implementations:

- IIOP providing RPC services;
- AnsaFlow, based upon the RTP flow protocol to provide multi-media flows;

IIOP and AnsaFlow are considered ‘native’, as the DIMMA nucleus is aware of the existence of these protocols, and is able to automatically locate and load the shared objects when required.

9.6.2 Adding new Protocols

Non-native protocol implementations may be added to DIMMA with great simplicity: all that DIMMA requires of the new protocol implementation is that it:

1. implements the Protocol interfaces described in §9.3.1
2. makes itself known to DIMMA by a simple registration process. This is currently implemented as an entry in a text file that describes for the protocol:
   (i) its name (e.g. “ProtocolIIOP”): used by DIMMA to determine the name of the shared object containing the protocol implementation
   (ii) its type (which may be RPC or Flow): binders need to be aware of the capabilities of a protocol
   (iii) its Interface Reference tag (e.g. TAG_INTERNET_IOP): see §5.5 for a discussion of DIMMA interface references)

9.6.3 Protocol Loading

When a DIMMA application is started, no protocols are linked or loaded with the process.

The DIMMA capsule contains a single *protocol loader*, which has knowledge of the native protocols, and makes use of protocol registry information to determine if any user defined protocols exist.

The *protocol loader* offers facilities to the Binders to access a protocol by Interface Reference tag, or provides iterators whereby a Binder may iterate over the set of Flow or RPC protocols.

The first time that a protocol is requested, the loader will load and dynamically link it with the application process and return a protocol reference to the Binder. Subsequent requests for access to the protocol are met by simply returning a reference to the loaded protocol.

9.6.4 Protocol selection

Protocol selection is the business of the Binders (see §6). An explicit binder will always choose which protocol to use [this is specified as a QoS parameter for the Explicit Binder], the implicit binders are pre-configured with a default policy for protocol selection.

The DIMMA 2.0 implicit binders are configured with the following policies:

- the RPC server binder will create an ODP channel for each RPC protocol known to DIMMA. This set of alternate channels to the interface implementation is exported as a sequence of protocol profiles within an Interface Reference as described in §5.5);
- in a similar fashion to the RPC server binder the Flow receiver binder will create an ODP channel for each Flow protocol known to DIMMA.
- the implicit RPC client binder will iterate over the set of protocol profiles within an Interface Reference, until it find a protocol tag that the protocol loader recognises and can provide a protocol implementation for. This protocol is then used to provide a path to the implementation;
- in a similar fashion to the RPC client binder the Flow transmitter binder will create an ODP channel for the first Flow protocol that it recognises in the Interface Reference's set of protocol profiles.

9.7 IIOP Implementation

DIMMA RPC is supported by an implementation of IIOP within the protocol framework. This section provides an overview of that implementation.

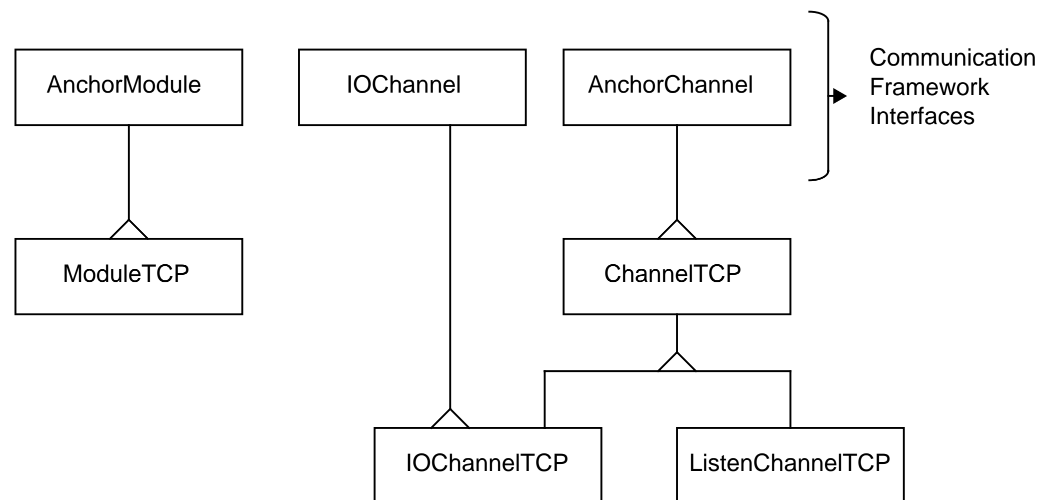
Within the protocol itself, there are distinct layers:

- TCP, which provides the Anchor Module/Channel for the protocol stack. The implementation of the TCP layer is independent of IIOP, and is intended to be available for re-use in other connection oriented protocols which require TCP transport;
- the IIOP layer itself, which provide the Ready module and Multiplex Channel implementations. All knowledge of the IIOP protocol is encapsulated within this layer;
- the session layer which provides support for thread multiplexing over an IIOP channel

9.7.1 TCP Layer

The class hierarchy for this protocol layer is illustrated in figure 9.3 The TCP layer is an implementation of various interfaces prescribed by the communication framework.

Figure 9.3: TCP Layer Class hierarchy



`ChannelTCP` provides the implementation common to all Channels in the TCP layer.

`ListenChannelTCP` is a specialisation of `ChannelTCP` that models the socket upon which a server listens for new connections. Its implementation of the `AnchorChannel` Ready method is capable of accepting a connection request on a listen socket, and manufactures an instance of `IOChannelTCP` to provide basic IO services for the new connection.

`IOChannelTCP` is a specialisation of `ChannelTCP` that models a connected socket (client or server). It provides Transmit/Receive methods to allow IO to be performed to/from the specified connection. In addition its implementation of the `AnchorChannel` Ready method will dispatch an (upcall) to an upper level Ready module, when invoked.

`ModuleTCP` is simply a factory for objects of type `ListenChannelTCP` and `IOChannelTCP` (note it is used a factory for objects of type `IOChannelTCP` only at the client end, at the server end `ListenChannelTCP` performs this service)

9.7.2 IIOP layer

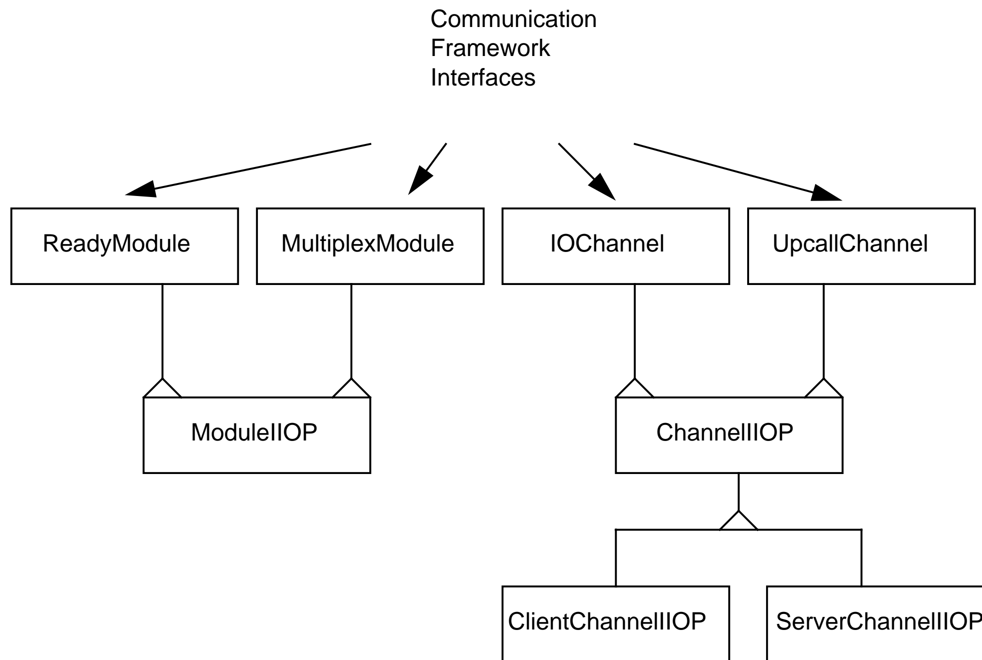
The class hierarchy for this protocol layer is illustrated in figure 9.4 The concrete classes within the IIOP layer provide implementations of framework specified interfaces.

`ModuleIIOP` is an implementation of both a `ReadyModule`, and a `MultiplexModule`. The class is intended to encapsulate the knowledge of the IIOP protocol.

As a `MultiplexModule`, `ModuleIIOP` provides factory methods to create objects of type `ChannelIIOP`. The methods offer the possibility to create a `ServerChannelIIOP` or open a `ClientChannelIIOP`. In both cases a lower (transport) `IOChannel` implementation must be provided, in the latter case the address of the remote Endpoint must also be supplied.

As a `ReadyModule`, `ModuleIIOP` provides an upcall interface whereby a lower protocol layer (i.e. TCP) may alert the Module when data is available on a lower channel.

Figure 9.4: IIOP Layer Class hierarchy



Class `ChannelIIOP` is an abstract class providing shared implementation between Client and Server IIOP channels.

`ClientChannelIIOP` provides the client side channel implementation. It offers both Transmit and Receive methods to support a 'push down'/'pull up' model for requests and responses, and an Upcall method to allow a 'push up' of responses from a server.

`ServerChannelIIOP` implements essentially the same interfaces as `ClientChannelIIOP`, but in this case the Transmit method creates a response, and the Receive/Upcall methods are for incoming requests.

9.7.3 Client Session Layer

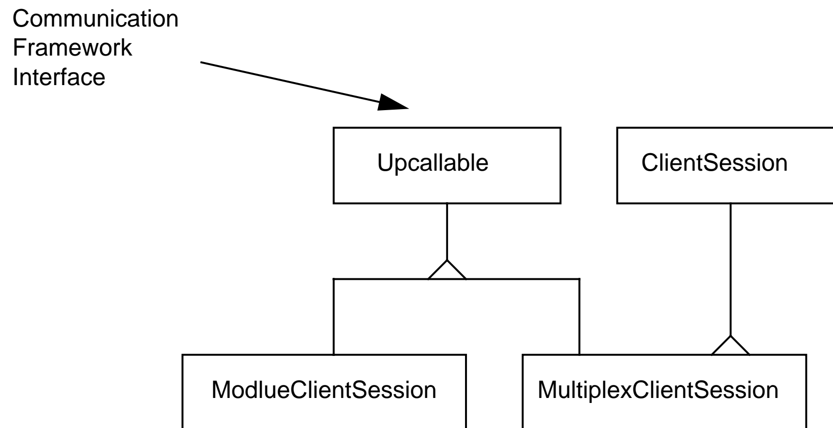
The class hierarchy for this protocol layer is illustrated in figure 9.5 The session layer is present, in essence, to provide a task multiplexing and de-multiplexing service required in a multi-threaded environment.

The basic `ClientSession` class provides a simple implementation of a null session layer. It provides a synchronous Call method, and an asynchronous Transmit method (for oneway operations) for use by upper layer software.

The `ClientSession` implementation is intended for use in a channel that has a dedicated task, and hence no requirement for task or thread multiplexing. The Call method is implemented as a simple Transmit followed by a Receive on the lower level IOChannel.

The `MultiplexClientSession`, in conjunction with `ModuleClientSession` provides a thread multiplexing service. `MultiplexClientSession` implements the Call method by invoking Transmit on the lower level `IOChannel`, and then blocks the client thread awaiting a response. `ModuleClientSession` provides a de-multiplexing service to dispatch incoming responses to the correct blocked client session.

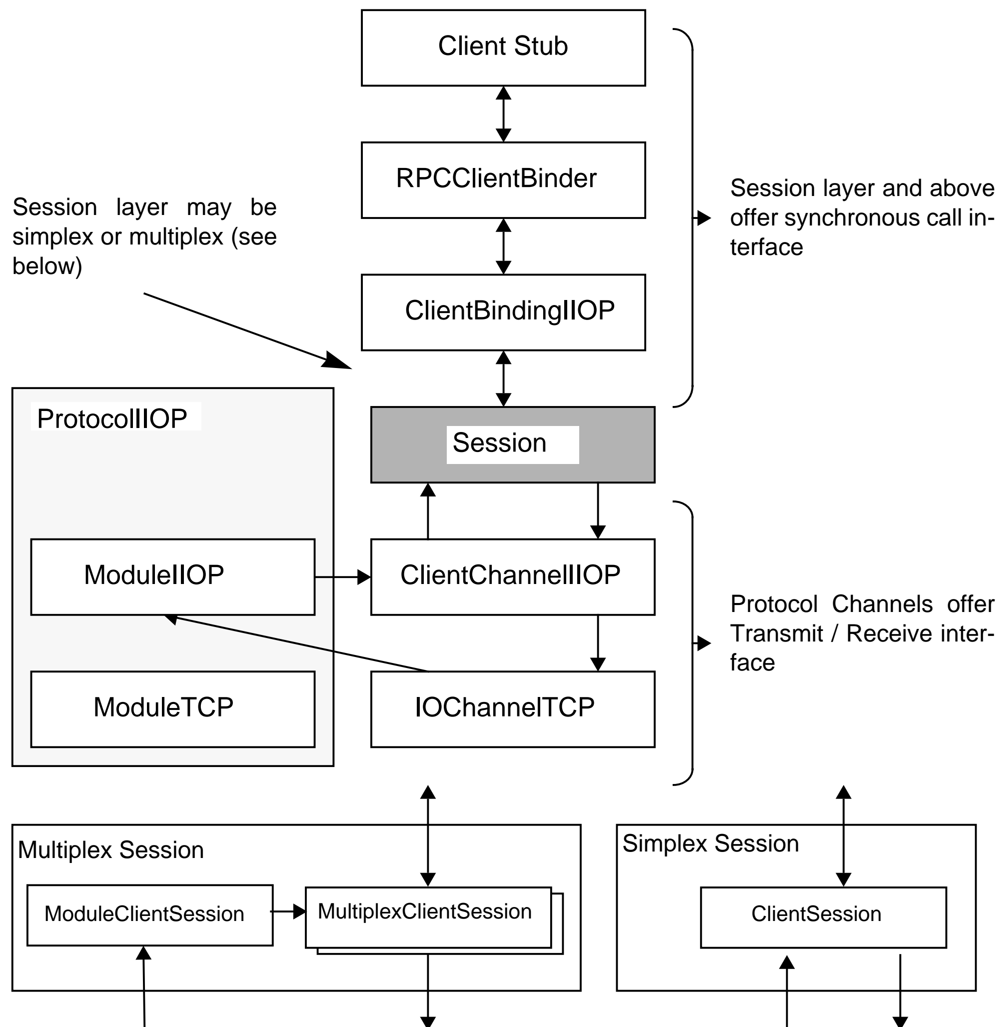
Figure 9.5: Session Layer Class hierarchy



9.7.4 Complete Client-side Protocol Stack

Figure 9.6 provides an illustration of the composition of the IIOP client side of

Figure 9.6: IIOP Client side protocol stack



the protocol stack.

The classes that are used to compose individual layers of the IIOP Protocol stack (TCP, IIOP and client session) have been described in the preceding sections.

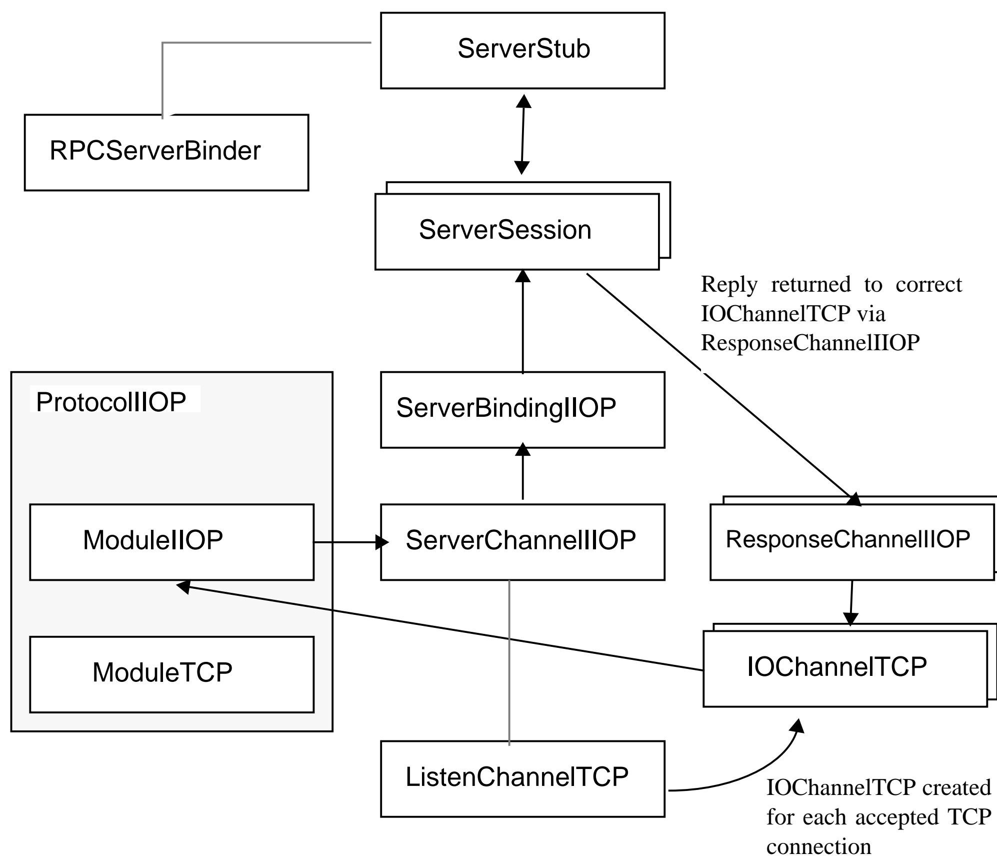
Above the session layer a client binding exists (`ClientBindingIIOP`), which provides a protocol independent "cap" to the overall ODP channel. The binding offers a Call method, with Request/Response arguments that are instances of Marshaller / unMarshaller interfaces, rather than the Buffers used at lower layers.

In the default client side configuration shown, there is a protocol independent implicit RPC client binder between the binding and the stub. Initially, when the stub is first created only the Binder exists. At an appropriate time (see §9.6.4) the Binder will make use of the protocol specific binder services (offered by the `ProtocolIIOP` object) to construct the full ODP channel.

9.7.5 Server-side Protocol Stack

Figure 9.7 provides an illustration of the composition of the IIOP server side of

Figure 9.7: IIOP Server side protocol stack



the protocol stack.

The RPC server binder is attached to the stub, and creates the ODP channel when an interface reference is export from the capsule.

At the TCP layer, the `ListenChannelTCP` is created to listen upon the TCP port that is advertised in the interface reference.

For each connection accepted by the `ListenChannelTCP` an `IOChannelTCP` is created to service the connection to the client. Incoming requests on an `IOChannelTCP` are dispatched to the upper layer `ModuleIIOP` to be de-multiplexed to the appropriate `ServerChannelIIOP`.

Context onto the originating `IOChannelTCP` is maintained by a `ResponseChannelIIOP`, so that responses from the server interface implementation can be transmitted back to the correct client.

Server Sessions within the current `ProtocolIIOP` implementation are simpler than the equivalent client sessions: a new server session is created to service each incoming request (however the unit of concurrency within which a server session executes is controlled by a QoS parameter).

9.8 Flow Implementation

DIMMA supports Flow interfaces within the protocol framework. Flow interfaces are asynchronous, one-way method calls and support continuous multi-media streams from Transmitters to Receivers. The implementation of the Flow protocol is called `ProtocolAnsaFlow`.

A complete AnsaFlow communications channel consists of two Channel layers with a Binding layer to ‘cap’ the Channel:

- The Connectionless Channel UDP Layer which is the anchor for the channel stack and encapsulates a network endpoint which is a UDP socket.
- The Channel RTP Layer which encapsulates knowledge of the RealTimeProtocol headers attached to messages, and offers a Connected Channel interface above the Connectionless layer below.
- The BindingAnsaFlow Layer which encapsulates knowledge of how to Transmit messages down the AnsaFlow Channel stack and how to Upcall messages received.

The `ProtocolAnsaFlow` object is similarly layered to support manufacture of AnsaFlow communications channels. The two Channel manufacturing layers are:

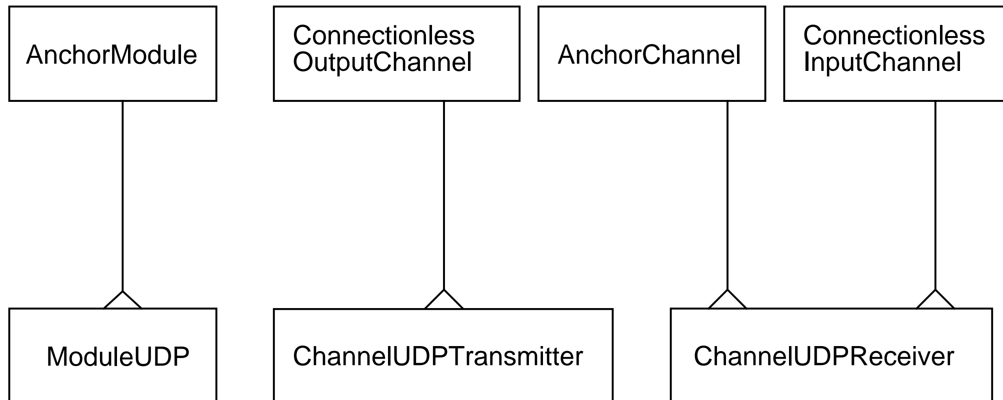
- `ModuleUDP` is the anchor module which creates `ChannelUDP` objects communicating on a network endpoint with a given UDP QoS configuration.
- `ModuleRTP` is a Ready Module and a Multiplex Module, supporting connected `ChannelRTP` objects over a connectionless transport layer.

The `ProtocolAnsaFlow` object itself manufactures `BindingAnsaFlow` objects when establishing a binding.

9.8.1 UDP Layer

The class diagrams for the UDP layer are shown in figure 9.8. `ChannelUDPTransmitter` does not inherit from `AnchorChannel` because `AnchorChannel` is used by Listeners to inform them of message arrival, but Transmitters never receive messages (there are no replies from Flow messages).

Figure 9.8: UDP Layer Class Hierarchy



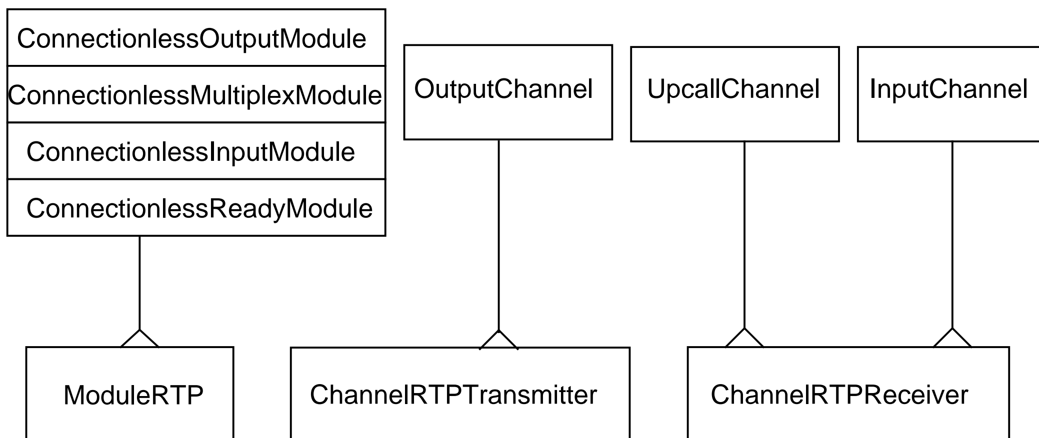
`ModuleUDP` passes `ChannelUDPs` a `UDPQoS` object when it constructs them. This mostly specifies low level socket configuration, but also contains a `ListenerTaskPolicy`, specifying whether the Listener on a `ChannelUDPReceiver`'s socket should be the capsule listener, the protocol listener, a dedicated channel listener, or none.

Some extra implementation inheritance is not shown. `ModuleUDP` inherits from `ResourcePool`, and `ChannelUDPs` inherit from `PoolResource`, allowing easy management by `ModuleUDP` of the Channels it has created.

9.8.2 RTP Layer

The class diagrams for the RTP layer are shown in figure 9.9.

Figure 9.9: RTP Layer Class Hierarchy



`ChannelRTPTransmitter` does not inherit from `UpcallChannel` because Transmitters never receive messages (there are no replies from Flow messages).

ChannelRTPTransmitters use ModuleRTP to transmit messages, allowing multiplexing of ChannelRTPTransmitters over Connectionless transport channels. The Connectionless channel layer below uses ModuleRTP's ConnectionlessReadyModule interface to inform it that a message is ready to be received.

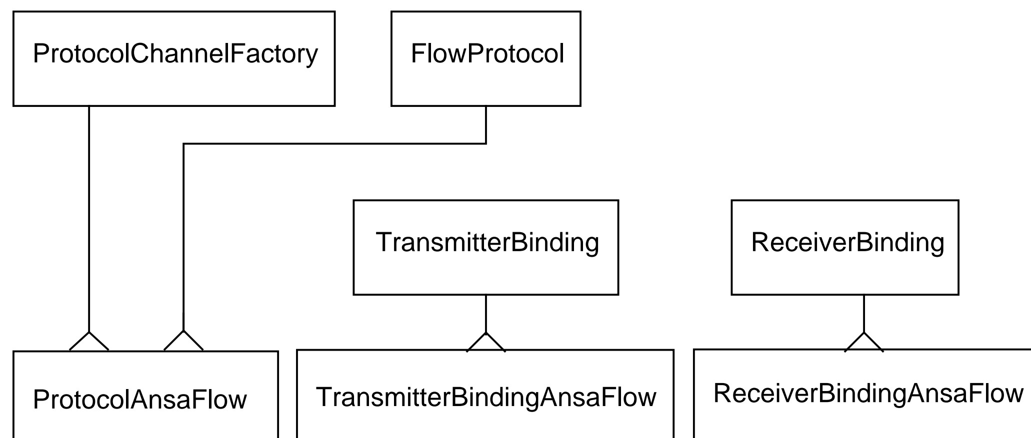
ModuleRTP passes a ChannelRTPReceivers an RTPReceiverQoS object when it constructs it. This holds an RTP Buffer allocator. The channel uses the buffer allocator for buffer resource management on incoming messages. Buffer management on the transmit side is handled at the binding level.

Some extra implementation inheritance is not shown. ModuleRTP inherits from ResourcePool, and ChannelRTPs inherit from PoolResource, allowing easy management by ModuleRTP of the Channels it has created.

9.8.3 Binding Layer

The class diagrams for the AnsaFlow Binding layer are shown in figure 9.10.

Figure 9.10: AnsaFlow Binding Layer Class Hierarchy



The BindingAnsaFlow layer provides Transmit and Upcall of Flow information.

ProtocolAnsaFlow passes TransmitterBindingAnsaFlow a TransmitterBindingAnsaFlowQoS object when it constructs it. This holds an RTP Buffer allocator. The binding uses the buffer allocator for buffer resource management of outgoing messages. Buffer management on the receive side is handled at the RTP level.

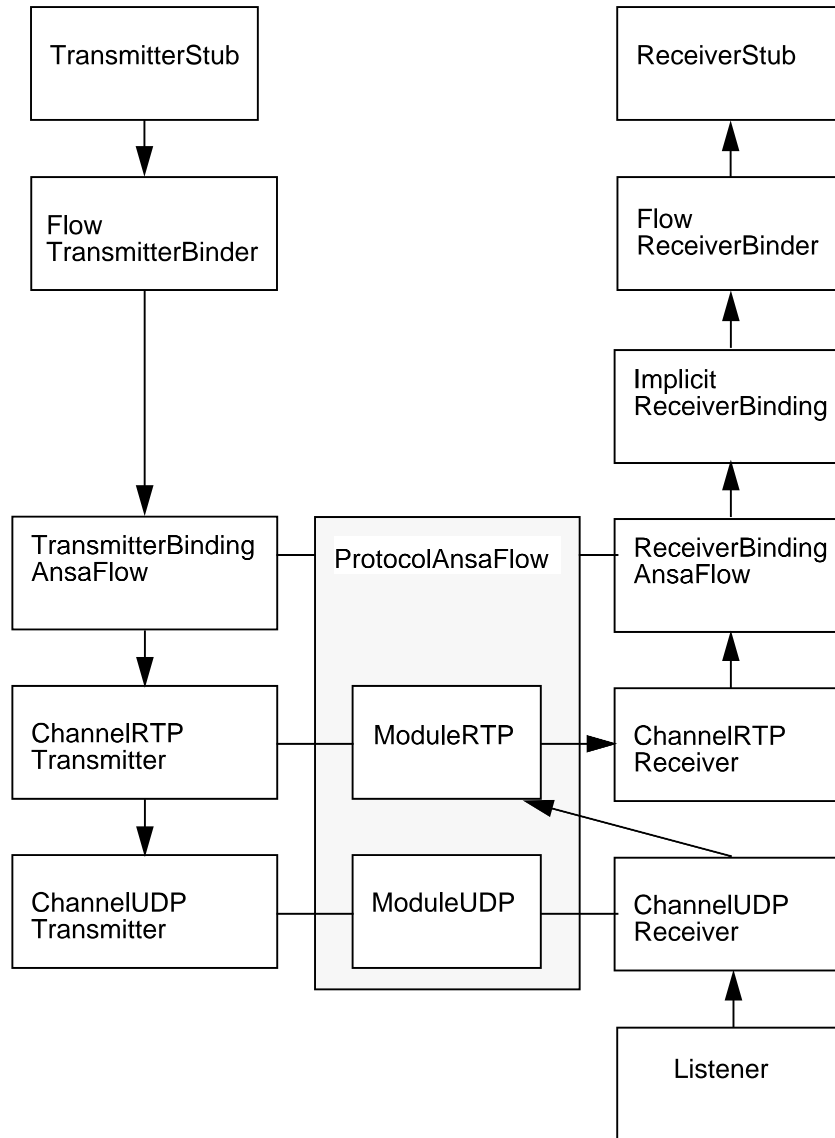
Both Transmitter and ReceiverBindingAnsaFlow objects receive a BufferOverflowPolicy object in their QoS. This controls whether the buffers managed by the channel stack may be dynamically extended or not.

ProtocolAnsaFlow passes ReceiverBindingAnsaFlow a ReceiverBindingAnsaFlowQoS object when it constructs it. This holds a Threading allocator. The binding upcalls its FlowDispatcher with messages in the context of a Task obtained from the allocator. No session management is needed as there will be no replies to the transmitter from a Flow upcall.

9.8.4 Complete Protocol Stack

The composition of a `ProtocolAnsaFlow` communication channel is shown in Figure 9.11.

Figure 9.11: Protocol AnsaFlow Structure



The `FlowReceiverBinder` is attached to a `ReceiverStub`, and creates the ODP Channel when an interface reference is exported from the capsule. The `ImplicitReceiverBinding` may sit above other Flow implementations, allowing transmitters a choice of protocol to communicate with the receiver.

A `TransmitterStub` uses a `FlowTransmitterBinder` as a binding. The binder creates a `TransmitterBindingAnsaFlow` when it is first used. The message to be transmitted is then sent to the `ChannelRTPTransmitter` where RTP headers are added. The `ChannelRTPTransmitter` caches the address of the receiver channel. It sends the message to that address using its `ModuleRTP`. This module uses the `ChannelUDPTransmitter` to transmit the message onto the network.

When the network delivers the message to the receiver's network endpoint, the receiver channel's `Listener` informs the ChannelUDPReceiver. This in turn informs the `ModuleRTP`. The `ModuleRTP` requests a buffer from the `ChannelRTPReceiver` above the `ChannelUDPReceiver`. The `ChannelUDPReceiver` then receives the message into that buffer. Then the `ChannelRTPReceiver` removes the RTP headers and upcalls its `ReceiverBindingAnsaFlow`. This gets a `Task` from its threading allocator, and upcalls its `FlowDispatcher` in the context of that `Task`.

If implicit binding is used, no network address is supplied when the receiver-side communication channel is created. The `ChannelUDPReceiver` requests a network endpoint from the operating system, and the EndPoints address gets incorporated into the `ChannelODPAddress` published in the interface reference. The transmitter-side communication channel transmits to this address.

If explicit binding is used, a well-known network address *may* be supplied to the explicit binder. If a well-known address is used this way, then:

- The address may be a Multicast IP address, allowing multicast flows from a transmitter to multiple receivers all bound to the same address.
- The transmitter may be constructed and started before the receiver. UDP does not require the receiver to be listening.

10 Threading

10.1 Overview

DIMMA uses the concept of threads to provide processing concurrency both within the DPE and in application objects. The underlying engineering mechanism used to provide real concurrency[1] are POSIX threads (see [POSIX]). These are not presented directly to applications nor to the generic DPE, in order to avoid unnecessary dependencies on operating system specific code. Instead the interface to the thread functionality is presented through the `Task` abstraction.

The `Task` abstraction in isolation would not be sufficiently flexible to provide fine grain control over concurrency within servers. As a result, DIMMA provides a variety of concurrency control abstractions which may be scheduled over `Tasks`.

All `Threading` abstractions support the `Threading` interface so that alternative implementations may be substituted without change to the calling code. This reflects the general policy in DIMMA which is to control behaviour through substitution of different mechanisms (implementation selected at bind time), rather than relying on run-time selection each time through the main code path.

10.2 Tasks

Tasks are modelled after Java threads since the latter provides a simple yet powerful interface. A DIMMA `Task` implements the `Threading` interface and is itself implemented as a thin layer over POSIX threads.

A class wishing to make use of processing concurrency has two options:

- It may inherit directly from `Task or`
- it may inherit from `Runnable` and be subsequently *installed* at run-time in an existing or newly created `Task`

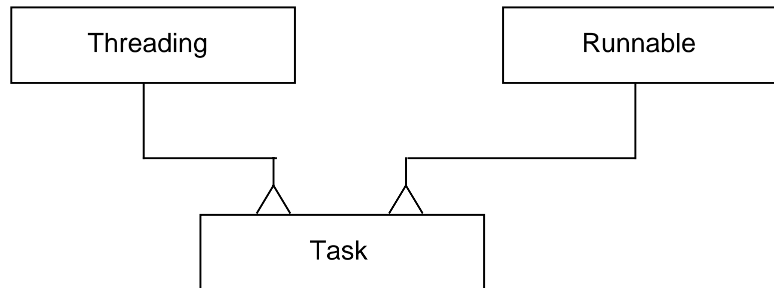
Either way, the key is the `Runnable` interface which is also implemented by `Task`. `Runnable` defines a single abstract method called `Run`() which will be called when the task starts running. Any class wishing to run in a separate thread of execution must implement (override) the `Run`() method.

The `Threading` interface provides control methods to control starting, stopping and awaiting the completion status of the associated task.

The class hierarchy of `Task` is illustrated in figure 10.1.

1. Real concurrency on a uni-processor is typically achieved through pre-emptive time slicing.

Figure 10.1: Task class hierarchy



10.2.1 Tasks and Resource Pools

The `Task` abstraction described above cannot be used with the pool based resource reservation mechanism (see section 8.3). A task executes the `Run`() method of its associated runnable and then exits, i.e. it discards the underlying POSIX thread. Hence it would be difficult to return the deleted task to a resource pool!

To circumvent this restriction, another threading abstraction is provided called `PooledTask`. Like the `Task` class, this supports the `Threading` interface but with different semantics which allow it to be placed in a resource pool. A `PooledTask` methods are defined to operate on the associated `Runnable` rather than on the underlying task. For example, a call on `Join`() will wait for the currently executing runnable to complete (i.e. when the `Run()` method returns), as opposed to waiting for the completion of the processing task.

Having executed a runnable, a `PooledTask` waits on a condition variable for a new runnable to be installed. At this point, the `PooledTask` may be returned to a resource pool.

10.3 Lightweight Threads

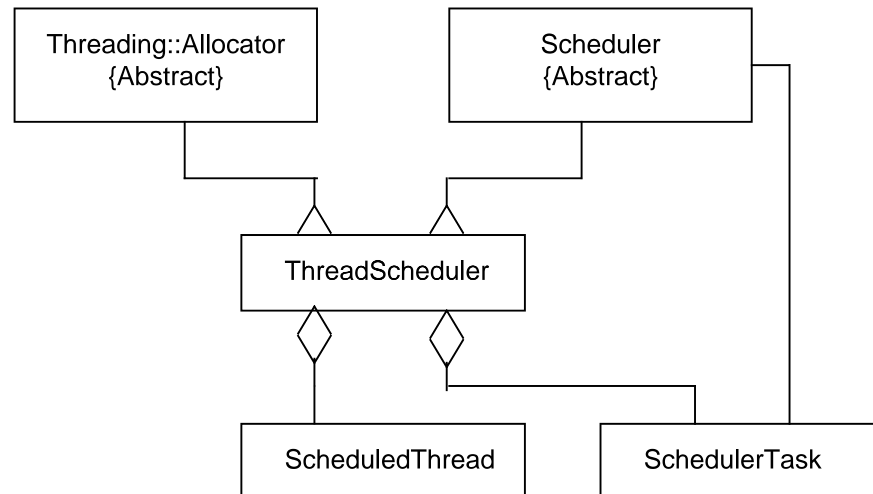
The lightweight thread abstraction `ScheduledThread` brings fine grain control to scheduling over POSIX threads. A `ScheduledThread` may be regarded as a unit of potential execution that, once assigned to a real unit of concurrency such as a `Task`, will remain with it until the `ScheduledThread` completes. Therefore, the control of scheduling is limited to the allocation of threads to tasks.

Lightweight threads are typically used to control invocations on a server. Each incoming invocation is allocated a lightweight thread and this in turn is assigned a processing task according to some scheduling policy. A considerable degree of control is possible by varying the scheduling policy and the number of tasks allocated to the scheduler.

The main components and their relationships are shown in figure 10.2.

The central component is the `ThreadScheduler` which acts as a resource pool or allocator for `ScheduledThreads`. This implements the `Threading::Allocator` interface which is used by clients to obtain

Figure 10.2: Lightweight threading components



ScheduledThreads. A ScheduledThread implements the Threading interface which provides the client with transparency from the actual implementation, i.e. the ScheduledThread could just as easily be a Task. A ScheduledThread may be associated with scheduling parameters such as a priority which the ThreadScheduler uses to add newly allocated ScheduledThreads to its queue.

A ThreadScheduler is also associated with a number of SchedulerTasks which provide the real processing resource. A SchedulerTask calls the Scheduler interface of the ThreadScheduler to obtain the next ScheduledThread to run from the queue.

10.4 Null Threads

The notion of a null thread; one that provides no real concurrency, may seem a little curious. However, it has several uses:

- In the context of resource control, a null thread may be configured in place of a Task to deliberately avoid a thread context switch
- A null thread implementation may be used to support an instance of DIMMA on a platform which has no thread support

A null thread is represented by the class NullThread which implements the Threading interface. However a call on the Start() method of a NullThread will cause the calling thread to immediately run the associated Runnable.

10.5 Single-threaded DIMMA

DIMMA may be compiled to run on a platform without thread support. In this case the only implementation of the Threading interface available is NullThread. In addition, in order to preserve maximum transparency, a Task is mapped onto NullThread.

A single-threaded instance of DIMMA has its limitations. In particular a capsule waiting for a reply to an invocation, cannot process subsequent requests until the invocation completes. This can give rise to deadlock situations where two capsules call each other, i.e. capsule A invokes an operation on B which in turn tries to invoke another operation on A. The reason for this limitation is due to the way in which the capsule uses its one and only thread.

The single thread is used to process all protocol endpoints. In a server, this thread runs an instance of the `SharedListener` which listens for network activity on all the capsule's endpoints. Each incoming invocation will be immediately executed in the context of this thread. However, it is not possible to do this in a client as the single thread will be used by the client code to make invocations and so cannot be used to run the `SharedListener`[1]. Client invocations are instead processed synchronously, i.e. the calling thread also waits for the reply.

1. Actually it would be possible to write a specialised scheduler to perform centralised control of all a capsule's events, but this could hardly be considered transparent in the sense that a considerable amount of the nucleus code would be affected.

11 Locking

In a multi-threaded environment, it is necessary to have some means of synchronising different activities that manipulate shared data. This is required both by applications and in the DPE itself. In DIMMA, these facilities are built from POSIX mutexes and condition variables. As with threading, this implementation is hidden behind various DIMMA classes to avoid building in unnecessary operating system dependencies.

DIMMA provides two locking models. The first is a thin layer on top of POSIX mutexes and condition variables, whilst the second provides a higher level abstraction similar to Java's synchronisation statements.

11.1 Locking model

11.1.1 Class Mutex

Class `Mutex` provides mutual exclusion, i.e a single level lock which may be held by only one thread at any given time. Only two methods are provided: `Lock`() and `Unlock`(). A mutex should be regarded as a mechanism to project relatively small sections of code. If there is a need to wait for longer periods of time, the `Condition` class should be used.

11.1.2 Class Condition

Class `Condition` provides access to POSIX condition variables which allow concurrently running threads to await an event or condition. The `Condition` class provides the following methods:

- wait() which waits for an event
- signal() which signals an event to single waiting thread
- broadcast() which signals an event to all waiting threads

11.1.3 Class RecursiveLock

Mutexes are not recursive, i.e. if a thread already holding a mutex attempts to acquire it again, it will block indefinitely. In some cases this is inconvenient and the `RecursiveLock` class is provided to allow a mutex to be acquired any number of times. Note that the mutex must be released the same number of times before it becomes available to other threads.

11.2 Synchronisation model

This model of synchronisation is modelled after Java and offers a more natural way of performing thread synchronisation based on the notion of synchronisable classes and objects.

11.2.1 Class Synchronisable

Any class needing to be synchronised (specified as a parameter to the Synchronised macro) must inherit from `Synchronisable`. This endows the derived class with a `RecursiveLock` and condition variable which are manipulated by the following macros.

11.2.2 SynchronisedObject macro

The `SynchronisedObject` macro is used to declare a block of code synchronised on a specific object. Most often the object will be itself.

The `SynchronisedObject` macro defines a synchronisation object whose constructor acquires a mutex and whose destructor releases the mutex. By defining this object at the beginning of a block, the mutex is effectively held for the duration of the block, the destructor releasing the mutex when the object goes out of scope.

The normal mode of use from C++ is hence:

```
{
      SynchronisedObject(this);
      ...
      critical region
      ...
}
```

Note the use of braces to set the block scope appropriately.

11.2.3 SynchronisedClass macro

This is similar to the `SynchronisedObject` macro above except that if locks a class rather an object (instance of a class). This is useful when manipulating static variables within a class. It is implemented itself in terms of a static mutex within the `Synchronisable` class.

12 Trader

There was never an intention to produce a trader as part of DIMMA, but some sort of trading[1] capability was necessary in order for applications to obtain references to service interfaces. The trading service provided is extremely basic: when the server exports its interface reference, this reference is written in ASCII form to a file with the same name as the interface. When the client imports the reference, it looks for a file with the correct name, and reads the reference from it. This assumes that the file is visible to both client and server using something like NFS.

Initially support was also provided to interface to the ANSAware trader. This is no longer provided as part of DIMMA, since it made it ANSAware a requirement of DIMMA.

The code for the trader is found in $DIMMA/dpe/Trader, where the class `odp_ansa_Trader_Client` is defined. This class inherits from an `odp_ansa_Trader_Sig`, which is elsewhere typedef'ed to an `odp_ansa_Trader`. In addition to the methods for the trader client, this file also includes the `ansa_trader()` method in the `odp_Capsule_Manager_Sig` class. This method returns a reference to a local `odp_ansa_Trader`, and is invoked as:

```
ansa_Trader trader = capsule_manager->ansa_trader();
```

The trader class is referred to as `ansa_Trader`, which is typedef'ed to `odp_ansa_Trader`. This simplified changing between the simple DIMMA trader and the ANSAware trader: the invocations are similar, and by changing the typedef, the different implementation could be used.

The trader supports two functions: `export` and `import`. These take arguments as shown in Figure 12.1. The `export` method extracts the ODP reference and class name from the interface reference passed as an argument, converts the reference to a string, and writes it to a file with the name of the class name. For example the reference of an interface named `Echo` will be written to a file named `Echo`. The `import` method is passed the name, then opens the file, reads the reference, and creates a generic interface reference using this information, which is returned to the caller. The context argument may be used to specify a directory in which to put the generated file, if the client and server are not resident in the same directory.

1. More accurately a naming capability.

Figure 12.1: Trader methods

```
void export (odp_GenInvocationRef ir, const
odp_Char8 *context )
                   throw(unknownType
                         ) ;

odp_GenInvocationRef import (const odp_Char8 *
type,
              const odp_Char8 * context)
                   throw(unknownType,
                         unknownContext,
                         noMatchingOffers
                        ) ;
```

References

[APM.1392]

Otway D., *The ANSA Binding Model*; **APM.1392**, APM Ltd., Cambridge U.K., Jan 1995.

[APM.1980]

Hayton, R. J., *DIMMA Tracing*; **APM.1980**, APM Ltd., Cambridge U.K., April 1997.

[APM.1995]

Macmillan, I. A., *An Introduction to DIMMA*; **APM.1995**, APM Ltd., Cambridge U.K., May 1997.

[APM.2036]

Howarth, N. J., *Building DIMMA 2.0*; **APM.1983**, APM Ltd., Cambridge U.K., April 1997.

[APM.2037]

Howarth, N. J., *Writing a DIMMA Application*; **APM.2037**, APM Ltd., Cambridge U.K., June 1997.

[APM.2046]

DIMMA Team, *DIMMA Performance Analysis;* **APM.2046,** APM Ltd., Cambridge U.K., August 1997.

[ISO/IEC 95]

ISO/IEC 10746-3, *Reference Model of Open Distributed Processing*, Jan 1995.

[OMG 95]

The Object Management Group, *The Common Object Request Broker: Architecture and Specification*; **Revision 2**, OMG Headquarters, 492 Old Connecticut Path, Framingham, MA 01701, Jul 1995.

[POSIX]

POSIX, *IEEE POSIX Std 10003.4a*, Sept 1992.