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

ODP C++ API Design Overview

Dave Otway

Abstract

This document describes an Application Programming Interface for writing distributable programs in C++ which conform to the distribution semantics of the Open Distributed Processing standard's computational model.

It currently only describes the non real-time features. These are intended to be the lightweight foundation for a real-time API.

This document is not complete. it is intended as the baseline for work in the 1996-1998 workplan which will be discussed at the September TC.

APM.1555.01 **Approved** 30th September 1997

Technical Report

Distribution:
Supersedes:
Superseded by:

ODP C++ API Design Overview

ODP C++ API Design Overview

Dave Otway

APM.1555.01

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

Contents

1 Introduction

This document describes an Application Programming Interface (API) for writing distributable programs in C++ which conform to the distribution semantics of the Open Distributed Processing (ODP) standard's computational model.

1.1 Requirements

The ANSA Phase III Distributed Interactive Multi-Media Architecture (DIMMA) project, the RETINA project, the DCAN project and some of the ANSA sponsors all have similar requirements for a real-time Distributed Processing Environment (DPE[1]).

This real-time DPE must provide full interoperability. That is between real-time and non real-time programs and between real-time and non real-time DPEs.

Non real-time objects must also be portable between real-time and non real-time DPEs.

A real-time DPE must be able to provide performance guarantees to objects and synchronous sub-networks of objects in a generally asynchronous distributed system. As well as a real-time DPE, this requires an API that provides the required real-time constructs.

1.2 Background

ANSAware is not suitable as the basis for a real-time DPE because it is designed to optimise resource usage, provides no application control over resource allocation or scheduling and is not lightweight enough.

CORBA implementations are likely to suffer from the same problems because they have the same design objectives. Obtaining full access to the source code of a commercial ORB implementation has also proved problematic.

The ODP computational model includes all the semantics required and has now been adopted as an ISO standard.

The open systems distributed systems industry is moving firmly behind the CORBA specification, but this has none of the real-time functionality required and is not lightweight.

There is a definite trend away from using C towards C++.

1. In OMG speak this is known as an Object Request Broker (ORB).

1.3 Objectives

The major objective is to build a prototype modular lightweight real-time DPE framework (the DIMMA nucleus) and an API based on the ODP semantics and implemented in C++.

This will serve the specific project and sponsor real-time requirements previously described. But the wider requirements are to interoperate with non real-time DPEs and provide portability of non real-time objects.

Interoperability with non real-time DPEs will be provided by adding their protocol stacks into the DIMMA nucleus and generating DIMMA stubs from their IDL descriptions.

The portability of non real-time objects will be provided by designing the API to be portable to any DPE.

Two longer term goals are to feed our experience back into CORBA by demonstrating:

- what features have to be added in order to provide performance guarantees
- what features can be left out without losing essential functionality in order to make it lightweight

1.4 Relationship to CORBA

Although the ODP computational model provides exactly the right set of distribution semantics, the results of the ANSA program must be capable of being fed back into the market.

In order to simplify this process:

- the non real-time functionality will not arbitrarily diverge from CORBA without good reason, although subsetting by choosing only one of a set of alternatives is encouraged
- a CORBA personality module will be added to the API to provide forward compatibility with the CORBA C++ mapping, i.e. all non real-time programs written for this personality module will be portable to a standard CORBA implementation, but not vice versa

1.5 Status

This work is being done in four stages:

1. a minimal API for interworking
   [only basic type and interface arguments]
2. an equivalent non real-time API to ANSAware and CORBA
   [add constructed types and concurrency]
3. a real-time API
4. preprocessor support

The status of the software prototype is that a stage 1 API plus some constructed types has been implemented over ANSAware 4.1 and the DIMMA nucleus.

This document describes the stage 1 API.

2 The application programming model

2.1 Relevant engineering constructs

2.1.1 Capsules

Capsules are the engineering units of locality, failure and security. They provide hardware enforcement of an encapsulation boundary by protecting the code and data inside a capsule from being read or modified from outside the capsule.

On nodes with virtual memory support a capsule usually corresponds to an address space. On nodes without virtual memory support the node constitutes a single capsule.

2.1.2 Stubs

2.2 Major computational constructs

2.2.1 Objects

Objects are the computational units of locality and can not have any externally accessible data. All components of a object must always reside in the same capsule (as each other) and there is no protection between components of the same object.

Multiple objects can be co-located in the same capsule, but a single object may not be split between capsules. If multiple objects reside in the same capsule then their individual (intra-capsule as opposed to inter-capsule) encapsulation must rely on the integrity of the programmer and programming tools that constructed them.

The interactions between objects must only use the computational constructs defined in this model. But within an object, any programming construct supported by the local programming environment may be used. [This may limit the portability of an object but will not affect its interoperability.]

Objects represent a design choice about the minimum granularity of distribution.

Objects have factories to generate them and all objects generated by the same factory may be said to be of the same ”type”, but this never needs to be checked because objects are anonymous and may only be accessed via their interfaces.

2.2.2 Interfaces

Any object which wishes to provide a service to other objects must construct an interface via which its clients can invoke operations on it.

An interface consists of a set of operations and (optionally) some data. These operations and data constitute the implementation type[1] of an interface. An object can have multiple interfaces of the same type and multiple types of interfaces.

In order to preserve its object's encapsulation, an interface must not have any externally accessible data or be externally constructable.

Data which is to be shared between an object's interfaces can be declared as the object's data. Alternatively, an interface can access the data of another interface in the same object.

Interfaces to an object can be created by the factory that creates the object or by an invocation of an operation on an existing interface to the object.

2.2.3 Signatures

An interface signature specifies the interaction type[2] of an interface. It consists of a set of operation signatures.

1. Equivalent to a C++ concrete class definition.

2. Equivalent to a C++ abstract class declaration.

3 The computational API

An application programmer writes ODP conforming distributable C++ programs using only the computational API. This chapter describes the computational API and explains how to use it.

3.1 Programming environment

An application source program does not exist in isolation. Application programs using this computational API assume the following program environment.

3.1.1 Preprocessor macros

All preprocessor macros have names with an `ODP_` prefix.

3.1.1.1 ODP_NAMESPACE

Eventually all API definitions will be in the ODP namespace. But until namespaces are widely implemented by production C++ compilers, all the API global names will have an `odp_` prefix[1].

This prefix can be stripped off by defining the `ODP_NAMESPACE` macro before any header files are included.

```
#define ODP_NAMESPACE
```

This has an effect similar to that which the *using-directive*:

```
using namespace ODP ;
```

will have when the ODP namespace is implemented.

3.1.1.2 ODP_DEBUG

The definition

```
#define ODP_DEBUG  1
```

before any header files are included switches on extra run-time checking by defining the following macros (which can be defined individually):

```
#define ODP_CHECK_NIL   1
```

checks for nil pointer dereferencing;

```
#define ODP_CHECK_RANGE 1
```

checks for out of range accesses in sequences;

1. The API definitions cannot be nested inside an ODP class because some of the API constructs are implement by templates and template declarations must be global.

```
#define ODP_ROBUST 1
```

does extra checks on incoming messages.

3.1.2 Stub generator

Each remotely accessible interface will be described in an interface description file using an Interface Description Language (IDL). These IDL files will be processed by a stub generator to produce a signature header plus client and server stubs.

3.1.3 Header files

All application source files must include the header file `computational.hh`,

```
#include <computational.hh>
```

which contains all the definitions needed for the ODP computational API, but does not include headers for any “standard” libraries.

3.1.3.1 Signature headers

A application source file must also include the signature header files for all interfaces used or provided by objects in the source file. A header file may include signature and invocation reference definitions for more than one interface.

The naming convention used by the stub generator is that the type of the invocation reference for an interface is named with the IDL interface name prefixed by any enclosing IDL module names. The (abstract) signature class for the interface is named by giving the invocation reference type name a `_Sig` postfix.

A signature header may also include typedef and constant definitions.

3.1.4 Libraries

Applications programs must be linked with the ODP library and the library for the underlying DPE. Because the ODP API has to be mapped onto each DPE, there is a specific ODP library for each DPE. It is named by adding the `odp_` prefix to the DPE name.

3.2 Language mapping strategy

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.

3.3 Signatures

Signatures are normally generated from an IDL interface definition by a stub generator, so a programmer does not need be too concerned with the details.

A signature is represented by a C++ abstract class definition which publicly inherits the base `odp_Signature` class. A signature defines the operations and terminations of an interface.

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 ;
} ;
```

3.4 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. The stub generator inserts a typedef for the invocation reference 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.

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.

3.5 Objects

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

The object class must declare all its interface classes as friends. It must also declare its factory function as a friend. 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.

```
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 () ;
} ;
```

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.

3.6 Interfaces

An 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 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.

3.7 Module to object mapping

IDL modules just provide naming scopes. An object and its interfaces can be structured to match the module scoping so that the interface Manager for the object Bank implements the signature BankManager_Sig and can be assigned to invocation references of type BankManager, but this structuring is not mandatory. The interfaces defined in a single IDL module may be implemented as interfaces to multiple objects and a single object may provide interfaces defined in multiple modules.

3.8 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.

3.9 Terminations

The anonymous termination is handled differently from named terminations.

3.9.1 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.

3.9.2 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.

3.10 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.

3.11 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.

3.12 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() ;
}
```

3.13 Object factories

3.14 Any references

Invocation references only. No concrete types. In initial version types described by name.

3.15 Primitive types

Same as CORBA.

3.16 Constructed types

Same as CORBA but only accessible via smart pointers.

3.17 Memory management

Local garbage collection automatically handled by the invocation references and smart pointers.

3.18 Trading

Export and import using Any types.