A Generic Middleware Broker for Distributed Systems Integration

By

Richard Donald Slamkovic

A thesis submitted for the degree of Doctor of Philosophy

School of Computer Science and Information Technology,
Faculty of Applied Science,
RMIT University, Melbourne, Australia.

December 2005

Declaration

I certify that

- Except where due acknowledgement has been made, the work is that of the author alone;
- The work has not been submitted previously, in whole or in part, to qualify for any other academic award;
- The content of the thesis is the result of work that has been carried out since the official commencement date of the approved research program;
- Any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Richard D Slamkovic

December 2005

Acknowledgements

Firstly, I must express gratitude to my supervisors, Associate Professor James McGovern my senior supervisor, and Associate Professor George Fernandez my secondary supervisor. Without their encouragement and support, I do not think I would have made it to the end of this undertaking. Their continued guidance kept me on track when sometimes I would veer off on a tangent. I will always remember the times we shared both academically and socially.

I also wish to thank my wife Angela, who has patiently endured my part-time academic pursuits for most of the twenty-two years that we have been married. My two children Julian and Alicia also deserve some gratitude for still recognising their father after long periods of absence, locked away working on this thesis.

This research was independently funded and carried-out by myself using all my own equipment, except in the corporate case study. The corporation that participated in the case study has licensed components of the technology discussed in this paper, and currently this technology has a patent pending.

Contents

List of Figures

List of Tables

Summary

A wide range of middleware protocols have emerged over time to support enterprise information systems built on the client-server model. Typically, software components are built to support and access a particular middleware protocol. This restriction can be a major problem for organisations that wish to deploy multiple middleware protocols and be able to communicate across these heterogeneous protocols. The requirement for multiple middleware protocols and their integration has stemmed from the dynamics of business and technology, as businesses expand, merge or join in partnerships or alliances. New technology such as web technology, enterprise computing and e-commerce contribute to generating new business opportunities. A primary aim of middleware integration is to introduce new technologies, while protecting an enterprises' investment in reliable, useful software components that have been developed for particular protocols (so called legacy systems).

Seamless middleware integration requires conversion of the message format from the source of the request to that of its target. A number of solutions have been proposed and implemented, but they lack wide applicability and ease of use. This thesis proposes an improved solution based-on dynamic protocol-level systems integration using configuration, rather than programming. This allows large complex enterprises to extend and enhance their existing systems more easily. The major components of this solution are a Middleware Protocol Definition Language (MPDL) based on the Object Management Group (OMG) Interface Definition Language (IDL) that can describe a wide range of protocols declaratively, and a run-time environment, The Ubiquitous Broker Environment (TUBE), that takes these protocol descriptions and performs the necessary mediation and translation. The MPDL can describe a range of synchronous, asynchronous, object-based, and binary and text-based protocols. Each protocol need only be described once, and the framework provides a means to easily implement special extensions to the protocol. Further, this approach can be used as the

basis for developing new middleware protocols; the protocol used internally by TUBE is itself defined and executed using this approach.

A prototype system has been implemented and tested successfully across a wide range of scenarios that integrate a range of current middleware technologies. The system has also been tested in a large Australian corporation.

1 Middleware and Middleware Inter-operability

Middleware is a software layer that supports the interaction between components in a distributed computing environment. Middleware provides a set of services that allows client and server programs to communicate with each other across networks, and different computer systems. Middleware provides a means of separating functionality, to allow for independent development and deployment of software components that support presentation, business logic, and access to services, typically database access, for a specific application.

Middleware resides between the applications and the underlying operating system and network (as illustrated in Figure 1.1).

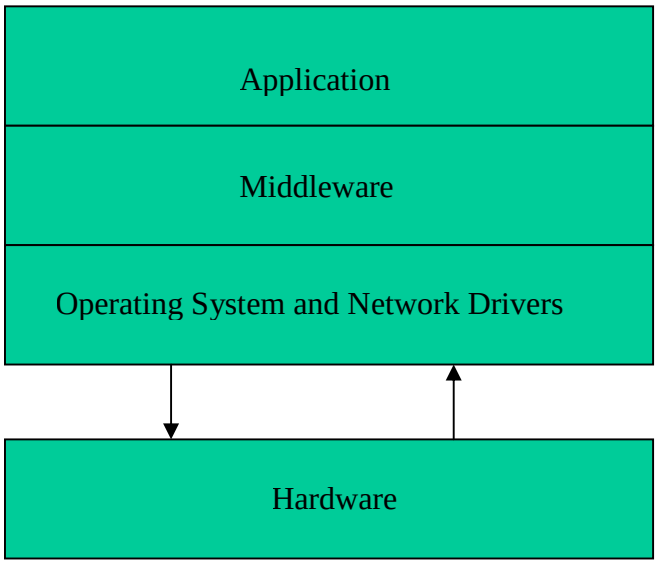

Figure 1-1: System layer context of Middleware

Since its emergence in the 1980's, the range of middleware has continually been expanded and improved. Major middleware types are synchronous procedural RPC (Remote Procedure Call) oriented middleware, such as DCE-RPC [Open Group, 1997]; asynchronous MOM (Message-Oriented-Middleware) based products, such as IBM's MQ-Series [MQ, 2001]; and transaction-oriented middleware including BEA's TUXEDO [BEA, 2001] and IBM's CICS [CICS, 2001]. Object-based middleware has been widely used, the best known of these being

OMG's CORBA [OMG, 2000], Microsoft's DCOM [Microsoft, 1998] and Java/RMI [RMI, 2003]. More recently, service-oriented architectures, such as Microsoft (.Net) and Java web services have given rise to middleware based on XML/SOAP [XML, 2003; SOAP, 2003].

Mobile computing environments present other challenges. Grace et. al. [2005] describe a reflective framework for the discovery and interaction of services in these environments. The middleware discussed in this paper focuses on the typical request-response model present in (usually) connected systems. Wireless connectivity and mobile environments are beyond the scope of this thesis.

In some cases, middleware is based on open or agreed standards, such as CORBA and XML/SOAP, while others, for example, DCOM, TUXEDO and CICS are proprietary. Often, organisations have their own in-house systems that do the work of third-party middleware. As a result of independent vendor product development, systems based on one of these methods are generally not directly protocol-level compatible with systems based on another. It is difficult to call a DCOM server from a CORBA client, although both systems are based on an object model. The implementations of these object-based systems are quite different [Orfali, Harkey and Edwards, 1996], and a CORBA client is not plug-compatible with a DCOM server, even if both run on a Windows (Intel) platform.

The need for system integration has stemmed from the dynamics of business and technology, as businesses expand, merge or join in partnerships or alliances. The introduction of new technology such as web technology, enterprise computing and e-commerce contribute to generating new business opportunities. Some organisations, particularly large diverse organisations, will have multiple middleware products and approaches. Different parts of the organisation may have adapted different middleware to best meet their needs in the past. In some cases, it may be desirable to maintain the legacy system, or the costs of replacement may be prohibitive. Companies are taking-over or merging with other companies, and small

companies increasingly have had to join global networks to compete locally (for example, “small companies, global networks” [Joia, 2000]). The resulting super-organisations typically include a mix of generally incompatible IT systems that require integration as quickly as possible to exploit the new business structures. The problem of protocol-level systems integration compounds if both companies use different operating system environments. The ability to maintain existing middleware and applications will result in significant cost savings in the context of new requirements or new application and middleware deployment.

Organisations continually update their IT infrastructure. Emerging middleware approaches, for example, XML/SOAP Web Services promise long-term benefits to the organisation, and it will be desirable to integrate new approaches into the existing environment.

The integration of middleware is a significant problem for these typically large organisations. Protocol-level integration of legacy systems with other systems has been reported to be a major challenge [van Steen, 1999] with no obvious general solutions. Low-level systems integration is difficult, because application semantics must be addressed and low-level manual data marshalling is often required [Emmerich, 2000; Blair, 1999; Geihs, 2001]. Vawter and Roman [2001] support this in the domain of current web-based systems stating that, “legacy integration is often the most challenging (if not the most challenging) task to overcome when building a web service”.

Current integration of middleware requires a significant amount of programming. The coding is either, commissioned by the organisation at considerable cost, or provided off-the-shelf by a vendor, as a “middleware connector”. In both cases, the solution is one-off, and limited to a specific middleware integration problem. This thesis describes an improved mechanism for the integration of a wide range of middleware. The aim is to allow organisations to more

easily have a client, written for one middleware protocol (for example a CORBA client), access a server implemented in another, such as a Web Service, a .Net [Microsoft, 2001] client or even a legacy system service. The approach will extend the useful life of client and server software, and enable organisations to adopt new middleware more easily, while maintaining existing infrastructure and applications. It does this by providing a declarative Middleware Protocol Definition Language (MPDL) that describes middleware, and a run-time engine called The Ubiquitous Broker Environment (TUBE) that takes compiled protocol descriptions and carries out requests across different middleware protocols. This research project addresses two broad questions, aimed at providing the base of a general and improved approach to middleware integration, based on describing protocols, rather than programming.

1. Is it possible to describe the characteristics of middleware protocols in a declarative language?
2. Can an engine be implemented that can take the protocol descriptions and carry out the run-time integration?

This research provides evidence that such a language and engine can be developed, to carry out all major current middleware integration. This approach will deal with new middleware, without modification or with relatively simple modification.

The rest of the thesis is organised as follows. Chapter 2 describes the nature of middleware, expands on the problem of middleware integration and provides an overview of other approaches to the problem. Chapter 3 introduces the TUBE approach. Chapter 4 defines the Middleware Protocol Definition Language (MPDL). Chapter 5 describes Message Processing and Data Marshalling and expands on the implementation infrastructure. Chapter 6 describes the actual implementation and its evaluation. Chapter 7 concludes the paper with a discussion

of some future research options. In addition, there a number of appendices that contain items commonly referred to within the body of the paper.

2 Middleware Inter-operability Requirements

2.1 Middleware

The core of middleware functionality is to support the client-server computing paradigm where applications are based on a program (a client) making a request of another program (the server or service), and the server providing a response to that request. The client and the server are independent and connected only through the interface of the server. The explicit linking or binding of a particular server to a client request is usually done at run-time and is part of the role of the middleware.

Client server computing allows applications to be built in a number of independent layers each with a number of different software components, possibly residing on different platforms. This inherent flexibility has the advantage of allowing different components to be changed without compiling unaffected components.

In client-server systems, the middleware carries out the following major high-level tasks.

i. It must set-up the interaction and carry out required initialisation.

ii. It must manage the interaction session, including sending messages, receiving messages, monitoring connections and handling errors and recovery. The sending and receiving of messages requires that the messages (client requests and server responses) are converted (marshalled and un-marshalled) to formats that can be transferred through networks and operating systems to the targeted servers.

iii. It must terminate the interaction.

2.2 End-Point Resolution

Middleware calls can be made by a proxy or stub generally created at compile time. Typically, middleware uses an IDL to facilitate run-time binding to servers. The IDL defines the interfaces of servers and the data exchanged between the client and the server. Before this exchange can occur, the middleware must determine the server address (that is, how to contact the server). This is what is referred to as the end-point. The end-point is comprised of protocol-specific information, such as the IP-address and port number of an Object Request Broker (ORB). Each middleware type has its own scheme for determining, and, or persisting this addressing information. This address could be, for example, the IOR (Inter-operable Object Reference) for a CORBA server, or a queue definition for MQ-Series. This provides the necessary information to send a message to, or communicate with, a defined interface using a particular protocol. A middleware broker must store these definitions against each interface defined for the particular middleware protocol. This facilitates the ability to change middleware configurations on an interface basis. There is no need to perform all activity using a single middleware type.

It follows that if a client can access a service across different middleware, that there may be more than one implementation of a service accessible through different middleware. Conventional middleware will fail if it cannot complete a request, even if an alternative could satisfy the request. The middleware determines a single service provider and marshals the call appropriately, and the onus is on the application developer to support any fail-over. The middleware only understands its own message formats and addressing schemes. For example, the default may be CORBA, and calls will target CORBA end-points (for example, an Interoperable Object Reference); however, an available alternative may be available through MQ-Series. Access to this service will not be attempted should the CORBA service request

fail. This is because the CORBA middleware is un-aware of the MQ configuration, and does not know how to contact the end-point; in this case, an MQ-Series queue. The ability to change the middleware dynamically offers the opportunity to access different servers to satisfy the same request.

Middleware is based on two major models, the Remote Procedure Call (RPC) model and the Message-Oriented Middleware (MOM) model.

2.3 The RPC Model

The purpose of the Remote Procedure Call model is to abstract the server location and method of communication from the client program. The client uses a local stub module to communicate with the server. The client is un-aware of the servers' location; it may be local (residing on the same machine) or remote. The stub contains the mechanism required to marshal and un-marshal parameters for requests and responses and communicate with the server. The stub is produced by an IDL (Interface Definition Language) compiler and is closely coupled to the RPC vendors' runtime libraries. In this synchronous mode of interaction, the client expects a response from the server in a finite amount of time. A timeout condition will occur if the response is not returned within a specified period. The RPC model is analogous to the line-based telephone system, in that the parties remain connected for the duration of the exchange. The following diagram shows the basic RPC model.

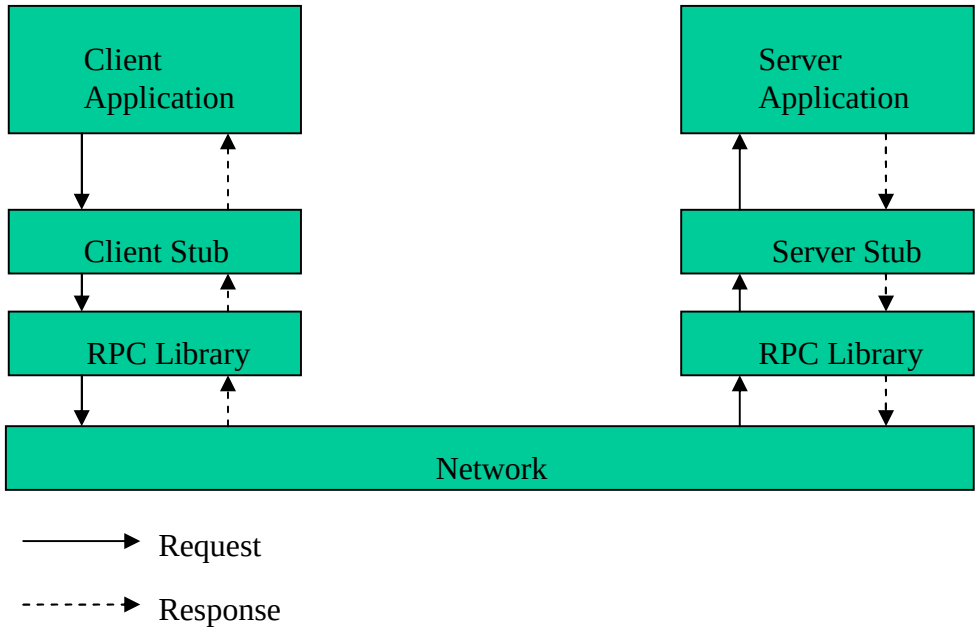

Figure 2-1: Remote Procedure Call Model

This more recent object-based middleware still follow this basic model. CORBA, COM and Java RMI all use IDL compilers to generate client-side stubs and provide runtime support libraries that perform the actual communication.

2.4 The Message-Oriented Middleware (MOM) Model

In contrast to the synchronous RPC-based model, the Message-Oriented model (generally) does not use an IDL compiler to generate stubs. The applications make calls to specific API modules to carry out the interaction. This involves placing messages onto queues and monitoring queues for response messages. This model is asynchronous, which means that the response may arrive at anytime. As with the RPC model, timeouts may be specified, although they are usually of a longer duration. The MOM model is analogous to the domestic postal system, where a letter is sent from one place to another and a response may or may not arrive. It is the responsibility of the person that posted the letter to monitor and follow-up on any expected response if it does not arrive. In the MOM model, this is the responsibility of the application. An extension to the MOM model is the publish-subscribe method. This provides

the ability for one application (the subscriber) to register interest in (subscribe to) the services offered by another application (the publisher). The subscriber can subscribe to events of interest by subject. For example, an application may wish to be notified when the price of certain stocks change; this application would subscribe to a subject like “price-movement”. This type of interaction is used in modern EAI/workflow systems such as TIBCO Businessworks (TIBCO 2005). The underlying protocol TIBCO Rendezvous is based on subject-based messaging.

2.5 Message Marshalling and Un-marshalling

Middleware generally works with messages in particular formats, either text or binary based. Text-based protocols include XML, HTTP and SOAP[1]. Binary protocols include object-based protocols (for example CORBA, COM and Java-RMI) and others such as DCE-RPC, which are not object-based. There are also Message Oriented Middleware (MOM) protocols such as MQ-Series and JMS [JMS, 2003]. Each protocol wraps or encapsulates the actual message content in different ways. SOAP for example wraps the content in a structure called a SOAP body, and then wraps this in another structure known as a SOAP envelope. The envelope also has an optional structure called a header. The message content referred to above is the application specific body of the message as defined for the interface.

The IDL defines the format and data types of the parameters passed between clients and servers. Middleware interoperability must be able to convert messages between different middleware formats, or provide a method that marshals the target interface into the desired protocol, based on the different middleware definitions.

[1] SOAP is actually XML carried over a (usually HTTP) network connection.

The user's data that is in the body of a message and forms the parameters of a call can be a combination of native types or complex types, declared and defined for each application. There are a limited number of native (intrinsic) data types: integer, long, short, character/byte, double/float and the bit. All other data types are constructed from these fundamental types. Once a mechanism for reading and writing these types is identified, structures of any complexity can be processed. The protocol defines the rules and mechanisms for reading and writing these basic types. These rules will then process any message over this protocol. For example, CORBA uses an encoding known as CDR (Common Data Representation) [OMG, 2000] for reading and writing basic data types. Once we define the rules of CDR or a callable library that implements the rules of CDR, it is possible to process CDR-based (CORBA) request and response messages.

Broadly speaking, messages between clients and servers, irrespective of communication mode, consist of two types of data, internal data and a payload. Internal data provides the necessary information to find a requested service, and to send responses back to the requestor. The payload is application-specific data needed to carry out the application functionality. The internal (control) data can be further divided into common data, that is, data that is needed for all middleware protocols, and protocol-specific data. The common data may contain varying values depending on the protocol, or appear in different places within a message in different protocols; however, they are needed for any message exchange regardless of protocol. These variables keep track of such things (among others) as; message lengths, sequence numbers, and whether the system is dealing with a request or a response. These variables are described in detail in Chapter 4. In the discussion that follows, these mandatory variables are referred to as common internal middleware variables. The protocol-specific data on the other hand, are variables, which only have meaning for a particular protocol. For example, a CORBA object-

key [OMG, 2000], which identifies the object to instantiate (or invoke) on the target end of a CORBA request.

2.6 Client-Server Interaction Modes

Middleware supports two primary modes of interaction or communication, synchronous and asynchronous. With synchronous communication, the client program makes a request of a server program, and then waits for a response. The client remains blocked or in a waiting state, until it receives a response from the requested service. Examples of middleware that support this mode of interaction are those based on Remote Procedure Calls (RPC), and include CORBA, COM, Java-RMI, SOAP (web-services) and DCE-RPC. With asynchronous communication, the client places a message on a message bus (or queue) of some kind and resumes processing. The request may require a response, but the client can continue to perform other tasks while waiting for the response. This approach is typified by Message Oriented Middleware (MOM) protocols such as MQ-Series and JMS.

Typically, interaction is restricted to one mode. True interoperability among middleware protocols will mean that calls should not only be able to be made across the same modes, but also across middleware using different modes of interaction, in some cases modes not directly supported by that middleware type. Consider the following example. A client only knows how to make a strictly synchronous request on a server, S1 using some synchronous protocol. Suppose the organisation introduces a new asynchronous middleware product and server S2, and wishes that client to access this new server. The client is unaware that the server implementation has been changed to use asynchronous queuing. A synchronous session needs to be held with the client, which is awaiting a response and is thus blocked. At the same time, a queue on the server-side must be monitored until a response that could come at anytime

arrives. When a response arrives, it is sent back to the waiting client. Ideally, this should be done without altering or compiling the client.

2.6.1 Conversational and Transactional Interaction

Within the synchronous and asynchronous interaction modes, there are two sub-types: conversational and transactional. In conversational mode the client and server have a "conversation", that is the interaction consists of multiple two-way message exchanges. Generally, in this type of exchange the client, the server or sometimes both needs to maintain the state of the exchange. One or both needs to know what point of the interaction they are up to in case the exchange is unexpectedly terminated. In the transactional mode, the client and server engage in a "transaction", which means they perform (generally) a single exchange of request and response. The communication is usually terminated once the client receives the response. There is also no need for either to maintain state.

Sometimes these modes are combined, for example, a conversational exchange may be employed in the context of a single transaction. In this case, there are usually special messages sent to mark the beginning and end of the transaction. The conversation occurs between these marker exchanges. This is more a function of the application than the middleware; however, the middleware must be able to support all these interaction variants.

In addition, middleware may also have some high level semantics above the standard synchronous and asynchronous communication modes. For example, the CORBA LOCATION-FORWARD response message (refer OMG, 2000) performs a re-direction task telling the client to resubmit the original request to a new target end-point, and the ability to handle these must be incorporated into any general approach to middleware interoperability.

2.6.2 Example Client-Server Interaction

A simple math server definition shown in Figure 1 illustrates basic middleware behaviour. The math server takes two (2) types of arguments, a char operator, from the domain <”A”, “S“, “M”, “D”> and up to two integer operands and returns an integer result. Figure 2 and Figure 3 below show the basic structure of a request and successful response message for an “add” operation of the numbers “1000” and “15” on the mathServer interface. The server may also return an exception or error condition. This is shown in Figure 4, where the div (divide) operation was called with “1000” and “0”. This is an illegal operation, and hence the server returns an exception. The exception is defined as a structure that contains one member, a string describing the error. It could however, be considerably more complex. The example exception shown is protocol-neutral, that is, it does not represent any specific protocol mapping. It is merely illustrative.

```
interface mathServer
{
    // request structure
    struct math_req
    {
        char   op_code;
        long   num1;
        long   num2;
     };


    // response structure
    struct math_resp
    {
        long   ret_num;
    };


    // define the exception
    exception mathException
    {
        string error_text;
    };

     // methods (services, functions, operations)
     void add(in math_req mr, out math_resp arsp) raises (mathException);
     void sub(in math_req mr, out math_resp srsp) raises (mathException);
     void mul(in math_req mr, out math_resp mrsp) raises (mathException);
     void div(in math_req mr, out math_resp drsp) raises (mathException);

};
```

Figure 2-2: mathServer IDL

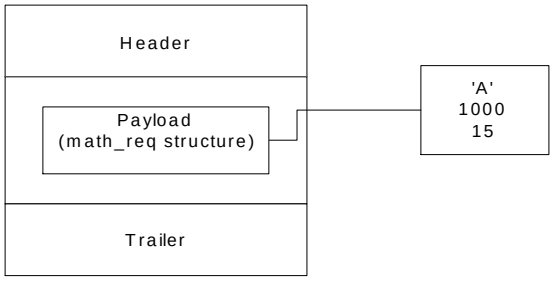

Figure 2-3: Structure of request message (highlighting payload)

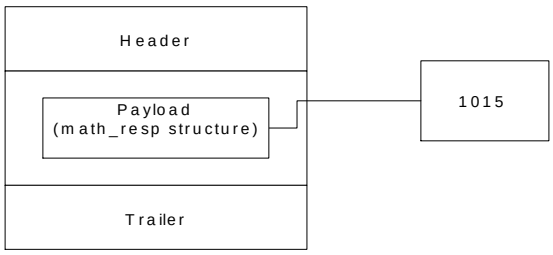

Figure 2-4: Structure of a successful response message (highlighting payload)

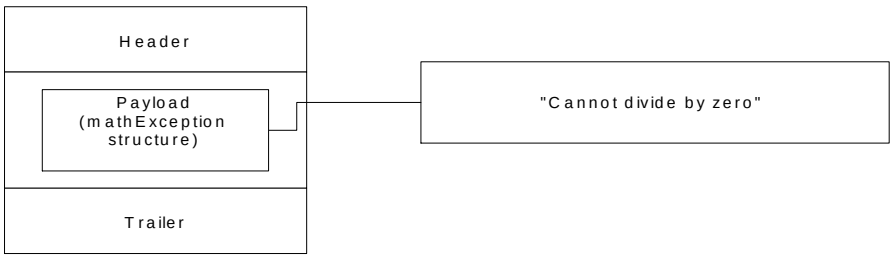

Figure 2-5: Structure of an unsuccessful response message with an exception as payload

The interface definition shows the payload for a request as a math_req structure (see Figure 2-3). The response is either a math_resp structure (see Figure 2-4) or some failure response (see Figure 2-5). The payload for a message is both the (serialised) input and input-output parameters to the operation, and the (serialised) response from the operation, whether successful or not. The example 'add' method is defined as void (returning no value) and therefore no return value is shown. The definition provides the information needed to marshal these structures. It provides the native types that constitute them. What is unknown is how to marshal them over a particular protocol. For example, integer (int) values may need conversion to text to be sent as XML. Does the protocol require the payload wrapped in some other structures, such as headers or trailers? Knowing the structure of the interface is not enough to interoperate across protocols. What needs to be added to the message and what

needs to be converted[2] to accommodate a particular protocol must be known as well. The address (in protocol-specific terms) of the end-point (target) is also required. This may be a host name and port number or a queue name, or perhaps a directory name. These items of "protocol structure" must be addressed in any general middleware broker.

2.6.3 Approaches to Middleware Inter-operability.

A number of middleware approaches are described in the literature and a number have been implemented as commercial products. A typical automated approach is that proposed by Dashofy et al. [1999] who investigated using various off-the-shelf middleware products to build bridges or connectors for distributed systems. They present their views from a software architecture perspective, restricted to the C2 [Taylor et al., 1996] architectural model. They have built software connectors that are specific to four middleware packages, Q, Polylith, RMI, and ILU. This means that to add support for another middleware package, a new specific connector (program) would need to be developed. Commercial tools such as those provided by commercial Enterprise Application Integration (EAI) products are similarly restricted. The majority of approaches require software to be re-written to communicate with each different middleware implementation. In contrast, this project aims for an easier and more flexible approach in which rules and middleware characteristics are specified in a repository, for the broker to provide the connection and transformation for different middleware protocols, as well as for legacy systems.

The broad classifications for different approaches to inter-operability are:

- Handcrafted solutions consist of writing ad-hoc software to implement each interoperation requirement. Although this approach is widely applicable, and very commonly used, it is labour-intensive and requires considerable expertise not always available. Such approaches are also difficult to maintain

[2] This may be character-set conversion such as ASCII to EBCDIC and conversion of numbers to strings and so on.

over time. Each implementation is required to provide code that performs traversal of complex data structures.

- Proprietary approaches (commercial EAI products) usually result in the user being locked-in with a proprietary solution. These approaches also require a considerable amount of coding to implement all but the most trivial applications.

- Architectural approaches provide mainly a high level modelling view of systems, and are not of much practical benefit to the low-level systems integrator. Certainly, they do not allow for any automated protocol-level integration.

- Specific middleware approach where systems such as ASTER [Issarny, 1998] provide an API that allows translation of different protocols to CORBA (that is, ASTER relies on a single middleware, CORBA, to provide all remote (RPC) and component services).

- Declarative and semi-automated approaches that provide a description of a protocol and an engine or set of procedures and data stores that can convert any protocol to any other protocol.

The object of this research fits into the category of declarative and semi-automated approaches. The advantage of this type of approach is that programming is minimised and a protocol need only be defined once. The disadvantage is that a MPDL that can handle a wide range of protocols is necessarily complex, as will be the engine that carries out the integration at run-time.

Obviously, such approaches are of commercial value, and some of the relevant work appears in the realm of patents, rather than in the published literature. An approach using a protocol definition language is described in a US patent held by Holzmann [1998]. The language described is very low-level, assembly-like language, and is not aimed to be easily understood by a wide range of developers. Both client and server nodes involved in the message exchange must implement the protocol that defines the language. The first message in every exchange is the description of the protocol that the parties will use for communication. For example, if both nodes wanted to use IIOP (Internet Inter-Orb Protocol), the initiating node would have to send the description of IIOP to the receiving node before any actual data could be exchanged. It is not clear from the published patent document whether a protocol as complex as IIOP can be handled by this mechanism. Further, the nodes involved in the message exchange must be able to implement both message conversions. For example, if the initiating node is using IIOP and the receiving node is using SOAP (HTTP), both must be able to carry out the needed conversion.

Kuznetsov [2004] describes a method of generating message translation at runtime. This is achieved by creating an XSL [XSL, 2005] mapping of the source and target interfaces. This mapping is used at message processing time to generate native object code that transforms the message from a source interface to a target interface representation. This appears to cater more for interface mapping rather than different middleware formats. The approach is XML/XSLT focused. There is mention of parsers for various types of definitions (for example, ASN.1, C++) that describe the data structures of the interface, not the characteristics of the protocols.

2.6.4 Criteria for General Middleware Inter-operability

Although protocol translation is recognised as an important problem, there is no generally accepted single way to do this, and there is considerable scope to improve on the current approaches. Other approaches that provide re-configurable middleware frameworks [Coulson et al, 2002], do not directly address protocol translation. The protocol translation is catered for by a plug-in based architecture, implying that the framework user will need to either source (from a third party) or program the required plug-in to perform the translation. A general middleware broker must work with a range of existing middleware protocols, and be easily adapted to any new middleware protocol. It should minimise programming extensions to the middleware integration tool when adding new middleware protocols, preferably being able to describe middleware protocols in a completely declarative manner, and in a language that can be easily used by a wide range of developers. The characteristics of an improved and generic middleware broker can be summarised in the following five major requirements.

Requirement 1: A general middleware broker must be able to describe all middleware protocols in an easy to use, declarative way. This requirement governs the overall design of the MPDL that must minimise its semantics, and must make use of existing standards and languages. The overall aim is to provide the capability to describe protocols to application developers, and to minimise procedural programming. Specifically the broker must:

i. provide a framework for implementing any special semantics, such as interaction semantics or data coding and de-coding functions,
ii. describe a middleware product once for integration across all protocols,
iii. be easily extended by programming in an understandable and maintainable way,

iv. special protocol handling code is written once, not for every interface. This allows optimal re-use of code and uniform treatment of all interfaces over the protocol.

Requirement 2: A general middleware broker must be able to convert data and message formats from any middleware protocol to that of any other middleware protocol. Specifically, the broker must:

i. deal with different transport formats (text or binary),
ii. deal with all native data types, complex data, and user defined data.

Requirement 3: A general middleware broker must be able to seamlessly and transparently deliver requests in one protocol to a service in another protocol, and deliver responses back to the requestor. This requirement lies at the core of how the functionality is implemented. Specifically, the broker must:

i. support different modes of interaction (synchronous or asynchronous),
ii. access servers through different protocols, without re-compiling application programs,
iii. support deployment at one end only, so that a server-receiving node will receive and respond to requests in its own protocol, without any need to process this request in any different way.

Requirement 4: A general middleware broker must be able to choose between multiple options for providing a service. This requirement means that the broker must be able to provide the ability to dynamically locate alternative end-points, so that alternative services may be accessed, should they be available. One mechanism for supporting alternate end-

points is protocol prioritisation. This allows the user to list protocols in priority order. The broker will try each listed protocol in-turn until either, the message delivery is successful, or all protocols are exhausted. In the latter case, the client receives a failure indication.

Requirement 5: A general middleware broker must be able to effectively operate in a data intensive, real world environment. It must be easily adaptable to that environment, and it must perform efficiently.

TUBE aims to improve on current automated approaches, particularly those outlined by Holzmann [1998] and Kuznetov [2004]. It will provide an MPDL that is an extension of IDL and thus easily understood, it will be capable of describing complex protocols, such as IIOP (CORBA). The protocols and interfaces are independent of one another. That is, the interfaces are defined in IDL and the protocols are defined in MPDL. The MPDL define all the characteristics of the middleware, including end-points and any APIs or libraries required.

TUBE will allow nodes involved in message exchange to use whichever protocol best suits their needs, with no requirement for either end to implement a particular protocol. For example, if the initiating node is using IIOP and the receiving node is using SOAP (HTTP), neither need be aware of the others' protocol. The sender simply sends the message in IIOP and the receiver will receive it in SOAP over HTTP. The broker maps the protocols at one or both ends.

By keeping the interface definition separate from the encoding definition TUBE allows any protocol and interface combination to be catered for. TUBE does not limit translation to approaches such as XML using XSLT, as does Kuznetov [2004]. In fact, XSLT is not used at

all, and we argue that the binary format used in TUBE repositories is better suited for describing complex data structures.

The TUBE approach is neither interface nor protocol centric. One could simply load all interfaces into the Module Definition Repository and have them processed by a single protocol, or multiple protocols. Conversely, all protocols can be loaded into the Protocol Definition Repository, and then used to process a single interface or multiple interfaces. These combinations may be changed as required, without any coding.

Unlike some proprietary EAI products, which attempt to control workflow and broadcast (publish) each message on a universal messaging bus, TUBE only communicates with designated end-points. However, TUBE is capable of broadcasting or publishing to a universal bus, if that is required. Since TUBE will provide fully synchronous or asynchronous methods, the desired communication type may be changed at anytime without system impact. For example, if synchronous behaviour is required from an (essentially) asynchronous middleware platform (such as MQ-Series), TUBE will handle the synchronisation through blocking and buffering. If it is then required to go back to purely asynchronous, the application software does not need to change, provided that the protocol is supported for the called interface. This will allow remote modules to be developed independently, and for each to use the middleware that best suits their purposes. There will be no need for independent development groups to be familiar with each other's protocols.

The next chapter provides a broad overview of the approach taken in the design of TUBE. The main components and its operation are briefly described. The following chapters define each of the major components in more detail.

3 A General Approach to Middleware Integration

3.1 Middleware Protocol Descriptions: The Protocol Definition Repository

A key design goal is to be able to describe any middleware protocol in an easy to use declarative language. Modules and protocols are described in a Middleware Protocol Definition Language (MPDL). MPDL is a declarative scripting language (based on OMG-IDL) that enables protocol descriptions. The MPDL provides all binding information, interfaces and protocol end-points, all the information required to marshal and un-marshal data across different middleware protocols. The MPDL will also specify any specific programs that need to be invoked to support protocol migration. For example, a protocol that applies encryption and decryption to all data through specific functions would have these functions and their application included in the protocol descriptions.

The MPDL compiler creates a Protocol Definition Repository (PDR) that describes all the protocol specific information needed to make requests and receive responses through a particular protocol. An interface description compiler creates a Module Definition Repository (MDR) that defines all the module specific information, such as interfaces and data, needed to locate and bind to a particular module. The run-time engine will access these repositories to translate between protocols.

TUBE Protocol Definition scripts are submitted to the MPDL compiler (see Section 4.1), which converts this into an internal format stored in a Protocol Definition Repository (see Figure 3-1) for TUBE to process at runtime. This internal format is discussed in detail in Chapter 5. The TUBE Interface Definition Language (IDL) compiler processes the IDL definition of the interfaces that need to communicate and stores this information in its Module Definition Repository (see Figure 3-2). These data in conjunction with the protocol definition

(stored in the Protocol Definition Repository) is all that TUBE needs to convert messages between different middleware formats.

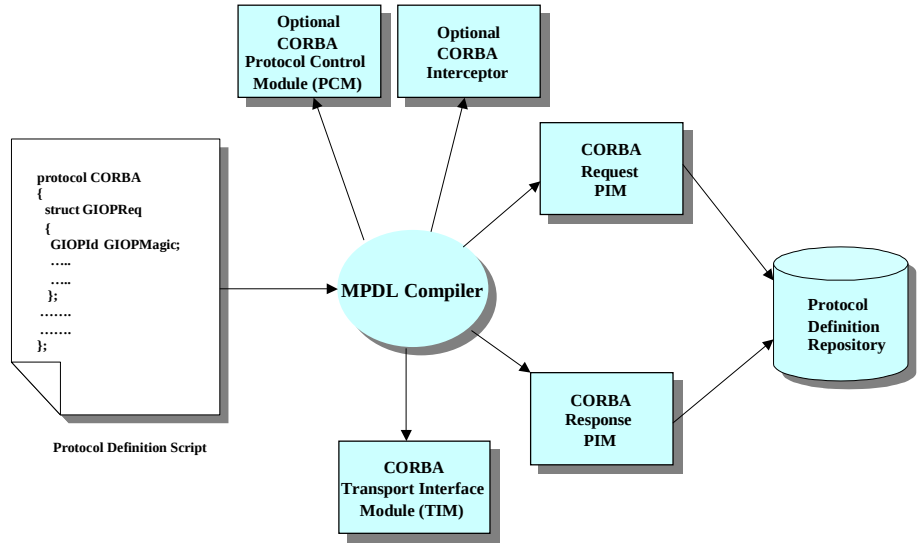

Figure 3-1: TUBE Protocol definition process

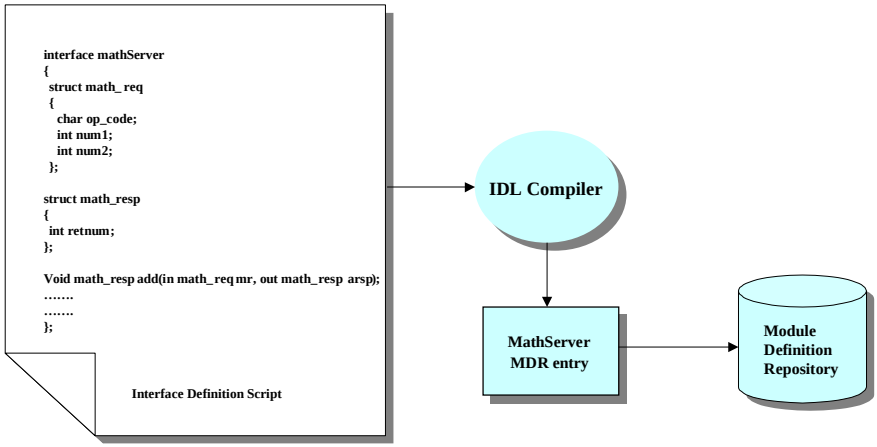

Figure 3-2: TUBE Interface definition process

3.1.1 Application Access Mechanisms to TUBE

Different applications will have varying requirements for how they will interact with TUBE.

The options available are:

- the XML-based API,
- the Java object-based API, and
- Protocol Interceptors.

In all cases, the application is un-aware of the protocol used to communicate with the server. For example, some applications may use the XML-based API. In the XML-based API, the application submits an XML request document and receives another XML document in response. The client only ever sees XML conforming to the module's IDL definition. In the case of the object-based API, the client supplies native Java objects as parameters and receives native Java objects in response. These API-based access modes are more suited to new applications. In contrast, for existing applications where the desire is not to write any new code or change any code, the Protocol Interceptor method is a better choice. Using this method, the application does not change in any way. The only change needed is in the configuration of the server address. This assumes that the server address is specified in some type of accessible configuration. If the server address is hard-coded or coupled to the application in some other way, then code changes cannot be avoided.

In the Protocol Interceptor mode, applications specify the service that they want by using the API of that service. For example, a CORBA client will invoke a method on an interface stub. These calls are intercepted by TUBE, which determines a service provider and marshals the call appropriately. As TUBE can be accessed directly through its own API, it supports the implementation of all application software in a protocol neutral way.

3.2 Multiple End-Point Resolution

TUBE has two main mechanisms for supporting multiple end-point definitions for the same interface. One is a protocol alias and the other is protocol prioritisation. The protocol alias

allows the same protocol to be used for marshalling, whilst targeting different servers. For example, a message is sent to a CORBA server and the server does not respond. This will be considered a failure. TUBE will now try the next protocol in the Distribution Priority Table. The protocol is CORBA2 and it has an alias of CORBA. This means that the end-point information is taken from the End Point Resolution Table (see below) entry for CORBA2, however the marshalling rules will use CORBA definitions. This allows multiple end-points to be configured for the one protocol and interface, thus providing a fail-over mechanism.

The other way that TUBE supports multiple end-points is by protocol prioritisation. If the required service is not available through a preferred protocol, then TUBE tries alternative protocols. For example, the default may be CORBA, and calls will target CORBA end-points (for example, an IOR); however, an alternative may be MQ-Series, which will be tried if a CORBA service cannot be reached. (The onus will be on the systems integrator to specify those protocols that are interchangeable for each interface).

The Distribution Priority Table stores the names of the various protocols supported for each interface defined in the Module Definition Repository. These protocols are stored in priority order. That is, starting by the preferred protocol, followed by each subsequent protocol. Each entry in the Distribution Priority Table corresponds to an entry in the End-Point Resolution Table. This table defines the communication parameters necessary to communicate with the interface over the specified protocol. In the case of CORBA, for example, this would be the IOR for a server that implements the desired interface. The information stored here depends entirely on the protocol. These two tables are used in conjunction by TUBE to determine where and how to send messages between different middleware.

3.3 The UBE (Ubiquitous Broker Environment) Architecture

Figure 3-3 shows the main components of TUBE. Systems that work through TUBE will use the TUBE API, or use their own middleware API, and have these calls intercepted and processed by TUBE. TUBE consists of four (4) main process components, in addition to its four (4) repositories.

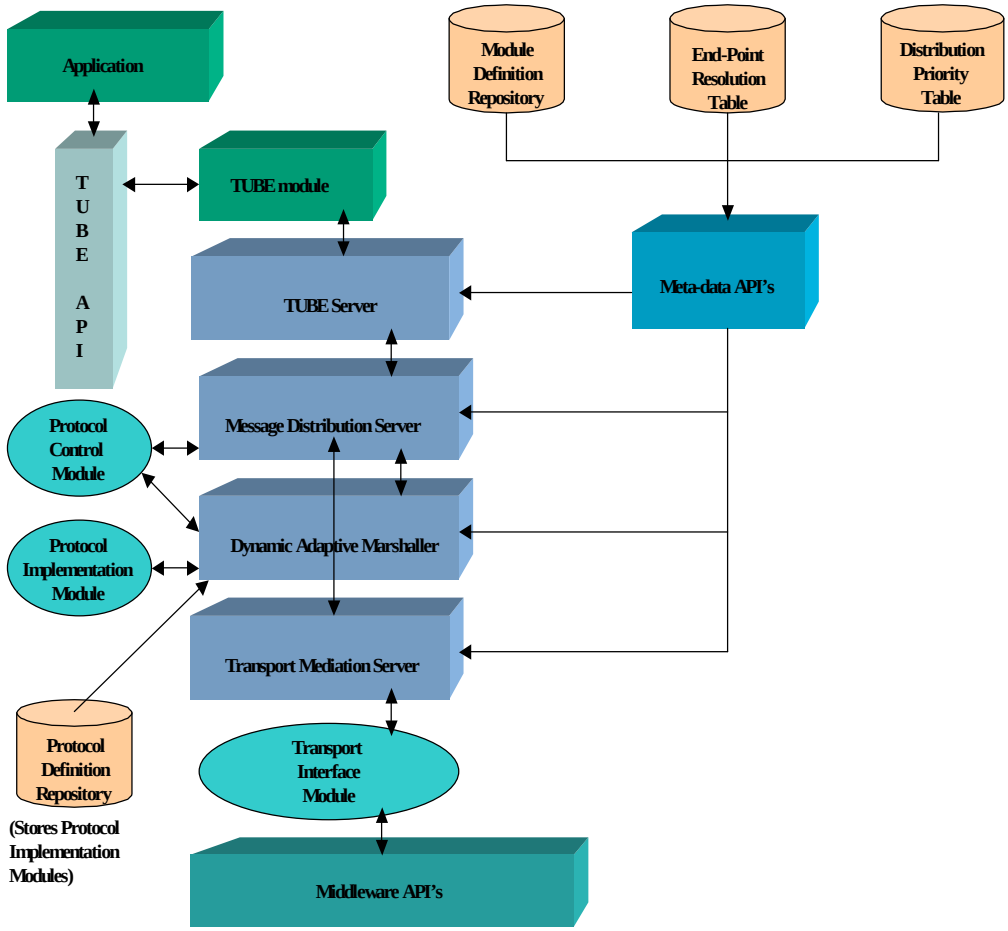

Figure 3-3: TUBE Component Architecture

3.3.1 Overview of TUBE Components

- The TUBE server provides the entry-points for the APIs. Both client code and TUBE internal code communicate through the interfaces provided.
- The Message Distribution Server (MDS) associates each request for a service with a particular protocol. The MDS requests data marshalling and un-marshalling, and

message transmission. The Message Distribution Server is broadly responsible for:

  - determining the source protocol of the message,
  - finding the preferred target protocol in the Distribution Priority Table,
  - determining the target end-point from the End Point Resolution Table, and
  - passing the message between the other components.

- The Dynamic Adaptive Marshaller (DAM) prepares requests for a particular protocol. Given a request, it looks-up the definition of marshalling rules for the requested protocol, and the target interface definition in the Module Definition Repository. It then marshals the target interface into the desired protocol, based on the definitions from both repositories. It also un-marshals from the source protocol into an internal protocol-neutral format (see Section 3.4). The DAM loads pre-compiled Protocol Implementation Modules (PIMs) that perform the actual marshalling and un-marshalling operations. The Protocol Implementation Modules contain the binary format of the rules defined in the protocol description.

- The Transport Mediation Server (TMS) is responsible for managing communications with the target end-point for the interface. It uses the information from the End Point Resolution Table, such as the IP-address and port number of an ORB, to determine the destination. The TMS passes this information onto a loadable module, known as a Transport Interface Module (TIM), which then handles the communication with the target. A Transport Interface Module is analogous to a Protocol Implementation Module (described above), except that the Transport Interface Module contains the rules for communicating with the target, rather than rules for marshalling data. The Transport Mediation Server and Transport Interface Modules are discussed in Section 5.7.

3.3.2 TUBE Repositories

- Module Definition Repository (MDR)

  The MDR stores the meta-definition of the particular interface. This includes the interface identifier and the data types of the parameters passed. This information is derived from the IDL for the interface.

- Distribution Priority Table (DPT).

  For each interface defined in the MDR, the DPT stores a list of protocols that can be used to communicate with this interface, stored in priority order.

- Protocol Definition Repository (PDR).

  The PDR stores the marshalling rules for each protocol. These rules are generic for each protocol and not specific to any interface stored in the MDR. The PDR consists of a collection of Protocol Implementation Modules.

- End-Point Resolution Table (EPRT).

  The EPRT stores the target communication address for each interface/protocol combination. This address could be, for example, the IOR for a CORBA server, or a queue definition for MQ series. This table stores the necessary information to send a message to, or communicate with, a defined interface using a particular protocol. The number and types of entries depends entirely on the protocol.

3.4 TUBE Internal Data Formats

TUBE uses different data formats internally depending on the situation. In the diagrams below the Protocol Independent Data Streams (PIDS) are the format used internally to pass data

between the TUBE API, the server and the DAM components. This format is referred to as a TLV (Type, Length, and Value) buffer. Each entry in the TLV buffer contains the information required to marshal that particular type into any protocol format defined in the Protocol Definition Repository. The Protocol Oriented Data Streams (PODS) on the other-hand consist of data marshalled into a protocol-specific format (for example, CORBA) by DAM. These are passed internally between DAM, the MDS, TMS and, if required middleware-specific APIs.

3.5 Communication Modes

It has already been mentioned that TUBE provides the ability to use either or both synchronous and asynchronous communication modes, and that the desired method can be changed at anytime without system impact. When it is required to switch from one mode to the other, all that is required is to change the configuration. This can be done on a per module/interface basis, even while the system is running; there is no need to shutdown and re-start the broker.

3.6 Out-bound Message Process Flow

The following scenario depicted in Figure 3-4 describes the process-flow of an out-bound message through TUBE:

1. The application call is passed to the TUBE API via the TUBE server.
2. The TUBE server passes the call to the Message Distribution Server.
3. The Message Distribution Server selects a protocol to try from the Distribution Priority Table.
4. The Message Distribution Server passes the interface/module identifier and the preferred protocol to the Dynamic Adaptive Marshaller.

5. The Dynamic Adaptive Marshaller reads the Module Definition Repository to determine the format and arguments for the call.
6. The Dynamic Adaptive Marshaller uses the protocol definitions stored in the Protocol Definition Repository to prepare the call for the particular middleware or application service.
7. The Dynamic Adaptive Marshaller passes the marshalled buffer back to the Message Distribution Server.
8. The Message Distribution Server passes the marshalled message to the Transport Mediation Server and tells it which protocol to use for transmission.
9. The Transport Mediation Server reads the End-Point Resolution Table to determine the host and port number required to communicate over this protocol.
10. The Transport Mediation Server attempts to communicate with the designated host using the appropriate communication parameters.

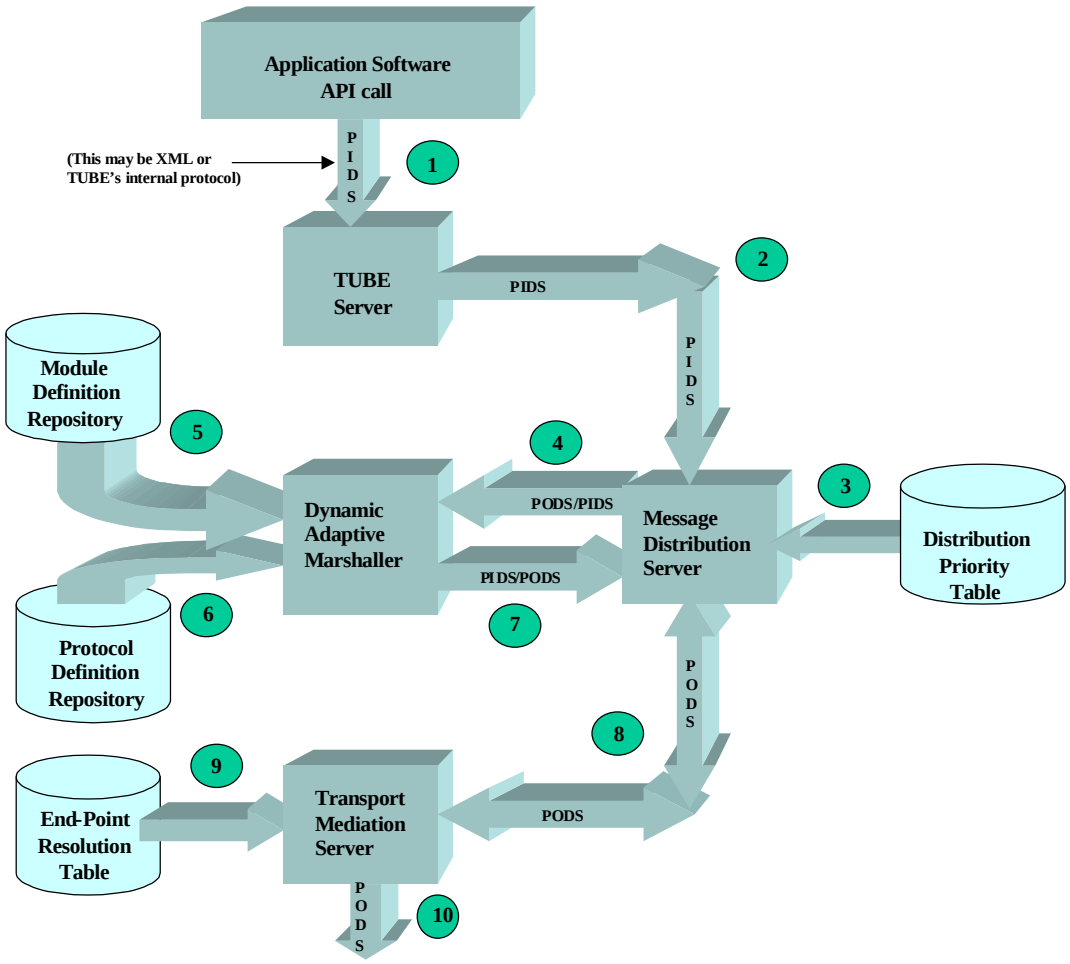

Figure 3-4: TUBE Out-bound message scenario

If communication with the preferred protocol fails, TUBE will try each subsequent protocol (in priority order). The application will only receive notification of communication failure once all the listed protocols have been exhausted. If communication succeeds, TUBE sends a positive notification to the application. The way that this occurs depends on the application's relationship with TUBE. If the application has invoked TUBE via the API, then TUBE will return the status directly to the application. If, on the other-hand, TUBE has intercepted an out-bound call made by a proxy or stub, then the status will be given to that module for return to the application.

3.7 In-bound Message Process Flow

The scenario shown in Figure 3-5 describes the process-flow of an in-bound message through TUBE.

1. The external call is intercepted by a TUBE module.
2. The interceptor uses the TUBE API to pass the message to the TUBE server, which passes the call to the Message Distribution Server.
3. The TUBE server passes the message to the Message Distribution Server in the protocol that it was received.
4. The Message Distribution Server looks-up the Distribution Priority Table to determine whether the message needs marshalling into another protocol. Steps 5,6,7 and 8 are only executed if the protocol needs to be converted by the Dynamic Adaptive Marshaller. If not then the message can be passed through to Step 9.
5. The Message Distribution Server passes the interface/module identifier and the preferred protocol to the Dynamic Adaptive Marshaller.
6. The Dynamic Adaptive Marshaller reads the Module Definition Repository to determine the format and arguments for the call.
7. The Dynamic Adaptive Marshaller uses the protocol definitions stored in the Protocol Definition Repository to prepare the call for the particular middleware or application service.
8. The Dynamic Adaptive Marshaller passes the marshalled buffer back to the Message Distribution Server.
9. The Message Distribution Server passes the (possibly converted) message to the Transport Mediation Server and tells it which protocol to use for transmission.
10. The Transport Mediation Server reads the End-Point Resolution Table to determine how to contact the end-point for this protocol. In this case, it determines that the end-point is local.
11. The Transport Mediation Server then passes the message to the “Target Application” on the local system.

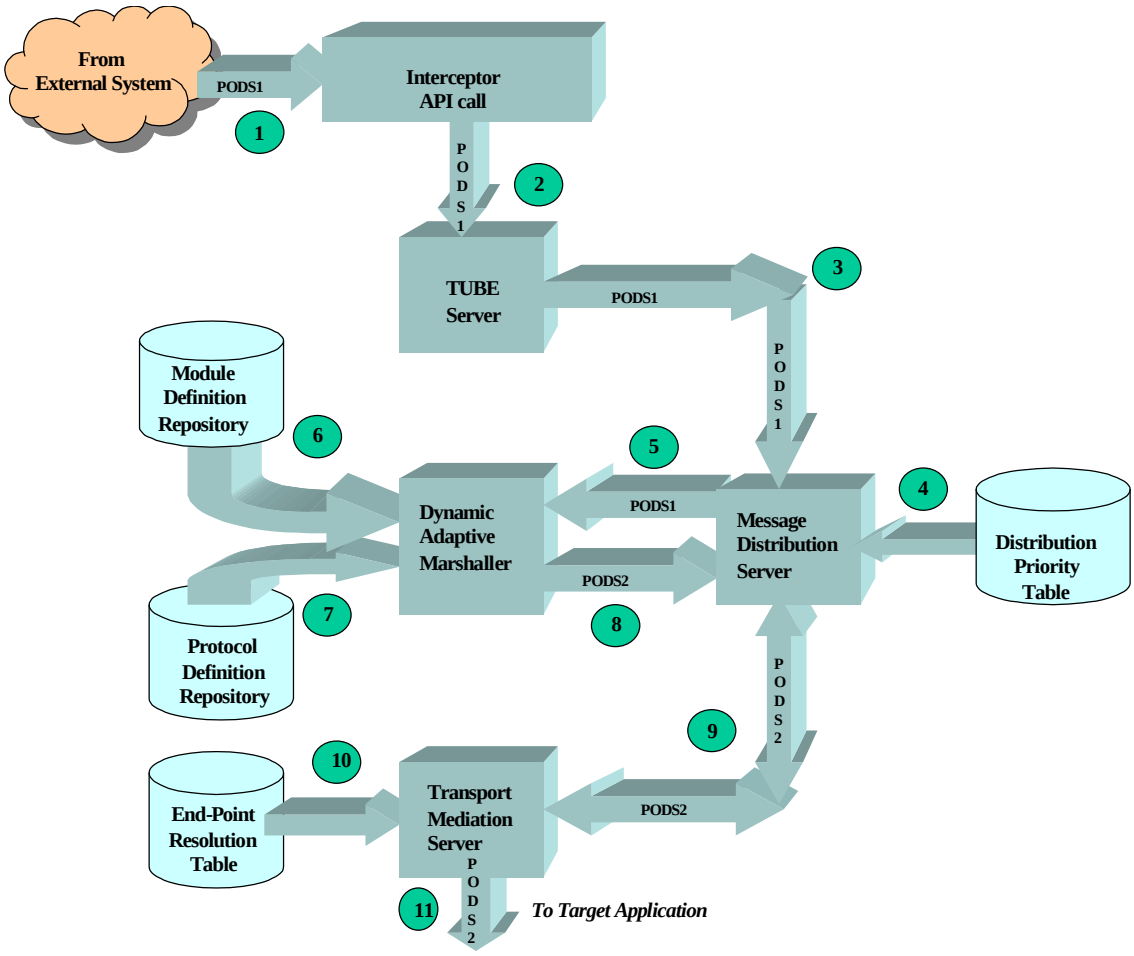

Figure 3-5: TUBE in-bound message scenario

This chapter has provided an overview of the components of TUBE to provide a context for a more detailed discussion of the key components, the MPDL, the process of message distribution, the marshalling of data, and communication management.

4 Middleware Protocol Definition Language (MPDL)

The mechanism for defining protocols should be abstracted from users (programmers). A wizard or some other abstraction can be provided as an interface. Broadly, the language will need to provide the following properties.

1. A grammar that defines the format of the message (including headers)[3].
2. A buffer format that defines the type of buffer for data transfer. For example as an un-interpreted array of bytes, an ASCII string or a CDR encoded array of bytes (CORBA IIOP).
3. A transport that defines the mechanism that moves messages between source and target end-points. These include message queues and communication sockets.
4. End-Points that define an address (in protocol-specific terms) to communicate with. Examples are ports for socket-based communication and queue names for queue-based (MOM) message exchange.

Candidate protocol definition languages include ASN.1 [Steedman, 1993], XML schema, an existing language (for example, Java), a BNF style grammar, or extensions to an existing definition language (for example, IDL). Some of these were discounted for the following reasons:

- ASN.1 was considered to be more suitable for defining applications and protocols that use the BER (Basic Encoding Rules), and not suitable for protocols based on other encoding rules (for example, CDR). The MPDL must be independent of encoding rules.

[3] ASN.1 and IDL are examples of grammar definitions.

- XML schema was seen as not flexible enough to (easily) describe some complex protocol semantics, such as variations in process based-on a field value (see Section 4.5). The MPDL must support the declaration of specialised semantics.
- Java was seen as too broad and would require the user to basically write the entire marshalling engine by hand. The MPDL must minimise the amount of code required to effect translations.
- BNF (Backus-Naur Form) was seen as too complex to be of general use. The MPDL must have an easily understandable grammar.

The selected representation for the MPDL is modelled on an already existing definition language, OMG IDL. Introducing a new language or syntax would (potentially) create a need for programmers to learn the language and (may) cause resistance to use, and the approach taken is to introduce new constructs and keywords based-on OMG IDL. This approach is attractive because:

- most programmers have some knowledge of IDL,
- there is a reduction in the learning curve when using a known idiom as a base, and
- the time to extend an existing syntax is less than would be required to define a completely new one.

Similar to the way an Interface Description Language (IDL) defines interfaces MPDL defines middleware protocol structure. The language defines the structure of both request and response messages, and all the information that needs to be defined in order to exchange messages with a server on behalf of a client. As a broker between disparate systems, TUBE will access MPDL protocol descriptions to convert from one client protocol to another to communicate with a server.

The MPDL (as stated above) is modeled on OMG IDL. A rationale behind using an existing language as a base is that most software engineers have some exposure to, or knowledge of it. This is mostly the case with IDL. It defines CORBA interfaces and is the description language for Java RMI [RMI, 2003]. OMG IDL itself is based on the original DCE RPC IDL [The Open Group, 1997]. Microsoft also has a language based-on extension to RPC IDL called MIDL (Microsoft Interface Definition Language). It primarily defines C++ COM interfaces [Box, 1998]. Using an existing language as a foundation reduces the learning curve for the users, and potentially reduces the development time for the MPDL and supporting tools.

4.1 MPDL keywords

Table 4-1 shows keywords and constructs[4] used to support protocol descriptions. Some of the keywords come directly from IDL.

Keyword	Description
protocol	Signifies the beginning of a MPDL script (protocol definition.)
request	Defines the structure of a request message.
response	Defines the structure of a response message.
%var%	Represents an internal common middleware variable. There are several of these explained in Table 4-2 below. An example is %operation%, which represents the operation or method to invoke on an object-based interface. It may be empty; its value depends on the protocol.
init	Defines a variable of the specified type with an initial value. Refer to the Variable-Definition Segment (page 41). As an example, we want to define an integer variable mynum and initialise it to the value one (1); we would write "init int mynum = 1;".
control	Specifies a field in the message to use as a switch (decision making) value. This allows us to handle different types of payload depending on the value of this field. For instance, we may receive an exception rather than the expected return value. The CORBA example below demonstrates this usage.
buffer	Signals the start of the payload (as defined in the interface) within the message. The processing follows the Module Definition Repository definition. The only exception to this is if some condition specified in a "control" clause has been met, and this specified the execution of another Code-Block.

[4] A complete formal description of MPDL syntax is given in Appendix I

Keyword	Description
var	Specifies a user-defined variable. We retrieve the values for these variables at marshal time from the End Point Resolution Table. During un-marshalling we read them from the input stream. In either case, the value is stored in the Variable-Definition Segment entry. An example of a user-defined variable is a CORBA object-key, we define it as follows: “byteSequence $objectKey$”. This means when we reach this point in the message, read a byteSequence structure and assign its value to the variable “objectKey”.
struct	We use this to define individual parts of the message, such as header, body or trailer. Each struct declaration causes the generation of a CODE_BLOCK (see Section 4.3). This allows different parts of the message to be handled out-of sequence. Where it may be necessary to re-marshal only some values. This is explained in the CORBA example.
declare	Define marshalling rules for a particular compound (complex) type.
bufferFormat	Defines how to encode/decode declared types encountered in the payload (refer to Section 4.5).
endPoint	Defines the communications end-point in protocol-specific terms.
external	Defines external classes that will provide marshalling functions for this protocol and communication management functions. If there are no external classes defined, TUBE will use its own to carry out these operations. The specified classes must implement specific interfaces. These classes may be used as wrappers around vendor-specific or home-grown APIs.
sequence	This causes the generation of a looping wrapper around the CODE_BLOCK, which marshals the defined type. This is closely associated to the %count% (internal) variable, which holds the value of the loop count. The marshaller must know from this definition, at what point and from where in the message, to read this value. In the case of marshalling, The marshaller will write this value into this point in the message. The encoding of the specified sequence then follows.
expr	Treats a string with substitutable parameters as a single expression (for example, the specification: “This” + “ + That” + “=” + “What” will produce “This+That =What”).
bin	Treats a binary string with substitutable parameters as a single expression (for example, the specification: %type% + %value% will produce 0x3”Fred”, assuming type contains 0x3 and value contains “Fred”).
bodyMember	How to process an IDL defined structure/item in the message payload. (See the HTML example in Appendix C).
codec_class	The class to call for reading and writing of native types.
pre_class	The class to call before un-marshalling a message payload.
post_class	The class to call after marshalling a message payload.
pre_method	The method to invoke on the pre_class.
post_method	The method to invoke on the post_class.

Keyword	Description
read	The position in the message where the pre_method should be called.
write	The position in the message where the post_method should be called.
complexStart	How to process the start of a complex (structured) item.
complexEnd	How to process the end of a complex (structured) item.
sequenceStart	How to process the start of a sequence.
sequenceEnd	How to process the end of a sequence.
sequenceItem	How to process an individual item within a sequence

Table 4-1: MPDL Keywords

The following table describes the common internal middleware variables that may appear in an MPDL definition. The entry referred to in the text, unless otherwise noted, is a record in the Variable-Definition Segment (see 4.3.1.2).

Variable	Description		
		Marshalling	**Un-marshalling**
endian	Defines the endian representation of the target host.	Value obtained from EPRT entry.	Value stored for reference only.
buffer_length	Specifies the overall length of the payload.	Encoded after payload.	Read before payload.
request_id	Ensures processing in correct sequence.	Read from entry and encoded.	Stored in entry.
isResponse	Determines if this is a response message.	Read from entry and encoded.	Stored in entry.
operation	Specifies the method to invoke. Only applies to protocols that support methods[5].	Read from entry and encoded or obtained from an ***XFORM*** map (see Figure 5.7).	Stored in entry.
		Marshalling	**Un-marshalling**
expect_resp	Specifies if this is a two-way invocation.	Read from entry and encoded.	Stored in entry.
num_bytes	The number of bytes in the next set of bytes.	Read from entry and used to write next block of bytes.	Stored in entry and used to read next block of bytes.
reply_status	The status of the communication session. Only applies to responses.	Read from entry and encoded. The value is protocol-specific.	Stored in entry.

[5] Protocols like CORBA, COM and SOAP support method invocations.

Variable	Description		
target_tlv	Read/Write the value from a TLV entry. We use the TLV primarily for payload processing.	Read from TLV entry and encoded.	Stored in TLV entry.
count	Internally created when we encounter “sequence” in MPDL.	Written at the start of a loop wrapper.	Read from stream at expected start of a loop.
array_size	Internally created when an item is defined as an ARRAY	Value obtained from Module Definition Repository entry.	Value obtained from Module Definition Repository entry.
sequence_size	The size (in elements) of the sequence to read/write	Read from entry and encoded.	Written from entry.

Table 4-2: Common Middleware internal variables

These are reserved words and are expected enclosed within '%' characters (for example, `%count%`).

4.2 An introduction to MPDL

The following protocol definition shows some of the basic concepts of MPDL. The protocol is a cut-down version of the W3C’s SOAP. This protocol is called SUDS (Simplified User-Defined SOAP). The basic structure of the protocol is as follows;

Envelope-Start
    Payload
Envelope-End

The SUDS protocol uses XML encoding. We will now show how this structure is represented in MPDL. Please refer to Table 4-1 for meanings of the keywords and constructs.

```
// define the STRING constants for the envelope
//start of a SUDS message
#define ENV_START “<SUDSEnvelope>”
// end of a SUDS message
#define ENV_END “</SUDSEnvelope>”

// define the class to use for marshalling
#define CODEC “TUBE.DAM.Marshall.PIM.XMLBuffer”

// now begin the actual protocol definition

protocol SUDS
{
      external
      {
         codec_class = CODEC;
      };

      // define structure of envelope start
```

```
      struct SUDSEnvelopeStart
      {
         String env = ENV_START;
      };

      // define structure of envelope end
      struct SUDSEnvelopeEnd
      {
         string enve = ENV_END;
      };

      // now define what a request and response look like
      request
      {
         SUDSEnvelopeStart ses;
      // This is the message payload (content)
         buffer req;
         SUDSEnvelopeEnd see;
      };

      response
      {
         SUDSEnvelopeStart ses;
         // This is the message payload (content)
         buffer resp;
         SUDSEnvelopeEnd see;
      };

};
```

Figure 4-1 MPDL Example

MPDL descriptions are compiled into Code-Blocks that are processed at run-time by a virtual machine. In TUBE, this virtual machine is referred to as the DAM (Dynamic Adaptive Marshaller). A Code-Block is comprised of one or more State-Blocks (see Table 4-3). A State-Block is a structure consisting of the following elements:

- Op-code
- A target variable to store the result of the operation
- One or more parameters for the operation
- Offsets into other data structures required by the operation

The operation referred to above is the operation indicated by the op-code. This is different from the invocation of an operation defined for an interface in IDL.

The Dynamic Adaptive Marshaller interprets the op-codes and executes the given instruction. Using a virtual machine allows the addition of functionality to the MPDL by expanding the range of op-codes.

The op-code is a symbolic value used to determine the operation to be carried-out. For example, the op-code READ_INT instructs the marshaller to read a signed 32-bit numeric value from the input source. Likewise, the op-code WRITE_INT instructs the marshaller to write a signed 32-bit value to the output target (see Appendix H for a table of all op-codes).

4.3 The MPDL Compiler

Although MPDL is based-on IDL, MPDL scripts are not compatible with IDL and therefore standard IDL compilers cannot process them, as they would not recognise the extensions. A special compiler processes MPDL definitions (also referred to as a script or protocol definition). The MPDL compiler reads the protocol definition and generates (by default) two types of output, a Protocol Implementation Module (PIM) and a Transport Interface Module (TIM). This chapter focuses on Protocol Implementation Modules. Transport Interface Modules are discussed in Section 5.7. Both of these modules are generated when the compiler is first run against an MPDL file. Subsequently, the compiler will only generate the Protocol Implementation Module unless instructed otherwise (see below). This is done to protect code that a user may have modified in a generated module. The compiler also has the ability to generate two additional modules, a Protocol Control Module and an interceptor. These modules are optional and are only generated if the required option is provided on the command-line. If no options are provided on the command-line, the compiler will only generate a Protocol Implementation Module. The compiler's available options are:

- -p generate a Protocol Control Module template
- -i generate a protocol interceptor module

- -t generate a Transport Interface Module

To avoid re-generation of a module and risk losing changes, the user omits the option for the module they wish to protect. For example to regenerate all but the Protocol Control Module, the command-line is `mpdlc –t –i protocol.mpdl` and to generate only the Protocol Implementation Module, the user types: `mpdlc protocol.mpdl`. The only module protected from overwriting by default is the Transport Interface Module. The only time it will be re-generated is if the –t option is specified.

4.3.1 Protocol Implementation Modules (PIMs)

There are two Protocol Implementation Modules generated for each protocol definition, one for handling requests and the other for handling responses. This simplifies the logic required in both the compiler's code generator and the runtime interpreter. The Dynamic Adaptive Marshaller loads the appropriate Protocol Implementation Module based-on the current message type (that is, request or response). The Protocol Implementation Module is comprised of Code-Blocks, derived from constructs within the protocol definition. For example, for each "struct" keyword encountered in the protocol definition, the compiler generates what we refer to as a Code-Block. As discussed, this Code-Block consists of a series of State-Blocks, where a State-Block (Table 4-3) consists of op-codes and state definitions, which define operations, variables (internal and user-defined) and initial values. Each op-code and state is (generally) associated with a source or target variable.

Type	Name	Description
Integer	op_code	Specifies the action to perform.
Integer	target_var	Target variable indicates the variable to use as the source or target for this operation.
State param	state_param	A state parameter entry (Table 4-4) for this op-code.
Integer	handler_offset	Offset to "declared" type handler map.
Integer	handler_name	Offset of handler name in Constant Segment.

Table 4-3: Structure of a State-Block

Type	Name	Description
Integer	Type	Type of this variable.
Integer	Use	Usage of this variable. Add or subtract from another value or use value as-is.
Integer	value_offset	Offset to constant value of this variable in Constant Segment.

Table 4-4: State Parameter entry

The following diagram illustrates the structure of a Protocol Implementation Module.

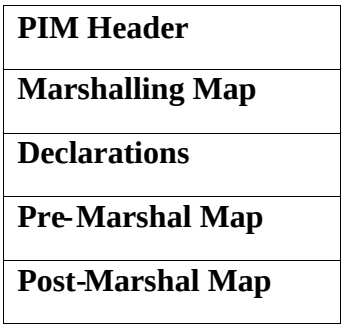

Figure 4-2: Structure of a Protocol Implementation Module

4.3.1.1 The PIM Header

The PIM header contains information and structures that assist in the loading and processing of the rest of the file. The header is comprised of the following fields.

Field	Description
File-identifier	This identifies this file as a valid Protocol Implementation Module
Marshalling class-names	The names of classes specified in an external clause. This can be empty.
Constant-Segment (CS)	Contains all constant values.
Variable-Definition Segment (VDS)	Contains information about all variables defined in the protocol definition.

Table 4-5: Structure of a PIM Header

The File-Identifier is a hexadecimal value that identifies this file as a valid TUBE Protocol Implementation Module.

The Marshalling class-names array specifies the names of the class that implements the TUBE.commsBuffer interface and any other classes specified using the “external” keyword

(see Table 4-1). These are the classes that will be used for all reading and writing operations whilst processing this Protocol Implementation Module. The actual disk layout of these items is an integer specifying the length of the array. Each entry in the array begins with an integer, which indicates the length of the name string, followed by the string. This string contains the actual name. A length of zero for the array signifies no class-names. In this case, the Dynamic Adaptive Marshaller will use default (internal TUBE) implementations for encoding and decoding of native values.

4.3.1.2 Constant Segment and Variable Definition Segment

The Constant-Segment stores the type, the length of the value and the actual value of all constant values used in the protocol definition. The MPDL compiler encodes offsets into this segment into instructions that require access to these values. Not all instructions access the Constant Segment. Table 4-6 shows the composition of a Constant Segment entry.

Type	Name	Description
Integer	Type	Type of constant.
Integer	length	Length of constant value.
Byte[]	Name	Name of constant.
Byte[]	Value	Constant value.

Table 4-6: Format of Constant Segment Entry

The Variable-Definition Segment (shown in Table 4-7) contains information about all the variables defined in the protocol definition. It stores the name, data type and a flag to define the variable as an internal or user-defined variable. If the variable has an initial value specified by an “init” clause (see Table 4-1), the compiler creates a new entry for the value in the Constant Segment.

Type	Name	Description
Integer	Flag	Flag to indicate if this is a system or user-defined variable.
Integer	Type	Type of this variable.
Integer	name_offset	Offset to name of this variable in Constant Segment.

Type	Name	Description
Integer	var_id	Symbolic identifier for this variable. This is –1 for a user-defined variable.

Table 4-7: Format of Variable-Definition Segment Entry

4.3.1.3 Marshalling Maps

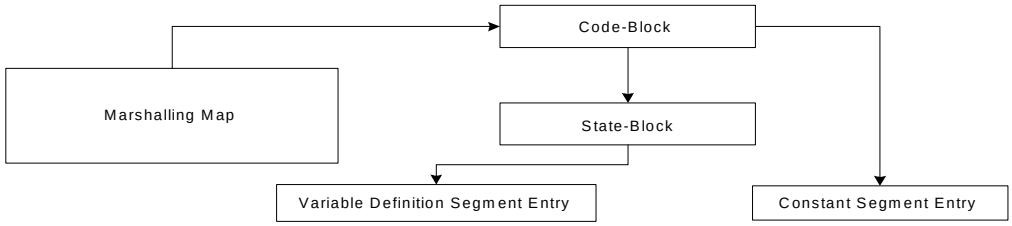

Figure 4-3: Structure of a Marshalling Map

Marshalling maps are collections of Code-Blocks. A Pre-Marshalling map contains the instructions to be carried-out before processing the message body (payload). The Marshalling Map has instructions for processing the body, and finally, the Post-Marshalling map provides the instructions to follow after the body has been marshalled. The Marshalling map, Pre-Marshal Map and Post-Marshal Maps all have the same basic structure (illustrated in 4.2 above) and consist of a number of op-codes and other information needed to carry out the specific task. These blocks contain all the necessary information for the execution of the operation. The Declarations section of the file contains pointers into these maps for Code-Blocks generated from “declare” (see Table 4-1) statements. These blocks contain all the code required to handle the declared type.

When the compiler encounters a simple (native) type in a struct definition, if it specifies an initial value, the compiler generates an entry in the Constant Segment and stores an offset to this value in a state-parameter entry (see Table 4-4). The compiler adds the entry to the Code-Block it is currently generating. If the variable does not have an initial value, the compiler generates a Variable Definition Segment definition as an empty slot for the value. This slot is

a placeholder for the value when it is read-in. It is also the source for the value when writing. Refer to Table 4-8 for a description of the runtime usage of this entry.

In the case of compound (declared) types, the compiler generates references to two separate Code-Blocks, one for reading and one for writing. These Code-Blocks have a type of USER-DEFINED and have an entry created in the Declarations section using the name of the structure with either a "_READ" or "_WRITE" appended. This modified name is stored in the Constant Segment and the Constant Segment index is stored in the definition entry. The MPDL compiler patches offsets to the actual Code-Blocks once it has completely processed the MPDL script. The instructions to handle the declared type are generated into the Marshalling map. The first instruction-block for handling this type contains a pointer to the modified name in the Constant Segment. This is how the compiler finds the value to patch into the declaration entry. This is also, how the Dynamic Adaptive Marshaller identifies and loads individual Code-Blocks at runtime.

The Constant-Segment is written to disk in its entirety. It is read-only at runtime. These values never change during the execution of the Protocol Implementation Module.

The compiler writes the Variable-Definition Segment to disk in the format shown above (Table 4-7). This is what the Dynamic Adaptive Marshaller reads when loading the Protocol Implementation Module. At runtime, another structure for storage of variable values is created for efficiency purposes. This runtime-only structure is called the Variable Value Table. The layout of the Variable Value Table appears below in Table 4-8.

Type	**Name**	**Description**
Integer	flag	Flag to indicate if this is a system or user-defined variable.
Integer	type	Type of this variable.
Integer	name_offset	Offset to name of this variable in Constant Segment.

Type	Name	Description
Object	var_value	The current value of this variable. This may be initially empty until the value is read.

Table 4-8: In-memory layout of Variable Value Table

The Variable Value Table stores the values for variables as they are read from the input source. If this value is being marshalled, then this entry is used as the source and the current value is written to the output target using either, user-supplied methods or internal (default) handlers. The native type is extracted from the Object wrapper for writing and educed from the native value into the wrapper when reading. This casting of native types to and from objects adds some processing overhead; however all data types can be handled in the same manner.

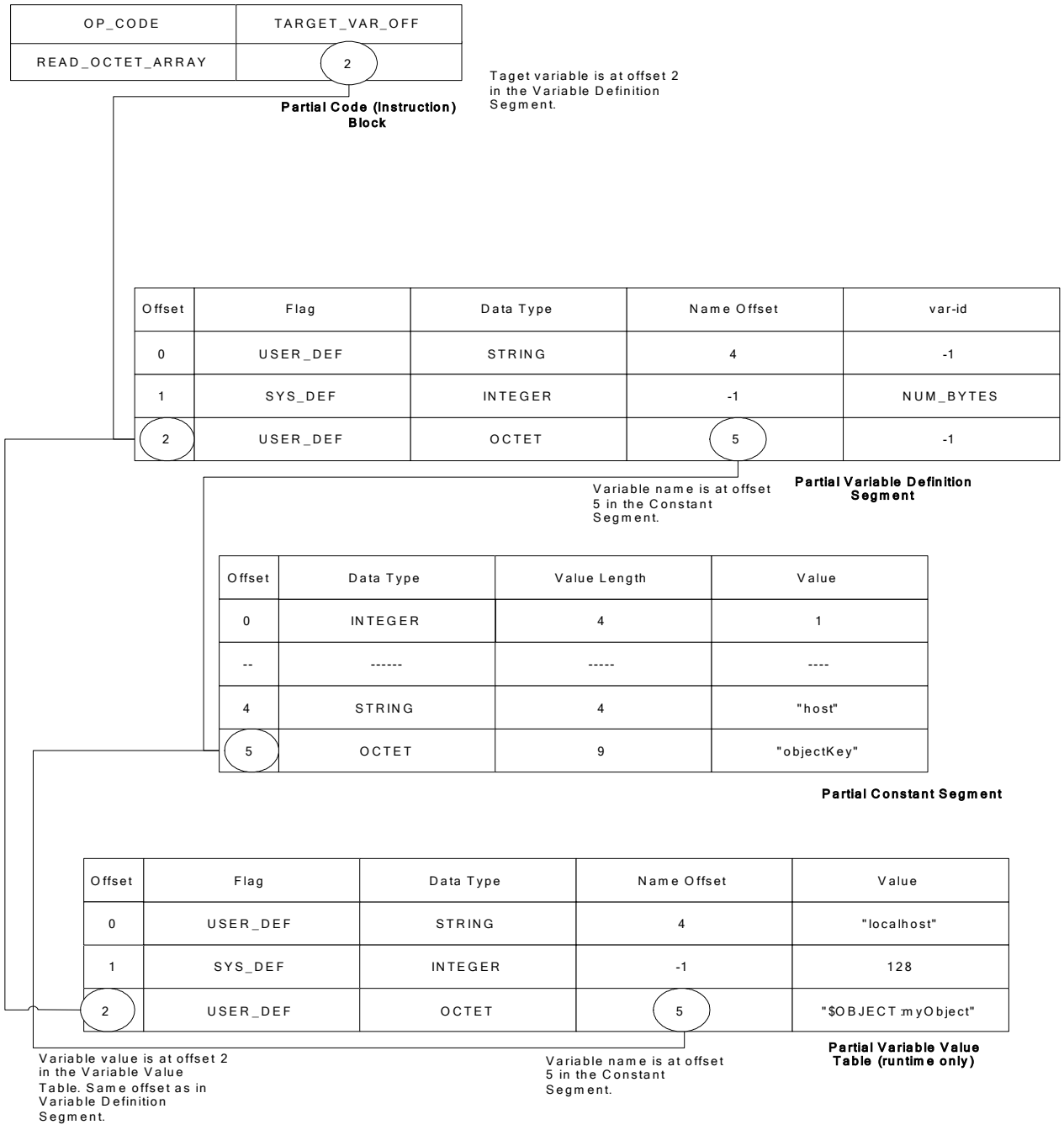

Figure 4-4: Mapping op-code target to variable value

Figure 4-4 above illustrates a read-octet operation for a target variable that has an offset of two (2) in the Variable-Definition Segment. By following this offset, the Variable-Definition Segment entry stores an offset of five (5) into the Constant Segment. This is where the name of the variable “objectKey” is found. Because this is a USER-DEFINED variable (indicated by the declaration “$objectKey$” in the MPDL script in Appendix A), initially this value is obtained from the End Point Resolution Table entry for this interface. This entry then remains constant for the life of the Protocol Implementation Module, unless explicitly changed by

invoking a set method or executing a Code-Block. When the value is read, it will be stored in offset two (2) of the Variable Value Table. This table is created only at runtime to manage the storage of actual values that are not constants. After the read, the entry at offset two (2) contains the value “$OBJECT:myObject”. When this is marshalled in a request, its value comes from this Variable Value Table entry.

4.4 The CORBA Example

An MPDL definition of CORBA using IIOP V1.0 is shown in Appendix A. Each construct and data member will be examined in detail showing how the PDL compiler processes them. Symbolic names will used to represent op-code and offset values. The actual numeric values are not relevant to our discussion, and symbolic names are easier to understand. It is also assumed throughout the discussion that the compiler has built a symbol table and other internal structures during the parsing phase. The discussion will concentrate on the code generated from these constructs, rather than their actual construction process. Most of the examples show the instructions generated for reading. It should be noted that for each set of read instructions generated, there is also a corresponding set of write instructions emitted.

The first entires in the file are any constant definitions. They are defined using the keyword #define. This is the same mechanism as used in C/C++ and OMG IDL. The compiler uses a pre-processing phase to replace any references to these definitions with their literal value.

The definitions in the example CORBA script are:

```
#define GIOPID “GIOP”,
#define HOST “host”, and
#define PORT “port”.
```

The values for HOST and PORT are used later in the endpoint definition clause.

The compiler now parses the pre-processed file and the first keyword encountered is “protocol” followed by the value “CORBA”.

```
protocol CORBA
{
```

This tells the compiler to generate the following three (3) filenames:

- CORBA_Req.PIM – defines rules for marshalling requests, and
- CORBA_Resp.PIM – defines rule for marshalling responses.
- CORBA_TIM – defines the communication parameters (explained later).

The “{“ character identifies this as the opening of the protocol definition script.

Next, follow three “typedef” statements. These behave the same way in MPDL as in standard IDL and programming languages such as C and C++, in that, they define an alias for the type. For instance, the following statement

```
“typedef sequence<octet, 3> reserved;”
```

causes the compiler to create a variable named “reserved” and whenever it encounters this variable to point to a Code-Block. The Code-Block will define op-codes for reading and writing a sequence of three octets. The definition for GIOP_MAGIC is very similar.

```
“typedef sequence<octet, 4> GIOP_MAGIC;”
```

The definition for “olist” specifies an octet sequence of unbounded length.

```
“typedef sequence<octet> olist;”
```

The important distinction to note here is that sequences of native items (such as octets) defined with “typedef” do not have their length encoded, and neither do we expect to read the

length during decoding. If the length is required when reading or writing, it must be defined using a “declare” clause (see byteSequence below) as explained next.

4.5 The bufferFormat Clause

Before the “declare” clause is described, the “bufferFormat” construct and how the Dynamic Adaptive Marshaller uses it in conjunction with the Module Definition Repository at runtime must be explained. The “bufferFormat” definition tells the Dynamic Adaptive Marshaller which Code-Blocks to use when marshalling the message, especially the payload. The payload can be made-up of either native types or constructed complex types. The complex types may contain native types and other complex types. The Dynamic Adaptive Marshaller must be provided with the marshalling instructions for the following standard constructed types:

- STRING – how to marshal a String
- BYTESEQ – marshal an arbitrary byte sequence
- ARRAY – marshal an array (fixed-size sequence) of native or complex types
- SEQUENCE – marshal a variable-length sequence of native or complex types
- OBJECTDEF – marshal an object definition (opaque object reference)

The Dynamic Adaptive Marshaller assumes that a message may only be comprised of a combination of those items and native types. If these instructions are not provided in the MPDL, there will be no handlers (Code-Blocks) generated, as there will be no “declare” clauses to define them. In this case, the Dynamic Adaptive Marshaller will use internal marshalling rules that may or may not be suitable for the particular protocol. For example, an object definition is protocol specific. If no instruction is given, the Dynamic Adaptive Marshaller will simply encode and decode an item defined (in the Module Definition Repository as an object), as an un-interpreted array of bytes. With reference to the MPDL

definition for an “objectDef” in the CORBA example, it can be seen that if the “declare” and “bufferFormat” statements were omitted, the default behaviour would not be suitable[6]. If the MPDL compiler encounters multiple bufferFormat statements, it throws an exception and terminates processing.

The next keyword encountered is “declare”. This is used for defining compound or complex types, which may be composed of many native and or other, compound types. Following is the definition of “byteSequence”.

```
declare byteSequence
{
 int %num_bytes%; // no of bytes in olist
 olist bytes;
};
```

Figure 4-5: Declaration of a byteSequence

This will generate op-codes that tell the Dynamic Adaptive Marshaller how to read and write an arbitrary sequence of bytes. A reference to an internal common middleware variable “`%num_bytes%`” (see Table 4-2) is defined. This indicates how many bytes (octets) to read or write next. This is followed by a reference to an “olist”, and the end of the declare clause, signified by “`};`”. The compiler will now create a Code-Block named “byteSequence_READ” with the following op-codes:

OP-Code	**Source / Target variable**	**Comment**
READ_INT		Read an integer from input
PUSH	Put on top of value stack	
POP	Get value on top of value stack	
ASSIGN_TO	NUM_BYTES	Assign value to internal NUM_BYTES
READ_OCTETARRAY		Read NUM_BYTES octets
PUSH		Put octet array on top of value stack. Caller will POP and retrieve value.
END_BLOCK		End of this Code-Block

Table 4-9: op-codes generated for reading a byteSequence

[6] For example, strings would not be null terminated.

From this point on wherever a reference to “byteSequence” read appears, the compiler will encode an instruction to load this Code-Block and execute it. Any other Code-Blocks that refer to this Code-Block will have a flag set that specifies a reference to a “USER-DEFIND” Code-Block. The instruction (in the referring block) will also have an offset (in the Constant Segment) to the name of this block. It must be noted that the compiler also creates a similar block containing write instructions.

The next declaration encountered is for an array. This entry specifies how the Dynamic Adaptive Marshaller should handle arrays. This is very similar to the byteSequence example, except that another special variable “array_size” is used to keep track of the number of actual entries. An array is a fixed-size sequence. The interpreter derives the upper-limit of the array dimension at runtime by referencing the Module Definition Repository entry for the particular interface being marshalled. Currently MPDL supports only single dimension arrays.

```
declare array
{
 int %array_size%;      // no of items in sequence/array
 olist bytes; // actual sequence of items (can be simple or complex)
};
```

Figure 4-6: Declaration for an array

The declaration for “nString” demonstrates the use of op-codes to add and subtract constant values to and from those currently being processed. The “`+ 1`” tells the compiler that there will always be one extra byte over the actual string length. Here the length of the string including the null byte is read, and then one subtracted from it. This is so the null is not consumed as part of the string. It is read separately and discarded. Conversely, when writing the string, first, the length is increased by one (1) and written. The string itself is written, followed by the null byte.

```
declare nString
{
 int %num_bytes% + 1;         // length of string_bytes (incl. null)
 olist string_bytes;          // the actual bytes of the string
 init octet null_byte = 0;    // the terminating null
};
```

Figure 4-7: Declaration of a null terminated string

The compiler generates the following Code-Block.

OP-Code	Source / Target variable	Comment
READ_INT		Read an integer from input
SUB	Subtract a value from the offset into the Constant Segment from the value just read	We have an offset to the value "1" in the Constant Segment.
PUSH	Put on top of value stack	We now have value - 1
USER_DEFINED		A declared code block
POP	Get value on top of value stack	
ASSIGN	NUM_BYTES	Assign value to NUM_BYTES
READ_OCTETARRAY		Read NUM_BYTES octets
PUSH		Put octet array on top of value stack. Caller will POP and retrieve value.
READ_OCTET		Read null byte
END_BLOCK		End of this Code-Block

Table 4-10: Op-codes for reading a null terminated string

The following "objectDef" declaration illustrates the usage of declared types within declared types.

```
declare objectDef
{
  nString repo_id;            // repository-id of object
  int profile_count;          // number of profiles in reference
  int profile_id;             // id of profile
  int length;                 // length of following stream
  short     version;          // IIOP version for this profile
  nString host;               // Host for this object
  short     port;             // Port for this object
  byteSequence object_key;    // Object key of target
};
```

Figure 4-8: declaration of an object reference

The table below illustrates the resultant Code-Block.

OP-Code	Source / Target variable	Comment
USER_DEFINED		A declared Code-Block
LOAD_BLOCK	repo_id	Load and execute the block named “nString_READ” and place the value in the variable “repo_id”
READ_INT	profile_count	Read an integer and assign it to the variable “profile_count”
READ_INT	profile_id	Read an integer and assign it to the variable “profile_id”
READ_INT	Length	Read an integer and assign it to the variable “length”
READ_SHORT	Version	Read a short and assign it to the variable “version”
LOAD_BLOCK	Host	Load and execute the block named “nString_READ” and place the value in the variable “host”
READ_SHORT	Port	Read a short and assign it to the variable “port”
LOAD_BLOCK	object_key	Load and execute the block named “byteSequence_READ” and place the value in the variable “object_key”
END_BLOCK		End of this Code-Block

Table 4-11: Op-codes for reading an object reference

The interpreter executes the instructions above whenever an “object” definition is encountered in the payload and the value being marshalled is defined as an “object” type in the Module Definition Repository. The statement “OBJECT=objectDef;” in the bufferFormat clause defines this association. The “bufferFormat” clause is the next construct encountered by the compiler.

The compiler next sees the “control” statement. The compiler writes the op-codes generated here (shown later) into the Pre-Marshal Map. These are loaded and executed just before marshalling the payload. When the interpreter encounters the special offset value START_PAYLOAD[7], it will search for a Pre-Marshal Map. If none is found, then the Dynamic Adaptive Marshaller will traverse the payload according to the Module Definition

[7] This offset indicates the position within the map where the payload is expected.

Repository definition for the interface being processed. Otherwise, if there is a map present, the Dynamic Adaptive Marshaller invokes an interpreter module to handle the specified tests.

```
control
{
    switch(%reply_status%)
   {
     case 0:
      buffer = body;                // follow MDR
     case 1:
      buffer = USER_EXCEPTION;      // follow Exception in MDR
     case 2:
      buffer = systemException;     // use declared structure
     case 3:
      buffer = objectDef;           // use declared structure
   }
};
```

Figure 4-9: The control clause

The statement above tells the compiler to generate some branching op-codes based-on the value of the internal variable reply_status. When the value of reply_status is read from the input at runtime it is examined and tested for the values: zero(0), one(1), two(2) and three(3). The value determines what action to take for encoding or decoding the payload. After executing the appropriate action, we exit this module and return to the main interpreter code. According to the rules specified above, the following process is executed:

- If the value is zero (0), ***false*** is pushed onto the stack. This indicates that the Module Definition Repository definition will be followed for the interface and the values are marshalled accordingly. In this case, the module returns a Boolean ***false***.

- If it is not zero (0) a test for one (1) is performed, and if this is true, the definition of the exception for this operation (as defined) in the Module Definition Repository is followed. Unlike the case for zero above where ***false*** is returned, for Module Definition Repository-defined exceptions ***true*** is returned to indicate that it is not the

standard payload; although, a Module Definition Repository definition is still being followed.

- Otherwise, the value is tested for two (2), and if this is true, the name of the Code-Block defined as “systemException_READ” is loaded and returned.

- Finally, the value is tested for three (3). If this is true, the name of the “objectDef_READ” Code-Block is loaded and returned. If none of the defined values exists, the Dynamic Adaptive Marshaller throws a marshalling exception. Table 4-12 shows the generated code.

OP-Code	**Source / Target variable**	**Comment**
EQ	REPLY_STATUS	Test if reply_status == 0
JUMP	LABEL_0	The test returns true. Jump to the given label.
EQ	REPLY_STATUS	Test if reply_status == 1
JUMP	LABEL_1	The test returns true. Jump to the given label.
EQ	REPLY_STATUS	Test if reply_status == 2
JUMP	LABEL_2	The test returns true. Jump to the given label.
EQ	REPLY_STATUS	Test if reply_status == 3
JUMP	LABEL_3	The test returns true. Jump to the given label.
PUSH	Exception	All tests failed. Push an Exception value onto the stack. This causes the interpreter to throw an exception.
JUMP	LABEL_4	Jump to the last label.
LABEL_0		
PUSH	False	Push a false value onto the stack. This is the return value. Therefore, we follow the Module Definition Repository.
JUMP	LABEL_4	Jump to the last label.
LABEL_1		This is a Module Definition Repository-defined Exception.
PUSH	True	Return true

OP-Code	Source / Target variable	Comment
LABEL_2		
PUSH	“systemException_READ”	Push the name of the block to decode a system exception.
JUMP	LABEL_4	Jump to the last label.
LABEL_3		
PUSH	“objectDef_READ”	Push the name of the block to decode an object definition.
JUMP	LABEL_4	Jump to the last label.
LABEL_4		
		Return value on top of stack.

Table 4-12: Op-codes for processing "control" clause

In summary, the module that performs the “control” instructions returns one of three values to the main interpreter. It returns ***false*** if the payload is marshalled by following the Module Definition Repository representation, or it returns ***true*** if the payload is a USER_EXCEPTION. A string value indicates that this module has pushed the name of a USER_DEFINED Code-Block (that was defined with the declare clause) onto the stack. The main interpreter loop will load and execute this Code-Block. After marshalling the payload, the interpreter will search for a Post-Marshal Map.

Unlike the control clause for Pre-Marshal Maps, there is no keyword to indicate the start of a post-marshalling map[8]. The compiler will always generate code to write-out the body (payload) length after marshalling the payload. Therefore wherever the variable “%buffer_length%” is encountered this tells the compiler that this is the payload length. Initially the length is marshalled as zero (0) and then re-written with the correct value after marshalling the body.

```
// Response message
struct GIOPRespMessage
{
 GIOPHdr hdr;
 int  %buffer_length%;  // the length of the following buffer (body)
 GIOPRespBody body;
};
```

Figure 4-10: Response message declaration showing buffer_length variable

[8] One may be introduced if it is deemed to add flexibility to PDL.

Statements that contain the buffer_length variable, such as the one above, automatically cause the compiler to create a Post-Marshalling Map. This map contains instructions to save the current point in the buffer, calculate the new position, write the length and return to the current position.

OP-Code	Source / Target variable	Comment
SAVE_POS	BUFFER_POS	Save current buffer position.
SET_POS	BUFFER_POS (POS=8)	Set the buffer position to the value of the constant at the offset given by the parameter.
SUB	A constant value of "12" from the buffer length. Header length (8) + length of integer (4) = 12.	Subtract the length of the header + the length of the integer from the buffer_length to give only payload length.
WRITE_INT	BUFFER_LENGTH	Write-out the value of the internal variable buffer_length
SET_POS	BUFFER_POS	Set the buffer position to the saved value.

Table 4-13: Post-Marshal Map for CORBA message

Next, the compiler encounters the "external" clause. This defines the full Java class-names (including packages) of the classes that the interpreter is to call for marshalling native types. Because CORBA uses CDR encoding for primitive data, the default TUBE marshaller is not suitable. Therefore, we define a special class to handle the CDR padding of the bytes that the Protocol Implementation Module reads or writes. This class needs to be defined only once in the MPDL. From then on, it will be available for marshalling any defined interface across this protocol, or it can be re-used with other protocols. For example, when the Protocol Implementation Module contains a READ_INT op-code, the Dynamic Adaptive Marshaller will call MYORB.marshaller.CDRBuffer.read_int() to obtain the value. Conversely, when WRITE_INT is encountered, MYORB.Marshaller.CDRBuffer.write_int(value) is called to output the value. This clause causes the compiler to populate the Marshalling class-name member of the PIM header (see Table 4-5).

The final construct in this example is the “endPoint” definition. It appears in MPDL as follows:

```
endPoint:"TCP"
{
//
// These are transport and protocol-specific items
//
string host = HOST;     // This is the host for the object
string port = PORT;     // This is the port on the host
};
```

Figure 4-11: MPDL endpoint definition for CORBA

The value following the “:” identifies the transport for the protocol, in this case “TCP” for IIOP. The Transport Mediation Server must find these values in the End Point Resolution Table entry for this interface. The compiler generates code into the Transport Interface Module for loading and using these values. As the definition for this endpoint defines the use of TCP/IP, the Transport Interface Module will use these values to create a sockets-based connection to the defined host on the designated port. The operation of Transport Interface Modules is covered in more detail in the section on the Transport Mediation Server (Section 5.7).

The next section will continue with the CORBA example and show how parts of the message may be re-marshalled.

5 Message Processing and Data Marshalling

5.1 The Message Distribution Server (MDS)

The Message Distribution Server is the component in the TUBE architecture that is responsible for managing the messaging life cycle and ensuring that clients either, receive the response in synchronous mode or are notified of responses in the asynchronous mode. The Message Distribution Server is the first and last module to handle a message and its subsequent response (assuming a two-way exchange). The Message Distribution Server is also responsible for determining the target end-point from the Distribution Priority Table and End Point Resolution Table, and providing that information to the other modules via an API. When clients elect to use the TUBE server directly via APIs, the TUBE server creates an instance of the Message Distribution Server to handle the message. Similarly, when a protocol interceptor intercepts a message, it uses an instance of the Message Distribution Server to manage the session.

In the following discussion, the assumption is that the system is processing a synchronous (two-way) message.

When a protocol listener intercepts a client request, the following sequence of events occurs:

- The listener creates an instance of the Message Distribution Server and passes the message to it.
- The Message Distribution Server passes the request to the Dynamic Adaptive Marshaller and waits for it to return a protocol-neutral representation of the request (a TLV buffer.).
- The Message Distribution Server now looks-up the Distribution Priority Table to ascertain the target protocol.

- The Message Distribution Server passes the TLV buffer back to the Dynamic Adaptive Marshaller for marshalling into the target protocol.
- After the Dynamic Adaptive Marshaller returns the marshalled request, the Message Distribution Server attempts to create an instance of a Protocol Control Module (discussed below) using the Java Reflection API. Depending on the result of the creation, the Message Distribution Server does one of two things:
    - If the creation is successful, the original request is passed to the Protocol Control Module for handling. The PCM is responsible for all further processing.
    - If the creation is unsuccessful, the Message Distribution Server passes the message to the Transport Mediation Server for transmission to the target end-point.
- The Message Distribution Server now waits for either the Transport Mediation Server or the Protocol Control Module to return the response.
- When the Message Distribution Server receives the response, it carries out the reverse of the above procedure; it uses the Dynamic Adaptive Marshaller to convert the response from the target protocol into the source protocol.

In the event that the transaction fails, the Message Distribution Server will attempt to send the message using the next protocol from the Distribution Priority Table. If the transaction cannot be satisfied, the Message Distribution Server returns a null value to indicate failure.

5.2 Protocol Control Modules (PCMs)

These modules provide higher-level protocol semantics than those required for marshalling. It is software written by a user. As an example, consider the CORBA MPDL definition (see Appendix A). In this script, there is a “control” clause, which is a switch statement that controls what sort of message payload the system is dealing with. The decision as to what to do with this payload after marshalling and return belongs to the Protocol Control Module. The Protocol Control Module implements the same switch logic as that specified in the control clause, with the addition of appropriate logic to handle the resultant payload. To help clarify, the following is another example based on CORBA MPDL using a response message. The Protocol Control Module must decide what to do with this response based on the value of the reply_status field of the message.

One of the values specified for reply_status in the control clause is a three (3), which signifies that the response payload is a CORBA object-reference (defined as objectDef in MPDL). To a CORBA client or server the value of three (3) actually means more than the type of response payload; it means that the response is a LOCATION-FORWARD response. This indicates that the original request has to be re-marshalled and submitted to the object whose reference is contained in the response message.

An attempt to support the specification of this logic in MPDL could result in an overly complex language; thus, these higher-level semantics are delegated to a user-supplied module. The MPDL still provides support for the marshalling of the various payload types, without however attempting to interpret their meaning. That is, the decision whether or not to re-submit the request to the new object is left to the Protocol Control Module. The Dynamic Adaptive Marshaller and TLV buffer APIs provide methods for retrieval and population of various fields within the message by name. Therefore, the Protocol Control Module makes a

request to the Dynamic Adaptive Marshaller to re-marshal the request using the new object-reference received in the response.

Only one Protocol Control Module is required for a given protocol, and this can manage any message for any defined interface handled by this protocol. Using a CORBA LOCATION-FORWARD response message as an example, the Protocol Control Module performs the following steps (illustrated in Figure 5-1):

1. Receive the original request from the Message Distribution Server
2. Send the message using the Transport Mediation Server and wait for the response.
3. Make a decision of what to do based-on the reply_status in the response
4. If the Protocol Control Module decides to re-submit the request
5. Use the Dynamic Adaptive Marshaller, TLV buffer and Transport Mediation Server APIs to set appropriate fields in the request with new values
6. Submit the new request via the Transport Mediation Server.
7. Return the response to the Message Distribution Server. If this response is null, the Message Distribution Server will attempt to send the message using the next protocol listed in the Distribution Priority Table, otherwise return the (non-null) response to the caller.

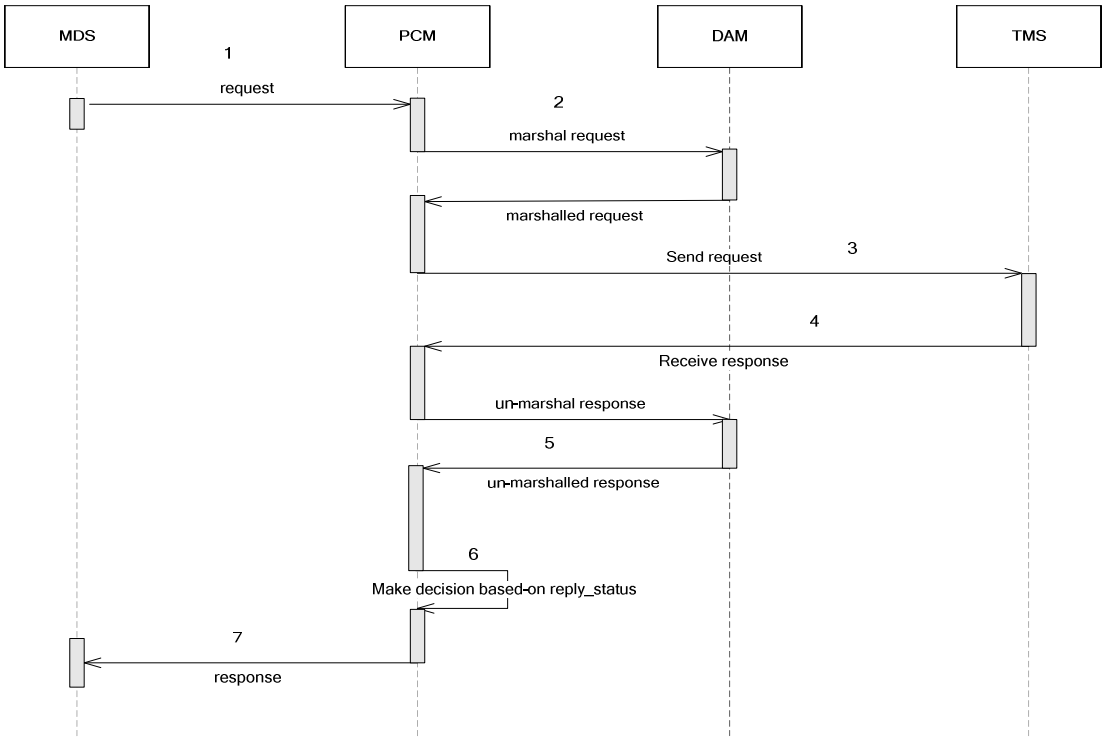

Figure 5-1: Message processing sequence with a PCM

The following is an example of Protocol Control Module logic to handle CORBA responses.

```
// This is an internal method to the PCM
// assume this module has received the variable instances as parameters
// from the handleMessage method that was called by MDS
// _tms is an instance of the Transport Mediation Server
// _dam is an instance of the Dynamic Adaptive Marshaller
// req_buff has come through from MDS and is in TLV format
// as TLVBuffer req_buff

private TLVBuffer makeDecision(DAM dam, TMS tms, TLVBuffer req_buff)
{
      Integer reply_status = null;

      reply_status = (Integer)_dam.getValue("reply_status"); // get
      reply_status
      // check reply_status value
      if (reply_status == null)
        return null;    // let MDS handle the error condition

      switch(reply_status.intValue())
      {
            case 0:
               // normal message response
               return null;         // return original response to MDS
            case 1:
               // user-defined EXCEPTION
               return null;         // return original response to MDS

            case 2:
               // SYSTEM_EXCEPTION
               return null;         // return original response to MDS

            case 3:
               {
               // LOCATION-FORWARD
```

```
                // get values required for creating new request
                String host = _dam.getValue("host");
                String port = _dam.getValue("port");
                String repo_id = _dam.getValue("repo_id");
                byte[] objectKey = _dam.getValue("objectKey");
                // set the new values into the request, all others are same
                req_buff.setValue("object_Key", objectKey);
                req_buff.setValue("repoId", repo_id);
                // set new values for end-point
                _tms.setValue("host", host);
                _tms.setValue("port", port);
                //
                // return the new request to send
                return req_buff;
                }
        }

}       // end of method
```

Figure 5-2: Partial code of CORBA Protocol Control Module

The Message Distribution Server does not attempt to interpret any of the messages, but simply routes the messages to the other components.

The basic operation of the Message Distribution Server is as follows:

1. Receive an in-coming message either from an interceptor or via the server API.
2. Invoke the Dynamic Adaptive Marshaller to un-marshal the source message into protocol-neutral (TLV) format.
3. Determine the target protocol and end-point from the Distribution Priority Table and End Point Resolution Table.
4. Call the Dynamic Adaptive Marshaller to marshal from the TLV format into the target format.
5. Invoke the Transport Mediation Server or a Protocol Control Module to perform the actual communication and await the response.
6. Use the Dynamic adaptive Marshaller to marshal the response into the source protocol format. The response is then returned to the interceptor, which returns the response to the client.

A major design goal of the Message Distribution Server, and for that matter all of TUBE, is to be protocol-neutral. The only parts that are (intentionally) protocol-specific are the Protocol Implementation Modules generated from the MPDL scripts. The Message Distribution Server uses the Protocol Control Module to make higher-level decisions about message processing. The Message Distribution Server passes the full request to the Protocol Control Module and delegates all further processing to it. The Message Distribution Server waits for the Protocol Control Module to return either, a response for the client or null. The following illustration (Figure 5-3) shows a runtime view of message processing using Protocol Control Modules.

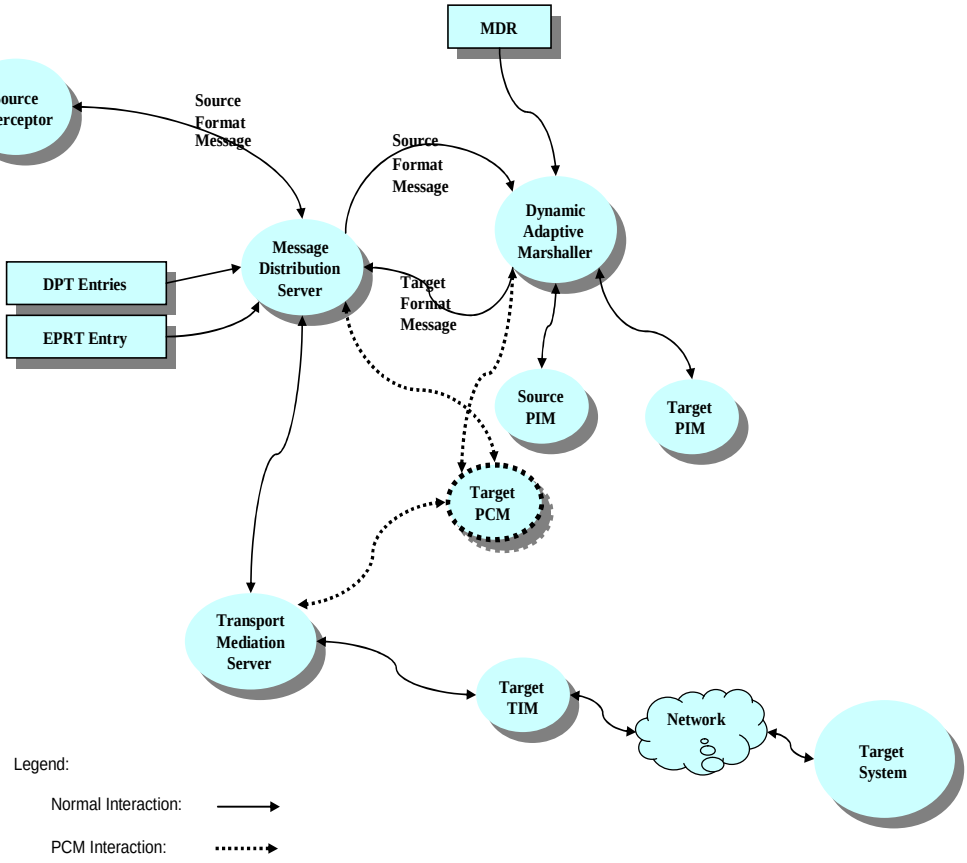

Figure 5-3: TUBE run-time processing showing PCM

5.2.1 Example of a Protocol Control Module and Multiple Interfaces

The following example shows how TUBE uses a Protocol Control Module to handle a situation that is more complex than the standard LOCATION-FORWARD response described above. The client and server are both CORBA-based.

Firstly, there are two interfaces involved. One is referred to as the agent and the other as the worker. The CORBA client performs the following sequence of calls, which are intercepted by the broker:

1. Create an instance of the agent interface.
2. Call a method on the agent to retrieve a reference to a worker.
3. Call methods on the worker to perform the actual application logic.
4. Call a method on the agent to release the worker reference.
5. Call a close method to release the agent

During the call in step 2, the agent returns a LOCATION-FORWARD response, which cannot be returned to the client. If TUBE returned this response, then the client would have direct contact with the agent and thus by-pass the broker for any further processing. Therefore, this response must be handled in a special manner. The Protocol Control Module provides the logic to re-marshal the response and submit it to the new object identified in the response.

The broker however, must still keep a reference to the original object so the client can call the close (step 5) method. When the worker interface is eventually returned, the broker must also store this reference. However, before the response is returned the broker must change the

object reference host and port properties to point to the TUBE CORBA interceptor. The references are cached using the combination of interface name and object key as the identifier.

The client now calls methods on the worker interface to retrieve customer and product data and perform any other necessary processing. After sometime, the client calls the worker release method on the agent, followed by a close on the agent itself.

The requirement to store multiple object references during the same session leads to some complex logic, which is isolated in the Protocol Control Module. However, this logic is now available to any interface that uses CORBA as the target protocol. This logic can handle an arbitrary number of interfaces and multiple LOCATION-FORWARD responses during a single session.

An issue that was confronted during this experiment was the passing of object references between differing protocols. For example, when a SOAP client performs the same sequence of calls as above it receives an opaque object reference in step 2. However, the reference has no meaning to the SOAP client, and it would not know how to handle the reference. Furthermore, how does the SOAP client provide this reference back to the broker to ensure the correct object is invoked for subsequent calls? This is very important if the message exchange is conversational in nature, that is, if it consists of a series of two-way requests and responses. Section 5.2 discusses this situation where both client and server use CORBA and how the CORBA Protocol Control Module stores this reference for the client. SOAP however, has no concept of an object as such [Vinoski, 2003]. All items transferred over SOAP are encoded into XML, therefore any object sent via SOAP must be serialised into XML. The SOAP specification Section 5 Encoding specifies rules for the serialisation of an object into XML. This is not an issue if dealing with a pre-defined object, for example a

Customer record. The structure of the Customer is defined in IDL therefore encoding into XML is straightforward. However, when a method is defined as returning an object, the situation is more complex. For example, what format to follow to encode the XML? There is no IDL that defines an object. What is an object? In OMG IDL, the keyword object is used to denote an opaque reference either, returned by or submitted to an ORB. The CORBA specification defines what an object reference consists of and therefore how to encode one. SOAP on the other hand has no such definition.

The mechanism used by TUBE to overcome this problem is to provide the SOAP client with a session token in the optional response header using the tag-name “objectRef”. This session token is a base64-encoded representation of the object key. The SOAP specification states, “the recommended representation of an opaque array of bytes is the 'base64' encoding defined in XML Schemas” [XSL, 2005 section 5.2.3]. The client provides this token in all subsequent requests in the optional SOAP header using the tag-name “objectRef”. The broker infrastructure understands this token to represent an object reference. When the request is un-marshalled, the value is placed into a special slot of the protocol-neutral buffer. The contents are stored in a location named “objectRef”. When executing a source Protocol Implementation Module, the Dynamic Adaptive Marshaller looks for this value and if found sets the appropriate value in the target Protocol Implementation Module’s variable value segment.

5.3 Modes of Interaction

In a synchronous mode operation, the client is in a blocked state while waiting for a reply. In the asynchronous mode, the client is also waiting, but can continue to perform other tasks whilst waiting. It is necessary to be able to handle both modes independently of one another, and be able to combine them. As an example, a client may make a synchronous request on a

server using the same protocol as always; the client is unaware that the server implementation has been changed to use asynchronous queuing. It is necessary to hold the synchronous session with the client, which is awaiting a response, and is thus blocked. At the same time, a queue on the server-side is monitored while waiting for a response that could come at anytime. When the response arrives, it is sent back to the waiting client. This entire process involves more than sending and receiving of the request and response; the system must marshal the data to and from the source and target protocols.

During the marshalling process, the message data need to be buffered and copied from the source to the target. If brokering a synchronous request over an asynchronous invocation to the server, this keeps the client blocked until the system has completely marshalled and sent the request message. The client continues to remain blocked until the response is returned.

5.4 The Dynamic Adaptive Marshaller

The Dynamic Adaptive Marshaller (DAM) is the name of the Virtual Machine (VM), or interpreter that executes the Protocol Implementation Modules discussed in the previous section. As the name suggests, this component must dynamically adapt to the protocol that it needs to marshal.

Once the Message Distribution Server invokes the Dynamic Adaptive Marshaller, the marshaller must dynamically adapt to the source protocol of the in-bound message, and to the target protocol of the out-bound message. The Message Distribution Server tells the Dynamic Adaptive Marshaller in which protocol the in-coming message is encoded. The Dynamic Adaptive Marshaller then searches the Protocol Definition Repository (PDR) for a request

Protocol Implementation Module that implements the un-marshalling rules for the particular protocol. If it does not find the required Protocol Implementation Module, the Dynamic Adaptive Marshaller returns an error. In this case, the Message Distribution Server will attempt to process the message using the next protocol listed in the Distribution Priority Table. This process continues until either, the Dynamic Adaptive Marshaller successfully loads a Protocol Implementation Module or all protocols have been attempted and failed. In this case, a null is value is returned indicating an error condition.

Once the source Protocol Implementation Module is located, it is loaded and the Dynamic Adaptive Marshaller checks the header for external class declarations. If it finds any, the Dynamic Adaptive Marshaller creates an instance of the classes using the Java Reflection API. Recall from the discussion in Section 4.3.1.1, that these classes must implement pre-defined interfaces; specifically the TUBE.commsBuffer interface. This allows the Dynamic Adaptive Marshaller to handle different buffer types and encoding schemes uniformly. Users are free to wrap or implement any underlying methods or formats that they choose within these classes.

The Dynamic Adaptive Marshaller calls pre-defined method signatures to read and write the different native data types. Therefore, if a user requires compression or encryption and does not want to reveal the algorithm in the MPDL definition, they can implement the algorithm in their external class. In this way, the details remain hidden, whilst still taking advantage of the Dynamic Adaptive Marshaller and a Protocol Implementation Module to perform the actual traversal of the interface and its data structures. This applies to any interface, regardless of its complexity.

Provided the interface is defined in the Module Definition Repository, the Dynamic Adaptive Marshaller and the Protocol Implementation Module ensure encoding of the message as per the rules specified in the MPDL definition for the protocol. The fact that values are encrypted with a proprietary algorithm does not interfere with the encoding and de-coding process.

This is a very powerful feature of the TUBE approach to message processing; special protocol handling code only needs to be written once, not for every interface. This allows optimal re-use of code and uniform treatment of all interfaces over the protocol.

The Dynamic Adaptive Marshaller uses the source Protocol Implementation Module to un-marshal the in-bound message into the internal protocol-neutral format. The next step in the process is to determine the target protocol. TUBE achieves this by using Message Distribution Server APIs to look-up the Distribution Priority Table (DPT) to determine which protocol has the highest priority. The Dynamic Adaptive Marshaller creates a request marshalling Protocol Implementation Module for the target protocol. The Dynamic Adaptive Marshaller then uses values from the TLV buffer to populate values within the target Protocol Implementation Module.

5.5 An Example of Middleware Interaction

TUBE's major objective to provide brokerage between different types of middleware is implemented by the following key mechanism:

- Storing interaction rules in Protocol Implementation Modules and Transport Interface Modules.

The major categories of information required by TUBE to mediate between disparate middleware are:

- On-the-wire protocol and payload format The Protocol Implementation Module encapsulates this information.
- Communication sessions. The Transport Interface Module contains all the communication logic for a particular protocol. The Transport Mediation Server section below discusses Transport Interface Modules.

  These communication sessions may be further decomposed into a number of operations. These are:

  - Session establishment (hand-shaking)
  - Session management
  - Session termination

Each in turn may require further de-composition, depending on the middleware in question. For example, session-management may involve simply sending data, or sending data and waiting for a response. The exact nature of the interaction depends on several factors: the target middleware, the session type (one-way or two-way) and the invoking application (interface) requirements.

The following example uses the mathServer interface to demonstrate a middleware interaction between a CORBA-based client and an MOM-based server. The discussion covers the mapping of object-based requests to non-object (procedural) requests.

As discussed earlier, the Module Definition Repository holds the definition of the interface. This is necessary because there is likely to be an impedance mismatch between the two middleware interfaces, such as for example, with CORBA, which is object-based, as opposed to MOM, which is message-based. The interface definition may need to be altered to reflect this. If mathServer is MOM-based, whereas its clients are CORBA-based, method calls in

CORBA must be properly mapped to MOM messages to ensure that the correct operation is performed by the receiving end.

The following (partial) definition is the original mathServer interface as used by CORBA clients and servers. As CORBA is based on objects and the object has methods associated to it, the client calls the specific operation directly on the object (for example, `obj.add(10, 9)`).

```
interface mathServer
{
    // request structure
    struct math_req
    {
        long  num1;
        long  num2;
    };

// remainder omitted
     void add(in math_req mr, out math_resp arsp) raises (mathException);
// remainder omitted
};
```

Figure 5-4: Partial CORBA definition of mathServer interface

The mathServer IDL defines four methods: add, sub, mul and div. To specify the operation to the MOM, the marshaller encodes the parameters using information from the Module Definition Repository. If the system sent the information as is (that is, with only math_req encoded), the MOM server would not know which operation to perform. Therefore, the IDL needs to be modified to reflect what MOM requires as established by the MOM server team. For example, let us assume that the MOM team established the following COBOL definition for the mathServer.

```
01 MATH_REQ.
   03 OP_CODE     PIC X VALUE SPACES.
             88    ADD_OP      VALUE ‘A’.
             88    SUB_OP      VALUE ‘S’.
             88    MUL_OP      VALUE ‘M’.
             88    DIV_OP      VALUE ‘D’.
   03        NUM1        PIC 9(4) VALUE 0.
   03        NUM2        PIC 9(4) VALUE 0.

01 MATH_RESP.
   03 RESP_NUM    PIC 9(4) VALUE 0.
```

Figure 5-5: COBOL definition of mathServer

For the remainder of the discussion assume that the server has been changed from CORBA to MOM-based and that the clients remain CORBA-based. The data structures math_req and math_resp are almost the same, except for the op_code in the request structure. The client development team creates the IDL shown in Figure 5-6.

```
interface mathServer
{
    // request structure
    struct math_req
    {
        char  op_code;  // **** This is the only change
        long  num1;
        long  num2;
     };


    // response structure
    struct math_resp
    {
        long  ret_num;
    };


    // define the exception
    exception mathException
    {
        string error_text;
    };

     // methods (services, functions, operations)
     void add(in math_req mr, out math_resp arsp) raises (mathException);
     void sub(in math_req mr, out math_resp srsp) raises (mathException);
     void mul(in math_req mr, out math_resp mrsp) raises (mathException);
     void div(in math_req mr, out math_resp drsp) raises (mathException);
};
```

Figure 5-6: The Modified IDL

It is worth noting that:

1. The interface remains largely un-altered.

2. The request and response parameters have not changed.

3. None of the object-oriented properties of the client interface has been violated.

4. Simply the op_code member has been added to the request structure.

The CORBA stubs need to be re-generated and compiled otherwise a runtime error will occur because the internal structure of a parameter has changed. This is necessary regardless of whether TUBE is used to broker messages or not. The only way to use MOM and avoid the

interface change is to allocate separate queues for each operation. In this case the MOM-based server would require logic to process requests in a different manner depending upon which queue they arrive on. Introducing the op-code simplifies this logic to a simple switch statement and does not create a dependency on queue names.

Clients may now use this interface with object-based and non-object based systems. If the IDL were left in its original state, the CORBA call obj.add(10, 9) would be encoded by TUBE into the MOM message as method-name serialised-parameters, for example:

```
add 10 9// spaces between values are for readability only
```

This is the default behaviour based on the IDL definition. The onus is on the systems integrator (the client development team in this case) to ensure that the definitions match. Conversely, if the call was being marshalled from a MOM message to a CORBA call and the IDL were in its original state, TUBE would not be able to determine which method to call. This is because TUBE only receives a sequence of bytes representing the math_req structure, and therefore it is not possible to determine the operation from the original math_req structure, since the necessary information is not there. Using the new IDL, a mapping is defined that instructs TUBE to use the op_code member of the request structure to determine the method to call on the CORBA object. There is still, however, a missing a link between the op_code value and the actual method-name. The following mapping definition provides this link.

```
<FieldMap action="operation">
<Field name="math_req.op_code" offset="0" type="byte" len=”1”>
<XForm Map="A,add M,mul D,div S,sub" />
</Field>
</FieldMap>
```

Figure 5-7: Example transformation (XFORM) map

The XML (fragment above in Figure 5-7) shows a method-name, is derived by either:

- Reading a byte from offset zero (0) in the in-bound buffer, and then mapping it according to the rules defined by the XML tag XForm. This is used when only a buffer of bytes is available, such as in an MQ or JMS BytesMessage.
- The op_code member of the math_req structure. This is used where the structure of the buffer (a SOAP or CORBA message for example) is known. The value is then derived according to the rules defined by the XML tag XForm. This shows, for example, that an 'A' is mapped to "add".

If the Dynamic Adaptive Marshaller cannot determine the operation from the protocol definition because the protocol has no mechanism for specifying methods, it will look for an XML document named ***interface.txf*** *(eg. mathServer.txf).* This document follows the format shown in Figure 5-7. If this file is not found, the Dynamic Adaptive Marshaller returns null to the Message Distribution Server, as it cannot determine the operation. The operation name is necessary as this is used to look-up the Module Definition Repository for the structure and order of parameters and return values.

5.6 From MOM to CORBA

The following example shows a complete translation from an in-bound MOM client request to a CORBA-based object request to illustrate the mapping process. The example shows the add operation with the decimal numbers 1000 and 15 respectively.

MOM Message Buffer (Hexadecimal, little-endian) as extracted by JMS PIM

```
00000 -- 41 ------------- ASCII character 'A'
00001 -- e8 03 00 00 ---- Decimal number 1000
00005 -- 0f 00 00 00 ---- Decimal number 15
```

Figure 5-8: MOM-based message buffer

Using the rules defined in the XML shown above the method name `add` is derived from the byte at offset zero in this buffer. This byte has the value 'A'. This identifies the operation to look-up in the Module Definition Repository so that the parameter list may be retrieved. The following diagram illustrates how the source Protocol Implementation Module builds the intermediate TLV buffer.

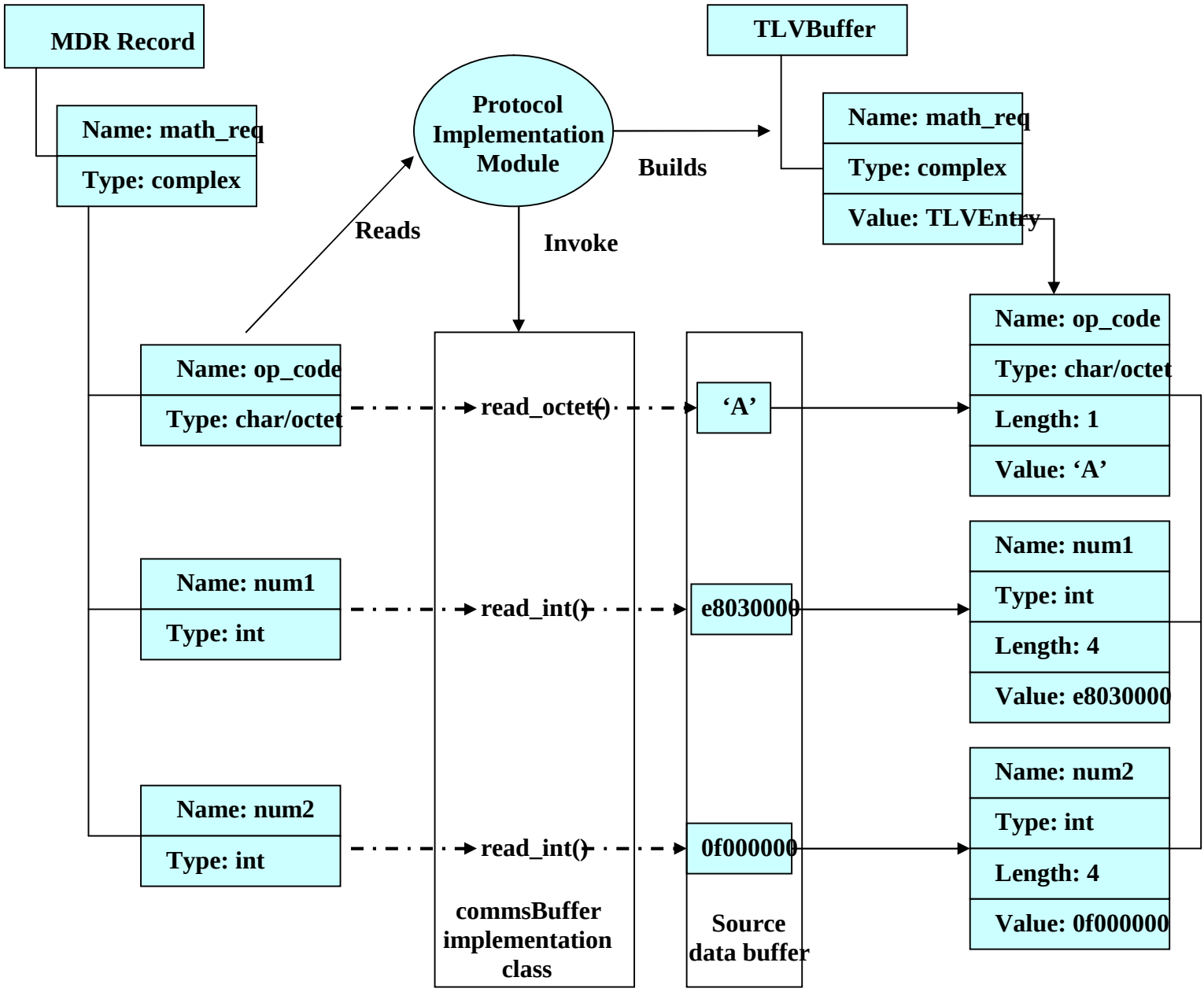

Figure 5-9: How the source Protocol Implementation Module builds the TLV buffer

The TLV buffer constructed in Figure 5-9 is passed to the target Protocol Implementation Module. The following sample shows how the CORBA Protocol Implementation Module marshals these values from the TLV buffer into the CORBA GIOP request buffer.

```
GIOP Header
   00000 -- 47 49 4f 50 – GIOP
   00004 -- 01 00 -------- IIOP version = 1.0
   00006 -- 00 ----------- Byte Order = Big-Endian
   00007 -- 00 ----------- Message Type = Request
   00008 – 3C 00 00 00 --- Message Length = 60 bytes (octets)
Request Header
   00012 -- 00 00 00 00 -- NULL (zero-length) Service Context List
   00016 -- 01 00 00 00 -- Request-id = 1
   00020 -- 01 ----------- Response Expected = true // two-way call
   00021 -- 00 00 00 ----- 3 Reserved octets
Object Key
   00024 -- 13 00 00 00 -- Length of Object Key (octet sequence) = 19
   octets
   00028 -- 2f 31 35 3332 2f 31 30 34 35 32 37 31 32 38 39 2f 5f 30
            00   /1532/1045271289/_0.
Operation and parameters
   00048 -- 04 00 00 00 -- Length of Method Name = 4
   00052 -- 61 64 64 00 -- NULL terminated string = “add”
   00056 -- 00 00 00 00 -- NULL (zero-length) Requesting Principal
   // from this point the data comes from the TLV buffer
   00060 -- 41 ----------- op_code = ‘A’
   00061 -- 00 00 00 ----- 3 bytes of CDR padding for alignment of 4 byte
                           boundary for C/C++ long (Java Integer).
   00064 –- e8 03 00 00 --- Decimal number 1000
   00068 -- 0f 00 00 00 --- Decimal number 15
```

Figure 5-10: GIOP buffer marshalled by CORBA PIM

The following excerpt from the End Point Resolution Table for the MathServer interface shows the specification for the remote object key at offset 28 in the example above.

```
<Interface Name="mathServer" Mode="synch" >
<CORBA ObjectKey="/1532/1045271289/_0" Host="192.168.1.3" Port="1978"
endian=”0”/>
</Interface>
```

Figure 5-11: Portion of End-Point Resolution Table

The CORBA Transport Interface Module uses the Host and Port values to establish communication with the remote ORB, and the CORBA Protocol Implementation Module uses the ObjectKey value to ensure that the correct object is invoked at the end-point. This value is encoded into the request as shown above.

Once the IDL definition is complete, the IDL is submitted to the TUBE IDL compiler, which populates the Module Definition Repository with the interface information. This information is protocol-independent. That is, the same Module Definition Repository definition is used to marshal CORBA, MQ, JMS or any other supported middleware protocol. The protocol marshalling rules are already contained in the relevant Protocol Implementation Modules and the transport (communication-level) interactions are defined in Transport Interface Modules.

5.7 The Transport Mediation Server (TMS)

The Transport Mediation Server is very similar in operation to the Dynamic Adaptive Marshaller (DAM) in that it uses loadable modules to perform its functions. These loadable modules are known as Transport Interface Modules (TIMs). The Transport Interface Module carries-out the communications with target end-points.

The characteristics of a Transport Interface Module are specified in the end-point definition of the MPDL description of a protocol. The Transport Interface Modules consist of Java code generated by the MPDL compiler. These modules implement a pre-defined interface and the compiler generates conformant basic skeleton code, which can be modified by the user.

The type of code generated depends on the transport specified in the end-point clause of the protocol definition. If the transport is MOM-based, such as JMS, the compiler generates a generic module that may be used with any JMS implementation. The user is free to implement

any higher-level message management semantics. By default the generated code provides the ability to match requests and responses using the correlation-id (see JMS, 2003).

The other generic model is for a sockets-based transport based on TCP/IP for protocols such as HTTP or CORBA. If the specified transport is not one of the pre-defined types, the compiler generates a basic skeleton only. This skeleton consists of empty method bodies and the user must fill in the actual implementation. This is useful in the case where for example, the user had an existing SNA [SNA, 2005] gateway. The user would fill-in the skeleton methods with the calls to the relevant SNA APIs.

This only need be written once and is then available for any interface to be exchanged over SNA. For example, assume the user wishes to retain their existing CORBA clients and have them communicate with an SNA server. This would now be possible because the user has a Transport Interface Module that interfaces to SNA. All the complexity of converting CORBA messages to SNA is provided by the protocol-mapping rules defined in the Protocol Implementation Modules. The user would of course, need to create an MPDL definition of SNA.

The Java classes implement the following pre-defined interface so that the Transport Mediation Server can use the same invocation mechanism without the need for dynamic method discovery. The following code-fragment shows the interface defined in Java.

```
interface TMSTim
{
  public void setup(EPRTProcessor arg);
  public boolean send(byte message[]);
  public byte[] receive(byte message[]);
  public void close();
};
```

Figure 5-12: Transport Interface Module interface definition

The methods are:

- `setup()` – this method is used for initialisation. The arg parameter is an instance of an EPRT entry. This contains name/value pairs read from the End Point Resolution Table for this protocol and interface combination. The EPRTProcessor provides APIs for reading these values.

- `send()` – this method is called to send a message. It returns a Boolean value of true if the send is successful or false otherwise. The Transport Mediation Server uses this value to decide if the receive method should be called. If the result is false, the Transport Mediation Server returns a failure indication to the Message Distribution Server. If the result is true and the message exchange is two-way (that is, a response is expected) the receive method is called. If the message is one-way and the result is successful, the Transport Mediation Server returns the result immediately.

- `receive()` – this method is called by TMS when waiting for a response. Whether the wait is synchronous or asynchronous depends on the implementation of the method.

- `close()` – this method is called to terminate a session. It may be used to clean-up and perform any necessary shut-down operations.

The following code illustrates a usage scenario of the `setup` method using JMS (MOM) as the transport.

```
public void setup(EPRTProcessor eprt)
  {
    // system writes to this
    String sendQueue = eprt.getElementValue("JMS", "Output");
    // system reads from this
    String recvQueue = eprt.getElementValue("JMS", "Input");
    // Timeout interval, how long to wait for a response
    String tout = eprt.getElementValue("JMS", "Timeout");
    if (tout == null)
    {
     m_Timeout = 60000;
    }
    else
      m_Timeout = Long.parseLong(tout) * 1000;


    // set-up the queueus
    setSendQueueName(sendQueue);
    setRecvQueueName(recvQueue);
  }
```

Figure 5-13: JMS TIM code fragment showing setup method

The code above firstly obtains the names of the queues and the timeout periods as specified in the End-Point Resolution Table, and then calls internal methods to interact with the JMS API. The entire JMS Transport Interface Module appears in Appendix F.

6 Evaluation of TUBE

In this chapter, the focus is on demonstrating to what extent TUBE meets the requirements outlined in Chapter 2. In the development of what is essentially an engineering artefact, there is a difficulty in proving or demonstrating that the design can be applied in all circumstances. In the case of TUBE, this would mean showing that:

i. all protocols could be described in the MPDL,
ii. that at run-time TUBE was able to correctly interpret these and allow seamless interoperation across protocols, and
iii. TUBE can be implemented in a range of real-world operating environments.

Since it is not practical to do this for all middleware protocols currently existing and future, in all their possible implementation environments the approach taken here is to select a representative set of current middleware protocols, and assess the applicability and performance of TUBE for these. The evaluation of TUBE will be based on a number of scenarios that assess its ability to meet the design criteria outlined in Chapter 2. These requirements are re-stated below. The assessment of a tool like TUBE needs to go beyond testing in "toy" environments, to potential real world applications. To this end, the applicability of TUBE to a realistic corporate operating environment is assessed using a case study from a live implementation in a large information-based Australian corporation.

Requirement 1: The MPDL must be able to describe all middleware protocols in an easy to use, declarative way. This requirement governs the overall design of the MPDL that must minimise its semantics, and must make use of existing standards and languages. The overall

aim is to provide the capability to describe protocols to application developers, and to minimise procedural programming. Specifically the broker must:

i. provide a framework for implementing any special semantics, such as interaction semantics or data coding and de-coding functions,
ii. describe a middleware product once for integration across all protocols,
iii. be easily extended by programming in an understandable and maintainable way, and
iv. special protocol handling code is written once, not for every interface. This allows optimal re-use of code and uniform treatment of all interfaces over the protocol.

Requirement 2: TUBE must be able to convert data and message formats from any middleware protocol to that of any other middleware protocol. Specifically, the broker must:

i. deal with different transport formats (text or binary),
ii. deal with all native data types, complex data, and user defined data.

Requirement 3: TUBE must be able to seamlessly and transparently deliver requests in one protocol to a service in another protocol, and deliver responses back to the requestor. This requirement lies at the core of how the functionality is implemented. Specifically, the broker must:

i. support different modes of interaction (synchronous or asynchronous),
ii. access servers through different protocols, without re-compiling application programs, and
iii. support deployment at one end only, so that a server-receiving node will receive and respond to requests in its own protocol, without any need to process this request in any different way.

Requirement 4: TUBE must be able to choose between multiple options for providing a service. This requirement means that the broker must be able to provide the ability to dynamically locate alternative end-points, so that alternative services may be accessed, should they be available. One mechanism for supporting alternate end-points is protocol prioritisation. This allows the user to list protocols in priority order. The broker will try each listed protocol in-turn until either, the message delivery is successful, or all protocols are exhausted. In the latter case, the client receives a failure indication.

Requirement 5: TUBE must be able to effectively operate in a data intensive, real world environment. It must be easily adaptable to that environment, and it must perform efficiently.

6.1 Selected Middleware Protocols

Table 6-1 shows the middleware protocols used in this evaluation. The MPDL descriptions appear in Appendices A through E.

Protocol	Primary Data Type	Object Based	Mode Supported	Special Extensions	Comments
CORBA	Binary	Y	Synch	Yes	A widely used binary, object-based, primarily synchronous protocol. Includes special semantics.
SOAP	Text	Y	Synch		Contemporary, text based protocol.
HTML	Text	N	Synch		Another example of a text-based Internet protocol (encoding).
JMS	Binary	N	Asynch		A contemporary asynchronous protocol.
Encrypted XML	Text	N	Synch		An example of special encoding/de-coding applied to extend a standard protocol format.
TUBE XML API	ALL	Both	Both		An example of the ease of use of TUBE's APIs.
TUBE Java API	ALL	Both	Both		Another example of the ease of use of TUBE's APIs.

Table 6-1: Evaluation Protocol characteristics

Each of these has been selected for the following reasons.

i. CORBA is a widely used and historically significant middleware protocol. It was the first important object-based protocol. CORBA is a complex protocol that includes extensions and special semantics. While it is no longer a major middleware protocol in

new applications, it is an important protocol from the perspective of integration in legacy systems.

ii. SOAP is the delivery vehicle for the latest SOA (Service-Oriented Architecture) systems. As the core protocol underlying the Web-Services that interact with these systems, SOAP is an important contemporary middleware protocol.

iii. JMS is the MOM (Message-Oriented Middleware) of choice amongst the J2EE (Java 2 Enterprise Edition) development and user communities. It is an asynchronous protocol by definition, in that a subscriber or listener registers interest in a queue, and a producer or publisher places messages onto that queue.

iv. HTML is another text-based Internet protocol. This was selected to test the ability of MPDL to represent individual items in a specified manner (refer to Appendix C).

v. Encrypted XML shows how a standard encoding (XML) may be extended with very little effort to produce a new protocol (encoding). One of the criticisms of XML is the large payload size. The W3C currently has a working group looking at the efficient exchange of XML [EXI, 2005]. Therefore, rather than encryption, other extensions, such as compression for payload reduction could be implemented.

vi. The TUBE XML-based API demonstrates the ability of an application to interact with TUBE using XML documents. This allows the application to reach servers using a variety of protocols that the application does not need to understand. It only needs to supply and receive IDL conformant XML.

vii. The TUBE Java-based API provides the same capabilities as the XML API, except that applications deal with pre-existing Java objects. This means the only coding change is to send and receive the objects via the TUBE API. Otherwise, the client developer must marshal and un-marshal those objects in code to and from whatever protocol the server is using.

6.2 Evaluation Operating Environments

TUBE can be implemented independently of operating environments, and a number of different operating environments were used in the experiments. The test configurations were as follows:

Servers:

- HP-UX CORBA server
- Windows XP-based server (Pentium M notebook)
    - Servers hosted
        - CORBA
        - JMS
        - SOAP
- Linux-based server (Pentium PC)
    - Servers hosted
        - CORBA
        - JMS
        - SOAP

Clients:

- Windows XP-based client (Pentium M notebook)
    - Clients implemented
        - CORBA
        - SOAP
        - JMS
        - XML API
        - Java API

- Linux-based client (Pentium PC)
    - Clients implemented
        - CORBA
        - SOAP
        - JMS
        - XML API
        - Java API
- Object Request Brokers (ORBs)
    - MICO C++-based open-source ORB [MICO, 2000]
    - JacORB Java-based open-source ORB [JacORB, 2004]
    - Sun J2SE internal ORB (part of Java 2 runtime system)
    - Proprietary 3rd-party ORB (used in corporate case study Sec 6.3.9)
- Message queues
    - Sun J2EE message queue (part of J2EE server runtime)
    - ActiveMQ Java-based open-source JMS implementation [ActiveMQ, 2005]

Each of the protocols has been implemented as an MPDL. Each of these need only be implemented once, and then can be used as either source or target protocols. An overview of the testing scenarios appears in Table 6-2.

The interfaces tested in these scenarios were the simple mathServer and more complex variations of the mathServer; the most complex variant appears in Appendix G.

Scenario	Source Protocol	Target Protocol	Source Data Type	Target Data Type	Source Object Oriented	Target Object Oriented	Client Mode	Server Mode	Comments
1	CORBA	SOAP	Binary	Text	Y	Y	Synch	Synch	Demonstrates mapping of binary and text data
2	SOAP	CORBA	Text	Binary	Y	Y	Synch	Synch	Demonstrates handling of special semantics, specifically CORBA LOCATION_FWD response.
3	JMS	CORBA	Binary	Binary	N	Y	Asynch	Synch	Demonstrates ability to handle both synchronous and asynchronous interactions
4	CORBA	JMS	Binary	Binary	Y	N	Synch	Asynch	Demonstrate sending from synchronous to asynchronous
5	JMS	SOAP	Binary	Text	N	Y	Asynch	Synch	Demonstrates ability to handle both synchronous and asynchronous interactions using another synchronous server.
6	SOAP	JMS	Text	Binary	Y	N	Synch	Asynch	Demonstrate sending from synchronous to asynchronous using another synchronous client
7	CORBA	Encrypted XML	Binary	Text	Y	N	Synch	Synch	Demonstrates ability to encrypt entire payload after marshalling.
8	Encrypted XML	CORBA	Text	Binary	N	Y	Synch	Synch	Demonstrates ability to decrypt entire payload before marshalling.
9	TUBE XML API	ALL	ALL	ALL	Both	Both	Both	Both	Demonstrates TUBE's XML-based API
10	TUBE Java API	ALL	ALL	ALL	Both	Both	Both	Both	Demonstrates TUBE's ability to interact with native Java objects.
11	CORBA	HTML	Binary	Text	Y	N	Synch	Synch	Further demonstrates converting from an object-based binary protocol to text.

Table 6-2: Scenario Matrix

The protocols in the table above were all tested as source and target protocols to and from each other. The standard source protocols to same target protocol scenarios were also tested (for example, CORBA to CORBA). These are not included in the table for brevity.

In all the scenarios tested, the same engine (the Dynamic Adaptive Marshaller) processed all the interfaces over all the protocols. The protocol structure and marshalling rules reside in the specific Protocol Implementation Module.

All the scenarios address requirement 1, in that all the protocols employed are defined in MPDL (1i,ii) and existing clients and servers required no changes. Use of clients based-on the TUBE APIs demonstrates easy extensibility (1iii). Other requirements shared by all scenarios are: the ability to deal with varying data types (2ii), the ability to access servers using different protocols without change to applications (3ii), and ability to choose between multiple service providers (4). Requirement 4 is demonstrated in all scenarios by modifying the End-Point Resolution Table entries for the tested interfaces and removing specific protocols. For example, the Distribution Priority Table entry specified CORBA followed by SOAP. By removing the CORBA entry, the system automatically used SOAP. Replacing the CORBA entry caused the system to revert to CORBA.

6.3 Evaluation Scenarios

6.3.1 Scenario 1.

CORBA[9] clients to SOAP servers specifically address requirement 2, showing the handling and transformation of binary data to text and vice versa (2i). This also demonstrates how quickly a new protocol may be added to an existing environment. Aside from the optional coding of a SOAP server, the definition and support for SOAP was achieved within a day

[9] The CORBA client calls were tested with both C++-based and Java-based CORBA clients. This was to test language independence of the brokering process.

rather than days or weeks. This demonstrates the power of the MPDL and Protocol Implementation Modules.

6.3.2 Scenario 2.

SOAP clients to CORBA servers address the handling of protocol-specific semantics. Specifically, the CORBA LOCATION-FORWARD response message demonstrates the usage of Protocol Control Modules to handle special interaction semantics (1i,iv). This complexity was introduced and covered in detail in the previous chapter. Extensive performance analysis is beyond the scope of this thesis; however, some timings were captured during this test. The results show that using TUBE to broker the communication and perform the marshalling adds only a small overhead of approximately 0.4 – 0.6 seconds. The test consisted of the following operations:

- establish connection to agent
- handle re-direction (connect to object)
- send login request
- receive login response
- send get customer details request
- get customer details (20k payload)
- release object
- release agent

The test was carried-out ten (10) times using a CORBA client calling directly to the ORB, and then another ten (10) times using a SOAP client calling the ORB via TUBE. The following table shows the results of each test. The Timings were displayed by the client at the end of each invocation. These tests were carried out on a Pentium M Notebook.

Direct Calls (seconds)	SOAP intercepted (seconds)
2.463	2.963
2.593	2.800
2.353	2.853
2.413	2.915
2.393	2.893
2.423	2.553
2.453	2.993
2.563	2.863
2.463	2.743
2.473	2.865

Table 6-2 Test result timings

6.3.3 Scenarios 3 and 4.

JMS clients to CORBA servers address the requirement of supporting different interaction modes (3i). Specifically, it demonstrates an asynchronous client interacting with a synchronous server. Scenario 4 demonstrates a synchronous client interacting with an asynchronous server.

6.3.4 Scenarios 5 and 6.

These scenarios address the same issues as scenarios 3 and 4 using a different synchronous protocol, in this case SOAP. This scenario was designed to test whether SOAP communicating with an asynchronous protocol would behave any differently.

6.3.5 Scenarios 7 and 8.

These scenarios demonstrate the ability of TUBE to mediate between standard protocols (for example, CORBA) and home-grown (for example, encrypted XML) protocols. This further demonstrates the compliance with requirements 1(iii, easy extensibility), 2 (conversion of data and message formats) and 3(iii, support deployment at one end only).

6.3.6 Scenarios 9 and 10.

These demonstrate requirements 1(iii, easy extensibility) and 3(iii, support deployment at one end only) by proving the ease of using the TUBE API. The same API-based clients were used for all interfaces across all protocols. Command-line parameters specified which interfaces and services to invoke. Each interface's End-Point Resolution and Distribution Priority Table entries were modified to ensure that different servers could be invoked if the preferred one was not available (4).

6.3.7 Scenario 11.

This scenario further demonstrates TUBE's ability to convert from binary to text by defining an MPDL description of HTML (2).

6.3.8 Using TUBE's APIs

The scenarios discussed were also tested using TUBE's internal APIs as the originating clients. Firstly, a simple Java client was written. This client reads XML documents from disk and submits them to servers using TUBE's XML-based API. The requests are sent to different servers over various protocols. The client receives an XML document in response. The other TUBE API mechanism tested is the Java-based API. With this API, clients populate and

submit a real Java object with their request and they receive a real Java object in response. If there are no Java objects that implement the interface defined in the IDL, the TUBE IDL compiler has the ability to generate stub classes. These classes consist of getter and setter methods only. A class is generated for each high-level structure (struct) definition in the IDL. The following code fragments illustrate the usage of both API variants.

```
// read XML file
String payload = getXMLPayload(filename);
Jclient jc = new JClient(interfacename, methodname, payload);
// send request and wait for response
String sr = jc.processMessageWait();
// print results
System.out.println("Server Response for :” + methodname);
System.out.println(sr);
```

Figure 6-1: TUBE XML-based API example

```
// create a client-side instance of a TUBEObject
// parameters to constructor: interfacename, objectname, operation
// e.g. “myinterface”, “myreq”, “mymethod”
TUBEObject o = new TUBEObject(args[0], args[1], args[2]);

/**
* An invoke that takes "real" java objects as parameters
*/
math_req req = new math_req();
req.setnum1(1000);
req.setnum2(15);
req.setop_code(‘A’);
// invoke using our request object
// The call to getReturnObject (below) actually causes the invoke to occur
// The invokeByObject method simply sets-up the request parameters
TUBEObject ret = o.invokeByObject(req);
// supply an instance of the top-level object to be populated
math_resp mr = new math_resp();
mr = (math_resp) ret.getReturnObject(mr);
if (mr != null)
{
   System.out.println("Class is: " + mr.getClass().toString());
   mr.print();
}
```

Figure 6-2: TUBE Java object-based API example

The protocol that TUBE uses internally to implement its object-based API is itself processed like any other protocol. That is, defined in MPDL and the generated Protocol Implementation Module executed (interpreted) by the Dynamic Adaptive Marshaller. This internal protocol is BinTLV (an abbreviation of Binary TLV). When a client creates an instance of TUBEObject

(see Figure 6-2) it actually creates a proxy object that uses BinTLV to communicate with the TUBE server. The client interacts with native Java objects and is un-aware whether the server is local or remote. This can easily be extended to support other object-based protocols such as Microsoft .Net.

6.3.9 The Corporate Case Study

The company involved in the case study is a large Australian corporation with an annual turnover of more than $1 Billion and approximately 130 fulltime employees in their Information Technology department. They have a variety of hardware platforms, including Sun and HP running UNIX, and Intel servers running Microsoft Windows. The desktops run Microsoft Windows.

For approximately six years, the company have attempted to achieve real-time integration between their front-end CRM system and their legacy core system. They have attempted to use MOM-based integration, which was partially successful and works with about three simple transactions. However, modifying customer details and placing an order in the CRM system, and then having those details reflected in the back-end and the order recorded has eluded them. One of the main reasons they have not been able to integrate has been that the MOM-based approach provides no business context, and therefore the data update cannot be applied accurately.

A small prototype was developed that uses CORBA as the integration protocol at the server end. The client uses SOAP. The company had no support for SOAP-based clients. Therefore, a core component of TUBE was introduced into the environment. This component renders CORBA requests from XML/SOAP and generates XML/SOAP response documents from CORBA objects. The TUBE IDL compiler is used to generate XSD [XSD, 2005] schemas

used for XML validation by the workflow engine at runtime. The prototype tested the following process:

- A web-based client sends the customer and order details using SOAP.
- The SOAP server uses the TUBE component to populate a CORBA request object from the XML.
- The SOAP handler sends the request to the server and waits for a response.
- The SOAP handler passes the response object to the TUBE component.
- The TUBE component returns a fully populated XML document (based-on the contents of the object) to the handler.
- The handler then returns this response to the client.
- The client validates the response against the generated schemas (XSDs)

This process allows integration between the core customer management system and the company's front-end CRM system via a workflow bus. This extension of the prototype is implemented in their production environment.

The following experiment tests TUBE's ability to intercept and re-direct messages from existing clients to new servers using differing protocols and implementation languages.

The back-end application has an existing C++-based GUI client. The client is re-directed to a TUBE interceptor, which marshals the requests into SOAP and sends them to another TUBE node that converts them back to CORBA, and then submits the request to the remote HP-UX server (see Figure 6-3 below). This test demonstrates that the CORBA client could be re-directed to a SOAP server with a simple configuration change, in a real application in a production environment. It also demonstrated that introducing the TUBE SOAP node allowed

SOAP clients to access the core legacy system. The SOAP support was achieved by creating and compiling an MPDL definition of SOAP (see Appendix B), and then adding SOAP to the Distribution Priority and End Point Resolution tables. The clients did not require any modification or re-deployment. The changes were confined to client configuration files. In the case of the C++ client this meant a change to the Windows registry and in the case of the SOAP client, the specification of a new URL. The C++-based client was swapped back to CORBA and then diverted to JMS without the user noticing any disruption or changes in application behaviour. The integration was completely invisible. These tests were also carried-out using a Java-based client.

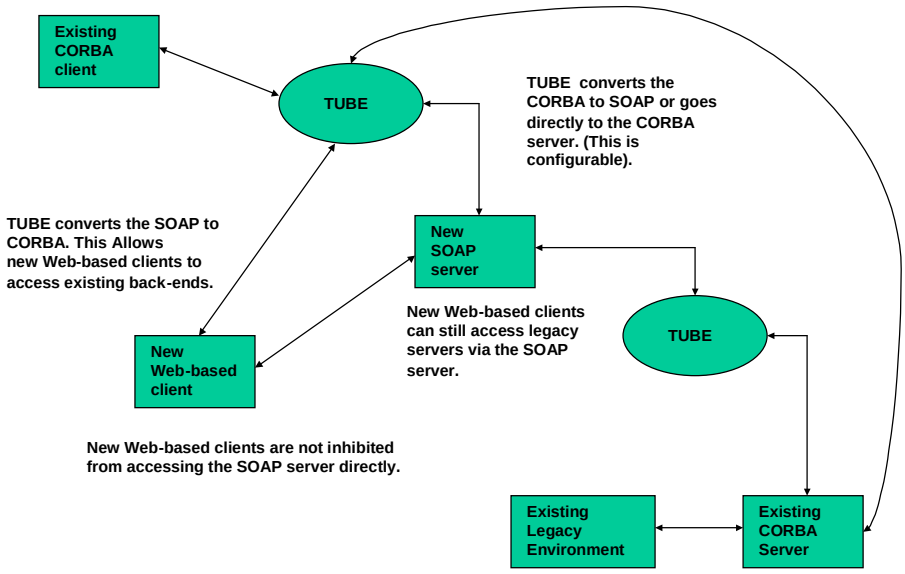

Figure 6-3: The CORBA re-routed to SOAP scenario

6.4 Summary

Overall, the scenarios demonstrated that TUBE meets its stated design objectives. It allows new protocols to be added to IT environments in a fraction of the time that existing methods do. SOAP was defined and available for integration within a day. Likewise, with all the other MPDL defined protocols. The ability of the MPDL compiler to generate Transport Interface Module templates and the loadable Protocol Implementation Modules significantly reduces the amount of coding and configuration required to introduce new protocols. TUBE proves that it is possible to introduce new protocols through configuration. This allows preservation of existing IT systems. As new protocols and access mechanisms become popular, (for example, SOA and web-services) the protocols that they employ (such as SOAP) may be added quite quickly with minimal intrusion. This allows the existing legacy back-end servers to remain un-altered. Clients using the new access protocols may be deployed without concern for the access protocol of the server. This results in reduced costs and extended longevity (future-proofing) of existing assets.

7 Conclusion

This research has developed an architecture, a mechanism and a tool-set that makes systems integration between disparate types of middleware much easier to implement. It allows an organisation to introduce new middleware while preserving existing investment in middleware and systems developed for them, avoiding re-development of these existing systems.

This research has looked at a number of aspects of middleware integration: the nature of the language to describe middleware protocols, the architecture and nature of the run-time system needed to process requests using the protocol description, and the appropriateness and effectiveness of the proposed solution.

The project has developed a language that can specify a range of middleware types. The language describes most of the characteristics of protocols including end-points, in a declarative form. The project has also examined the trade-off between the effort in representing features declaratively and procedurally, and some features have been implemented as programmed procedures; for example, the usage of Protocol Control Modules to handle protocol-specific semantics. Most protocols and most features have been able to be implemented declaratively in a Middleware Protocol Definition Language (MPDL) that is interpreted at run-time. A middleware protocol need only be described fully in the MPDL and any user-defined procedures once, and it can be re-used for every interface processed over that protocol, as a request or a response.

The research has also specified and implemented a run-time system that interprets the middleware protocols and delivers protocol specific requests to the required service. This part

of the project addressed the architecture of a generic middleware broker, the nature of components, repositories, Application Programming Interfaces (API) and their behaviour, including the ability to dynamically access multiple middleware protocols transparent to the requestor.

Finally, this research looked at the effectiveness of the approach from the perspective of reducing the amount of programming needed to integrate protocols, and the limitations of this approach.

The research shows that it is possible to develop an extensible language that describes all types of middleware. A single interpreter component, the Dynamic Adaptive Marshaller is capable of processing all the middleware definitions, using loadable protocol-specific Protocol Implementation Modules that are generated from the formal protocol descriptions. There is no special application processing required to convert binary protocols into text and vice versa. At run-time, the relevant Protocol Implementation Modules are loaded and executed to translate between different representations.

TUBE provides in-built handlers for CDR, XML, ASCII text and raw binary data with user-defined extensible marshalling classes catering for special encoding. The user-defined classes may be called either, before, during or after payload processing. Therefore, TUBE can be used to traverse all the complex structures defined in an interface, and then call the special class to, for example, compress or encrypt the payload before transmission.

The TUBE approach makes it possible to encapsulate all the communications logic and configuration into a single component, the Transport Mediation Server. This also uses loadable modules called Transport Interface Modules. These modules are generated from the

protocol definition. Protocol-specific control code is isolated into a single Protocol Control Module. This code is written only once and is then available for use by any interface processed over this protocol.

The thesis has demonstrated that TUBE works for a number of the most important protocols. These protocols have demonstrated a range of characteristics and challenges for inter-operation, including the ability to move between synchronous and asynchronous protocols, between text and binary-based protocols and object and non-object-based protocols. The approach has also demonstrated its applicability in a real-world production environment.

This research has extended the non-programming approaches to middleware integration, provided a language for describing all middleware protocols, and provided a runtime system capable of processing the protocol descriptions. This is potentially an important contribution as it means that a significant amount of coding and work involved in integrating existing systems can be avoided. Organisations will be able to add new middleware while maintaining their existing software. In some cases, they may be able to maintain existing technologies, and easily rollback to that technology, should there be problems with the new technology. Further, organisations can keep valuable legacy systems and integrate them into new applications using contemporary middleware.

7.1 Future Work

This project has also identified potential future work. It would be desirable to extend the language so that more complex interaction rules can be described. For example, the logic contained in Protocol Control Modules may be able to be provided in the definition and then generated into the module. The user would not need to write code, but simply specify the rules for the logic. This has the potential to reduce the amount of code required to handle special protocol semantics.

The ability to specify the communication interactions and have them converted into op-codes the way that Protocol Implementation Modules are is another possible extension to the MPDL. This would further reduce the amount of code the user has to deal with. One possible method is to have the instructions represented as pseudo-code in the MPDL. The invocation of external APIs then occurs by op-codes. Experiments are necessary to determine if it is possible to specify such interaction rules without overly complicating the language.

Another improvement is the extension of the MPDL to cater for definition of more recent and complex versions of CORBA and IIOP, such as the version of IIOP used in Java RMI. These later versions have extra features such as message fragmentation. TUBE currently has no support for this. The current implementation also assumes that security credentials and transaction contexts are handled at the application layer and that TUBE is simply a conduit between the two communicating systems; providing protocol mapping.

The recent advent of SOA (Service Oriented Architecture) has inspired the idea of extending TUBE to provide a SOA enabling framework. A service repository stores a high-level view of the available services and links them back to the IDL defined interfaces in the Module Definition Repository. The user searches the repository for a service, say get-customer-details; this is linked back to the Customer interface. The system presents the user with a view of all the methods available within this service. The user can then select the methods they wish to invoke and see what the parameters are. The user then configures a client to access the methods. TUBE can generate the client skeleton and wrapper classes, and then the user fills in their application logic. In addition, of interest is investigating other integration mechanisms, such as Generative Communications [Hesselbring, 1998] and continuous media interaction [Fitzpatrick et. al, 1998].

The Middleware Protocol Definition Language and the supporting runtime environment (TUBE) described in this paper provide a powerful and easier approach to the integration of heterogeneous middleware systems. Organisations that adapt TUBE will be greatly reducing the risk associated with changing technology, being able to incrementally introduce new middleware protocols, and maintain existing applications. This preserves their existing IT investments and helps to insulate them from the impacts of technology changes.

8 References

ActiveMQ 2005, Product Home Page
http://www.activemq.org

BEA 2001, TUXEDO product documentation
http://www.bea.com/products/tuxedo

Blair, GS, Blair, L, Issarny, V, Tuma, P, and Zarras, A 2000, 'The role of software architecture in constraining adaptation incomponent-based middleware platforms' in *IFIP/ACM international Conference on Distributed Systems Platforms* (New York, New York, United States, April 03 - 07, 2000). Multimedia Middleware Workshop. Springer-Verlag New York, Secaucus, NJ, Pages 164-184.

Box, D 1998, *Essential COM*, Second Edition, Object Technology Series, Addison-Wesley, New York.

CICS 2001, IBM Product Documentation
http://www.ibm.com/products/cics

Coulson, G, Blair, GS, Clarke, M, and Parlavantzas, N 2002, 'The design of a configurable and reconfigurable middleware platform', *Distributed Computing.* vol. 15, no. 2, Pages 109-126.

Dashofy, EM, Medvidovic, N, and Taylor, RN 1999, 'Using off-the-shelf middleware to implement connectors in distributed software architectures' in *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, Pages 3-12.

Emmerich, W 2000, 'Software engineering and middleware: a roadmap' in *Proceedings of the Conference on the Future of Software Engineering* (Limerick, Ireland, June 04 - 11, 2000). ICSE '00. ACM Press, New York, NY, Pages 117-129.

EXI 2005, Efficient XML Interchange Working Group – The World Wide Web Consortium
http://www.w3.org/XML/EXI

Geihs, K, 2001, 'Middleware Challenges Ahead' *IEEE Computer*, June 2001, Pages 24 – 31.

Grace, P, Blair, GS, and Samuel, S 2005, 'A Reflective Framework for Discovery and Interaction of Heterogeneous Mobile Environments', ACM SIGMOBILE Mobile Computing and Communications Review, 9(1), pp 2-14, special section on Discovery and Interaction of Mobile Services, January 2005

Hasselbring W, Roantree M, 1998, 'A Generative Communication Service for Database Interoperability,' in Third International Conference of Cooperative Information Systems *coopis* , p. 64, 1998.

Holzmann, G J 1998, United States Patent No. 5,826,017, October 20th 1998

Issarny, V, Bidan, C, and Saridakis, T 1998, 'Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype' in *Proceedings of the international Conference on Configurable Distributed Systems* (March 04 - 06, 1998). CDS. IEEE Computer Society, Washington, DC, Pages 207-214.

JacORB 2004, Product Home Page
http://www.jacorb.org

JMS 2003, Java Message Service – Sun Microsystems product documentation.
http://java.sun.com/products/jms

Joia, LA 2000, 'Information Technology for Relational Business Ecosystems: A Case Study in the Brazilian Engineering Industry.' in *Journal of Global Information Management*, vol. 8, no. 3, July 2000, pages 24-33

Kuznetsov, E 2004, United States Patent No. 6,772,413, August 3rd 2004

MICO 2000, Product Home Page
http://www.mico.org

Microsoft 1998, 'Distributed Object Model Protocol Specification, Version 1.0', Microsoft Corporation, January 1998.

Microsoft 2000, .Net Home Page
http://www.microsoft.com/net

MQ 2001, MQ-Series IBM product documentation
http://www.ibm.com/mqseries

OMG 2000, The Common Object Request Broker: Architecture and Specification, Revision 2.4, Object Management Group Inc, Framingham Mass., October 2000.

Orfali, R, Harkey, D, and Edwards, J 1996, *The Essential Distributed Objects Survival Guide*. First Edition, John Wiley and Sons, 1996.

RMI 2003, Java Remote Method Invocation – Distributed Computing for Java.
Sun Microsystems Whitepaper.
http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html

Schmidt, DC, Levine, DL, and Mungee, S 1998, 'The design of the TAO Real-Time Object Request Broker', in *Computer Communications*, vol. 21, no. 4, April 1998, Pages 294-324.

SNA 2005, Systems Network Architecture – IBM Corporation
http://www.redbooks.ibm.com/abstracts/sg245291.html

SOAP 2003, Simple Object Access Protocol 1.1 – The World Wide Web Consortium
http://www.w3.org/TR/2000/NOTE-SOAP-20000508

Steedman, D 1993, *ASN.1 The Tutorial & Reference*., Technology Appraisals, 1993.

Taylor, RN, Medvidovic, N, Anderson, KM, Whitehead, EJ, Robbins, JE, Nies, KA., Oreizy, P, and Dubrow, DL 1996, 'A Component- and Message-Based Architectural Style for GUI Software' in *IEEE Transactions on. Software Engineering,* vol. 22, no. 6, June 1996, Pages 390-406.

The Open Group 1997, *Introduction to OSF DCE,* Open Group Product Documentation, F201, The Open Group, November 1997.

TIBCO 2005 Product Home Page
http://www.tibco.com

van Steen, M, Homburg, P, and Tanenbaum, AS, 1999, 'Globe: A Wide-Area Distributed System' in *IEEE Concurrency* vol. 7, no. 1, Jan. 1999, Pages 70-78.

Vawater, C, and Roman, E 2001, 'J2EE vs. Microsoft.NET, A comparison of building XML-based web services', *report prepared for Sun Microsystems Inc*., The Middleware Company, June 2001 Page 13.

Vinoski, S, 2003 'It's Just a Mapping Problem', in *IEEE Internet Computing May-June* 2003, Pages 88 – 90.

Waddington, DG. and Coulson, G 1997, 'A Distributed Multimedia Component Architecture', in *Proceedings of the 1st international Conference on Enterprise Distributed Object Computing* (October 24 - 26, 1997). EDOC. IEEE Computer Society, Washington, DC, Page 334.

XML 2003, Extensible Markup Language – The World Wide Web Consortium
http://www.w3.org/XML

XSD 2005, XML Schema Definition – The World Wide Web Consortium document
http://www.w3.org/XML/Schema

XSL 2005, The Extensible Style Sheet Family – The World Wide Web Consortium
http://www.w3.org/Style/XSL

Appendix A

MPDL CORBA Example

Following is the MPDL script, which defines the items discussed in Section 4.4. Please refer to that section for an explanation of the definitions and keywords.

```
// ********************************************************************************
// OMG CORBA (IIOP) Version 1.0
//
// Items surrounded in % symbols are:
//                      internal TUBE variables (eg. buffer length - %buffer_length%)
// Items surrounded in $ symbols are :
//                      user-defined variables (eg. ObjectKey --> $ObjectKey$)
//
//                      The user-defined variables are expected to be found in either;
//                                      - The End Point Resolution Table (EPRT)
//                                      - The XFORM map for protocol mapping rules (different to
//                                        marshalling rules)
//
// The generated PIM will look in both of these repositories and generate a
// runtime error if the name is not found.
// ********************************************************************************

#define GIOPID "GIOP"               // first four bytes of CORBA message
#define HOST “host”                 // constant for host
#define PORT “port”                 // constant for port

protocol CORBA
{
        typedef sequence<octet, 3> reserved;
        typedef sequence<octet> olist;
        typedef sequence<octet, 4> GIOP_MAGIC;
        //
        // This is an arbitrary sequence of bytes
        // This is referenced in the bufferFormat
        // statement below.
        //
        declare byteSequence
        {
         int %num_bytes%;           // no of bytes in olist
         olist bytes;
        };

        //
        // An idl sequence<..., size>
        // This is a bounded (fixed-size) sequence
        // The only difference between an array and a sequence
        // is that arrays don't have a length encoded because
        // their size is fixed!
        // This is referenced in the bufferFormat
        // statement below.
        //
        declare array
        {
         int %array_size%;          // no of items in sequence, arrays don't have this encoded
                                    // although, we can calculate it at runtime
         olist bytes;               // actual sequence of items (can be simple or complex)
        };
```

```
//
// An idl sequence<...>
// This is an un-bounded sequence (see array above for a bounded sequence)
// This is referenced in the bufferFormat
// statement below.
//
declare sequence
{
  int %sequence_size%;     // no of items to read/write
  olist bytes;             // the items as defined in IDL
};

//
// CORBA uses a hybrid (between C & Pascal) String structure
// The length precedes the bytes (as in Pascal) and the String is zero (null) terminated (as in C)
// Therefore the number of bytes to write and read is always one more than the actual string
// length.
//
// define an nString – a null terminated string
//
declare nString
{
 int %num_bytes% + 1;          // length of string_bytes (incl. null)
 olist string_bytes;           // the actual bytes of the string
 init octet null_byte = 0;     // the terminating null
};

//
// declare a CORBA Object Reference
//
declare objectDef
{
  nString repo_id;                    // repository-id of object
  int     profile_count;              // number of profiles in reference
  int     profile_id;                 // id of profile
  int     length;                     // length of following stream
  short   version;                    // IIOP version for this profile
  nString host;                       // Host for this object
  short   port;                       // Port for this object
  byteSequence object_key;            // Object key – includes length and byte[]
};

//
// define how to handle a (CORBA) System Exception (ref. CORBA spec.)
//
declare systemException
{
 nString exc_repoid;           // repository-id of Exception
 int minor_code;               // minor-code
 int completion_status;        // completion-status
};
```

```
//
// This is how to treat a buffer - payload
// This item is referenced using the keyword "buffer"
// in any subsequent declarations below
// This declaration statement can only appear once!!!!
//
bufferFormat
{
 STRING = nString;              // marshal a string
 BYTESEQ = byteSequence;        // marshal a buffer of bytes
 ARRAY = array;                 // marshall a sequence<..., size>
 OBJECT=objectDef;              // an Object definition
                                // all other items are assumed to be native or constructed
                                // from those above
 SEQUENCE=sequence;             // marshal an un-bounded sequence (sequence<…>)
};


//
// The following control clause uses the reply_status member of the request body
// to perform some decision-making. The payload in the response message may be
// either of four (4) different types depending on the value of  reply_status.
//          0 - normal payload as per MDR definition of operation
//          1 - a USER defined EXCEPTION as defined in the MDR entry
//          2 - a SYSTEM EXCEPTION of a fixed format
//          3 - an OBJECT_REFERENCE as encoded for a
//          LOCATION_FORWARD response (see CORBA spec.)
//
control
{
   switch(%reply_status%)
  {
      case 0:
          buffer = body;                    // follow MDR
      case 1:
          buffer = USER_EXCEPTION;          // follow Exception in MDR
      case 2:
          buffer = systemException;         // use declared structure
      case 3:
          buffer = objectDef;               // use declared structure
  }
};

//
// The “external” clause defines our own CDR marshalling class.
// This class implements interface TUBE.commsBuffer and supplies methods to marshal
// native data types using CDR encoding (refer to CORBA spec).
// When we have to marshal an int. The DAM will call read_int or write_int on this
// class
//
external
{
   codec_class = “MYORB.marshaller.CDRBuffer”;          // the full classname
};

// Define a Request
Request
{
 GIOPReqMessage message;
};
```

```
// Request message
struct GIOPReqMessage
{
 GIOPHdr hdr;
 int %buffer_length%;                // the length of the following buffer (body)
 GIOPReqBody body;
};

// Common GIOP header
struct GIOPHdr
{
 init GIOP_MAGIC GIOPId =GIOPID;     // 'G"I"O"P' - first 4 bytes
 init octet majver = 1;              // major version, default 1
 init octet minver = 0;              // minor version, default 0
 init octet flags = %endian%;        // the endian-ness of the host
 init octet msg_type = %isResponse%; // built-in flag determines message type
};

//
// This is how a service context is encoded
//
declare ServiceContext
{
 int context_id;
 byteSequence contextData;
};



// This is a list (sequence) of service contexts – will generate a loop wrapper
// around the declared code-block
typedef sequence<ServiceContext> contextList;

// Request body
struct GIOPReqBody
{
 contextList clist;                  // sequence of ServiceContexts
 int       %request_id%;             // request-id
 octet     %expect_resp%;            // do we expect a response
 reserved res;                       // 3 reserved bytes
 byteSequence $objectKey$;           // another byte sequence
 nString %operation%;                // declared type (NULL terminated string)
 byteSequence req_principal;         // another byte sequence
 buffer params;                      // a buffer, which can contain various
                                     // parameters (native/constructed) follows MDR format
                                     // uses bufferFormat clause above
};

// Define a Response
Response
{
  GIOPRespMessage response;
};

// Response message
struct GIOPRespMessage
{
 GIOPHdr hdr;
 int       %buffer_length%;          // the length of the following buffer (body)
 GIOPRespBody body;
};
```

```
        // Response body
        struct GIOPRespBody
        {
         contextList clist;                    // a byte sequence of the form <length><bytes....>          int
        %request_id%;              // request-id
         int        %reply_status%;            // a reply code, identifies the response format
         buffer response;                      // a buffer, which can contain various
                                               // parameters (native/constructed) follows MDR format
                                               // uses bufferFormat clause above
        };

        //
        // These are values that are encoded into the EPRT
        // The generated TIM will look for these items in the EPRT entry
        //
        endPoint : "TCP"                       // The transport (eg. TCP, HTTP, JMS etc)
        {
        //
        // These are transport and protocol-specific items
        //
        string host = HOST;                                    // This is the host for the object
        string port = PORT;                                    // This is the port on the host
        };
};
```

Appendix B

MPDL SOAP example

```
// ********************************************************************************
// W3C SOAP (Simple Object Access Protocol) defined in TUBE MPDL
// ********************************************************************************

#define XML_SIG "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
#define ENV_STR "<SOAP-ENV:Envelope xmlns:SOAP-
ENV=\"http://schemas.xmlsoap.org/soap/envelope/\">"
#define BODY_STR "<SOAP-ENV:Body SOAP-
ENV:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\">"
#define ENDBODY "</SOAP-ENV:Body>"
#define ENDENV "</SOAP-ENV:Envelope>"
// URL is defined in EPRT entry
#define NS "<" + %operation% + " xmlns=\"" + $URL$ + "\"" + ">"
#define RQHEAD "<" + %operation% + "Request>"
#define RSPHEAD "<" + %operation% + "Response>"
#define RQEND "</" + %operation% + "Request>"
#define RSPEND "</" + %operation% + "Response>"

#define NSEND "</" + %operation% + ">"

// This protocol "SOAP"  is encoded by a built-in TUBE marshalling class for "XML"

#define CODEC "TUBE.DAM.Marshall.PIM.XMLBuffer"
#define HOST “host”                    // constant for host
#define PORT “port”                    // constant for port

protocol SOAP
{
        external
        {
          codec_class = CODEC;
        };

        // request specific header
        struct SOAPReqHeader
        {
                string sig expr(XML_SIG);
                string env expr(ENV_STR);
                string bdy expr(BODY_STR);
                string ns expr(NS);
                string rqh expr(RQHEAD);
        };

        // common trailer
        struct SOAPTrailer
        {
                string xx expr(NSEND);
                string yy expr(ENDBODY);
                string zz expr(ENDENV);
        };
```

```
        // response specific header
        struct SOAPRespHeader
        {
                string sig expr(XML_SIG);
                string env expr(ENV_STR);
                string bdy expr(BODY_STR);
                string ns expr(NS);
                string rsph expr(RSPHEAD);
        };

        struct SOAPRequestMessage
        {
                SOAPReqHeader rhdr;
                buffer req;
                string rqe expr(RQEND);
                SOAPTrailer rtail;
        };

        struct SOAPResponseMessage
        {
                SOAPRespHeader hdr;
                buffer resp;
                string rse expr(RSPEND);
                SOAPTrailer tail;
        };

        // Define a Request
        request
        {
                SOAPRequestMessage reqm;
        };

        // Define a Response
        response
        {
                SOAPResponseMessage respm;
        };

        //
        endPoint : "HTTP"                       // The transport (eg. TCP, HTTP, JMS etc)
        {
        //
        // These are transport and protocol-specific items
        //
        string host = HOST;                                     // This is the host for the object
        string port = PORT;                                     // This is the port on the host
        };
};
```

Appendix C

MPDL HTML example.

```
/* ---------------------------------------------------------------------------
// The internal TUBE STRBuffer class implements the commsBuffer interface
// and handles the reading and writing of the payload. TUBE handles
// all the complex data structure traversals.
//
// The user doesn't need to write ANY code!
//
// ---------------------------------------------------------------------------*/

// header
#define HDR "<HEAD>"
#define ENDHDR "</HEAD>"
#define BODYSTART "<BODY BGCOLOR=\"LIGHTBLUE\">"
#define BODYEND "</BODY>"
#define REQTITLE "<H1><FONT COLOR=\"DARKGREEN\">" + %interface% + " : " + %operation% + "
Request </FONT></H1>"
#define RESPTITLE "<H1><FONT COLOR=\"DARKGREEN\">" + %interface% + " : " + %operation% + "
Response </FONT></H1>"
// an individual item within a complex item or stand-alone
#define BODYLINE "<TR><TD><H2><FONT COLOR=\"DARKBLUE\">" + %name% + " : " +
"</FONT><FONT COLOR=\"RED\">" + %value% + "</FONT></H2></TR></TD>"
#define START "<HTML>"
#define END "</HTML>"
// start of a complex item
//#define CSTART "<TABLE BORDER=\"10\" BGCOLOR=\"LIGHTBLUE\" CELLPADDING=\"4\"
<TH><H3><FONT=\"DARKBLUE\">" + %name% + "</H3></TH></FONT>"
#define CSTART "<TABLE BORDER=\"10\" BGCOLOR=\"LIGHTBLUE\" CELLPADDING=\"4\"
<TH><H3><FONT=\"DARKBLUE\">" + %name% + "</H3></TH></FONT>"
// end of a complex item
#define CEND "</TABLE><TABLE>"
// start of a sequence
#define SEQSTART "<TABLE BORDER=\"10\" BGCOLOR=\"GREEN\" CELLPADDING=\"4\"
<TH><H2>" + "Table size : " + %sequence_size% + "</H2></TH>"
// end of a sequence
#define SEQEND "<FONT BGCOLOR=\"LIGHTBLUE\"></FONT></TABLE>"
// an individual item within a sequence
#define SEQITEM "<TR><TD BGCOLOR=\"PURPLE\"><H3>" + %name% + "</TD>      <TD
BGCOLOR=\"GREEN\">" + %value% + "</TD></TR></H3>"

//
// uses internal STRBuffer class for reading/writing
//
#define CODEC "TUBE.DAM.Marshall.PIM.STRBuffer"

protocol HTML
{
        external
        {
          codec_class = CODEC;
        };
```

```
        // This is how to process a member of the payload
        bodyMember
        {
                string detail expr(BODYLINE);           // variable line definition
        };

        // How to handle the start of a complex item
        complexStart
        {
                string cs expr(CSTART);
        };

        // How to handle the end of a complex item
        complexEnd
        {
                string ce expr(CEND);
        };

        // What to do at the start of a sequence / array
        sequenceStart
        {
                string ss expr(SEQSTART);
        };

        // What to do at the end of a sequence/array
        sequenceEnd
        {
                string se expr(SEQEND);
        };

        // How to handle each item within a sequence
        sequenceItem
        {
                string si expr(SEQITEM);
        };

        // Define a Request
        request
        {
                string rqhs expr(START);
                string rqhe expr(HDR);
                string tt expr(REQTITLE);
                string eh expr(ENDHDR);
                string rqh expr(BODYSTART);
                buffer req;
                string rbe expr(BODYEND);
                string rqe expr(END);
        };

        // Define a Response
        response
        {
                string rsphs expr(START);
                string rsphe expr(HDR);
                string tte expr(RESPTITLE);
                string ehe expr(ENDHDR);
                string rspb expr(BODYSTART);
                buffer resp;
                string rspbe expr(BODYEND);
                string rspe expr(END);
        };

};
```

Appendix D

MPDL definition for encrypted XML protocol.

```
// ------------------------------------------------------------------------
// Test creating a <type="int"> style protocol
//
// The internal TUBE STRBuffer class implements the commsBuffer interface
// and handles the reading and writing of the payload. TUBE handles
// all the complex data structure traversals.
//
// The user doesn't need to write ANY code!
//
// This file utilises the "external" feature to invoke:
//                                  - encryption post-MDR traversal
//                                  - decryption pre-MDR traversal
//
// The encryption and decryption methods are supplied in user-defined classes.
//
// ------------------------------------------------------------------------

// request header
#define RQHDR "<" + %operation% + "Request>"
// response header
#define RSPHDR "<" + %operation% + "Response>"
// body (payload) members
#define LINE "<" + %name% + " type=\"" + %type%  + "\">" + %value% + "</" + %name% + ">"
// start of a complex item
#define CSTART "<" + %name% + ">"
// end of a complex item
#define CEND "</" + %name% + ">"
// request trailer
#define TAIL "</" + %operation% + "Request>"
// respose trailer
#define TAIL2 "</" + %operation% + "Response>"
// start of a sequence
#define SEQSTART "<" + %name% + ".size>" + %sequence_size% + "</" + %name% + ".size>"

// cryptography class definitions
#define CRYPTO_IN "TUBE.DAM.Marshall.testCodec"
#define CRYPTO_OUT "TUBE.DAM.Marshall.testCodec"                // same class
#define DECRYPT "read"
#define ENCRYPT "write"
// this class implements TUBE.DAM.Marshall.commsBuffer
#define CODEC "TUBE.DAM.Marshall.PIM.STRBuffer"
//
// uses internal STRBuffer class for reading/writing
//
protocol TestNew
{
        // here are the definitions for the classes that will be invoked
        external
        {
                pre_class = CRYPTO_IN;          // load before un-marshalling
                post_class = CRYPTO_OUT;        // load after marshalling
                pre_method = DECRYPT;           // call before un-marshalling
                post_method = ENCRYPT;                  // call before un-marshalling
```

```
        codec_class = CODEC;            // call during marshalling/un-marshalling
    };

    // This is how to process a member of the payload
    bodyMember
    {
        string detail expr(LINE);       // variable line definition
    };

    // How to handle the start of a complex item
    complexStart
    {
        string cs expr(CSTART);
    };

    // How to handle the end of a complex item
    complexEnd
    {
        string ce expr(CEND);
    };

    // What to do at the start of a sequence / array
    sequenceStart
    {
        string ss expr(SEQSTART);
    };

    // Define a Request
    request
    {
        read:CRYPTO_IN;                 // call when un-marshalling
        string rqh expr(RQHDR);
        buffer req;
        string rqt expr(TAIL);
        write:CRYPTO_OUT;               // call when marshalling
    };

    // Define a Response
    response
    {
        read:CRYPTO_IN;                 // call when un-marshalling
        string rsph expr(RSPHDR);
        buffer resp;
        string rst expr(TAIL2);
        write:CRYPTO_OUT;               // call when marshalling
    };
};
```

Appendix E

MPDL definition for JMS.

```
// ********************************************************************************
// JMS - based message handler
// ********************************************************************************

#define CLASSZ "TUBE.DAM.DAMBuffer"

#define INQ "Input"
#define OUTQ "Output"
#define TOUT "Timeout"

protocol JMS
{
        //
        // The “external” clause defines our own binary marshalling class.
        // This class implements interface TUBE.commsBuffer and supplies methods to marshall
        // native data types using binary encoding
        // When we have to marshall an int. The DAM will call read_int or write_int on this
        // class
        //
        external
        {
            codec_class = CLASSZ;                    // the full classname
        };

        // Define a Request
        request
        {
                buffer req;
        };

        // Define a Response
        response
        {
                buffer resp;
        };

        // use generic MOM Transport Interface Module
        endPoint:"MOM"
        {
                string Input = INQ;
                string Output = OUTQ;
                string Timeout = TOUT;
        };
};
```

Appendix F

Transport Implementation Module for JMS.

```
//============================================================================
// TUBE-Auto generated transport handler for: Test from file: JMS.mpdl
// Generated by TUBE MPDLCompiler V 1.2 --- Mon Jan 19 16:19:58 EST 2004
//
// ***** Modified to insert comments ******
//============================================================================

package TUBE.TMS.TIM;

import TUBE.EPRTProcessor;
import TUBE.TMS.connection;

import javax.jms.*;
import javax.naming.*;

public class JMS_TIM implements TMSTim
{
    private Context                 jndiContext = null;
    private QueueConnectionFactory  queueConnectionFactory = null;

    private Queue                   sendQueue = null;
    private QueueConnection         sendConnection = null;
    private QueueSession            sendSession = null;
    private QueueSender             queueSender = null;

    private Queue                   recvQueue = null;
    private QueueConnection         recvConnection = null;
    private QueueSession            recvSession = null;
    private QueueReceiver           queueReceiver = null;

    private BytesMessage            message = null;
    private boolean                 m_DebugOn = false;

    private String                  m_correlID = null;      // for sequencing messages
    private long                    m_Timeout;              // timout period for responses
    private boolean                 m_Waiting = false;      // not waiting for response



    /**
     *
     * @param args     the queue to use
     */
    public JMS_TIM()
    {
        System.out.println("Creating JMS_TIM......");
    }

   /*
    * Receive messages
    */
    public byte[] receiveMessage()
    {
               if (queueReceiver == null)
                      return null;

               byte retbuff[] = null;
               long now = System.currentTimeMillis();
               if (m_Timeout == 0)
          m_Timeout = 60000;

               long then = now + m_Timeout;

               try
               {
         System.out.println("JMS_TIM.receiveMessage() - waiting on " +
recvQueue.getQueueName());
          while (true)
          {
              Message m = queueReceiver.receive(1000);
```

```
            if (m != null)
            {
                    if (m instanceof BytesMessage)
                    {
                        BytesMessage bm = (BytesMessage) m;
                        System.out.println("JMS_TIM.receiveMessage()- Reading byte message:
");
                        retbuff = new byte[100];
                        // is it ours?
                        if (m_correlID.equals(bm.getJMSCorrelationID()))
                        {
                          System.out.println("JMS_TIM.receiveMessage(): CorrelationID = " +
m_correlID);
                          bm.readBytes(retbuff);
                          m_Waiting = false;
                          break;
                        }
                    }
                    else
                    {
                        System.out.println("JMS_TIM.receiveMessage(): not a BytesMessage");
                        break;
                    }
            }

            //
            // Waiting for a response, check if timed-out
            //
            if (m_Waiting)
            {
//                if (m != null)
//                    System.out.println("JMS_TIM.receiveMessage(): found CorrelationID = " +
m.getJMSCorrelationID());

//                System.out.println("JMS_TIM.receiveMessage():now=" + now + ", then=" + then
+ ", timeout=" + m_Timeout);
                now = System.currentTimeMillis();
                if (now > then)
                {
                    System.out.println("Timeout in JMS_TIM.receiveMessage():" + (m_Timeout /
1000) + " secs. elapsed - Aborting.");
                    return null;
                }
            }
          }
               }
               catch (Exception e)
               {
          System.out.println("Exception occurred in JMS_TIM.receiveMessage(): " +
e.toString());
          e.printStackTrace();
                return null;
               }

               return retbuff;
    }

  /**
  * terminate the connection
  */
    public void close()
    {
        if (sendConnection != null)
        {
            try
            {
                sendConnection.close();
                if (recvConnection != null)
                      recvConnection.close();
            }
            catch (JMSException e)
            {
                System.out.println("Exception occurred in close(): " + e.toString());
            }
        }
    }

    /**
     * print a message in debug mode
     */
```

```
    private void logMessage(String msg)
    {
        if (m_DebugOn)
            System.out.println(msg);
    }

    /**
     *
     * @return the byte array received
     */
    public byte[] receive()
    {
        return receiveMessage();
    }

    /** @param buff the message
     */
    public boolean send(byte[] buff)
    {
      try
      {
                if (sendSession == null)
                        return false;

          message = sendSession.createBytesMessage();
          m_correlID = "ID:" + System.currentTimeMillis();
          System.out.println("JMS_TIM.send() - creating CorrelationID: " + m_correlID);
          message.setJMSCorrelationID(m_correlID);
          message.writeBytes(buff);
          System.out.println("JMS_TIM.send()");
          queueSender.send(message);
          // need to test if one-way invocation
          // don't set if it is
          m_Waiting = true;             // we are waiting for a response
          return true;
      }
      catch (Exception e)
      {
          System.out.println("Exception occurred in send(): " + e.toString());
          e.printStackTrace();
                return false;
      }

//      return false;
    }


     private void setRecvQueueName(String qname)
     {
        System.out.println("Queue name is " + qname);

        try
        {
            jndiContext = new InitialContext();
        }
        catch (NamingException e)
        {
            System.out.println("Could not create JNDI API " + "context: " + e.toString());
            //System.exit(1);
        }

        /*
         * Look up connection factory and queue.  If either does
         * not exist, exit.
         */
        try
        {
            // pick-up our requests from here
            recvQueue = (Queue) jndiContext.lookup(qname);
            recvConnection = queueConnectionFactory.createQueueConnection();
            recvSession = recvConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            queueReceiver = recvSession.createReceiver(recvQueue);
            recvConnection.start();
        }
        catch (NamingException e)
        {
            System.out.println("JNDI API lookup failed: " + e.toString());
            //System.exit(1);
        }
        catch (JMSException je)
```

```
        {
          System.out.println("Exception in setRecvQueueName(String queue_name)" +
je.toString());
        }
     }

     private void setSendQueueName(String qname)
     {
        System.out.println("Queue name is " + qname);

        try
        {
            jndiContext = new InitialContext();
        }
        catch (NamingException e)
        {
            System.out.println("Could not create JNDI API " + "context: " + e.toString());
            //System.exit(1);
        }

        /*
         * Look up connection factory and queue.  If either does
         * not exist, exit.
         */
        try
        {
            // put responses here
            queueConnectionFactory = (QueueConnectionFactory)
jndiContext.lookup("QueueConnectionFactory");
            sendQueue = (Queue) jndiContext.lookup(qname);
            sendConnection = queueConnectionFactory.createQueueConnection();
            sendSession = sendConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            queueSender = sendSession.createSender(sendQueue);
            sendConnection.start();
        }
        catch (NamingException e)
        {
            System.out.println("JNDI API lookup failed: " + e.toString());
            //System.exit(1);
        }
        catch (JMSException je)
        {
          System.out.println("Exception in PAQueueReceiver(String queue_name)" +
je.toString());
        }
     }


     public void setUp(EPRTProcessor eprt)
     {
               // These values come from the EPRT
         String sendQueue = eprt.getElementValue("JMS", "Output");     // we write to this
         String recvQueue = eprt.getElementValue("JMS", "Input");      // we read from this
         String tout = eprt.getElementValue("JMS", "Timeout");
         if (tout == null)
         {
            m_Timeout = 60000;
         }
         else
             m_Timeout = Long.parseLong(tout) * 1000;

         setSendQueueName(recvQueue);
         setRecvQueueName(sendQueue);
     }
}
```

Appendix G

Complex mathServer interface IDL.

```
/*****************************************/
/* Test complex typedefs in TUBE idl    */
/*****************************************/

interface mathServer8
{
      typedef octet my_byte;
      typedef string my_string;
      typedef sequence<my_string> fred;
      typedef sequence<my_byte, 30> ralph;
      typedef sequence<my_string, 10> george;
      typedef long my_index;


      struct math_req
      {
       char op_code;
       my_index num1;
       long num2;
       fred f;
       ralph r;
      };

      typedef math_req my_MathReq;

      typedef sequence<math_req> junk;

      struct math_resp
      {
       long resp_num;
      };

      my_index add(in math_req mr);
      long sub(in math_req mr);
      long mul(in math_req mr);
      long div(in math_req mr);
      long test1(in math_req mr, in fred f);
      long test2(in math_req mr, inout ralph r);
      void test3(in math_req mr, inout math_resp r, inout george g);
      long test4(in junk jj);

};
```

Appendix H

MPDL Op-Codes for Protocol Implementation Modules

OP-Code	Description	Numeric Value
Native Type Handling Instructions		
READ_OCTET	Read a single octet (byte) from a source.	1000
WRITE_OCTET	Write a single octet (byte) to a target	1001
READ_OCTET_ARRAY	Read an array of octets (bytes) from a source.	1002
WRITE_OCTET_ARRAY	Write an array of octets (bytes) to a target.	1003
READ_SHORT	Read a short (2 byte) value.	2000
WRITE_SHORT	Write a short (2 byte) value.	2001
READ_INT	Read a Java int (C++ long).	3000
WRITE_INT	Write a Java int (C++ long).	3001
READ_LONG	Read a Java long (C++ long-long).	4000
WRITE_LONG	Read a Java long (C++ long-long).	4001
READ_DOUBLE	Read a double value.	5000
WRITE_DOUBLE	Write a double value.	5001
READ_FLOAT	Read a floating-point value.	6000
WRITE_FLOAT	Write a floating-point value.	6001
String Handling Instructions		
READ_STRING	Read a string value (used mainly for XML).	7000
READ_CSTRING	Read a CORBA string value (int, string, null).	7001
READ_JSTRING	Read a Java string value (short, string).	7002
READ_PSTRING	Read a Pascal string value (int, string).	7003
READ_NSTRING	Read a null terminated string value (for example, C/C++ string, null).	7004
WRITE_STRING	Write a string value (used mainly for XML).	8000
WRITE_CSTRING	Write a CORBA string value (int, string, null).	8001
WRITE_JSTRING	Write a Java string value (short, string).	8002
WRITE_PSTRING	Write a Pascal string value (int, string).	8003

OP-Code	Description	Numeric Value
WRITE_NSTRING	Write a null terminated string value (for example, C/C++ string, null).	8004
Sequence Handling Instructions		
READ_ARRAY	Read a fixed-length sequence of items.	10000
WRITE_ARRAY	Write a fixed-length sequence of items.	10001
READ_SEQUENCE	Read a variable-length sequence of items.	11000
WRITE_SEQUENCE	Write a variable-length sequence of items.	11001
LOOP	The start of a looping sequence.	14000
LOOP_END	The end of a looping sequence.	14001
Miscellaneous Instructions		
SAVE_POS	Save current buffer position.	100
SET_POS	Set buffer position.	101
SET_VAR	Set the value of an item.	102
ADD	Add two values.	103
SUB	Subtract one value from another.	104
SAVE_LEN	Save the length of the buffer.	105
WRITE_LEN	Write the length of the buffer.	106
EQ	Test for equality.	200
NEQ	Test for in-equality.	201
GT	Test if a > b.	202
LT	Test if a < b.	203
LTE	Test if a <= b.	204
GTE	Test if a >= b.	205
OR	Test for a OR b.	206
AND	Test for a AND b.	207
JUMP	Jump to the given LABEL	208
LABEL	The target of a JUMP instruction.	209
EXPR	Evaluate a string expression and perform substitutions.	220
BINEXPR	Evaluate a binary expression and perform substitutions.	221
INVOKE	Invoke an external class.	300
PUSH	Push a value onto the top of the stack.	800
POP	Pop a value off the top of the stack.	801
LOAD_BLOCK	Load a named Code-Block.	999
END_BLOCK	Signal the end of a block.	-1
USER_DEFINED	Start of a user-defined (declared) block.	13000
ASSIGN	Assign the value to current item.	15000

Table H.0-1: MPDL Op-codes for Protocol Implementation Modules

Appendix I

MPDL grammar

```
<specification>::= <protocol_spec> <semicolon> ;

<protocol_spec>::= “protocol” <identifier> “{“ <definitions> “}” ;

<definitions>  ::= + <definition> <end_point_opt> ;

<definition>   ::= <typedef>
                 | <declare_clause>
                 | <external_clause>
                 | <buffer_format_clause>
                 | <complex_start_clause>
                 | <complex_end_clause>
                 | <sequence_start_clause>
                 | <sequence_item_clause>
                 | <sequence_end_clause>
                 | <body_member_clause>
                 | <control_clause>
                 | <struct_type>
                 | <request_decl>
                 | <response_decl>
                 ;

<typedef>      ::= “typedef” <type_declarator> ;

<type_declarator>      ::= <simple_declarator>
                         | <sequence_type>
                         ;

<sequence_type> ::=     <sequence1> | <sequence2> ;

<sequence1>    ::= “sequence” “<“ <simple_type_spec>
                 |            “,” <numeric_constant> “>“
                 ;

<sequence2>    ::= “sequence” “<“ <simple_type_spec> “>“


<buffer_format_clause> “bufferFormat” “{“ <member_decls> “}“

<member_decls> ::= + <member_decl> “=” <identifier> <semicolon> ;

<member_decl>  ::= <string_buff_decl>
                 | <object_buff_decl>
                 | <array_buff_decl>
                 | <seq_buff_decl>
                 | <byte_seq_buff_decl>
                 ;

<string_buff_decl> ::= “STRING” ;

<object_buff_decl> ::= “OBJECT” ;

<array_buff_decl> ::= “ARRAY” ;

<seq_buff_decl>   ::= “SEQUENCE” ;
```

```
<byte_seq_buff_decl> ::= “BYTESEQ” ;

<declare_clause>  ::= “declare” <identifier> <decl_member_list> <semicolon>
;

<decl_member_list>::= *<declare_member> ;

<declare_member>  ::= <declarator> <identifier> <semicolon> ;

<declarators>   ::= * <declarator> ;

<declarator>    ::=     <simple_declarator>
                  |     <complex_declarator>
                  ;

<simple_declarator>     ::= <native_declarator> ;


<native_declarator>     ::= <floating_pt_type>
                          | <integer_type>
                          | <char_type>
                          | <boolean_type>
                          | <octet_type>
                          ;

<complex_declarator>    ::= <identifier> ;

<floating_pt_type>      ::= <float> | <double> ;


<float>         ::=     “float” ;

<double>        ::=     “double” ;

<integer_type> ::|      <int>
                  |     <long>
                  |     <short>
                  ;

<int>           ::=     “int” ;

<long>          ::=     “long”  ;

<short>         ::=     “short” ;

<char_type>     ::=     <char>  ;

<char>          ::=     “char” ;

<boolean_type> ::=      “boolean” ;

<octet_type>    ::=  ”octet” ;


<struct_type>   ::=  “struct” <identifier > <struct_body> ;

<struct_body>   ::=  “{“ <members> “}” <semicolon> ;

<members>       ::= *<member> ;


<member>        ::=     <native_member> “;”
                    |   <tube_member>
                    |   <user_member>
```

```
                    |  <init_member>
                    ;

<native_member>  ::= <simple_declarator> <identifier> ;

<tube_member>    ::= <simple_declarator> <tube_var_decl> ;

<user_member>    ::= <simple_declarator> <user_var_decl> ;

<init_member>    ::= "init" <simple_declarataor> <identifier> "="
<assign_var> ;

<assign_var>     ::= <const_decl>
                   | <var_decl>
                   ;

<var_decl>       ::= <tube_var_decl>
                   | <user_var_decl>
                   ;

<tube_var_decl> ::= "%" <tube_var> "%" ;

<user_var_decl> ::= "$" <identifier> "$" ;

<tube_var>       ::= <request_id>
                   | <expect_resp>
                   | <buffer_length>
                   | <endian>
                   | <is_response>
                   | <num_bytes>
                   | <reply_status>
                   | <sequence_size>
                   | <array_size>
                   | <count_kw>
                   | <target_tlv>
                   ;

<request_id>     ::= "request_id" ;

<expect_resp>    ::= "expect_resp" ;

<buffer_length> ::= "buffer_length" ;

<endian>         ::= "endian" ;

<is_response>    ::= "isResponse" ;

<num_bytes>      ::= "num_bytes" ;

<reply_status>   ::= "reply_status" ;

<sequence_size> ::= "sequence_size" ;

<array_size>     ::= "array_size" ;

<count_kw>       ::= "count" ;

<target_tlv>     ::= "target_tlv" ;

<string_expr>    ::= "expr" "(" <constant_expr_list> ")" <semicolon> ;

<constant_expr_list> ::= <constant_expr>
                       | <constant_expr> "+" <constant_expr_list>
                       ;
```

```
<constant_expr> ::= <numeric_constant>
                  | <string_constant>
                  | <var_decl>
                  ;

<external_clause> ::= “external” <external_body> ;

<external_body> ::= “{“ <external_members> “}” <semicolon> ;

<external_members> ::= *<external_member> ;

<external_member> ::= <codec_clause>
                    | <pre_clause>
                    | <post_clause>
                    | <pre_meth_clause>
                    | <post_meth_clause>
                    ;

<codec_clause> ::= “codec_class” <literal_assignment> ;

<pre_clause> ::= “pre_class” <literal_assignment> ;

<post_clause> ::= “post_class” <literal_assignment> ;

<pre_meth_clause> ::= “pre_method” <literal_assignment> ;

<post_meth_clause> ::= “post_method” <literal_assignment> ;

<literal_assignment> ::= “=” <string_literal> <semicolon> ;

<complex_start_clause> ::= “complexStart” <clause_body> ;

<clause_body> ::= “{“ <clause_members> “}” <semicolon> ;

<clause_members> ::= *<clause_member> ;

<clause_member> ::= <simple_declarator> <identifier> <clause_expr>
<semicolon>;


<clause_expr> ::= <string_expr>
                | <binary_expr>
                ;

<complex_end_clause> ::= “complexEnd” <clause_body> ;

<sequence_start_clause> ::= “sequenceStart” <clause_body> ;

<sequence_end_clause> ::= “sequenceEnd” <clause_body> ;

<sequence_item_clause> ::= “sequenceItem” <clause_body> ;

<body_member_clause> ::= “bodyMember” <clause_body> ;

<control_clause> ::= “control” <control_body> ;

<control_body> ::= “{“ <switch_stmt> “}” <semicolon> ;

<switch_stmt> ::= “switch” “(“ <var_decl> “)” <switch_body> ;

<switch_body> ::= “{“ <switch_cases> “}” ;
```

```
<switch_cases> ::= *<switch_case> ;

<switch_case> ::= “case” <numeric_constant> <colon> <switch_act> ;

<switch_act> ::= “buffer” “=” <buffer_opt> <semicolon> ;



<buffer_opt> ::= <body_opt>
               | <user_exc_opt>
               | <identifier>
               ;

<body_opt> ::= “body” ;

<user_exc_opt> ::= “USER_EXCEPTION” ;

<request_decl> ::= “request” <req_resp_body> ;

<response_decl> ::= “response” <req_resp_body> ;

<req_resp_body> ::= “{“ <req_resp_members> “}” <semicolon>;

<req_resp_members> ::= *<req_resp_member>
                     | <buffer_item>
                     ;

<buffer_item>  ::= “buffer” <identifier> ;

<req_resp_member> ::= <identifier> <identifier> <semicolon> ;
                    | <read_clause> <semicolon>
                    | <write_clause> <semicolon>
                    ;

<read_clause>   ::= “read” <colon> <string_constant> ;

<write_clause>  ::= “write” <colon> <string_constsnt> ;

<binary_expr>   ::= “bin” “(“ <constant_expr_list> “)” <semicolon>;

<end_point_opt> ::= ?
                    | <end_point_decl>
                    ;

<end_pont_decl> ::= <end_point_hdr> <end_point_body> <semicolon> ;

<end_point_hdr> ::= “endpoint” <colon> <string_constant> ;

<end_point_body> ::= “{“ <end_point_members> “}” ;

<end_point_members> ::= *<end_point_member> ;

<end_point_member> ::= <simple_declarator> <identifier> <semicolon> ;

<semicolon>     ::= “;” ;

<colon>         ::= “:” ;

<numeric_constant>     ::= *<number> ;

<string_constant>      ::= <dquote> <any> <dquote> ;

<number> ::= *[0-9] ;
```

```
<dquote> ::= “ ;

<any>    ::= *[any character] ;
```

List of Acronyms and Terms

API	**Application Programming Interface**
ASN.1	**Abstract Syntax Notation One**
BNF	**Backus-Naur Form**
CDR	**Common Data Representation**
COM	**Common Object Model**
CORBA	**Common Object Request Broker Architecture**
CS	**Constant Segment**
DAM	**Dynamic Adaptive Marshaller**
DCE	**Distributed Computing Environment**
DCOM	**Distributed Common Object Model**
DPT	**Distribution Priority Table**
EAI	**Enterprise Application Integration**
EPRT	**End-Point Resolution Table**
HTTP	**Hyper-Text Transfer Protocol**
IDL	**Interface Definition Language**
IIOP	**Internet Inter-ORB Protocol**
IOR	**Interoperable Object Reference**
J2EE	**Java 2 Enterprise Edition**
J2SE	**Java 2 Standard Edition**
Java-RMI	**Java Remote Method Invocation**
JMS	**Java Message Service**
MDR	**Module Definition Repository**
MDS	**Message Distribution Server**
MOM	**Message-Oriented Middleware**
MPDL	**Middleware Protocol Definition language**
OMG	**Object Management Group**
ORB	**Object Request Broker**
OSF	**Open Software Foundation**
PCM	**Protocol Control Module**
PDR	**Protocol Definition Repository**
PIM	**Protocol Implementation Module**
RPC	**Remote Procedure Call**
SNA	**System Network Architecture**
SOA	**Service-Oriented Architecture**
SOAP	**Simple Object Access Protocol**
TIM	**Transport Interface Module**
TLV	**Type Length and Value**
TMS	**Transport Mediation Server**
TUBE	**The Ubiquitous Broker Environment**
VDS	**Variable Definition Segment**
VVT	**Variable Value Table**
W3C	**World Wide Web Consortium**
XML	**Extensible Mark-up Language**
XSD	**XML Schema Definition**
XSL	**Extensible Style-Sheet Language**
XSLT	**Extensible Style-Sheet Language Transformations**