DIMMA – A Multi-Media ORB

Douglas Donaldson, Matthew Faupel, Richard Hayton, Andrew Herbert,
Nicola Howarth, Andre Kramer, Ian MacMillan, Dave Otway, Simon Waterhouse
<*firstname.lastname*@ansa.co.uk>

APM Limited, Poseidon House, Castle Park, Cambridge, UK. CB3 0RD
Tel: +44 (0) 1223 515010 Fax: +44 (0) 1223 359779

Abstract

DIMMA – A Distributed Interactive Multi-Media Architecture – is an open distributed processing (ODP) platform that facilitates the production of distributed applications. It has particular support for those applications that have soft real-time constraints, and those that make use of multi-media.

DIMMA consists of a portable layer of distribution engineering (middleware), together with tools to interface applications to this engineering. The most popular commercial ODP platform is the Common Object Request Broker Architecture (CORBA) whose definition is managed by the Object Management Group (OMG). In recognition of this popularity, DIMMA supports a CORBA compliant API so that CORBA applications may be easily ported to or from the DIMMA platform.

This paper describes the example implementation of DIMMA. It gives an overview of the motivations behind the design of DIMMA and then highlights some key features of the DIMMA implementation. It concludes with a brief analysis of DIMMA’s performance.

1. Introduction

1.1. The goal of the DIMMA project

Object Request Brokers (ORBs) have been developed to provide support for distributed object-oriented applications, hiding much of the complex engineering needed to implement distribution. These can now be regarded as stable technology; the most wide-spread ORB architecture, OMG’s CORBA [[6]], was first introduced in 1987, and many commercial and public-domain implementations of their CORBA ORB specification are now available.

However, combining this provision of distributed object-orientation with support for multi-media is not straight-forward, as it imposes a number of requirements on the ORB:

- Support for specifying flow interfaces

  Current ORBs only support RPC type interfaces, i.e., call an object and (possibly) receive a return value. Multi-media applications often require handling of continuous flows of data being transmitted and received.

- Control over resources used

  Many types of multi-media service are particularly sensitive to quality of service. For instance, a video stream must deliver its frames at a regular and timely rate in order to avoid the picture being jerky or corrupted. This requires the system to guarantee that the resources are available to deliver this quality of service, which in turn requires the system to provide the programmer with control over those resources, possibly including those provided by an underlying real-time OS.

- Support for new protocols to be easily added

  CORBA ORBs can work correctly while only supporting one RPC protocol (IIOP). Due to the diverse nature of multi-media though, there are a much larger number of protocols that a multi-media ORB may have to support, and more are being developed all the time. This requires a multi-media ORB to provide an easy method for adding new protocols.

- Minimum necessary footprint

  Multi-media services make heavy use of system resources (buffers, CPU time, network bandwidth). For this reason, the ORB itself should have the minimum necessary use of such resources so as to avoid impacting the quality of the services that it is supporting. Furthermore it should be scaleable to allow its use in a range of environments from small devices to large switches.

The goal of the DIMMA project was to produce an architecture for a multi-media ORB that met these requirements, and a practical implementation of that architecture, making as few assumptions as possible about the host OS. It aimed to create a small, efficient, modular ORB, with support for flows and resource control, and the flexibility to add new protocols and new styles of application interface.

1.2. Project history

The DIMMA project began in April 1996 and made four formal releases of an implementation of DIMMA over the course of a year and a half. The first year concentrated on developing the architecture and then testing the ideas with an initial implementation. The first release of DIMMA 1.0 in November 1996, was built in a modular fashion and supported both RPC and flow protocols, made available to the programmer through either a CORBA or an ODP-based interface. ISO RM-ODP [[2]] significantly influenced the design, which in turn was influenced by earlier work at APM [[3]].

Experience gained from this first release led to a restructuring of the code base to improve its generic and modular features and to add more debugging support for the application programmer. This reworked version was released in April 1997 as DIMMA 1.1.

Having established the generic framework for resource control in DIMMA 1.1, this framework was populated to ensure that all aspects of resource usage by all supported protocols could be controlled by the application programmer through the medium of QoS parameters passed in through explicit binding. Minor changes also allowed new protocols to be dynamically loaded at runtime. This version 2.0 of DIMMA fulfilled all the original goals of the project and was released in May 1997.

Finally, some time was devoted to analysing the performance of DIMMA, and significant improvements in its speed and robustness were achieved. A resultant optimised version 2.01 of DIMMA was released in September 1997.

1.3. How DIMMA differs from CORBA

CORBA was designed to offer applications maximum transparency to the concerns of distribution and to offer portability across a wide variety of operating systems. As yet though, CORBA does not address the needs of an increasingly large class of applications that must deliver their results within a particular time scale (soft real-time), nor does it address the need for communication of continuous flows of data such as audio or video.

DIMMA was designed to explore how the needs of such applications may be met in the context of an ODP platform. To this end it provides applications with control over their allocation and use of resources through quality of service (QoS) parameters, and supports multi-media flows through flow interfaces. These facilities may be regarded as extensions to CORBA.

In contrast to some rather monolithic CORBA implementations, a major feature of the DIMMA design is its open flexible component architecture. DIMMA is highly configurable: it supports multiple protocols – new protocol implementations may be dynamically incorporated into an application at run-time – and QoS parameters allow an application programmer to exercise a high level of control over the configuration of the components that are used to provide an engineering channel.

1.4. The application of DIMMA

DIMMA is intended primarily as an experimental vehicle. The focus is on using DIMMA to identify the needs of multi-media and real-time distributed applications in terms of proposed core ORB facilities and to prototype the resultant ideas.

Although DIMMA will run "out of the box", it is anticipated that it will be of greatest value to those who wish to customise the core ORB. The internal structure is very flexible and the components are built according to a set of well defined frameworks. For example, new protocols may be easily added using the DIMMA protocol framework which facilitates the reuse of layered modules.

The explicit binding model, together with the flexible resource reservation mechanisms allow a considerable range of applications to be built: applications that are not possible on today's standard commercial ORBs. This should be of interest to those involved in telecommunications applications such as video on demand.

1.5. Paper structure

The remainder of this paper highlights particular features of the design of DIMMA and how it has been extended to support the achievement of the goals outlined above. It concludes with a summary of the performance of the final version of DIMMA, demonstrating the results possible when explicit control of quality of service is given to the application programmer.

2. DIMMA Structure

2.1. Overview

DIMMA is constructed as a set of small components that may be combined in many different ways to suit the diverse needs of applications. This also acknowledges the fact that the real world needs ORBs that are high performance, down-sizeable and scaleable. In this sense, DIMMA may be regarded as a microkernel ORB.

The components of DIMMA are considered to be subdivided into groups and arranged in layers; this is also reflected in the source structure. This layering is depicted in Figure 2-a.

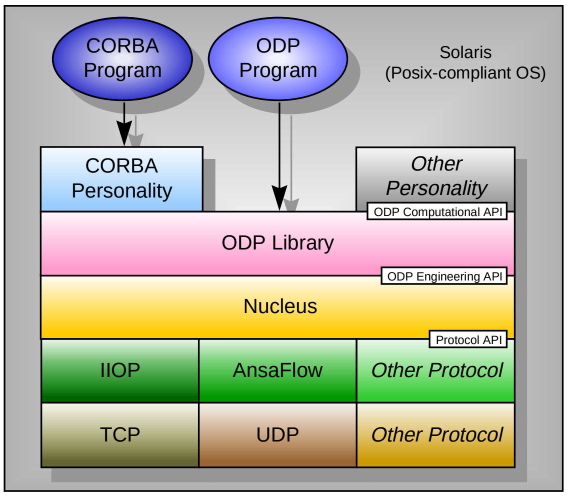

Figure 2-a. Structure of DIMMA

DIMMA provides two interfaces for the application programmer: a proprietary one based on the ODP-RM concepts [[2]] – the ODP Computational API – and a CORBA extended subset called Jet. Jet is implemented as a personality built on top of the ODP facilities and hence shares a number of features with the former. Further personalities could be added if required.

Both APIs are mapped onto a common ODP Engineering API by the ODP Library in order to facilitate hosting on different ORBs (or Nuclei in ODP terminology). The DIMMA nucleus supports the ODP Engineering API directly, whilst adapters can be provided for other ORBs. To date only one experimental adapter has been produced, which was to allow applications written using Jet or ODP to communicate over APM's ANSAware product.

The DIMMA nucleus provides the distribution engineering apparatus such as binders and communications protocols. Currently two protocols are provided: the CORBA Internet Interoperability Protocol 1.0 (IIOP) for interoperability with other vendors' ORBs and a proprietary protocol (ANSA Flow), optimised for transporting multi-media flows. New protocols can be added within a well-defined framework; furthermore existing protocol components (e.g., the TCP part of the IIOP stack) can be reused by these new protocols as required.

2.2. CORBA personality

Jet, the CORBA personality, supports most of the CORBA C++ language mapping and has an associated IDL compiler, though it does not support any CORBA-style repositories. It is intended for application writers who are already familiar with CORBA and who may wish to port their applications between DIMMA and another vendors ORB.

2.3. ODP Library

The ODP Computational API is based on the concepts of the ODP Reference Model [[2]]. It provides the minimum facilities necessary for writing distributed applications but also affords considerable flexibility. For example, unlike CORBA, an ODP application object can export multiple interfaces.

It is intended primarily for hosting personalities such as Jet rather than for writing applications. For this reason there is no IDL compiler provided and hence an ODP application writer must provide their own stubs.

The ODP-RM concepts supported by the ODP computational API are: objects; operations; interfaces; terminations, and invocations. In addition, the concepts of *signature* and *invocation reference* are introduced, where a signature defines the operations and terminations of an interface, and an invocation reference identifies an interface in the computational viewpoint.

2.3.1. 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 declares its factory and interface classes as friends, to allow them to access the private object data. The object has no public methods or data, as construction is via the factory and all further access via the interfaces.

2.3.2. Operations, interfaces and signatures

An ODP interface is characterised by a signature and some behaviour. A signature corresponds to the interface type, which in turn defines a set of operations, and the ODP interface itself provides a particular implementation of those operations.

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:

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

The ODP interface is then represented by a C++ object defined by a class which publicly inherits from both the interface's signature and an odp_Interface class template, which provides implementation that is shared amongst all interfaces. 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:
     Pence balance(BankPin mpin) ;
};
```

The interface inherits from odp_Interface a pointer to the object on which it is an interface. 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.

2.3.3. 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 all of a named termination's results are passed as arguments of its exception. In the signature example above, invalidPin is a named termination.

The anonymous termination is handled differently from named terminations. 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.

2.3.4. Invocations and references

An operation in a (local or remote) interface is always invoked via an invocation reference. These are implemented as typed smart pointers to an (abstract) signature. This pointer is generated from the odp_InvocationRef template class like so:

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

The 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 in invocations are mapped onto their C++ 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. Constructed types are passed by value, interfaces are passed by reference.

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

2.4. Target platform

To exploit the full range of QoS controls offered by DIMMA, the DIMMA nucleus must be hosted on an operating system able to provide the necessary soft real-time facilities. DIMMA makes use of Posix interfaces [[1]] to access OS resources; in particular the Posix threads interface is relied upon to support control of multi-tasking. The current version of DIMMA has been tested and released on Solaris 2.5.

3. Communications Framework

The purpose of the communications framework is to simplify the production of new protocols by encouraging and facilitating reuse, through providing general components and standard interfaces, and through providing a uniform approach to resourcing for quality of service.

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 message multiplexing
- dispatches messages to stubs
- interfaces with OS 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”. In other words, modules are the static representation of available protocol layers, while channels are the objects created dynamically when a binding is set up using that protocol, which manage the resources associated with the binding.

Client call and server reply messages are presented to the top channel and go down the channel stack until they reach the lowest level channel (called an anchor channel) 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 or server object.

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. Session objects are assigned for each concurrent flow or invocation on a stub. They act as state machines, preserving any information required by each caller and linking the calling threads to the messages that are passing through the channel to carry out their operations.

4. Flow Support

4.1. Overview

There is a class of applications for which the operational RPC mechanism is inappropriate. These applications deal naturally in continuous flows of information rather than discrete request/reply exchanges. Examples include the flow of audio or video information in a multimedia application, or the continuous flow of periodic sensor readings in a process control application.

A flow has a distinct type and an associated direction with respect to the binding, e.g., video information might flow out of a producer binding associated with a camera and into a consumer binding associated with a TV monitor on which the output of the camera is to be displayed. It follows that flow interface types exist in pairs that are related by the reversal of the flow.

The type of flow is characterised by the set of possible frames that it can support. For example, a video flow might be able to carry both MPEG and JPEG frames.

DIMMA extends the functionality of a basic ORB in three ways in order to support flows:

1. The ODP library provides a means to represent and manipulate flows
2. The CORBA IDL compiler understands a new type of flow interface, mapping them on to the appropriate ODP library facilities
3. An example flow protocol is provided to demonstrate the use of flows.

Additionally, the use of flow interfaces generally requires the ability to specify QoS parameters; the support for this is dealt with in more detail in subsequent sections.

4.2. Flow Interfaces in IDL

The IDL extension to support flows proved very simple from the user’s point of view: all that is changed is that the keyword “flow” is used instead of “interface” in CORBA IDL. It actually required more changes behind the scenes though to support this, such as checks in the IDL compiler to ensure that the operations within the flow were legal (e.g., no inout parameters).

A flow interface is modelled in Jet IDL in terms of one-way operations and will result in the generation of both producer and consumer components, in an analogous way to client and server components generated from an operational interface. The main difference is that the operations are one-way and data is unidirectional with respect to the binding. The operations within a flow interface correspond to frame types and these are further described by the parameters of the operation.

For example, a simple video flow consisting of a single frame type (Frame1) consisting of a frame number and the video image data (image), could be described in IDL as follows:

```
flow Video
{
    void Frame1( in long frame_no,
        in string image );
};
```

Note that both parameters are described as input (in) and that the operation returns no result (void type).

The ODP standard defines an additional concept called stream which is described as a set of unidirectional flows, e.g., a TV stream might be considered a logical entity consisting of an audio and video flow. DIMMA does not implement the stream concept directly, although in principal, a stream binding could be constructed by an application from a set of flow bindings.

Although flows appear to be similar to operational interfaces in IDL, they are distinct entities and a flow cannot inherit from an interface, nor can an interface inherit from a flow.

4.3. Example protocol

The protocol that was added to demonstrate flows within DIMMA is called ANSA Flow. It is a lightweight but fully functional multicast protocol based on RTP [[4]] over UDP. Note though that it is not a full implementation of all possible RTP packet types; just the parts of it needed to support ANSA Flow’s proprietary packets.

5. Resource Management

5.1. Overview

DIMMA provides for application control and management of resources through the use of QoS parameters. All resources needed by an application affect QoS, but many are managed by the OS and/or the network, and the application has only limited influence over them. This constrains what can be achieved in terms of guaranteeing QoS.

In practice, with current OSs, an ORB may control sharing within a capsule (process), e.g., the number of tasks, or how channels are multiplexed. It may exert some influence over inter-capsule behaviour, e.g., via task priorities. To do any better than that requires a specialist resource-aware OS and a resource-aware network protocol.

DIMMA QoS parameters are specified in terms of attributes such as buffers, threads, communication endpoints and so on. Mapping between application specific QoS parameters (e.g., jitter, frame rate, etc.) and these “engineering” parameters remains the responsibility of the application.

The DIMMA QoS infrastructure allows the application explicit control over the resource allocation policy to be used within a given channel. For example, buffer allocation may be defined to be “on demand”, or a set of buffers may be preallocated for the channel’s exclusive use. A few examples are given to clarify the possibilities.

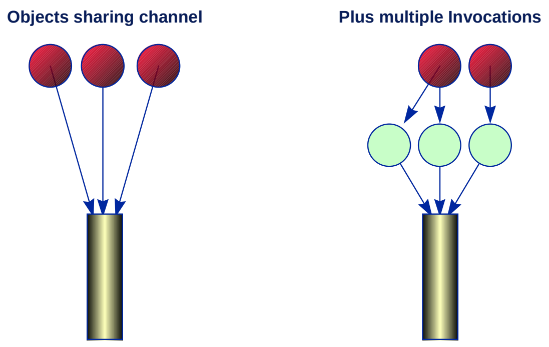

Figure 5-a: Possible channel sharing policies

Figure 5-a illustrates two possible policies for multiplexing objects to channels. The left-hand diagram shows multiple objects sharing a single channel. The right-hand diagram shows the same, with the addition of session objects maintaining state information to allow multiple invocations. The channels themselves may be subdivided into multiple layers, each with further multiplexing.

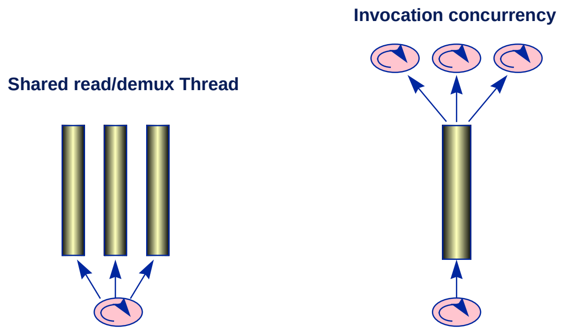

Figure 5-b: Possible thread sharing policies

Channels may share a read/demultiplex thread. This is shown on the left-hand side of Figure 5-b. The channel itself may be shared between multiple application-level threads as shown on the right-hand side. One extreme (not shown) is one thread per capsule, i.e., the single threaded case.

Figure 5-c gives an example of two possible resourcing policies for a complete end-to-end channel.

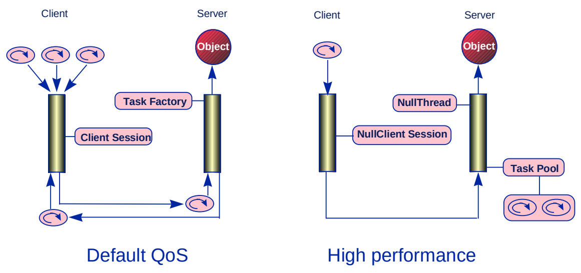

Figure 5-c: Two channel resourcing strategies

The left-hand diagram gives a typical policy for "default" quality of service. The calling threads share a common channel, and at the server side, there is only a single listener thread which, on receipt of an incoming call, causes a new thread to be created to handle the call and then passes control to that thread (which is destroyed on call completion). All resources are allocated on demand through factories and deleted when finished with.

The right-hand diagram shows a policy for high-performance quality of service. The client has a dedicated channel so there is no need to maintain session information, and there is no context switch on calling; the thread waits for the response. At the server side, when a call is received, the receiving thread calls the server object and returns the result, so again there is no context switch. While the call is being executed, a new thread is drawn from a pool ready for the next receive. When the call has been completed, that thread is returned to the pool. All resources used by the configuration are drawn from pre-allocated pools and returned when finished with.

5.2. Generic resource framework

The aim of the components of the DIMMA resource framework is to provide a generic way of controlling all resources while allowing different specific resource control policies to be implemented. To achieve this, all resources are allocated through a generic Allocator interface:

```
template <class AResource>
class ResourceAllocator
{
public:
    virtual AResource * Allocate() = 0;
    virtual ~ResourceAllocator() {};
};
```

Behind this interface, different policies can be implemented, e.g., "create a new resource each time it is needed" (factories), or "take a pre-allocated resource from a pool" (pools). The resources themselves implement a generic Resource interface to allow them to be freed once finished with:

```
class Resource : public Queueable
{
public:
    Resource() {}
    virtual ~Resource() {}
     virtual void Free() {}
};
```

Buffers, sessions and threads can all be controlled via this framework, and different policies can be put together for different parts of a protocol to form a wide range of overall resourcing policies. Buffer and session allocation is straight-forward; threading requires a little more explanation.

5.3. Threading

DIMMA offers both single-threaded or multi-threaded operation, switchable at compile time through a configuration flag. The latter is implemented internally using Posix threads; the former is made available to produce a high-performance executable, or for platforms that don't support multi-threading. Since CORBA does not define any standards for multi-threaded operation, DIMMA provides its own interfaces to control threading.

The DIMMA model of threading provides two abstractions: Threads and Tasks. Threads are a unit of potential concurrency and are scheduled over tasks by DIMMA. Tasks represent a unit of real concurrency and are implemented as Posix threads [[1]] (bound to a lightweight process on Solaris). Tasks are scheduled pre-emptively by the underlying operating system and it is the applications responsibility to ensure that access to data shared between tasks is properly synchronised.

Three threading policies are provided with DIMMA; these control what is done when there is a possibility of a context-switch within the protocol. The "null task" policy means that the calling task carries on executing, thus avoiding a context switch. The "normal task" policy means that control is handed over to another task and the calling task returns. The "scheduled thread" policy allows a specified number of prioritised threads to be scheduled over a number of tasks. The threads are added to a prioritised queue and the first on the queue is executed as and when a task becomes available.

5.3.1. Client threading

Client objects directly control their concurrency by creating tasks appropriately. The interface to DIMMA tasking borrows heavily from Java [[5]]. A DIMMA task executes an instance of a Runnable interface, the latter being an abstract class from which a concrete class should inherit. The Runnable interface defines a single method called Run which will be the entrypoint for the task. A simple example of its use is as follows:

```
#include <Runnable.hh>
class MyRunnable : public Runnable
{
  Runnable::status_t Run();
  ...
};

main()
{
  Runnable * runnable = new MyRunnable;
  Task * myTask = new Task;
  myTask->Install( runnable );
  myTask->Start();
  (void) myTask->Join();
  delete myTask;
}
```

5.3.2. Server threading

A server object shares the same engineering for threading as that provided for client objects but the use is typically different. The reason for the difference is that server interfaces are normally upcalled by a nucleus task, rather than being executed by a task provided by the server object. It is possible to specify whether the operation should execute in the nucleus task, a new server task or whether a new thread should be queued for subsequent execution by an associated task.

5.4. Specifying resource requirements

In order to provide any kind of bounded QoS, it must be possible for an application to communicate its requirement to the underlying infrastructure, both distribution layer and host operating system.

DIMMA supports a resource reservation model and allocates resources according to the specified QoS when an application establishes a binding, e.g., when a client binds to a server. Resources are reserved for all parts of the underlying channel, e.g., communications resources, buffers, tasks, etc. The binding model used by DIMMA to support the specification of QoS is dealt with in the next section.

6. Binding

6.1. Overview

With standard RPC systems, binding is done *implicitly*, i.e., when a remote reference is first used, a binder sets up an appropriate communications channel to it. This is the model adopted by the majority of ORBs and provides maximum transparency to the application in terms of hiding irrelevant engineering details. However, there is a trade-off: transparency implies little or no control over the QoS of the binding that is established.

When quality of service can be specified, binding must be done *explicitly* (because the binder will not know which of a range of possible quality levels is wanted); this requires extra support from the binding apparatus. Although explicit specification of Quality of Service parameters can be provided for RPC protocols as well, it is essential for flow protocols. QoS is specified on a per-connection basis so as to allow the same service to be accessed with different levels of quality, e.g., a video source could be provided at different levels of quality over different bandwidth connections.

DIMMA supports both implicit and explicit binding models.

6.2. Implicit binding

Implicit binding in DIMMA is provided by Binder objects which implement a predefined binding policy, and make use of default QoS parameters. In keeping with the DIMMA component philosophy, the implicit binders may be replaced by application specific implicit binders which implement a different binding policy appropriate to the application.

6.3. Explicit binding

To meet the needs of multi-media and real-time applications, DIMMA provides a model of explicit binding. Explicit binding allows both the QoS and the time of binding to be controlled and hence allows resource reservation. The downside is the increased complexity of the mechanism required to establish the binding.

An explicit binding is accomplished in several stages and is bootstrapped using the implicit binding mechanism. An application wishing to offer a service must do so via a binding manager which offers its own interface as a proxy for the real service. The binding manager is responsible for placing each party's local explicit binder in communication with the other. These in turn will set up the local bindings with the specified QoS and establish the network connection. This procedure is illustrated in Figure 6-a.

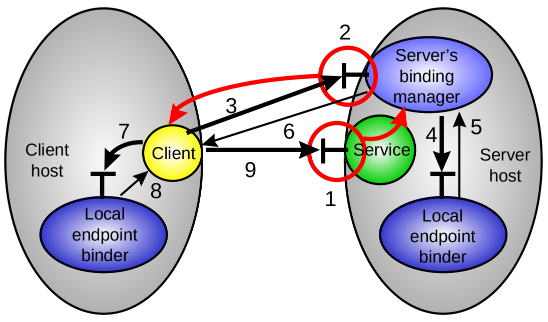

Figure 6-a: Explicit binding

1. The service interface is exported via the server's binding manager.
2. When the service interface reference is exported (e.g., as a call parameter or via a trader), a reference to the server's binding manager interface is passed instead.
3. The client, on binding to the service, calls the binding manager passing it the parameters for the binding (e.g., QoS requirements).
4. The binding manager hands off the binding request to the server's local endpoint binder, which creates a service endpoint and…
5. …returns the details to the binding manager, which in turn…
6. …returns the details to the client.
7. The client then passes the remote endpoint details plus other binding parameters to its local endpoint binder, which sets up the binding and…
8. …returns an invocation reference to the client…
9. …which the client then uses to call the service.

All these machinations though are transparent to the client application, which simply imports what it believes to be the interface reference for the service and binds to it.

6.4. Binding and resource control

The protocol-specific QoS specification provided during explicit binding is passed to the top level protocol binder at each end of the binding. In the usual case where the protocol is made up of a number of layers, the QoS specification for the top level layer will contain a pointer to the QoS for the next layer down, and so on. When creating the channel for the binding, each layer extracts the QoS for the next layer down and then passes that to a create channel request on the lower layer recursively until the bottom layer is reached. The bottom layer extracts the resourcing information from the QoS parameters it is given to create its channel and then returns that to the next layer up, which repeats the process until the top layer is reached again. Thus each layer resources its part of the channel appropriately.

To give a concrete example, QoS for the IIOP protocol is specified using objects of the following classes (access and constructor methods have been removed to save space):

```
class TCPQoS
{
    ...
protected:
    E_ReadingTaskPolicy f_policy;
    int f_receiveSize;
    int f_transmitSize;
    Buffer::Allocator* f_bufferAllocator;
};

class AcceptorTCPQoS : public TCPQoS
{
    ...
private:
    Threading::Allocator*
        f_taskAllocator;
    IOChannelTCP::Allocator*
        f_channelAllocator;
};

class IIOPQoS : public GenericQoS
{
    ...
private:
    TCPQoS* f_TCPQoS;
    Buffer::Allocator*
        f_writeBufferAllocator;
};

class ClientIIOPQoS : public IIOPQoS
{
    ...
private:
    ClientSession::Allocator*
        f_clientSessionAllocator;
};

class ServerIIOPQoS : public IIOPQoS
{
    ...
private:
    Threading::Allocator*
        f_TCPReadingTaskAllocator;
    IOChannelTCP::Allocator*
        f_IOChannelTCPAllocator;
    Threading::Allocator*
        f_serverSessionAllocator;
};
```

Note first that the IIOP QoS specification contains a pointer to a TCP QoS specification, and secondly that there is a variable for each possible policy option within the protocol, e.g., whether to allocate buffers from a pool or a factory, what the tasking policy should be and so on.

When the IIOP binder is requested to create a channel, it extracts the TCP QoS from the IIOP QoS it has been given and then passes it to the TCP module's create channel method. That being the lowest layer, it creates the TCP channel and passes it back to the IIOP layer, which then uses it and the IIOP-specific QoS to create an IIOP channel.

6.5. Implications for network endpoints

Supporting Quality of Service affects how network endpoints are used. For a standard RPC system with implicit binding, all connections to a service can be multiplexed through the same network endpoint because they all have the same (default) quality of service. See Figure 6-b.

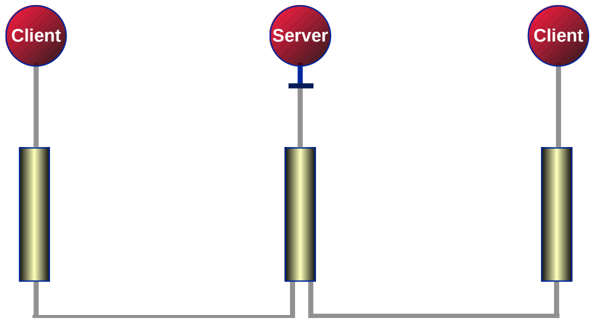

Figure 6-b: Standard ORB sharing a single network endpoint

With an ORB such as DIMMA, which supports access to the same service with different levels of quality of service, a server must be able to offer multiple network endpoints in order to support the different QoS (Figure 6-c).

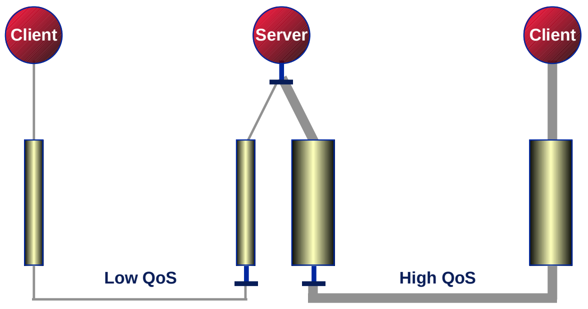

Figure 6-c: Multiple endpoints for varying QoS

Although supporting QoS requires support for multiple network endpoints for the same service, this extra level of complexity is masked within the binding mechanism outlined above, and all the application ever sees is a standard invocation reference that encapsulates the details necessary to distinguish between multiple possible endpoints.

6.6. Implications for binders

Supporting explicit specification of QoS has a number of implications – there is a trade-off between the amount of control allowed to the application over QoS and the transparency to the application of the mechanisms being used.

As resource usage and thus QoS is specific to each protocol, a binder can no longer be generic. This requires the application to understand the QoS details of each protocol it uses, i.e., it too can no longer be generic. It also affects the dynamic loading of protocols – if QoS is protocol specific, how do we determine and select from the possibilities offered by a dynamically loaded protocol?

This problem of lack of transparency can be mitigated by defining a more generic sort of QoS and then having each protocol be able to map from the generic QoS to its own specific QoS.

DIMMA has support for this concept of generic QoS, which is called engineering QoS. It is at a higher level than protocol specific QoS, though it does not attempt to provide very high level "application" QoS. It is the responsibility of the implementer of a protocol to map this engineering QoS into terms that are applicable to the protocol. The two protocols supplied with DIMMA, IIOP and ANSA Flow, both implement this mapping.

The options currently configurable with engineering QoS are:

- Processing concurrency, which defines the maximum number of tasks to run concurrently. A special "null task" value indicates that the message reading task is to process the operation.
- Message concurrency, which defines the maximum number of messages that can be simultaneously processed (and hence the size of the buffer pool). If this is greater than processing concurrency, message processing is scheduled over the available tasks as described earlier in section 5.3.
- Buffer size, which defines the size of the buffers in the pool and hence the maximum size of a message.
- Channel policy, which defines whether or not channels are multiplexed on transport connections.

By having this engineering QoS it allows application programmers to write applications that control QoS without having to hard-code decisions about protocol use throughout the program. It also means that the choice of which protocol to use can be deferred until runtime, allowing dynamically loaded protocols (that were not necessarily known about at design time) to be used.

7. Performance

7.1. Improvements

For DIMMA 2.01 we undertook a performance analysis and then tuned the system based on the results. These are the major performance improvements that we implemented compared to earlier systems:

- Marshalling

  The standard iostream libraries that were used for marshalling in DIMMA 2.0 carry out costly mutex locking. We wrote our own replacement routines which eliminated this.

  We used C++ templates for decisions on marshalling, e.g. byte-ordering and float representation, which removed some frequently executed runtime tests.

- IIOP

  We removed redundant marshalling and unmarshalling, and provided a QoS configuration that doesn’t change thread in the client session thus avoiding a costly context switch.

- TCP

  Reading a TCP message requires reading a header containing a length and then reading the remainder of the message. This was previously done using two `recv` calls, each of which executes costly locking code. This was replaced with a single `recv` call that just tries to read as much as possible in one go into a buffer, i.e., it usually reads both the header and the trailing message in one go.

7.2. Results

There were large performance gains in DIMMA 2.01. The default QoS configuration was 50% faster than DIMMA 2.0, and is comparable with commercial ORBs. The high-performance QoS configuration is 2-3 times faster than commercial ORBs. Descriptions of the default and high-performance QoS configurations are given earlier in Figure 5-c.

The tests were carried out using 5,000 iterations of a remote call using IIOP on a 167MHz Sparc Ultra 1 with 128Mb of memory. The first two tests used CORBA oneway operations; the second two used standard operations. The “no param” tests used an operation with no parameters, the “null string” tests used an operation with a single string parameter, which was passed a zero length string.

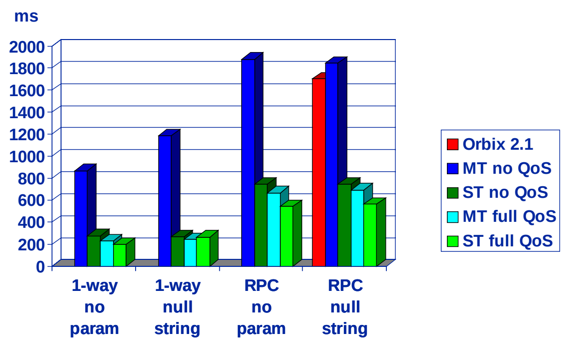

Figure 7-a: Performance results

- Orbix 2.1 - a popular commercial CORBA ORB from Iona
- MT no QoS - multi-threaded build, default QoS configuration
- ST no QoS - single-threaded build, default QoS configuration
- MT full QoS - multi-threaded build, high-performance QoS
- ST full QoS - single-threaded build, high-performance QoS

8. References

[1] POSIX, IEEE POSIX Standard 10003.4a; September 1992.

[2] ISO/IEC, *Reference Model of Open Distributed Computing*; **ISO/IEC 10746-3**, January 1995.

[3] Li, G., *An overview of real-time ANSAware 1.0, Distributed Systems Engineering Vol 2. No. 1;* **ISSN 0967-1846**, March 1995.

[4] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V., *RTP: A Transport Protocol for Real-Time Applications;* **RFC 1889**, Audio-Video Transport Working Group, January 1996.

[5] Gosling, J., Joy, B., Steele, G., *The Java^{TM} Language Specification;* **ISBN 0-201-63451-1**, Addison Wesley Longman Inc., August 1996.

[6] The Object Management Group, *The Common Object Request Broker: Architecture and Specification, revision 2.2;* **OMG 98-02-33**, OMG Headquarters, Framingham MA, U.S.A., February 1998.