

Chapter 6 Service Architectures

Gordon S. Blair and Geoff Coulson

Lancaster University

6.1 Introduction and Motivation

The telecommunications industry is undergoing a period of rapid change with deregulation, globalisation and, crucially, diversification—particularly in the area of service provision. In particular, telecommunications operators are increasingly offering a wide range of services, including multimedia services, over an diverse communications infrastructure, e.g. incorporating wireless networks. To date, such services have often been developed in a rather ad-hoc manner, with little attention paid to service engineering or indeed to underlying service architectures. This chapter focuses on the latter topic, i.e. *the emergence of appropriate service architectures to support the creation of next generation telecommunications services* (chapters ??, ?? and ??, discuss complementary issues of service engineering). Particular attention is given to the emergence of *middleware* platforms and technologies that underpin such service architectures.

The goals for service architectures (and associated middleware platforms) are as follows:

- To support the *rapid creation* and subsequent *rapid deployment* of new services, for example by providing a *higher level programming model* for the creation of services and also by encouraging *re-use*.
- To overcome problems of *heterogeneity* in the underlying infrastructure in terms of network and device types, system platforms and programming languages, etc., and hence to offer greater *portability* and *interoperability* of services.
- To provide more *openness* in service construction, for example to promote *maintenance* and *evolution* of often highly complex software.

The chapter is structured as follows. Section 6.2 discusses early developments in the area of service architecture, with section 6.3 then examining existing and emerging architectures, focusing mainly on distributed object and component technologies. Emphasis is placed on the rationale for such approaches. Following this, section 6.4 investigates the application of such technologies in the telecommunications industry. Section 6.5 discusses some challenges facing existing service architectures, including the need to support user and device mobility and also the increasing prevalence of multimedia services. A number of emerging solutions are also considered including reflective middleware

technologies and the Object Management Group's Model Driven Architecture. Finally, section 6 contains some concluding remarks.

6.2 Early Developments

Given the nature of the area, it is essential that services architectures are supported by appropriate standardisation, whether in terms of de jure or de facto standardisation processes. This is essential to ensure portability, interoperability and openness as discussed above. The first relevant step in standardisation was the creation of the Reference Model for Open System Interconnection by the International Standards Organisation (ISO OSI). This standard defines the now famous 7-layer model for communicating processes as illustrated in figure 6.1.

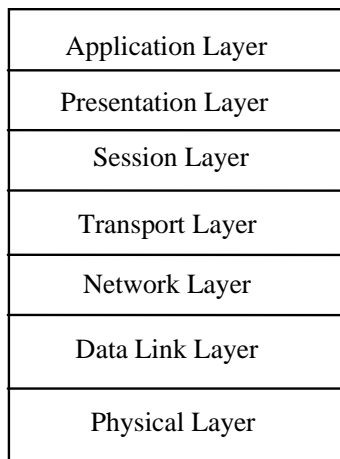


Figure 6.1 The ISO OSI 7-layer Model

This layered structure should not be viewed as an implementation architecture; rather it is a reference model providing concepts and terminology for reasoning about communications protocols, and also an abstract framework (cf. meta-architecture) which can be populated by a selection of appropriate technologies. For example, the TCP/IP protocol stack can be viewed as an instantiation of the lower layers of this architecture. (Note that similar comments apply to the ISO/ITU-T RM-ODP as discussed in section 6.3.1 below).

While OSI offers an open approach to service construction, it does not in itself constitute a service architecture for next generation telecommunication services. For example, OSI is fundamentally based on abstractions over message passing, and does not attempt to hide this from the user. In other words, the approach does not provide the higher level programming abstractions we seek. In addition, OSI is limited in scope, for example when dealing with broader service creation issues such as security and fault-tolerance.

To overcome such limitations, subsequent research has focused on the development of distributed systems technologies that have the goal of offering a higher level of *transparency* (i.e. hiding aspects of distribution), including support for non-functional properties like security and fault-tolerance. The first approaches in this area were based on the classic *client-server model* as depicted in figure 6.2.

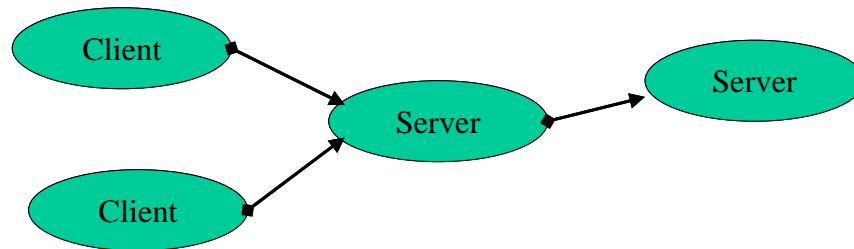


Figure 6.2 The Client-Server Model

In this model, processes can take the role of either clients (requesting a service) or servers (providing a service). Such roles can also change over time; for example, a server can become a client of another server. The model is supported by an appropriate protocol supporting the associated request/ reply style of interaction, most typically a *remote procedure call* protocol that makes interactions appear to the programmer to be similar to a standard procedure call in the host language.

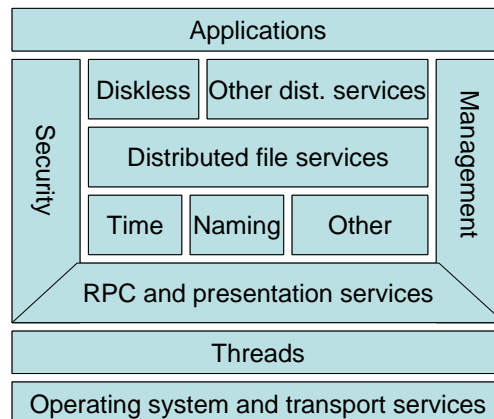


Figure 6.3 The Open Group's DCE

The client-server approach has been highly influential and has led to the emergence of *middleware* technologies like the Open Group's Distributed Computing Environment (DCE) (see figure 6.3). As can be seen, this is based on a Remote Procedure Call protocol (DCE RPC) together with a platform-independent thread service, enabling the creation of multi-threaded servers (and indeed clients).

Crucially, the architecture also comprises a series of value-added servers that can be accessed by applications offering security, persistency of data, etc.

DCE has been successfully deployed in the telecommunications industry. The approach is still however rather limited:

- It is still quite low level in terms of the programming model offered to service developers.
- There is little in the way of software engineering methodologies and tools to support the client-server approach.
- Non-functional properties must be programmed explicitly, e.g. by making calls to appropriate system servers (security servers, etc).

In addition, DCE is inflexible to deploy due to the fact that, although the various modules shown in the figure are separable *in theory*, they are tightly coupled *in practice*. Thus, for example, the RPC service can't be isolated, because this requires the security service, which in turn requires the time service, etc.

As a result of these limitations, attention has more recently focused on more supportive service architectures based on *distributed object technologies* or *component-based software development*. We discuss these developments in some depth in section 6.3 below.

6.3 Current Architectures

6.3.1 RM-ODP – A Meta-Framework for Distributed Computing

Introduction

The Reference Model for Open Distributed Processing (RM-ODP) is a joint standardisation activity by both ISO and the ITU-T. It is therefore a *de jure* standard produced following a period of international consultation to reach a technical consensus.

The aim of RM-ODP is to enable “the development of standards that allow the benefits of distribution of information processing services to be realised in an environment of heterogeneous IT resources and multiple organisational domains” (ISO/ITU-T 1995a). It is important to stress however that RM-ODP does not itself prescribe particular standards for open distributed processing. Rather, as with OSI, it provides a framework to enable specific standards to emerge and hence should be considered a meta-standard for open distributed processing. It is therefore essential for RM-ODP to be sufficiently generic to enable a range of standards to be accommodated. The main value of RM-ODP is in providing a common set of concepts for the field of open distributed processing.

The standard consists of four parts. The heart of the standard is Foundations (Part 2) (ISO/ITU-T 1995b), which defines basic modeling concepts for distributed systems. This is followed by Architecture (Part 3) (ISO/ITU-T 1995c) which constrains the basic model by introducing concepts which a conformant RM-ODP

system should embody. The standard also includes a Overview (Part 1) (ISO/ITU-T 1995a), providing motivation and a tutorial introduction to the main concepts, and Architectural Semantics (Part 4) (ISO/ITU-T 1995d) that provides a formalisation of RM-ODP concepts.

Note that many of the RM-ODP ideas to be described below were initially developing in the ANSA/ ISA project (van der Linden 1993). This work also produced a prototype platform, ANSAware (APM 1993), as a partial implementation of the ANSA architecture.

Major Concepts in RM-ODP

An Object-Oriented Approach The most important contribution of RM-ODP is to develop an *object-oriented approach* to distributed computing, hence tackling the major limitations of the client-server approach outlined above. In particular, the motivations for adopting such an approach were as follows:

- To provide a higher level of abstraction for the programming of distributed systems (incorporating features like encapsulation, data abstraction, support for evolution and extensibility).
- To enable the use of object-oriented analysis and design methodologies in a distributed setting.

One major problem in object-oriented computing is the lack of an agreed terminology for the subject; for example, Meyer has identified at least three quite distinct uses of the term *class* (Meyer 1997). One major contribution of the RM-ODP standard is therefore to provide a precise and unambiguous *object model* tailored to the needs of open distributed processing. This model features for example definitions of terms like object, interface, role, template, factory, class, type, subclass, and subtype. A complete presentation of the object model is however beyond the scope of this chapter (the interested reader is referred to (Blair and Stefani 1998) for a comprehensive treatment).

One interesting feature of the RM-ODP object model is its support for *multimedia programming*. In particular, multimedia is supported by three complementary features:

- As well as supporting conventional operational interfaces (i.e. interfaces containing operations that can be invoked in a request/ reply fashion), RM-ODP also offers *stream* and *signal* styles of interface. Stream interfaces support the production or consumption of continuous media data such as audio or video, whereas signal interfaces serve as emitters or receivers of asynchronous events.
- RM-ODP enables the creation of *explicit bindings* between interfaces, the end result being the creation of an object that represents the communications path between the interfaces. There are three styles of such bindings, i.e. operational, stream or signal, corresponding to the styles of interface discussed above.

- The explicit binding mechanism described above provides intrinsic support for *quality of service (QoS) management*. First, the QoS of a binding can be specified at creation time and appropriate steps taken to ensure that desired levels of guarantees are met (e.g. resource reservation or admission control). Second, as the binding is an object, appropriate steps can be taken to manage the binding at run-time (e.g. through monitoring and adaptation of QoS).

Again, further discussion of this aspect of the object model can be found in (Blair and Stefani 1998).

The Concept of Viewpoints The second important contribution of RM-ODP is its development of a *viewpoint-based methodology* for the design of distributed systems. Viewpoints are intended to deal with the inherent complexity of distributed systems by partitioning a system specification into a number of partial specifications, with each partial specification (or viewpoint) offering a complete and self-contained description of the required distributed system targeted towards a particular audience. The terminology (language) used for each description is therefore tailored towards its target audience. Viewpoints are as central to RM-ODP as the 7-layer model is to OSI. It should be stressed however that viewpoints are not layers. Rather, they are more correctly viewed as projections of the underlying system.

RM-ODP defines five viewpoints, namely the Enterprise, Information, Computational, Engineering and Technology Viewpoints. The different viewpoints have corresponding viewpoint language each of which shares the general object modeling concepts described above; each language can be thought of as a specialisation of this general model for the particular target domain. A summary of the various viewpoints and models is given in table 6.1.

<i>Viewpoint</i>	<i>What it addresses</i>	<i>Example modeling concepts</i>
Enterprise	Business concerns	Contracts, agents, artifacts, roles
Information	Information, information flows and associated processes	Objects, composite objects, schemata (static, dynamic, invariant)
Computational	Logical partitioning of distributed applications	Object, interface, environmental contract
Engineering	Distributed infrastructure to support applications	Stubs, binders, protocol objects, nodes, capsules, clusters
Technology	Technology procurement and installation	Implementation, conformance points

Table 6.1 RM-ODP Viewpoints and Languages

The different viewpoints outlined above effectively create a *separation of concerns* in the specification of a distributed system. The five viewpoints

collectively provide a complete specification of the system which would enable a RM-ODP compliant implementation to be developed.

The one added difficulty of introducing such viewpoints is that of ensuring consistency between the different specifications. This is a difficult problem and comprehensive solutions are currently beyond the state of the art. One approach is to consider the use of appropriate formal specification techniques for each viewpoint and then employ mathematical analysis to ascertain consistency (Bowman *et al.* 1996).

Distribution Transparency and Associated Functions To support a desired level of abstraction in RM-ODP, it is necessary to offer a high level of *distribution transparency*, i.e. to hide the complexities of distribution and heterogeneity from the user. The RM-ODP approach is to support *selective transparency* whereby the programmer can elect for a given level of transparency. A number of different levels of transparency are possible; for illustration, a selection of the main distribution transparencies is given in table 6.2.

<i>Transparency</i>	<i>Concern</i>	<i>Effect</i>
Access	Means of access to objects	To mask differences in data representation or the invocation mechanism employed
Location	The physical location of objects	To enable objects to be accessed by a logical name
Failure	The failure of an object	To mask invocation failure from the user through an appropriate recovery scheme
Migration	The movement of objects	To mask the fact that an object has moved
Relocation	The movement of objects involved in existing interactions	To mask the fact that an object has moved from current users of the object
Replication	Maintaining replicas of objects	To hide the mechanisms required to maintain consistency of replicas
Persistence	Maintaining persistency of data across interactions	To hide the mechanisms required to maintain the persistency
Transaction	Maintaining consistency of configurations of objects	To hide the mechanisms required to maintain consistency of configurations over multiple invocations

Table 6.2 Distribution Transparencies in RM-ODP

RM-ODP also defines a number of so-called *functions* that enable desired levels of transparency. Essentially, the requirements for distribution transparency are expressed in the Computational Viewpoint, and the Engineering Viewpoint is then responsible for meeting the desired level of transparency by employing the appropriate transparency functions. The standard also defines other supportive functions, most notably the *trading* function which acts as a broker for service

offers in the distributed environment. This service is central to RM-ODP (and to the Computational Viewpoint in particular) and hence the trading function merits a separate document in the standardisation of RM-ODP (ISO/ITU-T 1997).

RM-ODP and TINA

As mentioned above, RM-ODP is a meta-standard that can be specialised for a number of domains. Crucially, given the goals of this chapter, there has been a significant effort in specialising the concepts of RM-ODP for the telecommunications industry, resulting in the Telecommunication Information Networking Architecture (*TINA*) (Dupuy *et al.* 1995). This architecture was created by the TINA-C consortium which featured many of the world's leading telecommunications companies, centred on a core team located at Bellcore in New Jersey. TINA is intended to provide a framework for the development of future telecommunications networks. The aim is to define an architecture to support improved interoperability, to be able to re-use both software and technical specifications and to have greater flexibility in the design and deployment of the distributed applications that comprise a telecommunications network. A further aim is to provide a path of evolution and integration from existing telecommunications architectures such as the Intelligent Network (IN) (Abernethy and Munday 1995) and the Telecommunication Management Network (TMN) (ITU-T 1991).

The overall TINA architecture is illustrated in figure 6.4.

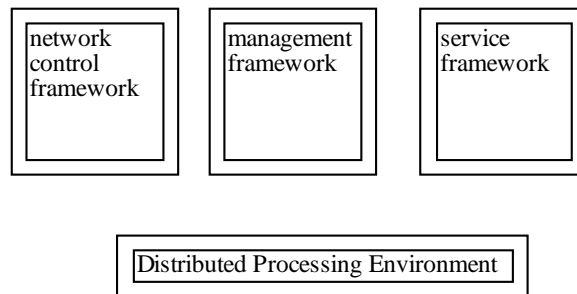


Figure 6.4 The TINA Architecture

TINA places a *distributed processing environment* (DPE) at the heart of its architecture. This DPE adopts the RM-ODP object model and is also organized along the lines of the RM-ODP Engineering Model. The TINA DPE provides a uniform platform for the execution and deployment of the different distributed applications that can be found in a telecommunications network. TINA has identified three broad classes of applications, with associated architectural models as defined below:

- The *network control architecture* defines a generic set of concepts for describing and controlling transport networks in terms of two frameworks.

The *network resource information framework* is inspired by ITU-T Recommendation G.803 and defines a generic network architecture, organised around transmission independent and switching technology independent notions of network elements, how they are related (aggregations) and topologically interconnected, and how they can be configured to provide an end-to-end path. The *connection management framework* then describes generic components for the control of end-to-end communication paths in a network. This framework is organised around notions of connection graphs that describe logical and physical topologies of network connections.

- The *service architecture* defines the principles, concepts, and basic classes necessary for the design, deployment, operation and management of information services in a TINA environment. It includes notions such as user agents, service and communication sessions, access management, and mediation services.
- The *management architecture* defines a set of principles for the management of the components in the distributed computing infrastructure, and in the network control and service architectures described above. The architecture covers two main areas. The first, referred to as *computing management*, is responsible for the management of computers, the DPE and related software. The second, referred to as *telecommunications management* is concerned with the management of information network services and software related to the underlying network. The latter is further divided into service, network and element management in a similar manner to the above-mentioned TMN. Finally, the architecture partitions management into a number of *functions* including the OSI functions of fault, configuration, accounting, performance and security.

The breadth of the architecture should be apparent from the above description of the main components in TINA. This illustrates the level of complexity of dealing with heterogeneity in modern telecommunications environments.

A final noteworthy contribution of TINA is the definition of a role-based business model for telecommunications. This model identifies the following roles: *service provider* (provides and manages the service), *consumer* (the customer of a service), *retailer* (sells the service to consumers on behalf of service providers), *broker* (puts consumers in touch with retailers), and *communication provider* (provides the network connectivity to let it all happen). This simple model gains its power and flexibility from the fact that roles are distinguished from actual stakeholders; for example the service provider and communication provider roles may both be played by a single telecommunications company or, equally well, by separate telecommunications content provider companies. This property makes the business model applicable in a very wide range of situations as further discussed in section 6.4.2 below.

6.3.2 Specific Distributed Object Technologies

Introduction

The work on RM-ODP has encouraged the emergence of a range of specific middleware technologies adopting the object-oriented paradigm. The most significant players in this area are listed below:

- *The Object Management Group's CORBA* The Common Object Request Broker Architecture (CORBA) is a standardisation activity promoted by the Object Management Group (OMG). This organisation is sponsored by a number of IT and telecommunications companies with the aim of promoting an open object-oriented framework for distributed computing. The particular goal of CORBA is to provide the mechanisms by which objects can transparently interact in a distributed environment.
- *Microsoft's COM and DCOM* COM is functionally quite similar to CORBA but is generally restricted to Microsoft platforms. It does however have some significant differences including support for multiple interfaces (as in RM-ODP) and also interoperability at the binary level. Distribution is supported by DCOM, which in turn is based heavily on the Open Group's DCE RPC technology.
- *Java RMI* RMI is tightly bound into the Java language and supports the transparent remote invocation of methods defined on Java objects whether in a separate address space or on a separate machine. RMI is now supported by other facilities such as Jini and Enterprise Java Beans offering a comprehensive platform for distributed computing (albeit restricted to a single language environment).

Further details of these technologies can be found in (Emmerich 2000).

In this chapter, we focus on CORBA because of its unique role in providing a language and platform independent environment for distributed objects. It is also interesting to note that for similar reasons the TINA consortium has identified CORBA as the technology of choice to serve as a basis for the TINA DPE (see below).

Focus on CORBA

OMG and the Object Management Architecture CORBA is supported by the OMG as part of an initiative to develop a comprehensive Object Management Architecture (OMA) for distributed object-oriented computing. The OMG is a non-profit organisation sponsored by over 800 organisations including computer manufacturers, software companies, telecommunications companies and end users. The OMG's overall objective is to "promote the theory and practice of object technology for the development of distributed computing systems". The approach adopted by OMG to achieve this goal is "to provide a common architectural framework for object-oriented applications based on widely available interface specifications... conformance to these specifications will then make it possible to

develop a heterogeneous application environment across all major hardware platforms and operating systems”.

The Object Management Architecture is defined in figure 6.5.

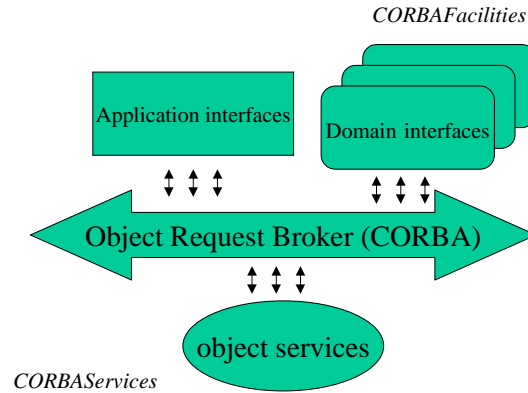


Figure 6.5 The Object Management Architecture

As can be seen, the Object Request Broker (ORB) has the central role in this architecture, providing the mechanisms whereby objects can transparently make requests and receive responses (we examine this in more detail below). The ORB is then supplemented by the following:

- The CORBA developer can provide an arbitrary number of application objects offering specific services through *application interfaces*.
- In doing so, they can make use of a range of object services (*CORBAServices*), including support for functions like naming, trading, transactions, and security.
- There are also a range of specific interfaces for specific application domains (*CORBAFacilities*), including areas such as financial services, healthcare and real-time systems. These interfaces are specified by associated *Domain Task Forces*. Crucially, there is a very active task force for the telecommunications domain as discussed in section 6.4.1 below.

Interestingly, the original scope of the OMG has been extended to encompass design notations for distributed objects. In particular, the OMG produced a standard for the Unified Modeling Language (UML) in 1997 and continue to support extensions and refinements to this standard.

The CORBA Object Model and IDL The CORBA object model defines an *object* as “an identifiable, encapsulated entity that provides one or more services that can be requested by a client”. To access an object, clients issue a *request* for a service. Requests consist of the target object, the required operation name, zero or more actual parameters and an optional *request context*. The request context is used to specify additional information about the interaction. This can convey information about either the client or the environment; for example, the context can

be used to define a user's preferred priority or transactional information. Should a request fail, an *exception* is raised in the client object.

Objects in CORBA support a single interface described in terms of the OMG's Interface Definition Language (IDL). IDL is a language independent, declarative definition of an object's interface. It is not expected that the object implementation shares the typing model of IDL. Rather, a mapping must be provided between IDL and each host language. IDL closely resembles C++ but with added features to support distribution.

The main features of IDL are best illustrated by example. The following IDL specification defines a simple stock control system:

```
// OMG IDL definition
interface inventory
{
    // attributes and type definitions

    const long MAX_STRING = 30;

    typedef long part_num;
    typedef long part_price;
    typedef long part_quantity;
    typedef string part_name<MAX_STRING+1>;

    struct part_stock {
        part_quantity max_threshold;
        part_quantity min_threshold;
        part_quantity actual;
    };

    // operations

    boolean is_part_available (in part_num number);
    void get_name(in part_num number, out part_name name);
    void get_price(in part_num number, out part_price price);
    void get_stock(in part_num number, out part_quantity
        quantity);
    long order_part(in part_num number, inout part_quantity
        quantity, in account_num account);
};
```

The *in*, *out* and *inout* annotations define whether the associated parameter carries data into the object, returns results from the object or a combination of both. The remainder of the IDL is relatively straightforward and is not described further.

Note also that IDL supports interface inheritance whereby a derived interface can be created by inheriting the specification of one or more base interfaces. This is not shown in the above example.

The Object Request Broker The role of the Object Request Broker (ORB) is to enable requests to be carried out in a heterogeneous distributed environment. A client can issue a request on an *object implementation* and the ORB deals with finding the object, sending the request to the appropriate object and preparing the object to receive and process the request and return the result back to the client. The ORB therefore implements a level of distribution transparency (actually location and access transparency in terms of Table 6.2).

The overall architecture of the ORB is captured in figure 6.6.

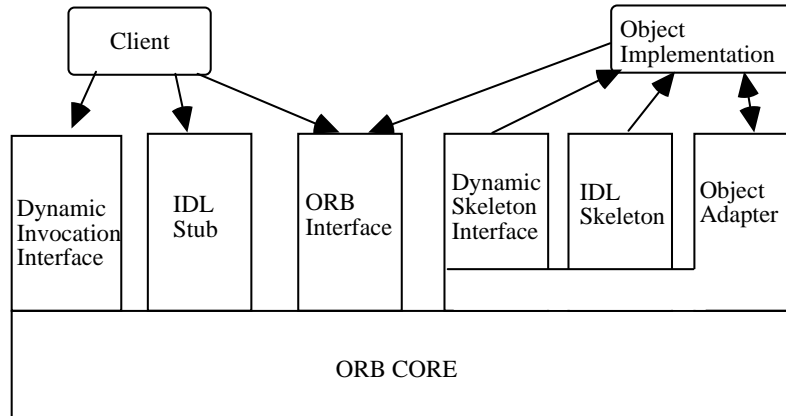


Figure 6.6 The Object Request Broker

The *ORB Core* is the lowest level in the ORB architecture. This is generally implemented as a library that is linked into both the client and the server. This library for example offers support for sending requests using CORBA's *General Inter-ORB Protocol* (GIOP¹). An application programmer will typically not access the ORB core directly but will access the higher level interfaces indicated in the diagram.

IDL Stubs provide the mechanism for clients to be able to transparently issue requests, dealing with the *marshaling* of parameters and the *unmarshaling* of results (marshaling is the mapping of typed parameters on to a flat, programming language independent, message format; unmarshaling is the inverse process). The stubs are normally pre-compiled from the IDL definitions of the required object implementations and linked into the client's address space. Hence, the set of stubs is *static*. The *IDL Skeleton* interface offers the same service at the receiving end of the request. Skeletons identify the required procedure, unmarshal the parameters, up-call the object implementation to request that the operation be carried out and then deal with marshalling the results and returning them to the client. Again, this mechanism is static in nature.

To overcome the static nature of stubs, the client can use the *Dynamic Invocation Interface*. Using this interface, the client must manually construct a request indicating the required operation, the parameters, etc. This is obviously more difficult but the flexibility is sometimes required in applications that must operate on objects not known at compile time (e.g. debuggers and browsers). CORBA also supports an analogous *Dynamic Skeleton Interface* on the implementation side.

¹ OMG also defines the *Internet Inter-ORB Protocol* (IIOP) which is an instantiation of GIOP that runs over the Internet's TCP/IP protocol stack.

Finally, objects have access to an *Object Adapter* interface. This interface effectively provides a “life-support environment” for CORBA objects. In earlier versions of the architecture, this was under-specified leading to a number of disparate implementations of the interface. To overcome this, the OMG have now specified a Portable Object Adapter (POA), offering fully specified support for functions like object activation and passivation, and the management of object references. To encourage flexibility, the POA is now policy driven in areas such as persistency and thread management.

6.3.3 From Objects to Components

Motivation

Distributed object technologies have proved to be highly successful in encouraging the uptake of distributed systems technology in a wide range of application domains (including telecommunications). However, a number of significant problems have been identified with the distributed object approach:

- Little support has been provided for the third party development of objects and for their subsequent integration into applications.
- It is difficult to deploy and subsequently manage large-scale configurations of objects.

In addition, significant difficulties have been reported in dealing with the extensive range of interfaces and services offered by current platforms. For example, when dealing with CORBA, the programmer must deal with a number of complex interfaces including the POA (see section 6.3.2 above), as well as explicitly inserting calls to the various CORBA services to obtain desired non-functional properties like transactions and persistence.

Recently, a number of *component technologies* have emerged in response to these difficulties. Szyperski (Szyperski 1998) defines a *component* as “a unit of composition with contractually specified interfaces and explicit context dependencies only”. In addition, he states that a component “can be deployed independently and is subject to third-party composition” (Szyperski 1998). A key part of this definition is the emphasis on *composition*; component technologies rely heavily on composition rather than inheritance for the construction of applications, thus avoiding the so-called *fragile base class problem* (and the subsequent difficulties in terms of system evolution) (Szyperski 1998). To support third party composition, components support explicit *contracts* in terms of *provided* and *required* interfaces. The overall aim is to reduce time to market for new services through an emphasis on programming by assembly rather than software development (cf. manufacturing vs. engineering).

In terms of middleware, most emphasis has been given to *enterprise (or server-side) component technologies*, including Enterprise Java Beans (EJB) and the CORBA Component Model (CCM). In such enterprise technologies, components typically execute within a *container*, which provides implicit support for

distribution transparencies, in terms of transactions, security, persistence, location transparency, events, and resource management. This offers an important separation of concerns in the development of server-side applications, i.e. the application programmer can focus on the development and potential re-use of components to provide the necessary application logic, and a more “distribution-aware” developer can then provide a container with the necessary non-functional properties. As well as support for distribution, containers also provide additional functionality including life-cycle management and component discovery.

In the following section, we describe the CORBA Component Model in more detail. Note that this is conceptually similar to the Enterprise Java Beans model, but generalised to offer full language-independence. Other important component technologies include SUN’s Java Beans technology, a component technology restricted to a single address space, and Microsoft’s .NET, which, like Java, relies on the existence of an underlying virtual machine.

Focus on the CORBA Component Model

Overview The *CORBA Component Model (CCM)* is an integral part of the CORBA v3 specification. (Other important extensions to the architecture include improved *Internet integration* through for example firewall support, a URL-based naming scheme, and links to appropriate Java technologies, and also *quality of service support* in the form of an asynchronous messaging service and also minimum, fault-tolerant and real-time specifications.) CCM provides a language-independent, server-side component technology supporting the implementation, management, configuration and deployment of future CORBA applications (Wang *et al.* 2001). There are essentially three main parts to the technology:

- An underlying component model.
- A container framework offering implicit security, transactions, persistency and event management.
- A packaging technology for deploying binary, multi-lingual executables.

We look at each in turn below.

The Component Model A component in CCM is defined in terms of a number of interfaces of different styles (cf. RM-ODP). In particular, the following styles are supported:

- *Facets* are interfaces that the component provides to the outside world, corresponding to ‘provided’ interfaces as discussed above. A given facet looks very much like a traditional CORBA interface; it can be invoked synchronously via a CORBA request or asynchronously via the asynchronous messaging that is part of CORBA v3.
- *Receptacles* are equivalent to ‘required’ interfaces; before a component can operate, its receptacles must be connected to facets offered by other components. This embodies an important part of the contract in terms of the usage of this component.

- The CCM also supports a publish-subscribe style of interaction through *event sources* and *sinks*. Event sources are emitters of information, which are then loosely connected to zero or more event sinks through an appropriate event channel (cf. the Observer pattern (Gamma *et al.* 1994)).
- Finally, a CORBA component exposes a set of *attributes* that can be used by various configuration tools to enable declarative configuration of the component.

The above features are supported by an extended IDL that has new keywords representing the new concepts. For example, facets and receptacles are declared using the keywords `provides` and `uses` respectively. The interested reader is referred to (Seigel 2001) for a treatment of this extended IDL.

Internally, a CORBA component consists of implementations of each of the facets and event sinks, together with the set of attributes. A component also exposes additional interfaces, including the *home interface* which supports lifecycle operations such as the creation or deletion of particular instances, and the equivalent interface which allows legacy software (including, interestingly, EJB software) to be presented as CORBA components.

The Container Framework A CORBA component can be placed in an appropriate container that provides implicit management of that service. In particular, the container performs the following tasks on behalf of components:

- The management of resources, e.g. in terms of activation or passivation of instances.
- The setting of appropriate POA policies.
- The calling of CORBA services at appropriate points to achieve the desired non-functional properties of the component, e.g. in terms of security.

The container also undertakes to inform the component of important events; the component can provide handlers to respond to such events.

The component/ container approach represents an extremely high level of transparency in terms of distributed systems development. With this approach, the programmer can focus on the development or re-use of components, and the associated interconnection logic, and can then rely on the infrastructure to manage the complexities of distribution. This overall approach is summarised in figure 6.7 (taken from (Wang *et al.* 2001)).

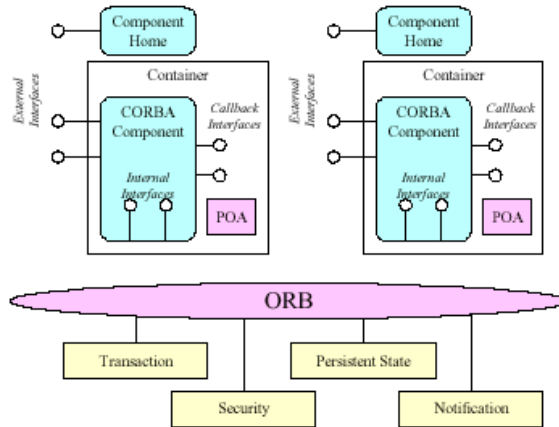


Figure 6.7 The CCM Container Model

A given container can also be configured according to key *policies*. For example, the *servant lifetime policy* can be used to establish a policy for activation or passivation of components (i.e. activate on first invocation/ passivate on request completion, activate on first invocation/ passivate on transaction completion, activate on first invocation/ destroy explicitly, activate on first invocation/ deactivate when container needs to reclaim resources). Policies can also be established in other areas including security, transactions and persistency.

The Packaging Technology Once completed, components can be packaged together with appropriate meta-data in an assembly file (.CAR file). The meta-data comprises information on the component, its dependencies, and its assembly instructions. In all cases, this meta-data is captured using XML templates.

More specifically, the meta-data consists of the following key elements:

- A *software package descriptor* containing a description of the overall package in terms of, for example, the author, the implementation tools, the IDL, etc.
- A *component descriptor* describing, for example, the required container policies.
- A *component assembly descriptor* describing a set of components and how they should be interconnected.
- A *property file descriptor* describing initial attribute values for a given component.

Some of this information is collated from a programmer supplied *CIDL* description. CIDL (Component Implementation Definition Language) is a declarative language that complements the (extended) CORBA IDL, and which describes additional properties of the eventual implementation.

This information is then sufficient for deployment tools to unwrap the component and install it in the distributed environment thus considerably easing the task of installing distributed systems software.

6.4 Applying the Technologies

It is fair to say that the middleware technologies described in this chapter have so far been primarily applied to business-oriented computing environments. For example, they are often used to facilitate interworking in company networks, to integrate legacy applications into distributed systems, and to provide back-end web services in areas such as accounting, e-commerce and database integration. However, in addition to this, there has already been a significant use of middleware technology in the telecommunications industry. This was initially seen in the development of TINA in the mid 1990s as described above. More recent developments are described in this section.

6.4.1 The OMG's Telecommunications Domain Task Force

A clear example of the trend toward the use of middleware technology in telecommunications is the work carried out by the OMG's Telecommunications Domain Task Force (OMG 2001a). This task force, which dates back to 1997, seeks to influence the development of CORBA and its associated services to meet the needs of the telecommunications industry. It has been responsible for the introduction of a number of new standards into the OMG's stable. Among these are the following core standards (others are discussed in section 6.4.2):

- The *Control and Management of A/V Streams* specification (OMG 1997) defines a framework in which CORBA objects can manage audio and video streams; the streams themselves are assumed to be implemented outside the CORBA environment. 'Manage' in this context refers to functions such as establishing connections, starting and stopping the flow of data, and adding new parties to multiparty sessions.
- The *Notification Service* (OMG 1998a) (and the associated Management of the Event Domain specification (OMG 2000a)) standardises publish-subscribe oriented communication. The Notification service lets consumer objects 'subscribe' to a 'topic' and thereby obtain information made available by producer objects that have previously 'published' the information under that topic. The service holds published information persistently so that it is not necessary for the publishers and subscribers to interact synchronously. This communication style is particularly useful for loosely coupled systems for which scalability is a prime concern (e.g. control and management in telecommunications networks).
- The *CORBA/TCAP* (OMG 1998b), *CORBA/TMN* (OMG 1998c) and *CORBA-FTAM/FTP Interworking* (OMG 2000b) specifications define ways in which CORBA can interoperate with, respectively, the TCAP,

Telecommunications Management Network, and File Transfer Access and Management/File Transfer Protocol standards to facilitate the use of CORBA in network management. Also in this area, the Telecom Log Service (OMG 1999) enables the logging of management events and the querying of log records.

Most recently, the Telecommunications task force has finalised a standard for Wireless Access and Terminal Mobility (OMG, 2000c). This defines an architecture and interfaces to support wireless access and end-system mobility in CORBA. Mobility of both clients and server objects is supported through an architecture that supports handoffs between base stations in a similar way to Mobile IP (Perkins 1996). GIOP tunneling² is employed between the base stations involved in a CORBA invocation to hide the fact of mobility from both client and server.

6.4.2 Middleware Technology in Open Service Provision

Recently, a number of initiatives have emerged that are aimed at using middleware technology in *open service environments* in telecommunications networks. An open service environment is one in which anyone, not only the telecommunications network provider, is permitted to create and manage services (e.g. a multimedia content archive or a private virtual network).

TINA has been fundamentally influential in this area through its role based business model that was briefly discussed in section 6.3.1 above (see also chapter ?). Given this business model, it is natural to identify a number of standardisable interfaces in an open service environment. For example, it is necessary to define interfaces between the consumer and the retailer, and between the service provider and the communication provider. In this environment, Parlay (Parlay 2001) and JAIN (Sun 2001) are standardisation initiatives defining interfaces that allow controlled access by third parties (e.g. service providers and retailers) to the network functionality owned by telecommunications companies. Parlay defines its interfaces in CORBA IDL while JAIN is a Java based technology and defines its interfaces in Java (interestingly using the Java Beans component model). But both initiatives have broadly similar aims and there is current work on convergence between the two. Both Parlay and JAIN are covered in more detail in chapter ?.

In a similar area, the OMG's Telecom Service Access and Subscription (OMG 2000d) standard has recently been adopted. This standard includes a set of interfaces whereby communications providers can offer third party enterprises secure access to telecommunications network functions like call control and user location. The standard also supports the straightforward integration of billing and payment for such services.

² Generally, the term *tunnelling* refers to an encapsulation of one protocol within another. A tunnel is used to ship data between two administrative domains that use a protocol that is not supported by the internet that connects them.

Work is also beginning on populating the other parts of the TINA business model (i.e. not just interfaces to the communication provider role). For example the OMG has recently issued a white paper on an “Open Services Marketplace” that aims to develop CORBA specifications in this area. This work, however, is still at a relatively early stage and specifications are yet to emerge.

6.4.3 Middleware Technology in Programmable Networking Environments

A final area of current research interest in applying middleware technology to telecommunications is the use of these technologies in *programmable networking environments*. These provide network programmability at a lower level than the Parlay and JAIN initiatives mentioned above. The primary aim here is to render networks more programmable in terms of their fundamental packet forwarding behaviour. The basic approach is to open up the network at the level of routers and switches so that new functionality can be introduced onto routers or switches. For example, packet schedulers, traffic shapers, routing algorithms and firewalls can be dynamically loaded and configured using this approach.

A good example of recent work in this area is Columbia University’s Genesis project (Campbell 1999). This introduces the abstraction of *network spawning* in which new ‘child’ virtual networks can be dynamically created on top of an existing ‘parent’ network (the ultimate parent is the physical network). The degree of resource sharing between parent and child can be closely controlled so that child networks are capable of providing predictable quality of service, e.g. in terms of throughput and delay. Thus, spawned networks can add QoS support and dynamicity to the traditional private virtual network abstraction that typically is concerned only with security. Genesis uses CORBA to control the process of network spawning and each newly spawned network has its own CORBA-based control infrastructure.

More information on the area of programmable networking can be found in the proceedings of the Open Signalling (OPENARCH 2001) or Active Networking (IWAN 2000) conference series.

6.5 Meeting Future Challenges

Although there is a steadily increasing uptake of middleware technology in telecommunications, the telecommunications domain is evolving so fast that keeping up is a major challenge. In this section, we examine a number of specific areas in which this evolution is particularly challenging. Then we briefly discuss approaches that are being adopted in the middleware research community to address these challenges.

6.5.1 Emerging Challenges

The first major problem area is *mobility*. Although the above-mentioned OMG standard provides basic support for mobility in terms of supporting handoffs and GIOP tunnelling, much remains to be done. One key characteristics of mobile applications is that their connectivity can vary widely over the lifetime of an application session. When a user's machine (e.g. a mobile PDA) is connected to a fixed, wired network, connectivity may be good, with high throughput and reliability, and low latency and jitter. At other times, however (e.g. when the user is travelling), the PDA may experience either total disconnection, or intermittent low-grade connection (low throughput and reliability, and high latency and jitter) though a wireless LAN or mobile phone link. Traditional request/ reply communications protocols like CORBA's GIOP are ill suited to such an environment. They do not work at all during periods of disconnection and work inefficiently over low grade and intermittent connections. What is required is 'built-in' tolerance to temporary disconnection and the ability to communicate asynchronously during periods of disconnection.

Another mobility related challenge is to provide distributed systems services that are sensitive to the user's current *location*. For example, if a user accesses a roadmap service, the underlying service discovery infrastructure should implicitly return a map that relates to the user's current location. Similarly, if a user prints a document, the request should be transparently routed to a local printer in the same room or building. Current service discovery mechanisms like the CORBA naming or trading services do not meet these requirements as they can only deal with relatively static services and have no concept of location. Although work in this area is proceeding (e.g. see (Arnold *et al.* 1999)), there is still much to be done before location based services can be widely deployed.

A further requirement arising from mobility is the desire to establish *ad-hoc networks* of mobile users who need to come together for some transient purpose—e.g. a business meeting or a team of mountain rescue personnel who need to collaborate on a rescue mission. In such ad-hoc networks, each user should have transparent access to services provided by other users as well as to other services that happen to be locally situated.

A second major challenge arises from the emergence of *ubiquitous computing*. This involves large and highly heterogeneous networks of primitive devices like domestic appliances, devices attached to in-car local area networks, and wearable computers. Such an environment imposes particularly demanding requirements for decentralised management, which current middleware technologies do not deal with well. Current systems are typically managed manually and usually break when key services become unavailable or networks partition. Furthermore, the ubiquitous computing environment demands that the middleware runs on very primitive devices with few resources (especially memory and CPU capacity). This is difficult or impossible with current technologies that employ client and server side libraries occupying many megabytes of RAM.

A further challenge arises from the increasingly central role of *multimedia*—particularly continuous media like audio and video. Although the above mentioned CORBA standard for the control and management of A/V streams enables the *control* of media streams, it does not provide a framework for in-band processing of such streams. For example, one cannot implement sources, sinks or filters for streams in the CORBA middleware environment. This can easily lead to fragmented system design and competing resource management strategies in end-systems.

A final challenge arises from the need to integrate middleware technologies into the broader application-level software environment—especially the Internet and the World Wide Web. This area has received attention in CORBA v3 as discussed in section 6.3.2 above. However, there is little experience with these extensions and fully seamless integration is still to be achieved.

6.5.2 Emerging Solutions

The problem areas identified above are now widely acknowledged by the middleware research community and work is underway to address them. In this section, we focus on two complementary research approaches that promise to address many or most of the above problems: these are *reflective middleware*, and the quest for *model driven architectures*.

The basic approach of *reflective middleware* is to ‘open up’ the middleware platform implementation at runtime so as to be able to inspect, configure and reconfigure the platform internals. In other words, the approach is to abandon the previously dominant ‘black box’ paradigm in favour of a ‘white box’ paradigm in which the platform implementation is made selectively visible to applications. The essence of the approach is to provide a ‘causally connected self-representation’ (or CCSR, or ‘meta-model’) of the middleware platform. ‘Causal connection’ means that changes made to the model are ‘reflected’ in the represented platform, and vice versa. As an example, given a meta-model that represents the current structure of the middleware in terms of a topology graph of constituent components, we could inspect the structure of the system simply by observing the structure of the meta-model, and we could implicitly remove, replace or add components simply by manipulating generic graph operations defined on that meta-model.

Reflective middleware is typically structured in terms of reusable components that can be appropriately composed to yield a desired middleware profile. Note that in this approach the middleware platform *itself* is built using components; this differs from the existing commercial component models discussed above which provide application level component functionality *on top of* a traditional black-box middleware platform. As an example of the use of componentised reflective middleware, a cut-down profile can be specified for a PDA or sparsely resourced ubiquitous computing node (e.g. client-only libraries can be specified, or multi-threading or media-stream capability can be left out). To assist the composition process, the component models, although lightweight in comparison to the

commercial application levels models, typically support the provided/ required style of component composition discussed in section 6.3.3. This explicit specification of requirements makes it easier to accurately predict the effects of composing sets of components in a given manner.

The fact that reflective middleware facilitates the *dynamic* manipulation of components is particularly useful in a mobile computing environment in which it is desirable to maintain application continuity—albeit at reduced quality—when passing from one connectivity domain to another. For example, as one moves from a good to a poor quality network connection, one might alter the compression scheme used by a video stream by replacing one compression component with another. This, however, is only one example; given appropriate system support there are numerous situations in which it is useful to reconfigure running middleware platforms. In particular, the reflective middleware approach seems well placed to address the dynamicity requirements arising from ubiquitous computing and continuous media support. It also appears promising in managing the evolution of software over longer time periods. Further details of research on reflective middleware can be found in the literature (e.g. (Coulson 2000), (RM 2000), (Kon *et al.* 2000), (Blair, *et al.* 2001)).

The concept of *model driven architecture* (OMG 2001b) was first proposed by the OMG in late 2000. Essentially the MDA concept is to raise the level of programming abstraction so that an abstract service specification (expressing ad-hoc application logic) called a *platform independent model* (PIM) can be automatically mapped, through a sophisticated tool chain, to a suitable middleware based implementation—which is called a *platform specific model* (PSM). The motivation for the MDA concept is the proliferation of middleware level technologies that are already available (e.g. CORBA, Microsoft's DCOM, Java RMI, or Web-based platforms) or becoming available (e.g. XML/SOAP, Microsoft's .NET or Sun's ONE). It was felt that a new approach was needed to avoid the need to retool every time a new technology comes along, and to facilitate interworking between these technologies.

In more detail, PIM-level specifications are expressed in the OMG's Universal Modelling Language (UML) and then 'compiled' to generate an implementation in a preselected set of (PSM) middleware technologies. Currently CORBA is the only PSM technology supported but the list is expected to grow as the model driven architecture paradigm gains acceptance. As well as mapping to one or more selected PSMs, the tool chain also generates a set of 'bridges' that enable transparent interworking between parts of the system that might be implemented in different middleware technologies. A standard set of distributed systems services are implicitly available, via bridging, from any middleware implementation environment. At present, these are the standard CORBAServices which, however, have been renamed as Persistent Services in the MDA. Note that the 'compilation' process may not be entirely automatic in that some 'glue' code may need to be provided by the programmer. However, the amount of work required would be

expected to be minimal compared to the task of achieving the same result entirely manually.

Currently, the MDA is in an early state of development and the first MDA tools produced by OMG members are not expected until early 2002. However, several key parts of the architecture are already standardised. These are UML, a standard called XMI that specifies how to interchange XML meta-data, a meta-modelling repository called the Meta-Object Facility (MOF), and the associated Common Warehouse Model (CWM). Overall, MDA promises to be an important step forward in the automation of middleware based system development, particularly in combination with the reflective component based technologies discussed above. This combination promises not only a significantly improved development environment but also an environment in which existing distributed software can be adapted, reconfigured and evolved over time.

6.6 Conclusions

This chapter has examined the role of service architectures in supporting the creation and management of next generation telecommunications services. Particular attention has been given to the potential support offered by emerging middleware platforms and technologies.

A range of middleware technologies have been considered:

- Client-server technologies such as DCE.
- Distributed object technologies such as CORBA, Java RMI and COM/DCOM.
- Component-base technologies (particularly enterprise technologies) such as the CORBA Component Model, Enterprise Java Beans and .NET.

These represent a gradual evolution of middleware technologies towards the needs of modern service creation and deployment. In particular, the authors believe that component technologies are particularly well placed to meet the needs of the telecommunications industry in terms of offering a high level programming model, supporting a third party style of development, encouraging re-use and the rapid creation and evolution of services, ensuring that services are portable and interoperable, and also supporting their automatic deployment.

The chapter has also examined existing practice in applying such middleware technologies in the telecommunications industry. While the use of such technologies is not yet widespread, there are a number of interesting developments including the work of the OMG Telecommunications Domain Task Force, the use of middleware technologies in opening up networks (e.g. through JAIN and Parlay) and in managing programmable networking environments. There are however a number of research challenges that have not been met. Most importantly, existing architectures tend to be too complex and heavyweight for the telecommunications domain. This has for example been a major inhibitor in the uptake of technologies such as TINA. In addition, there is little support in the key areas of mobile and ubiquitous computing, or multimedia services. The chapter concludes that such

challenges demand a new approach to service architecture and points to interesting work in the field of reflective component-based middleware that offers one promising way forward to meeting the middleware needs of the future telecommunications industry.

Acknowledgements

The authors would like to acknowledge the support offered by the FORCES consortium (EPSRC Grant GR/M00275) in the development of the ideas presented in this chapter.

References

- Abernethy TW and Munday CA 1995 Intelligent Networks, Standards and Services. *BT Technology Journal* No 2, pp 9-20.
- APM Ltd. 1993 ANSAware 4.1 Application Programming in ANSAware. *Document RM.102.02. A.P.M.* Cambridge Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, UK.
- Arnold K O'Sullivan B Scheifler R Waldo J Wollrath A 1999 The Jini Specification. Addison Wesley.
- Blair GS Coulson G Anderson A Blair L Clarke M Costa F Duran-Limon H Fitzpatrick T Johnston L Moreira R Parlavantzas N Saikoski K 2001 The Design and Implementation of Open ORB v2. *Special Issue of IEEE Distributed Systems Online on Reflective Middleware*, <http://www.computer.org/dsonline/>.
- Blair GS and Stefani JB 1998 Open Distributed Processing and Multimedia. Addison-Wesley.
- Bowman H Derrick J Linington P and Steen MWA 1996 Cross Viewpoint Consistency in Open Distributed Processing. *Software Engineering Journal* Vol 11, No 1, pp 44-57.
- Campbell AT Kounavis ME Villela DA Vicente JB de Meer HB Miki K Kalaichelvan KS 1999 Spawning networks. *IEEE Network Magazine* Vol 13, No 4, pp. 16-29.
- Coulson G 2000 What is Reflective Middleware? Introduction to the reflective middleware subarea, *Distributed Systems Online Journal*, IEEE Computer Society, <http://boole.computer.org/dsonline/middleware/RMArticle1.htm>.
- Dupuy F Nilsson G Inoue Y 1995 The TINA Consortium: Toward Networking Telecommunications Information Services *IEEE Communications Magazine*, Vol 33, No 11, pp. 78-83.
- Emmerich W 2000 Engineering Distributed Objects. Wiley.
- Gamma E Helm R Johnson R Vlissides J 1994 Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- ISO/ITU-T 1995a ISO/IEC CD 10746-1 | ITU Recommendation X.901, Open Distributed Processing - Reference Model - Part 1: Overview.
- ISO/ITU-T 1995b ISO/IEC 10746-2 | ITU Recommendation X.902, Open Distributed Processing - Reference Model - Part 2: Foundations.

- ISO/ITU-T 1995c ISO/IEC 10746-3 | ITU Recommendation X.903, Open Distributed Processing - Reference Model - Part 3: Architecture.
- ISO/ITU-T 1995d ISO/IEC CD 10746-4 | ITU Recommendation X.904, Open Distributed Processing - Reference Model - Part 4: Architectural Semantics.
- ISO/ITU-T 1997 ITU Recommendation X.950, Open Distributed Processing - Trading Function: Specification.
- ITU-T 1991 ITU-T Recommendations M.3010, Principles for a Telecommunications Management Network Working Party IV, Report 28.
- IWAN 2000 Proc. 2nd Intl. Working Conf. on Active Networks (IWAN 2000), Tokyo, Japan.
- Kon F Román M Liu P Mao J Yamane T Magalhães LC and Campbell RH 2000 Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, IBM Palisades, New York.
- Meyer B 1997 Object-Oriented Software Construction. 2nd Edition, Prentice-Hall.
- OMG 1997 Control and Management of A/V Streams Specification: <ftp://ftp.omg.org/pub/docs/formal/00-01-03.pdf>.
- OMG 1998a Notification Service Specification: <ftp://ftp.omg.org/pub/docs/dtc/00-12-02.pdf>.
- OMG 1998b CORBA/TCAP Interworking Specification: <ftp://ftp.omg.org/pub/docs/telecom/98-10-03.pdf>.
- OMG 1998c CORBA/TMN Interworking Specification: <ftp://ftp.omg.org/pub/docs/telecom/98-10-10.pdf>.
- OMG 1999 CORBA Telecom Log Service Specification: <ftp://ftp.omg.org/pub/docs/formal/00-01-04.pdf>.
- OMG 2000a Management of Event Domain Specification: <ftp://ftp.omg.org/pub/docs/formal/01-06-03.pdf>
- OMG 2000b CORBA/FTAM-FTP Interworking Specification: <ftp://ftp.omg.org/pub/docs/telecom/00-02-04.pdf>.
- OMG 2000c Wireless Access and Terminal Mobility Specification: <ftp://ftp.omg.org/pub/docs/dtc/01-05-01.pdf>.
- OMG 2000d Telecom Service Access and Subscription Specification: <ftp://ftp.omg.org/pub/docs/dtc/00-10-03.pdf>.
- OMG 2001a Telecommunications Domain Task Force: http://www.omg.org/telecom/telecom_info.htm.
- OMG 2001b The OMG's Model Driven Architecture: <http://www.omg.org/mda/>.
- OPENARCH 2001 Proc. IEEE Conf. on Open Architectures and Network Programming, OPENARCH 2001, Anchorage, Alaska.
- Parlay 2001 The Parlay Group, <http://www.parlay.org>.
- Perkins C 1996 IP Mobility Support. *IETF RFC 2002*.
- RM 2000, Archive of the Workshop on Reflective Middleware (RM'2000), <http://www.comp.lancs.ac.uk/computing/RM2000/>.
- Seigel J 2001 Quick CORBA 3. John Wiley and Sons.

- Sun 2001 Java APIs for Integrated Networks, <http://www.sun.com>.
- Szyperski C 1998 Component Software: Beyond Object-Oriented Programming. Addison-Wesley.
- van der Linden R 1993 An Overview of ANSA. Document APM.1000.01, A.P.M. Cambridge Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, UK.
- Wang N Schmidt DC O’Ryan C 2001 Overview of the CORBA Component Model. Component-Based Software Engineering: Putting the Pieces Together, George Heineman and Bill Council, eds., Addison-Wesley, Reading, MA.