18

Intercessory Objects within Channels

Barry Kitson

Telecom Research Laboratories, 770 Blackburn Road, Clayton, Victoria, 3168, Australia

The management of interactions between components of a distributed system necessitates mechanisms for monitoring activity, and influencing behaviour, within *channels* binding those components. A low-level mechanism, based on *intercessory objects*, is considered here as a foundation for the construction of higher level management functions.

The role of intercessory objects in a distributed system is discussed, with emphasis on the underlying distributed processing environment and on the structure of the channel. The presence of intercessory objects introduces additional management requirements to the system, including a need for the distributed processing environment to provide support for the configuration of these objects within the channel. Restrictions placed on this configuration by distributed processing environment implementations are discussed, as is the value of generic interfaces for the implementation of reusable intercessory objects.

Keyword Codes: C.2.4; K.6.4
Keywords: Distributed Systems; System Management

1. INTRODUCTION

A *distributed processing system* (or *distributed system*) is a software application, typically consisting of components supported by a *Distributed Processing Environment* (DPE). A DPE provides basic infrastructure support for a distributed system, and is usually sufficiently generic to support a wide range of applications. Notable emerging DPEs include APM's ANSAware [1], OSF's DCE [2] and CORBA [3] implementations such as IBM's DSOM, Digital's ObjectBroker, Hewlett-Packard's ORB Plus and IONA's Orbix.

A DPE provides basic management facilities and resources to the system it supports and, more significantly, embodies aspects of an architectural model or framework which influences the system structure and system management. The architectural models by which distributed systems are constructed and managed are extremely important in realistic applications, as they provide a basis for integration and interworking between independently developed distributed systems. The standardisation of these models for open distributed processing systems is a significant activity in organisations such as the International Standards Organization (ISO) [4] and the Object Management Group (OMG).

One of many significant requirements of a management model for distributed systems is the need to manage run-time interactions between system components. To monitor and control these interactions, a management system requires notification of significant events during each interaction, and also requires access directly into the *channel* [5] binding sys-

K. Raymond et al. (eds.), *Open Distributed Processing*

tem components. Each of these management capabilities can be provided by introducing the concept of an *Intercessory Object* (IO) into the management model.

Intercessory objects are inserted into channels to providing basic access to information collection and control functions. Higher-level management functions spanning OSI fault, configuration, accounting, performance and security management functional areas [6] may be constructed from these basic services. IOs may also be used to provide base functionality for debugging distributed applications, and may be seen as a generalisation of the *transparency object* concept from work on the Reference Model for Open Distributed Processing (RM-ODP) [5].

Section 2 gives a brief overview of important features of object-based DPEs, and describes a model of channels into which intercessory objects are introduced in Section 3. These models are based heavily on RM-ODP work. Some early thoughts on implementation and configuration management issues are outlined in Section 4, before the application of IOs, and their relationships to transparency objects and other concepts, are discussed in Section 5.

2. DISTRIBUTED PROCESSING ENVIRONMENT FEATURES

A distributed processing environment provides the infrastructure on which components of a distributed system operate, and supports an architecture describing the nature of these components. Object-based architectures, such as that of RM-ODP, encapsulate components of distributed systems, allowing external access to internal state only via well-defined interfaces. Conforming DPEs support this object model and provide the communications infrastructure necessary for objects comprising a distributed application to interact. This includes support for addressing and binding functions.

2.1. The object model

The object model defined in architectures such as ISO's Reference Model for Open Distributed Processing (RM-ODP) [5] and adopted into the Telecommunications Information Networking Architecture (TINA) [7] is depicted in Fig. 1. Objects encapsulate state, and provide services to other objects via the interfaces they support. Typically, each operational interface consists of at least one *operation*, equivalent to a member function or method in traditional object models. The *interface type* specifies the operations available on instances of that type, and an *operation type* specifies the types of argument and return parameters. Interactions in these models commonly adopt Remote Procedure Call (RPC) semantics, blocking the invoking thread during normal execution of remote operations. Non-blocking operations, equivalent to message passing, are also provided.

For each interaction, a *client* and a *server* object may be identified. The client object initiates an interaction by invoking an operation on an interface of the server. Communications between these objects is handled by the DPE, via a *channel* [5]. The channel may consist of a number of important component objects, as outlined in Section 2.3.

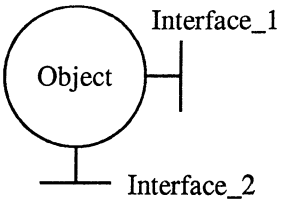


Figure 1: A multi-interface object. This shows the object model of TINA and RM-ODP.

2.2. Generic interfaces

Within OMG work, the Common Object Request Broker Architecture introduces the concept of a Dynamic Invocation Interface (DII) [3], and recent contributions have defined the closely related Dynamic Skeleton Interface (DSI) [8]. The DII allows objects, acting as clients, to access interfaces without requiring that associated client stubs be available. The DSI provides analogous capabilities on the server side of a binding. A server can use the DSI to service calls to arbitrary interface types.

In programming languages such as C and C++, the DII and DSI each appear as an Application Programming Interface (API) expressed in the native language. The DII includes functions to specify an interface to be used, operations to be called, arguments to be passed and values to be returned. The DSI is similar, but with slightly different addressing semantics. In languages where types are treated as first-class entities, such as Scheme, there is much greater scope for representing all interfaces consistently at the application level, but the underlying facilities are invariant across language mappings.

These APIs are implementations of a class of operational interface which will be termed a *generic interface* (GI) here. A generic interface may be specified in CORBA Interface Definition Language (IDL), for example, and has the important property that it is capable of representing operation invocations on any other interface type that can be represented in the model. The operations of a GI are equivalent to those at the lower levels of stub marshalling routines. Specifically, the marshalling routines for base and standard types, and for operation invocations, are represented in a GI.

For management systems, a standard GI provides a significant capability. At the simplest level, it allows information specific to the interactions between particular managing and managed systems to be represented independently of the interfaces by which they communicate. The GI is capable of supporting interactions with any interface type. It is the property of genericity, and the fact that mappings between any *application-specific* interface type and the GI type can be made, that is the value of a GI in the context of this discussion. This will be expanded in Section 4.1.

2.3. Binding objects and the channel

The channel between the client object and the server object is created by the distributed processing environment during the binding of the client to the server's interface. However, it should be noted that many of the channel components are produced at compile time and may be shared for implementation efficiency. The channel will typically contain marshalling and de-marshalling functions and other protocol-specific communications mechanisms. Although these components are not normally implemented with all the support given to application level objects by the DPE, it is convenient to depict them as logical objects, as shown in Fig. 2.

The channel objects of Fig. 2 form stacks of functionality associated with the client and server sides of the SInt service interface. A particular DPE may implement only some of these objects and may not, for reasons of efficiency, encapsulate functionality into objects identifiable as those shown in the figure. However, the objects of Fig. 2 represent specific channel capabilities germane to discussions in following sections.

The Proxy and GI Proxy objects shown in Fig. 2 provide encapsulations of the addresses of, or pointers to, the next objects in the sequence supporting interactions in the channel. As such, the proxies identify points were control flow may be redirected. This concept helps clarify discussions below.

The Proxy objects are dependent on the type of the interface being bound. In the example, the service interface is identified as having type SInt. Each Proxy object supports an interface of type SInt and makes use of a similar interface. Each GI Proxy object is similarly associated with a generic interface type GI, which is standard throughout the DPE.

A Proxy object, or more correctly the SInt interface of a Proxy object, will typically appear to the application programmer as a programming language dependent representation of the server, instantiated within the client code. The Proxy object makes use of the marshalling stub object, Mshl Stub, to perform the high level marshalling of arguments and de-marshalling of return parameters associated with the operation invocation. The exact nature of the Mshl Stub object is implementation dependent. In most currently available DPEs, these stubs involve relatively complex operations determined at compile time, based on information contained in the SInt interface type definition. However, given appropriate programming language support there is no reason why this same functionality cannot be developed at run-time.

The Mshl Stub object is responsible for mapping the high level SInt representation of the interface into a low level representation such as that provided by a GI. Some DPEs, particularly those based on CORBA, already support an explicit DII at the application level. Use of such a DII is equivalent to direct use of the GI interface by the client, as shown in Fig. 2. Even if the DPE does not make a DII available to applications, functionality equivalent to the GI interface will exist in the marshalling structures of the channel.

In some DPE implementations, it may only be possible to implement the Mshl Stub object if the SInt type is known at compile time. This depends on programming language support, but it should be noted that some implementations will allow the construction of this object at run-time, and others will not.

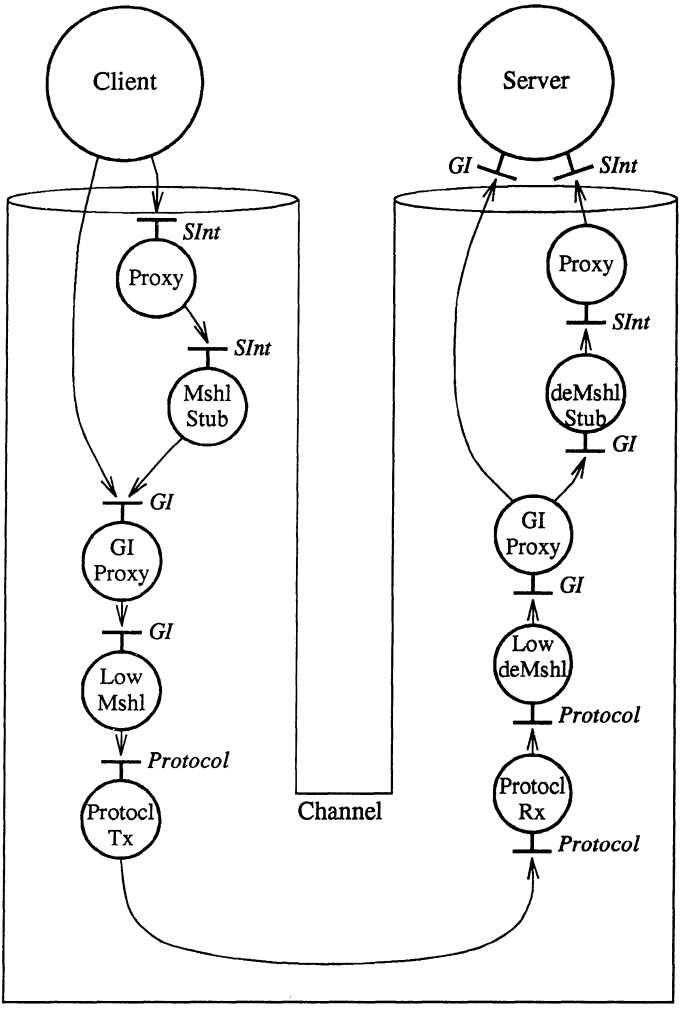


Figure 2: Channel binding a client to a server interface. The channel consists of a number of logical objects forming stacks below the client and server. A particular distributed processing environment may implement the functionality of only some of these objects when binding the client and server of service interface type `SInt`.

The GI Proxy object performs a function similar to the Proxy object described above, allowing another level of indirection. The addressing encapsulated by this object may be used to select protocols for communications, for example. The LowMshl object provides a protocol-specific mapping from the GI interface type to the representation used by the underlying communications protocol object (Protocol). Such protocol objects are responsible for the actual transmission of information across the network.

On the server side of the channel, there is a similar stack of logical objects. It should be noted that Proxy and GI Proxy objects on the server side perform identical functions to their counterparts on the client side. In fact, the implementations of these objects on each side of the channel need not be different.

Another important point is that the server in the example may be offering the SInt interface in its own right, with a complete mapping of the interface type into the application code, or alternatively, as a specific set of supported GI functions. In the former case, a high level de-marshalling stub, deMshlStub is required, while in the latter, this and the Proxy above are not required.

For example, a server might implement a large range of functionality internally, and may decide at run-time to make only some of this functionality available to clients. The exact functionality, or operations available on an interface for the use of clients, may be dependent on the nature of those clients and security or other constraints. The server can use the GI to give access to this functionality as determined at run-time. The client may be unaware of this, and may view the interface as an instance of the SInt type, for example, binding to it via the channel shown in Fig. 2.

3. INTERCESSORY OBJECTS

The purpose of intercessory objects is to allow almost arbitrary functionality to be added to the channel binding clients to server interfaces. Ideally, an application programmer should be able to implement an intercessory object just as any application level object in the distributed system, to provide arbitrary complexity of function. It should also be possible to implement an IO independently of any particular distributed system, so that reusable libraries of IOs may be assembled and effectively utilised. These requirements place restrictions on the implementations of intercessory objects themselves, and also on the mechanisms by which they may be inserted into the channel on the client or server side.

As an example, consider an intercessory object which notifies a managing system whenever RPC invocations of operations are commenced or completed on an interface of type SInt. An IO which performs this function is depicted in Fig. 3.

Such an intercessory object may be inserted into the channel on either the client or server side, and may perform actions without affecting the operation of the client or server themselves.

When inserting an intercessory object into a channel stack, it is necessary to allow for the inclusion of additional intercessory objects in future. The mechanism for including objects in a sequence supporting a channel involves breaking the sequence, redirecting calls to the intercessory object, and then directing calls from the intercessory object back through the objects in the channel after the insertion point. This redirection is central to

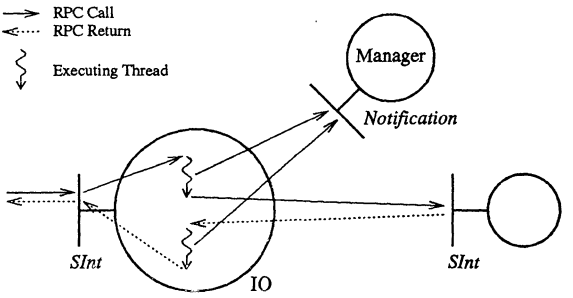
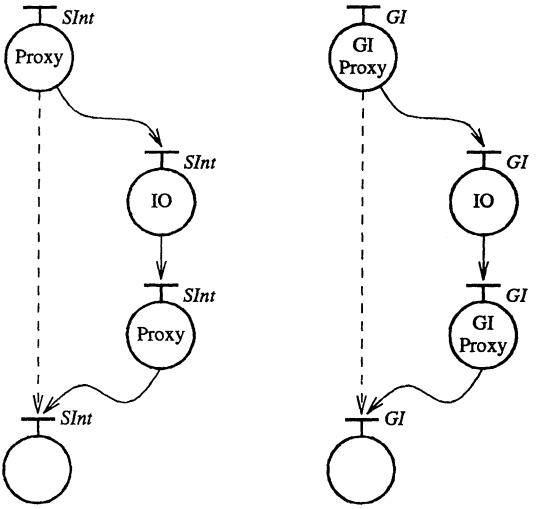


Figure 3: Intercessory object used to monitor interactions. The intercessory object `IO` executes a short sub-thread, or sequence of operations, before and after passing an invocation on to another object in the channel stack. In each of these sub-threads the management system is notified via a call on a `Notification` interface. (The returns from these RPCs are not shown for simplicity.)



(a) Specific Interface (b) Generic Interface

Figure 4: Insertion of an intercessory object. The insertion of an intercessory object `IO` in a client stack is shown (a) where the object `IO` is implemented in a fashion specific to the `SInt` interface type bound by the channel and (b) where the `IO` object is generic, and is implemented in terms of the generic interface type `GI`. The initial sequence of objects in the channel is indicated by the dashed arrow.

the functionality of the Proxy objects described in Section 2.3. Insertion can only take place after such a Proxy object, and in order to allow for future insertions further in the channel after a given intercessory object, another Proxy object should be added after each inserted intercessory object. The insertion process for this object is shown in Fig. 4(a).

An intercessory object to perform activities such as monitoring would typically be constructed with a view to reuse. The requirement for such an object is so common that a reusable version may even be supplied as part of the DPE management infrastructure. In any case, the object implementation is independent of the actual interface type being bound, and would therefore be most easily implemented as an object which both supports and uses interfaces of the generic GI type. Insertion of such an object into the stack supporting the client involves redirection of the thread of control to the intercessory object. The redirection is performed by a GI Proxy object in this case, since a GI Proxy is capable of calling the GI interface supported by the intercessory object. The insertion process is shown in Fig. 4(b).

Similar insertion of intercessory objects is obviously possible in the server stack, by redirecting Proxy or GI Proxy objects to call intercessory objects. In the case of the monitoring example considered above, it would obviously be usual to implement the relevant intercessory objects in terms of GI interfaces to assist with reuse.

The same approach may be taken in the case of interaction management activities which involve the intervention of the management system in interactions themselves. Such intervention may include adding and checking authentication information for security purposes, adding and using control information for management of transactions, re-binding in the event of server failure, or duplicating an interaction and directing it to multiple servers, possibly collating and combining results.

4. IMPLEMENTATION AND MANAGEMENT ISSUES

At the lowest level, the configuration of intercessory objects in the channel is controlled by the use of proxy objects. A proxy object should support an interface which allows the redirection of invocations to other objects in the client and server stacks and a reference to this interface should be made available to the system managing the channel. However, there are other implementation issues which affect configuration.

4.1. Generic interfaces and intercessory objects

As described above, there are two basic forms an intercessory object implementation can take. These forms are distinguished by the interfaces supported by the intercessory object. It should be noted that each particular IO will use the same interface type as a client that it provides as a server. An IO may be implemented in terms of a specific interface type or in terms of a generic type. The choice is a pragmatic one, but it is important to note that the limitations of particular DPEs and programming languages will limit the options available.

For programming languages which treat types as first-class entities, where it is possible to dynamically bind to representations of specific interface types, the definition of a GI type is not a significant issue, as equivalent functionality can be achieved within the language. But in many languages this is not the case, and the reuse of generic IOs is limited by the acceptance of the corresponding GI specification.

4.2. Sequencing

It is clear that the positioning of IOs in the stacks of both clients and servers is significant. Consider, for example, a replication IO which duplicates any invocation it receives to multiple servers, and a notification IO which notifies a management system of any interaction occurring within the channel. If the replication IO is placed after the notification IO, a single message will be received by the management system as the client calls the replicated interface. However, if the notification IOs are placed after the replication IO, multiple notifications will be received for each invocation.

Another sequencing issue is caused by the relationship between IOs based on the generic interface types and IOs based on application-specific interface types. In the case of channel objects depicted in Fig. 2, interface-specific IOs must be placed above generic IOs, on both the client and server sides of the channel, unless de-marshalling stubs are available on the client side and marshalling stubs on the server side. These stubs may be used to map specific interface types to generic interface types and vice versa.

As the semantics of IO functionality may depend on the sequence in which the IOs are invoked within a particular interface interaction, there is a requirement that this structure be managed in some way. Constraints, placed on these structures by IO implementations, complicate the management process.

4.3. Channel management

As the number of intercessory objects in a channel may change dynamically, and the sequence of IOs in the channel may change, it will be convenient to use a *binding object* to manage the internal configuration of the channel itself.

In many cases, intercessory objects will have additional interfaces specifically related to the operations they are performing or to the functionality they provide. An example is a replication IO which may be redirected to multiple servers, where these replicas are dynamically created and destroyed at run-time. The replication IO needs to be informed about the creation of new server replicas and deletion of old ones along with other reconfigurations which may take place in the replicated system. Information of this nature may be passed to the replication IO via such additional interfaces.

In any case it is possible, and indeed highly likely, that intercessory objects will require a status similar to that of application objects, requiring the ability to create and destroy interfaces dynamically. The types of these interfaces cannot be known to the client or server objects if the intercessory object functions are transparent to them. Therefore, management paradigms cannot make assumptions about these types.

5. APPLICATIONS

A number of management and related areas require the capabilities provided by intercessory objects. Transparency objects from RM-ODP, and *filters* and *smart proxies* from Orbix [9], the CORBA implementation from IONA Technologies, satisfy similar requirements.

5.1. RM-ODP transparency objects

RM-ODP [5] includes definitions of eight *transparencies* which may be provided to distributed systems by conforming environments. Each of these transparencies represents a possible feature of distribution which may be masked from an application by the underlying infrastructure or DPE. Examples include *location transparency*, which masks from a client the location of its server, and *replication transparency*, which masks from a client the fact that it interacts with multiple servers on each invocation.

These transparencies, at least five of which are directly related to interactions between system components, were to be implemented by partially standardised *transparency objects*. A replication transparency object, for example, may intercept an invocation from the client, and send a copy to each of a number of servers. The object may then collate the server responses for return to the client.

Following some work in this area, it became clear that the variability inherent in aspects of this function, such as collation strategies and behaviour in the event of reconfiguration, make standardisation of the associated transparency objects inappropriate.

However, consider the case that the replication transparency object is implemented as an intercessory object, with associated mechanisms for insertion into the channel. Now, the variable functionality could be seen as useful differentiation, and specific functionality could be defined by the developer of the application or the management system, the DPE vendor, or by a third party. Although the intercessory and transparency objects may behave identically, the IO concept does have the advantage of a model where mechanisms are specified for its insertion into the channel.

5.2. Orbix filters and smart proxies

Orbix, the CORBA implementation from IONA Technologies, also provides capabilities similar to those of intercessory objects. The channel model for this CORBA implementation is slightly complicated by the concept of an address space, corresponding to an operating system process, but access to the communication mechanisms is possible. In particular, Orbix filters provide direct access into the channel for server objects, and Orbix smart proxies provide similar access to the client stack. One difference though, is that a smart proxy will be bypassed if the CORBA DII is used for interactions. In that sense, a smart proxy is similar to the interface-specific IO defined in Section 3.

Of particular interest are the uses to which these capabilities are put in Orbix. Suggested applications [9] include caching of data within smart proxies to avoid unnecessary communications overheads, the creation of threads on operation invocations, and the passing of authentication information between clients and servers. These functions are all useful management applications, and the facilities provided by Orbix are essentially equivalent to those which could be provided by IOs of specific types.

5.3. Other management applications

The ability to monitor events taking place within the channel binding two objects has numerous management applications. The types of events which might be monitored include initiation and completion of an interaction on the client side, and initiation and completion of an operation call on the server side. The parameters passed during interactions are also of interest to the management system, as are outright failures and timeouts of interactions.

Monitoring events is useful for debugging and location of faults, and allows the measurement of interaction frequency and data flow, and the determination of usage for accounting and performance management. The management system may base the distributed system configuration on this information, in addition to an understanding of physical resources in the network supporting the distributed system.

Other examples of interaction management include the maintenance and transfer of authentication information for security purposes, and the use of alternative protocols to improve performance of data transmission.

Through the use of intercessory objects, management of this form may be undertaken by management components of the distributed system, without any significant impact on core application operation. The distributed application may be subdivided into managing and managed subsystems in such a way that management operations can be tuned, or even replaced, without making changes to any core application components.

More significantly, management requirements such as these are providing strong motivation for the development of facilities to introduce arbitrary functionality into the channels between objects comprising distributed systems. The value of IOs does not lie in their standardisation, but in the standardisation of the mechanisms which surround them.

6. CONCLUSIONS

Intercessory objects allow the insertion of control and monitoring capabilities into the channel binding objects in a distributed system, and this may be totally transparent to the objects involved. However, the concept of an intercessory object is not new. The transparency objects of ISO's RM-ODP, and the filters and smart proxies of IONA's Orbix, for example, offer similar functionality. The distinctions between IOs and these concepts, and the areas where standardisation work might be valuable, lie in the models describing the configuration of IOs.

With the current emergence of a number of commercial DPEs, and the desire in computing, telecommunications and other industries to build large-scale distributed systems, it is important that work on management of these systems develop further. Intercessory objects offer a mechanism on which many higher-level management activities can be constructed. Libraries of intercessory objects could provide a valuable tool set to perform a wide range of activities, and IOs could be selectively included into various points in the distributed system. Debuggers, and test harnesses and stubs, could be of particular value.

Structures and functions for management of distributed software systems are not yet well defined or understood, but it appears that intercessory objects may have an important role to play in the management of interactions, while introducing their own management requirements.

ACKNOWLEDGEMENTS

The permission of the Director, Telecom Research Laboratories, Telstra Corporation Limited, to publish this paper is hereby acknowledged. The author also wishes to thank Leith Campbell, Geoff Wheeler, Ajeet Parhar and Michael Warner of Telecom Research Laboratories, for their comments on drafts of this paper.

REFERENCES

[1] Architecture Projects Management Ltd., "Application Programming in ANSAware," RM.102.02, Cambridge, February 1993.

[2] Open Software Foundation, in *Introduction to OSF DCE*, Prentice Hall, New Jersey, 1992.

[3] Object Management Group, "The Common Object Request Broker: Architecture and Specification," 91.12.1, 10 December 1991.

[4] ISO/IEC and ITU-T, "Open Distributed Management Architecture — First Working Draft," ISO 8801, July 1994.

[5] ISO/IEC and ITU-T, "Draft Recommendation X.903: Basic Reference Model of Open Distributed Processing — Part 3: Prescriptive Model," 10746-3.1/X.903, February 1994.

[6] ISO/IEC ITU-T, "Information processing systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework," ITU-T X.700, 1989.

[7] William J. Barr, Trevor Boyd & Yuji Inoue, "The TINA Initiative," *IEEE Communciations Magazine* (March 1993).

[8] Object Management Group, "Universal Networked Objects," OMG TC Document 94.9.32, 28 September 1994.

[9] IONA Technologies, *Orbix Advanced Programmer's Guide, Version 1.2*, Dublin, Ireland, February 1994.