Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

ANSA Real-Time QoS Extensions

Guangxing Li, Dave Otway

Abstract

Binding, explicit binding, QoS, real-time..

APM.1094.00.06 **Draft** 23 January 1994

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Copyright © 1994 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

ANSA Real-Time QoS Extensions

Request for Comments (confidential to ANSA consortium for 2 years)

ANSA Real-Time QoS Extensions

Guangxing Li, Dave Otway

APM.1094.00.06

23 January 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1993 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

1	**1**	**Introduction**
1	1.1	Objective
1	1.2	Audience
2	1.3	The problem space
2	1.4	Outline of the document
3	**2**	**A framework for explicit binding**
3	2.1	The ANSA/ODP computational model
3	2.2	Binding
4	2.2.1	Implicit and explicit binding
4	2.2.2	The rule of explicit binding
4	2.3	Designing a binding model
5	2.4	A framework for explicit binding
6	2.5	A simple binder
7	2.5.1	The basic binder
7	2.6	A channel binder
8	2.7	A group binder
9	2.8	A switch binder
9	2.9	**Summary**
11	**3**	**Quality of Service**
11	3.1	QoS characteristics
11	3.2	Towards a QoS framework
12	3.2.1	QoS domain
12	3.2.2	QoS management
12	3.2.3	QoS basic mechanisms
13	3.2.4	QoS common expression
13	3.2.5	QoS transparency mechanisms
13	3.3	A language based approach to QoS
14	3.4	QoS and computation entities
14	3.5	Designing a QoS language
14	3.5.1	Expressing QoS
14	3.5.2	QoS items
15	3.5.3	Combination
15	3.5.4	Conformance check
15	3.5.5	Construction
15	3.5.6	BIND operation
15	3.5.7	Management/control interface
15	3.5.8	Domain
15	3.6	A simple QoS language
19	3.7	QoS and interface
20	3.8	QoS matching
20	3.9	QoS negotiation

20 3.10 Computational and engineering views
21 3.11 Summary

23 4 A model of real-time programming
23 4.1 Real-time programming model
23 4.2 Real-time communication
24 4.3 Real-time objects
24 4.3.1 Distributed object execution
25 4.3.2 ANSA object execution
26 4.3.3 Real-time objects
28 4.3.4 Real-time object invocation
29 4.3.5 Scheduling
30 4.4 Priority scheduling
30 4.4.1 Priority management and priority inheritance
31 4.4.2 Resource allocation and task preemption
32 4.4.3 Dealing with priority inversion
33 4.5 Deadline scheduling
34 4.6 Other scheduling paradigms
34 4.7 Application controlled rendezvous
35 4.8 Summary of the real-time programming model
35 4.9 Real-time communications
35 4.10 Towards a parallel protocol stack
36 4.11 Towards a timed RPC protocol
37 4.11.1 Discussion of problem
38 4.11.2 The protocol
40 4.11.3 Server deadline expiry
40 4.12 Towards a decomposable RPC protocol
42 4.13 Summary

43 5 A model of real-time QoS
43 5.1 QoS domains
43 5.2 Binders

46 6 Related work
46 6.1 IMAC
46 6.2 Lancaster work
46 6.3 OSI QoS framework
47 6.4 CNET work

48 7 Conclusion
48 7.1 Acknowledgment

1 Introduction

1.1 Objective

This document is aimed at the provision of an ANSA/ODP compliant framework for real-time applications. The framework provides a structured collection of concepts, their relations and tools which enable the design and implementation of ANSA/ODP compliant distributed real-time programming systems.

The design principles for the framework are

- abstraction: minimizing the amount of engineering detail that the application programmer is required to know, yet not preventing the programmer from exploiting the benefits of a distributed, real-time environment.
- automation: automatic tools are used to translate, optimize, transform and check application programs to minimize the complexity involved with the diverse and complicated real-time resource management activities.
- separation of concerns: most real-time computing are mainly concerned with resource management for guaranteeing a certain degree of Quality of Service (QoS). The principle of separation of concern states that, whenever possible, a single system resource should be manipulated separately from other system resources. The consequence of the principle is that it supports the construction of extensible systems through the ability to add new resources, and remove existing resources without disturbing un-related parties.
- integration: this principle states that real-time services (objects) are treated as first class citizens in a system environment. That is, operations and mechanisms (such as trading, security, monitoring, replication, location, migration and federation) provided for existing non-real-time components can be applied to, and used by, real-time components. The principle allows the evolution of the architecture from the development of individual real-time systems, to groups of real-time systems and then to the enterprise-wide real-time systems.

1.2 Audience

The audience of this document are expected to be the designers of distributed real-time environments. The audience are assumed to be familiar with the ANSA/ODP Computational Model [Rees 93] and the Engineering Model [Herbert 91].

1.3 The problem space

It is the stringent timeliness and performance nature of real-time applications that are the primary source of problems. These applications introduce new problems of behaviour and resource requirement specification and management in addition to the basic ANSA/ODP distribution problems.

It is commonly accepted that behaviour and resource requirement constraints can be addressed by QoS statements, while the implementation of QoS constraints can be addressed by binding facilities in ODP context [ISO/ODP 93].

Both QoS specification and binding model in current ANSA/ODP architecture are in early-development stage. This document provides a refined framework for extending the current ANSA/ODP work in the retrospect.

1.4 Outline of the document

Chapter 2 presents a detailed binding framework.

Chapter 3 gives a QoS framework.

Chapter 4 discusses a real-time programming model.

Chapter 5 presents the real-time programming model within the general binding and QoS framework.

Chapter 6 discusses related research.

Chapter 7 gives the conclusions

2 A framework for explicit binding

This chapter first reviews the current ANSA computational model, then discusses a framework for explicit binding.

2.1 The ANSA/ODP computational model

The current ANSA computational viewpoint is based on a location independent object-based model of distributed systems. In this model, interacting entities are treated uniformly as objects, which encapsulate states and behaviour. Objects are accessed through interfaces which define named operations together with constraints on their invocation. Interfaces are specified in an abstract data type language known as an Interface Definition Language (IDL) in ANSA and Interface Definition Notation (IDA) in ODP.

An activity takes place in the model when objects invoke named operations in an interface of another object. Services are made available for access by exporting interfaces to a database of service interfaces available in the system known as the trader. An object wishing to interact with a service interface must import the interface by specifying a set of requirements in terms of a typed name and attribute values. This will be matched against the available services and a suitable candidate selected.

Also central to the ANSA/ODP computational model is the notion of transparency whereby selected aspects of systems can be made invisible to applications. This is achieved by means of notional transparency managers interposed between the application and the support layers. It is an important principle that transparencies are selective and not prescribed by the system. This means that applications can choose to gain access to lower level functionality where required.

2.2 Binding

Binding is the process by which an activity in one object establishes the ability to invoke operations at an interface to some other object. A binding establishes and controls the communication sessions involving multiple objects so that their interactions are possible.

Binding is a complex process that may affect

- service preparation (setting up a service interface).
- service trading (service type checking, and service matching).
- examining the communication connectivity among the participants of an activity participants.
- activity instantiation.
- negotiating QoS parameters among the participants.

- controlling the validation of a binding.
- controlling the membership of a binding.
- changing QoS parameters, etc.

2.2.1 Implicit and explicit binding

According to the ODP terminology, a binding is an object, instantiated as a result of an explicit binding action, controlling the relationship between a set of bounded interfaces. A binding object has a control interface containing operations to change the set of interfaces it connects and the characteristics of the binding.

There are two kinds of bindings in the ODP model:

- implicit bindings --- which are established by the infrastructure without a explicit binding action
- explicit bindings --- which are established by an explicit binding action.

Current ANSA computational model does not cover explicit binding and it is this document's motivation for extending the model in the retrospect.

2.2.2 The rule of explicit binding

Generally, it is a binding that allows interoperation. An implicit binding is the one established according to a default rule. In ANSA engineering model, an implicit binding establishes the minimum functional support for interoperation. This is only sufficient when the interactions among objects are simple and of only functional requirements. If the interaction among objects become complicated and has QoS requirement (rather than just functional requirements), an implicit binding model (or a default action) is not enough. In this case, a facility is required in a computational model for

- allow the specification of individual interaction requirements, i.e. QoS.
- allow the management of QoS.

This facility is referred as explicit binding.

2.3 Designing a binding model

On designing a practical interaction model on which to base binding, the following aspects should be taken into account:

- interface implementation
- discovering interface
- type checking
- establishing and managing QoS

The first three aspects have been discussed in [Herbert et al. 93]. It is the last aspect still lacks treatment. Establishing and managing QoS is a complicated issue that is mainly determined by the infrastructure resource management functions. The treatment is therefore depends on the exact platform. For example, it is generally impractical to handling real-time QoS on a non-real-time platform.

Given this platform dependent nature of QoS, some important requirements for designing a binding model seem to be:

- simple --- a binding model has direct influence on the engineering design, it is therefore not ideal if there are any unrealistic assumptions on the technology available. For example, a design requires the existence of distributed scheduling is not a good design, simply because there are no generally useable solutions to the problem.
- generic --- QoS operations are normally problem oriented, and a binding model should be designed to be able to cater for individual problem domain. The varsity of different QoS solutions requires the model to be a generic framework, while relying on transparency tools to apply for specific problem areas.
- recursive --- the binding model should allow self-expansion, i.e. using the same conceptual model for the construction of better functional binding models. This allows the evolution of technologies.

2.4 A framework for explicit binding

This section proposes a framework for explicit binding that is driving from the design criteria given in the last section.

Traditionally in ANSA system, it is the nucleus that provides the required resource management functions for interoperation among distributed objects. Research and prototyping work on engineering design in the past has been concentrated on minimum functions, or a smallest nucleus. This has helped the port of ANSA testbench on various platforms, and the application of ANSA architecture on various problem domains. The nucleus interface is transparent to application programmers, and operated by implicit binding operations.

Recognizing it is the nucleus that provides the necessary resources for interoperation, an important aspect of an explicit binding model is the framework of how resources are managed for bindings for the provision of a required interaction pattern (or QoS). We propose to upgrade the ANSA engineering model with a set of resource managers each provides a brand of QoS for the particular resource it manages. Implicit binding operations (which uses the resource managers in a pre-defined manner with little or no QoS statements) will be used as before for default functional operations. Explicit binding operations are allowed to manipulate the resource managers explicitly with QoS statements to achieve the required interaction patterns.

Note: Editorial: this is in coherent with the intuitive meaning of explicit bindings: they are for the manipulation of engineering resources for the provision of QoS. This introduces the following issues:

Note: (1) computational representation of engineering mechanisms, for example, can we define engineering mechanisms with IDL and DPL? At least some interfaces should be defined to allow the manipulation of engineering.

Note: (2) a generic engineering framework for accommodating different resource managers. In other words, the engineering model should be an open, generic and recursive framework that allows the incorporation and expansion of resource managers for various QoS problem areas.This is in contrast to the simple and more closed view of engineering model required by implicit binding operations.

We propose an explicit binding framework with the following three components:

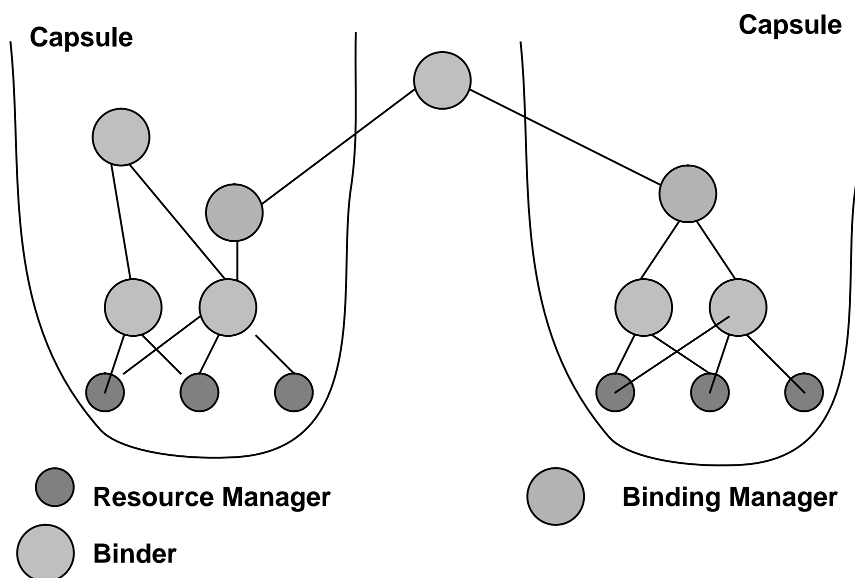
- resource managers --- which are the resource management interfaces for providing QoS within their capacity, these are the engineering components.
- binders --- which create bindings by using of resource managers, these are the transparency mechanisms to provide the computational view of engineering.
- binding mangers --- which are user programs that manipulate binders for individual applications or problem domains.

The three components are ANSA objects, i.e. units of distribution.

The framework has a hierarchy property.A binder functions as a QoS mapper, it maps a level of QoS to a set of low level QoS that can be satisfied by a set of low level resource managers. Such a model is recursive: a binder can be build on an existing set of binders and binding managers to provide higher level QoS and bindings. Figure 2.1 shows such a hierarchy model.

The model presents a hierarchy view of resource management. The lowest level is not assumed to manage networked resources. And a higher level may not be constrained to be a local resource manager. Engineeringly, only the lowest level resource managers and binders are necessary. Other higher level resource management activity can be modelled as application level binders (services). Such binders (or binding managers) can be designed in the same way as the lower level binders, to provide complicated QoS support, such as end-to-end QoS and group QoS.

Figure 2.1: A framework for explicit binding



2.5 A simple binder

This section describes a very simple binder conforming to the binding framework. The binder is directly driven from the ANSA implicit binding operations.

ANSA system creates *plug, socket* and *session* objects by implicit binding operations. A socket and plug pair consists of a communication channel through which a client can interact with a server. A session is a cache of a plug or a socket to store the end-to-end state required for one invocation and to synchronise the execution of the tasking and the communication system. From the binding point of view, a plug, a socket or a session are all partial components of a binding. A simple binding consists of one plug, one socket and one pair of sessions, while a complicated binding may consist of many plugs, sockets and sessions.

2.5.1 The basic binder

The basic binder (BasicBinder) is a local system service interface, it provides the required function for explicit control over sockets and plugs.

BasicBinder provides the following operation:

```
{socket} <- BasicBinder$BIND(svr_if) QoS
```

```
{plug} <- BasicBinder$BIND(clt_if) QoS
```

The BIND operation creates a socket when binding a server interface with a required QoS. It creates a plug when binding a client interface with a required QoS. Both the socket and plug are the created bindings: the management interfaces.

A socket (or a plug) is an interface through which explicit binding control operations are implemented. Some typical operations are:

- change QoS

```
socket$ReBind() newQoS
```

- unbind

```
socket$UnBind()
```

Apart from associating QoS to sockets and plugs, QoS can also be associated with sessions, which are created dynamically at invocation time:

```
clt_if$operation(args) QoS
```

A session pair is created implicitly (unlike plug and socket) as a cache of socket and plug, representing the dynamic resources needed for a single invocation. The QoS parameters associated with an invocation are implicitly associated with sessions, this is also done by the BasicBinder which manages sessions transparently from application programmers. This models the requirement for dynamic resource management, while the plug and socket binding models static resource management requirement. They are complimentary mechanisms, each has its own applications. For example, the socket and plug binding can be used to set up a private communication channel between a client and a server, while the session binding allows to choose the dynamic channel parameters based on each individual invocation.

2.6 A channel binder

Based on the basic binder, higher level binders can be built. This section describes a simple end-to-end binder --- a channel binding binder.

A channel binding or a channel consists of a plug and a socket. The QoS associated to a channel can be divided into two end-QoS, one for the plug and another for the socket. How the channel level QoS is transformed to the two end-QoS is a matter of channel QoS policy issue.

The channel binding model consists of three participants, two binding managers and the binder itself. For each end of a channel, i.e. the client and server site, there is a binding manager which controls the creation/destrction of local plugs, sockets and the association of QoS to them. A channel can be created by a channel binder, which can be either at a client site, server site or any other site.

The channel binder has a similar interface and operations like the basic binder:

```
{channel} <- ChannerBinder$BIND(clt_if, svr_if) QoS
```

This operation may operate as follows:

- find out the relevant binding manager interfaces through the clt_if and svr_if.
- drive out the two end-QoS, this may involve QoS negotiation between the binder and the two end binding managers.
- the server site binding manager sets up a socket.
- the client site binding manager sets up a plug.
- return a control interface as the channel binding.

A channel has similar QoS operations as a socket or a plug:

```
channel$ReBind() newQoS

channel$unBind()
```

2.7 A group binder

A group binding is a binding that may involve many clients and servers. Like the channel binding model, the group binding model consists of a few binding managers and the binder itself. The group binder has a group creation operation:

```
{group} = GroupBinder()QoS
```

This operation creates an empty binding group, which may have the following operations:

```
group$add_client(clt_if)
group$add_server(svr_if)
group$ReBind() newQoS
group$unbind_client(clt_if)
group$unbind_server(svr_if)
group$unbind()
```

The group talks to the relevant end binding managers to set up the relevant end bindings and other auxiliary services.

2.8 A switch binder

Suppose there is a server Q and a client P, and there is no direct network connectivity between the two, but there is an intermediate object X who has connectivity to both Q and R. In other words, Q and X share a common transportation protocol, P and X share a common transportation protocol, but not P and Q. In this scenario, it is impossible to use the implicit binding model to allow the interaction between P and Q. The problem is to design an explicit binding model to allow X functions as a network bridge or switch to allow the interoperation between P and Q.

The problem can be defined more specifically as follows. We want to have a bind operation to allow *"bind P->Q" through switch X*. Q has an interface Q_svr, P calls Q through X, i.e. X has a shadow interface X_Q_svt of Q_svr, it passes all calls to Q_svr; P is a client of X_Q_svr.

The switch binding model assumes each object has a binding management interface, say Q_Mgr, X_Mgr and P_Mgr.

The operation *"bind P->Q" through switch X* may work as follows:

- Call Q_Mgr to create the server interface Q_svr (create a socket);
- Pass Q_svr to X_Mgr, which has a JOIN operation works like:
  - — bind to Q_svr (create a plug, a client of Q_svr);
  - — create an interface X_Q_svr (create a socket), which is a shadow interface of Q_svr, it passes all calls to it to Q_svr through the plug.
- Pass X_Q_svr to P_Mgr, which then bind to the interface (create a plug, a client of X_Q_svr).

After the operation, a switch binding is set up; P and Q can interoperation. This scheme implements a software interceptor: in the sense that X needs to know the specific server interface type. Alternatively, X can be designed to provide a "short circuit" JOIN operation.

JOIN(socket, plug, policy) means:

- all incoming messages to the socket are passed to the plug at the transportation level, without walking up higher level protocols (such as marshalling, un-marshalling etc).
- all the returning messages to the plug are passed to the socket without walking up the higher level protocols, also at the transportation level.

This JOIN operation is an X nucleus provided one, allowing the nucleus functions as a software switch. By this operation, X does not need to know what is the server interface in Q, and therefore X can be a generic switch.

The policy parameter chooses the required message transformation policy applied when switching a message between the plug and the socket.

2.9 Summary

This section presents a framework for explicit binding. Some typical binding models are designed as binder examples in the framework.

3 Quality of Service

3.1 QoS characteristics

QoS is a generic mechanism which can express performance requirement for a user and performance provision for a server. It can also be used to conduct the negotiation of the required performance and the provided performance.

The main purposes of QoS are to express:

- the provided performance
- the required performance
- the required resources for the provision of a service
- the required resources for the access of a service

In our work, we are interested in QoS with the following characteristics.

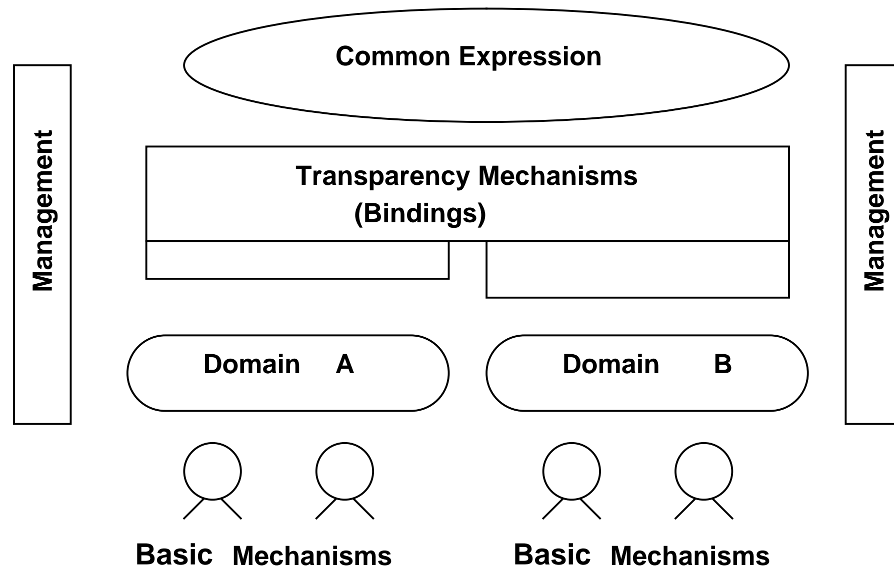
- QoS are categorised: for a particular class of application area, a particular QoS domain is required. A universal QoS domain for many applications is a non-goal of this research.
- QoS are behaviour and/or resource constraints: QoS are used to model the behaviour or resource requirement of a system. This can be either declarative or imperative.
- QoS are quantified: QoS can be itemized to individual quantified parameters.
- QoS are combinative: QoS items can be combined in various ways for the purpose of specification.
- QoS are directorial: QoS are hints in the selection of mechanisms and policies to meet the various requirements and the interaction between them.

3.2 Towards a QoS framework

As QoS are categorised and there are many different QoS domains, it is unrealistic to use the same mechanism for all of them. On the other hand, we are interested in a common architecture that can afford many QoS domains, it is therefore logical to work out a framework so that QoS domains can co-exist and relate to each other. A QoS framework provides a common conceptually model for the definition, organization, co-relating, management and engineering of different domains of QoS.

We propose a framework consisting of five major components (shown in Figure 3.1): QoS domain, QoS mechanisms, QoS management, QoS expression and QoS transparency.

Figure 3.1: A QoS Framework



3.2.1 QoS domain

A QoS domain is associated with a specific area of application or service. There can be many QoS domains, such as, transportation QoS, real-time QoS, multi-media QoS and OSI QoS.

A QoS domain defines a specific set of QoS parameters and their combinations. A QoS domain can be constructed upon other QoS domains. For example, a multi-media QoS domain may consists a transportation QoS domain and a real-time QoS domain.

3.2.2 QoS management

For each QoS domain, there are generic and/or specific QoS management tasks. The QoS management functions include:

- provision,
- initiation,
- change,
- enquiry,
- termination,
- monitoring,
- notification,
- negotiation

3.2.3 QoS basic mechanisms

For each QoS domain, there exists a set of basic mechanisms for the provision and management of the QoS items in the domain. This is the engineering support for the operation of QoS. For different QoS domains, this layer of

interfaces can be different substantially. There is no need for a common interface for all QoS domains.

3.2.4 QoS common expression

QoS are required to be associated with computational entities in the ODP architecture. As these computational entities are expressed in a generic language form, it implies the requirement for a common QoS expression.

Note: Editorial: a common QoS specification does not mean a common syntax, as demonstrated by DPL, it rather mean a common semantics, leaving the transparency tools to handle the difference in syntax, infrastructure etc.

There are two possible approaches for the specification:

- Application Programming Interface (API) approach --- for each QoS domain there is a set of QoS programming interface, this is the common approach in operating system practice.
- language-based approach --- a common QoS language is used to description QoS item and their combination. This approach is preferred because of its adaptability and portability as discussed in Section 3.3.

3.2.5 QoS transparency mechanisms

Given the semantic gaps and the varieties in the basic QoS engineering mechanisms, a QoS transparency layers is required to provide the necessary adaptation of the language level QoS to the basic QoS mechanisms.

The QoS transparency mechanism is another set of ODP engineering transparency tools, addressing non-functional requirements.

3.3 A language based approach to QoS

Many approaches have been developed for QoS specification and management. It is believed the API approach is useful but not adequate, and therefore a language based approach is both important and necessary. A language based approach to QoS provides a necessary level of abstraction for:

- hiding engineering details which are irrelevant to application programmers
- hiding arbitrary syntactic differences between technologies
- keeping concepts as independent and orthogonal as possible
- imposing the simplest possible conformance rules
- providing the maximum configuration flexibility

A language based QoS also allows the application of automation tools to

- compile application programs onto any suitable technology
- optimize application programs for any particular configuration
- transform declarative requirements into imperative statements
- check that the application program will execute correctly

3.4 QoS and computation entities

QoS can be associated both with service providers and users. The intermediate mechanisms for the association of provided QoS and the required QoS is called binding. Bindings become *first class citizen* constructs in the computational model as explained in the last chapter.

In relation with other computational components, QoS can be classified as:

- QoS associated with an interface template
- QoS associated with an instance of an interface template
- QoS associated with an object template
- QoS associated with an instance of an object template
- QoS associated with an activity
- QoS associated with a binding

Of these, the QoS associated with bindings are the most complicated ones. As demonstrated by our binding model, binding can be organized as end-system binding, end-to-end (channel) binding and group binding etc.; QoS associated with these bindings are therefore representing end-system QoS, end-to-end QoS and group QoS etc. The recursive binding model also implies the recursive nature of the QoS transparency mechanisms. Strictly, only QoS is the required computational extensions for performance oriented applications; while bindings are the engineering mechanisms for QoS implementation.

QoS associated with other computational entities (i.e. non bindings) can be seen as QoS templates, they are not activated until the relevant bindings are set up at run time.

3.5 Designing a QoS language

3.5.1 Expressing QoS

QoS rules may be needed to:

- identify relevant QoS domain
- identify relevant attributes in a QoS domain
- express quantitative measures for the identified attributes
- express combinations of individual attributes
- allow conformance check
- allow negotiation
- allow the construction of new QoS domains with existing domains
- describe domain dependent management/control interfaces

3.5.2 QoS items

The individual QoS items (or characteristics) depend on the individual platform or domain. For example, on a normal commercial real-time platform, QoS item can be a priority, a deadline or the allocated tasks etc. On a communication system, QoS items can be the required transportation QoS e.g. jitter, error-rate etc.

A QoS item can be expressed simply as a <name, value> pair, such as <Priority, 10>, or more meanfully as a <name, relation, value> triple, such as <Priority, =, 10> and <Jitter, <=, 100>.

3.5.3 Combination

It requires at least three forms of combinators to combine QoS expressions:

- logical combinators: and, or, not etc. Examples: "<priority, 10> and <deadline, 100>"
- order combinators: expressing the ordering of perceived satisfaction. For example, either QoS_1 or QoS_2, "either <protocol, TCP> or <protocol, IPC>"
- range combinators: expressing a QoS negotiation. For example, "<Rate, >=, 10> and <Delay, <=. 15>".

3.5.4 Conformance check

Conformance check is a sort of *QoS match* procedure. Conformance check can be both "syntactic" or "semantic". In the first case, it can be realised by a syntax analyser, by a type checker or by some match criteria. In the second case, it can be realised as parts of QoS negotiation procedure.

3.5.5 Construction

This is the component for the construction of new QoS domains with existing ones. It may be equivalent to some type constructors. For example, a "record", a "union", or a "set" constructor.

3.5.6 BIND operation

Each QoS domain has a BIND operation for the creation of bindings in the domain.

3.5.7 Management/control interface

Each QoS domain has a few management interfaces which provide operations for QoS related operations. For example, a QoS domain at least has a binding interface which is the result of the BIND operation.A binding interface has at least an UNBIND operation. Other operations of a management interface may be ChangeQoS, AddNewMember etc. Other management interfaces include notification interface, monitoring interface etc.

3.5.8 Domain

A QoS domain contains all or some of the components defined in section 3.5.2 to section 3.5.7.

3.6 A simple QoS language

ANSA system has already got one simple Trader constraint language for specifying QoS offers and constraints, and for matching QoS constraints to offer. The language was used in IMAC [Nicolaou 91] for simple multimedia QoS representation and negotiation. The language is therefore chosen to start with.

The short BNF for the constraint language is given in Figure 3.2.

Figure 3.2: Trader Constraint Language

```
<constraint> := <empty>
              | <expr>
              | <expr> -> <superlative>
              | -> <superlative>

<expr> :=       <expr> or <expr>
              | <expr> and <expr>
              | not <expr>
              | ( <expr> )
              | <nexpr> in <nexpr>
              | <nexpr> == <nexpr>
              | <nexpr> != <nexpr>
              | <nexpr> < <nexpr>
              | <nexpr> <= <nexpr>
              | <nexpr> > <nexpr>
              | <nexpr> >= <nexpr>

<superlative> := min [ <nexpr> ]
               | max [ <nexpr> ]

<nexpr> :=      <term>
              | <nexpr> + <term>
              | <nexpr> - <term>

<term> :=       <factor>
              | <term> * <factor>
              | <term> / <factor>

<factor> :=     <id>
              | <constant>
              | ( <nexpr> )
              | - <factor>

<empty> :=
```

In comparison with the requirements shown in the last section, the main deficiencies of the language are its inability to specify QoS attributes and the construction of new QoS domain. This is not a problem for the Trader in which the constraint expressions are used in a purely symbolic or syntactic sense. The extension for the definition of a QoS domain is necessary because our QoS expressions are required to be associated with bindings, i.e. each QoS expression has to be tied to a binding domain, or a binder. A QoS domain constrains the identifiers in a QoS expression to certain key words. Figure 3.3 shows our QoS domain definition language, which is an extended IDL language.

A QoS domain defines:

- attributes or KEYWORD and their types
- the BIND operation
- the management interfaces
- QoS constraint macros

Figure 3.3: QoS domain specification language

```
<spec> :=           <header> <needs> <body>

<header> :=         <id> : QoSDOMAIN =

<needs> :=          <empty>
                    | <needs> <need>

<need> :=           NEEDS <id> ;

<body> :=           BEGIN <declrations> END.

<declarations> := <empty>
                    | <declarations> <declaration>

<declaration> :=    <id> : TYPE = <type>;
                    | <id> : KEYWORD = <type>;
                    | <macro_header> : MACRO "<constraint>";
                    | <id> : <proc>;
                    | <id> : <interface>;

<proc> :=           OPERATION [ <arguments> ] <QoS_trailer>
                    RETURNS [ <results> ]

<interface> :=      INTERFACE <if_body>

<if_body> :=        BEGIN <if_specs> END.

<if_specs> :=       <empty>
                    | <if_specs> <if_spec>

<if_spec> :=        <id> : TYPE = <type>;
                    | <id> : <proc>;

<QoS_trailer> :=    <empty>
                    | QOS
```

The QoS constraint expressions (the Trader constraint language) are extended accordingly to allow the specification of *structured keyword* by using a dotted notation, such as the REX.protocol etc.

The QoS constraint language and the domain definition language constitute our QoS language.

To make these ideas more concrete, we will present some examples shown in Figure 3.4. and Figure 3.5.

Figure 3.4 defines two QoS domains: IPC and CHANNEL. The IPC QoS domain defines four keywords: EndType, Protocol, Rate and Address. They can be used as identifiers for constituting QoS expressions in the domain. An IPC QoS expression is show in Figure 3.5. IPC has one management interface, i.e. the binding interface, which provides two control operations: UnBind and ReBind. The IPC BIND operation takes any interface reference as argument and creates a IPC binding (interface) as result. The IPC domain models a typical inter-process communication scenario where communication has two parties: one socket and one plug. The BIND operation sets up a socket and a plug before communication can happen.

The CHANNEL QoS domain defines six keywords. Of these, Socket and Plug are of the type IPC.K, i.e. they are structured keywords. It is conventional to name the type of the record of the keywords in a QoS domain D as D.K; thus

IPC.K is the record type of the keywords in IPC. This models a layered domain: CHANNEL is built on top of IPC. CHANNEL has two management interfaces: one as the binding interface, the other for event notification. The BIND operation takes three interface references as arguments: a server interface, a client interface and a notification interface. The notification interface can be used to *call-back* the binding creator when there are relevant events to be reported.

Figure 3.5 defines a MEDIA QoS domain, which is built on top of the CHANNEL. It also illustrates examples of QoS macros and QoS expressions for each of the domains.

Figure 3.4: QoS domain and constraint examples

```
IPC: QoSDOMAIN =
BEGIN
   EndType: KEYWORD = {socket, plug};
   Protocol: KEYWORD = {TCP, UDP, MSNL};
   Rate: KEYWORD = INTEGER;
   Address: KEYWORD = STRING;

   IPC_binding: INTERFACE =
   BEGIN
     Result: TYPE = {ok, fail, retry}
     UnBind: OPERATION [] RETURNS [Result];
     ReBind: OPERATION QOS [] RETURNS [Result];
   END;
   BIND: OPERATION [IfRef: InterfaceRef] QOS RETURNS [IPC_binding].
END.

CHANNEL: QoSDOMAIN =
NEEDS IPC;
BEGIN
    Type: KEYWORD = {shared, non_share};
    Rate: KEYWORD = INTEGER;
    Address: KEYWORD = STRING;
    Protocol: KEYWORD = {one_way, two_way};
    Socket: KEYWORD = IPC.K;
    Plug: KEYWORD = IPC.K;
    CH_binding: INTERFACE =
    BEGIN
      Result: TYPE = {ok, fail, retry};
      UnBind: OPERATION [] RETURNS [Result];
      ReBind: OPERATION [] QOS RETURNS [Result];
     END.
    Notifier: INTERFACE =
    BEGIN
      Event: ANNOUNCEMENT OPERATION [event: INTEGER] RETURNS [];
    END.

    BIND: OPERATION [SvrIf: InterfaceRef,
                     CltRef: InterfaceRef,
                     NotifierRef: Notifier] QOS
           RETURNS [ CH_binding ];
END.
```

Figure 3.5: QoS domain and constraint examples: cont.

```
MEDIA : QoSDOMAIN =
NEEDS CHANNEL;
BEGIN
 Type: KEYWORD = {Audio, Video};
 Rate: KEYWORD = INTEGER;
 Delay: KEYWOE= INTEGER;
 Enconding: KEYWORD = {pal, mpeg, jpeg};
 Channel: KEYWORD = CHANNEL.K;

 M_binding: INTERFACE =
 BEGIN
  Result: TYPE = {ok, fail, retry};
  UnBind: OPERATION [] RETURNS [Result];
  ReBind: OPERATION [] QOS RETURNS [Result];
 END.
 audio(x,y):MACRO "(Rate>=x and Delay<=y)";
 video(x,y,z):MACRO "(Rate>=x and Delay<=y and Encodeing==z)";
 BIND: OPERATION [ Source: InterfaceRef,
                   End: InterfaceRef] QOS
                   RETURNS [M_binding];
END.
--QoS expression on IPC
(Type == Socket) and (Protocol == TCP) and (Rate == 100)

 --QoS expression on CHANNEL
 (Type == non_share) and (Protocol == two_way) and
 (Socket.Protocol == UDP) and (Plug.Rate == 1000)

 --QoS expressions on MEDIA
 (type == Video) and video(100, 10, pal)

 (type == Audio) and audio(1000, 6)

 --QoS expression and invocation
 USE IPC
 DECLARE {ipc_binder} : IPC CLIENT
 DECLARE {socket} : IPC.ipc_binding CLIENT
 DECLARE {svr_if} : Any_Server SERVER
 {socket} = ipc_binder$BIND(svr_if) (Protocol == TCP)
```

3.7 QoS and interface

QoS constraints can be associated with interfaces as QoS templates for simplifying QoS specification, interface trading and conformance test.

QoS constraints can be associated with an interface by extending interface type definitions, i.e. IDL, with QoS clauses. QoS clauses can be associated with individual operations or the whole interface at interface specification time. Figure 3.6 shows such an example.

QoS constraints specified at an interface type may be automatically inherited by an interface instance when it is created at binding time, where additional QoS constraints can also be imposed.

The QoS templates and the binding time QoS clauses can be organized into an interface reference, which can be used for interface trading and QoS conformance test.

Figure 3.6: IDL QoS Specification

```
VideoDev: INTERFACE =
NEEDS Media;
IF_QOS (Delay <= 100) and (Encoding == pal)
BEGIN
StreamIn: OPERATION [] (Rate <= 1000) RETURNS [];
StreamOut: OPERATION [] (Rate >= 100) RETURNS [];
END.
```

3.8 QoS matching

The constraint language provides well defined rules for matching a required QoS into offered QoS. For example, if the following QoS offers are made:

- (Protocol == TCP) and (Rate == 100)
- (Protocol == TCP) and (Rate == 1000)
- (protocol == UDP)

The constraint expression (Protocol == TCP) would return:

- (Protocol == TCP) and (Rate == 100)
- (Protocol == TCP) and (Rate == 1000)

Alternatively (Protocol == TCP) and (->max[Rate]) would return:

- (Protocol == TCP) and (Rate == 1000)

These rules can be used by a Trader for interface trading as illustrated by ANSA.

3.9 QoS negotiation

QoS negotiation requires a set of auxiliary facility for managing QoS. They are QoS domain related and should be defined as QoS domain management functions.

3.10 Computational and engineering views

Generally, our binding model is a way of structuring QoS into a manageable framework. There are four major components in the framework:

- binders: they provide a common BIND operation to generate various QoS domain specific bindings.
- a generic QoS language: it provides a common tool to specify various QoS expressions.
- QoS domain specific resource managers: for each QoS domain, there is a set of engineering mechanisms for the management and operation of bindings.
- a generic QoS language mapper: it links the common QoS expressions to domain specific binders and resource managers.

Figure 3.7: A QoS Transparency Framework

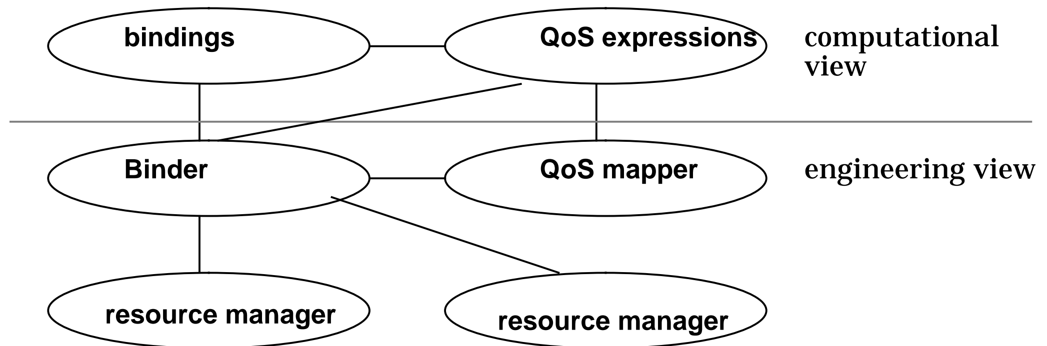


Figure 3.7 gives a graphical illustration of the framework in terms of computational and engineering views. The recursive binder model allows the use of bindings and QoS to achieve (recursive) layered QoS and their bindings.

3.11 Summary

This chapter presents a QoS framework.

4 A model of real-time programming

This chapter synthesis the earlier work on ANSA real-time architecture [Li and Otway 93] and experiment [Li 93a].

4.1 Real-time programming model

The essence of a real-time programming model is to provide the basic abstractions so that stringent timing constraints of real-time activities are respected (guaranteed at best).

The real-time programming model developed in [Li 93] is based on the ANSA computation and engineering models. As in the ANSA system, objects provide the basis for distribution, interfaces of objects provide service access points, and named operations of an interface provide the actual services. Abstractions, mechanisms and policies are developed to allow a programmer to access and control the resource allocation of the supporting environment. Tasks (representing processor resources) and communication channels (representing communication resources) are considered the most important system resources. Both static resource allocation --- the allocation of system resources to interfaces --- and dynamic resource allocation --- the allocation of system resources to invocations are supported. *Predictability, programmer control* and *mission criticality* are the main concerns of the real-time programming model.

4.2 Real-time communication

Real-time applications present more complicated functional requirements to the underlying communication systems. Three extensions aimed at making the ANSA communication system more suitable for real-time applications are identified:

- a parallel communication protocol stack to allow the preallocation of communication resources (a separate channel, for example) and the removal of layered multiplexing. This is required partially by the real-time programming model. The main gain of this design is that it allows the application to explore the communication QoS that the low-level operating environment can provide. For example, in an ATM environment, a channel (or a network circuit) is allowed to associate with various transportation QoS, such as jitter, delay, priority etc. Even in an ordinary operating environment, this design allows the choose of communication protocols, such as TCP, UDP, IPC etc.
- a timed RPC protocol to allow the association of deadlines with invocations. ANSA is an RPC based system. A basic goal of many RPC systems is to make the semantics of a remote call as close as possible to that of a local call. However, distribution cannot be completely ignored:

applications will have to deal with the possibilities of concurrent access to shared resources, variable latency in accessing resources and communication failures. The semantics of remote calls are implemented by RPC protocols. Two often referred to semantics are *exactly-once* and *at-most-once* executions. Real-time applications add another dimension to the problem: timeliness --- arbitrary delays associated with synchronous RPC invocations cannot be tolerated. Our solution to the timed RPC problem is the design of a dependable RPC protocol through which reasonable timing constraints (representing different trade-off between consistency and strictness) of a remote invocation can be specified clearly and enforced. This relieves the additional burden of having to monitor and manage timing constraints by application programmers during remote calls.

- a decomposable RPC protocol to allow the synthesis of the protocol to provide different levels of invocation semantics (such as exactly-one, at-most-once), so that an application programmer can customize the system to application-specific requirements of functionality and performance. This work is targeted at new transport protocols with QoS parameters in the operational interface.

The three designs are integrated within a coherent architecture to provide a communication infrastructure for real-time applications. *Predictability*, t*imeliness* and *performance* are the main concerns of the real-time communication system.

4.3 Real-time objects

This section discusses the real-time extensions of ANSA objects. The structure of the real-time objects is examined along with object invocation mechanisms, the handling of priorities and deadlines, resource allocations, scheduling mechanisms and policies, and the application's control over scheduling.

4.3.1 Distributed object execution

The use of an object-oriented data model and the client-server execution model makes the distribution of data and the processing implicit in nature. In non-real-time environments, object-oriented design has been successful in simplifying the design, implementation, and maintenance of software in many distributed systems.

Object interdependence can be classified into two categories: *static* interdependence --- the structural relationships between objects, and *dynamic* interdependence --- the interactions between objects. Many useful results are known about the static relationships between distributed objects. [Herbert 93a] [Blair 92]. Related concepts, such as abstract data typing, type checking and subtyping, are accepted and used widely. On the other hand, little consensus has been achieved on the execution view of objects. Many approaches to object execution have been proposed, some of which are the *active object* model [Black 86], the *passive object* model [Allchin 83], and the *actor object* model [Attoui 91].

For real-time applications, this execution aspect is of vital importance --- it has fundamental impact on the *predictability* of computational activities. Real-time object execution models are required to address not only how the computational activities are carried out, but also how shared resources are

used (i.e. the manner in which contention for system resources is resolved taking into account timing constraints of real-time activities). The latter issue is often neglected and considered irrelevant engineering detail in non-real-time computing. Distributed real-time systems must provide support for the specialized requirements of real-time communication, tasking, scheduling, and control. These requirements must be explicitly addressed in an object execution model, if the object-oriented approach is expected to be applicable to a real-time world.

4.3.2 ANSA object execution

The ANSA Object Execution Model (AOEM) is defined by the ACM and AEM. The AOEM can be summarised as follows.

- objects export services through interfaces.
- threads are created either explicitly for concurrent computational activities or implicitly by the invocations between objects. In the latter case, a thread embodies a distinct run-time agent for a client in its server side, representing the invocation on a computational interface.
- the infrastructure (capsule) is in charge of the management of resources (tasks, buffers etc.) in the system, and of their allocation to the different threads.

This means the system behaviour is completely dependent on the system's resource management policy. Also, the infrastructure offers no possibility of interacting with this management. Therefore, the resulting behaviour is totally non-deterministic, and nothing can be guaranteed; it depends entirely on the system workload.

4.3.2.1 ANSA object execution model deficiencies for real-time applications

To be more specific, ANSA Testbench is used as an example to detail the AOEM. In the Testbench, its time-sharing characteristics of tasking and scheduling can be summarised as follows:

- multiplexing of one thread queue. The queue is used for all interfaces within a capsule; and all system tasks are homogeneous --- they are allocated for serving any threads (requests on any interfaces).
- thread enqueue policy (and thus request service scheduling policy) is First Come First Service (FCFS).

Based on this single capsule-wide thread queue with a pool of tasks, the ANSA tasking system is very efficient at the task/thread resource sharing. However it imposes severe constraints on flexible and real-time scheduling. For example, it precludes the possibility of preallocating tasks for real-time interfaces (services). One aspect of the non-predictability caused by this design is if all system tasks have been assigned to some time-consuming non-real-time threads, newly arrived real-time requests (threads) have to wait until the completion of the non-real-time threads. Also this design precludes the possibility that an application performs its own resource management, synchronization and scheduling on the basis of services (interfaces) and tasks.

The simple FCFS thread enqueue policy precludes any real-time performance, when the object is executed in an open environment where time constrained and non-constrained operations are allowed to be requested dynamically.

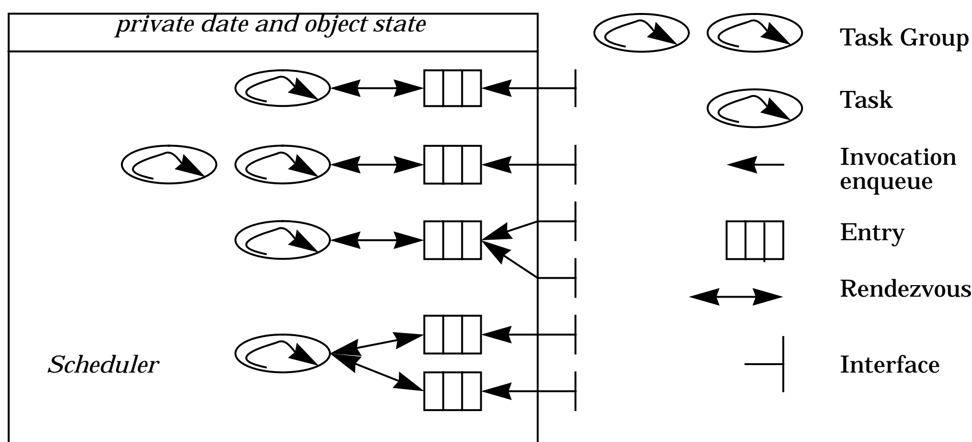
4.3.3 Real-time objects

A real-time object model can be obtained by extending the ANSA object execution model with explicit resource allocation and real-time scheduling support.

A real-time object is composed of data, one or more tasks of execution, and a set of interfaces. A new abstraction, ***entry***, is introduced as the basic mechanism for real-time scheduling.

An entry is a thread queue with a record of control data. An entry may be created dynamically, and interfaces of an object may be bound to it. When an interface is bound to an entry, each operation request on the interface will be transferred (by the infrastructure) to a thread enqueued on the entry. Any thread representing a computational activity is also spawned on an entry. The entry is an engineering concept which is confined within a capsule.

Figure 4.1: Real-time object illustration



In Figure 4.1, a graphical illustration of a real-time object is given.

Flexible tasking is based on the entry abstraction. System tasks may be allocated for each individual entry. The tasks allocated are dedicated to execute the threads on the entry. When executing a thread, a task is also allowed to *rendezvous* with other entries dynamically. A ***rendezvous*** of a task with an entry means that the task waits to accept and execute one thread on the entry. Different control parameters may be selected for each entry to choose a thread enqueue policy, a task/entry rendezvous policy, and to enforce concurrency controls. These policy issues are discussed in the further sections.

In such an object model with data, interface, entry and tasks encapsulated within a capsule, there is a choice of how many entries are allocated, which interface is attached to which entry, how many tasks are allocated to an entry, whether a task can rendezvous with a specific entry, and what kinds of resource scheduling policies are used.

The choice to allocate a new entry for some interfaces reflects the need to separate these interfaces from others for the purpose of resource management.

The number of tasks allocated to an entry not only enforces the real concurrency allowed for the execution of threads on the entry, but also affects

the real-time scheduling properties, for example, *preemptivity,* as explained later in Section 4.4.2.

The flexibility for allowing a task to rendezvous with an entry enables an application to have complete control over its virtual processor(s) based on its knowledge of the system state.

The user control over system tasking behaviour is further enhanced by the scheduling policy/mechanism separation used in the architecture (detailed in Section 4.3.5.

These resource management activities can all be done dynamically, increasing the flexibility and usefulness in an open dynamic environment.

Some typical system configurations are illustrated below. Their combinations are straightforward.

The simplest form (Figure 4.2) is *Shared Single Entry* configuration, in which all interfaces share a single entry with all tasks serving all incoming requests on all interfaces.

Figure 4.2: Shared Single Entry (ANSA) Configuration

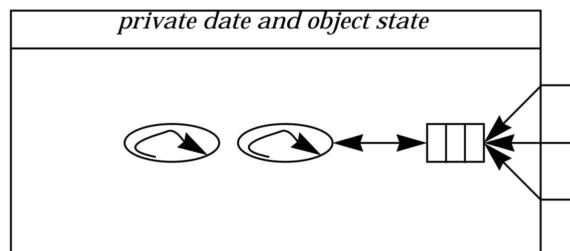
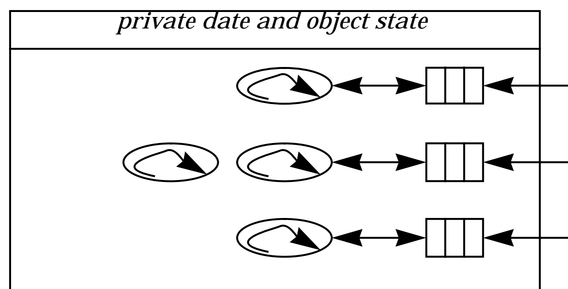
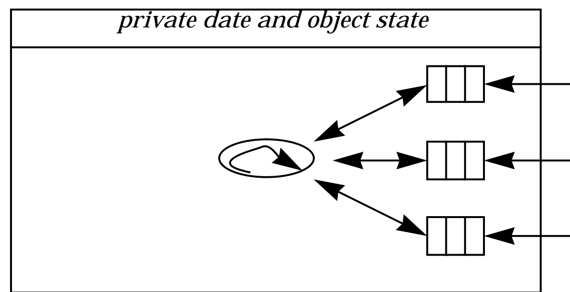


Figure 4.3: Multiple Single Entries



Another simple form (Figure 4.3) is *Multiple Single Entries*, in which each interface has its own entry.

Figure 4.4: Single Task Multiple Single Entry



Another interesting simple form (Figure 4.4) is *Single Task Multiple Single Entry*, in which the single task decides at its run-time which entry (interface) it would like to serve.

A combined configuration is illustrated in Figure 4.1. It contains the three simple configurations.

4.3.4 Real-time object invocation

The act of requesting that an operation of an interface be executed is termed an *invocation* (a synchronous call). Each invocation is conveyed as a message to the invoked object, and is then transferred to a thread in the capsule where the invoked object resides.

To support the mission-critical requirements, there must be some means to enable the urgency of a computational activity to be spread among all the nodes it needs to access; and that urgency information should be used by the system resource scheduler to resolve resource contention so that important or more urgent computational activities have better access to system resources. This is done in the real-time ANSA by allowing the association of an optional *priority* (criticality) and/or *deadline* with each invocation. As the invocation crosses the physical boundary and becomes a thread in the called object, this priority and/or deadline is passed and becomes a property of the thread, which may then be used as a scheduling parameter on the server site.

The priority and/or deadline of an invocation is independent of its contents (the invocation parameters) and context (the invocation thread). Allowing explicit invocation priority (and/or deadline) has several benefits: (1) it allows extra flexibility in conjunction with the server scheduler, in determining how the invocation is to be processed; (2) it allows a low-priority invocation to be sent from a high-priority task without having to enhance the server (thread) task's priority; (3) likewise, a low-priority thread may send a high-priority invocation to a server indicating the system has entered an urgent situation.

It should be pointed out that the priority and/or deadline is just a client's objective view of the criticality of an invocation; how that will affect the system resource management is also determined by the scheduling policy (the interpretation of the scheduling parameters) and the resources allocated for the service. This is further explained in the following sections.

4.3.5 Scheduling

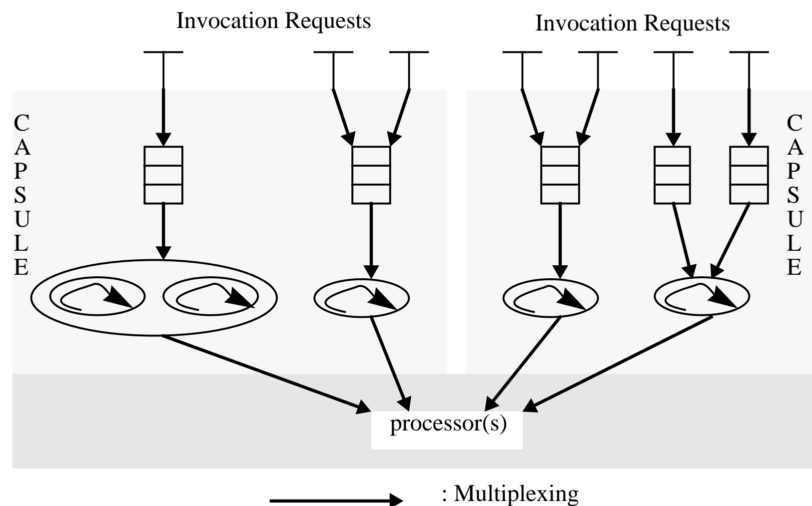
The main goal of the real-time ANSA tasking design is to allow the maximum control of scheduling at the application level. Care has been taken to achieve the balance between flexible and deterministic scheduling. A policy/ mechanism separation approach has been taken to address the diversity of real-time programming. Real-Time programming models have been devised for specific applications. Therefore, an ideal general purpose real-time support environment should provide multiple models of real-time programming. This can be supported by the multiple application-selectable scheduling policy modules on top of a shared set of scheduling mechanisms.

The system scheduling behaviour is defined in layers as:

- thread scheduling --- the rendezvous scheduler on each entry.
- task scheduling --- the nucleus scheduler on tasks.

Task scheduling and thread scheduling are two separate, but related scheduling domains. Task scheduling is defined in the nucleus or the underlying operating system kernel. Thread scheduling is defined per entry. Task scheduling manages multiplexing of task executions over processor(s). Thread scheduling manages the multiplexing of requests (thread) over tasks. Figure 4.5 illustrates the structure of this multiplex.

Figure 4.5: Threads, Tasks and Processor(s) Multiplexing



The primary function performed by multiplexing is the sharing of processor resources, which is similar to the multiplexing in communications systems and protocols for sharing communication resources [Nicolaou 91]. The use of separate entries to process requests on separate interfaces offers a number of (potential) advantages:

- allows the use of a specific scheduling policy (thread scheduling policy) suitable for each interface or each interface class.
- allows the possibility of using interface specific tasks to serve requests, and thus allows for more efficient resource utilisation.

- separate entries may be processed in parallel, thus increasing performance.
- allows the possibility of end-to-end scheduling and guarantees.
- preserves the modularity and separation of service interfaces.

The nucleus scheduler defines how the real processor(s) is assigned to tasks, i.e. it manages the context switches between tasks. Preemption is used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behaviour of the task execution.

There are two issues in thread scheduling management. One is how a thread is enqueued in an entry (with the assumption that the first thread in the queue is executed first). Such a policy may be a system defined one, like invocation priority based, invocation deadline based, or an application provided one.

Another issue is how a serving task rendezvous with a thread in an entry, i.e. how the thread scheduling parameters (priority and/or deadline) are used/ inherited by the task. This is defined by a task/thread rendezvous policy. Such a policy affects how the serving task competes for processor resources with other tasks. Priority inheritance is discussed further in section 4.4.1.

4.4 Priority scheduling

This section discusses the mechanisms needed to provide the *static priority based* scheduling model in the real-time ANSA framework. Static priority based scheduling is the most popular (and perhaps more important, supported) real-time scheduling method [Ada9X 93] [POSIX]. There are well-known analytic methods [Moitra 86] [Lehockzy 89] to decide the schedulability of a set of periodic or aperiodic tasks.

Though obviously related, priority and scheduling are different issues. Associating a notion of priority with an invocation is an intuitive way of structuring real-time applications. The *priority queuing* of threads in an entry is incorporated to support such a view of real-time applications.

While priority is a well defined and generally applicable notion, its role in ANSA task scheduling needs to be carefully examined. A clear definition of the *priority inheritance* (Section 4.4.1) and *priority ceiling* (Section 4.4.3}) --- used when the enforced synchronization during a task and a thread rendezvous --- is needed to understand how priority works on tasking.

4.4.1 Priority management and priority inheritance

A distinction is made between a task's ***static*** priority (that declared in its creation) and its ***dynamic*** priority (that is the static value potentially enhanced by a rendezvous or an explicit change of priority). It is the dynamic priority that is used by the nucleus (or operating system) schedule to determine the current system-wide *urgency* of a task.

The tasking model is designed to support a structured approach to priority management. Statically, the different task/entry/interface configurations allow important real-time services to be distinguished from non-real-time services. A dedicated entry may be allocated to real-time services, and high priority tasks may be allocated on the entry, so that request on the interface has better response time. Dynamically, a serving task may take into account the priority

of an invocation, and use this priority as its dynamic priority. This is called ***priority inheritance***.

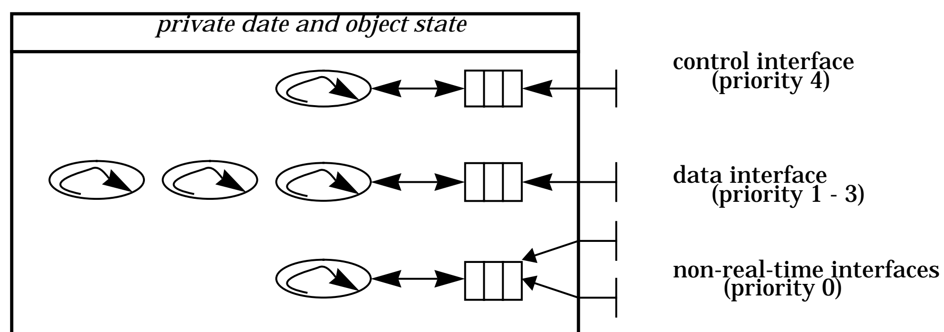
Two levels of priority inheritance schemes may be defined. They are called (basic) ***priority inheritance*** and ***transitive priority inheritance.*** In the first scheme, a serving task with a low priority raises its priority to the higher priority of an invocation request before it starts the service, and changes back to its original value after the service is completed. The second scheme is an extension of the first scheme to consider the situation when there are no waiting serving tasks and a high priority invocation request arrives. In this case, the invocation priority is compared with the priorities of the running serving tasks. If all of the serving tasks are running at priorities lower than the invocation priorities, one of the tasks is chosen to inherit the invocation priority. If at least one of the serving tasks is running at a priority which is higher than the invocation priority, then the invocation is enqueued in the entry.

4.4.2 Resource allocation and task preemption

Task preemption is a scheduling activity such that when a high priority task is ready to run, it starts processing immediately, by preempting a low priority running task (if any). Preemption is a basis of predictability.

In the real-time ANSA, task preemption may be caused by task allocation and/or priority inheritance. By allocating tasks of different priority to different entries, an application programmer may anticipate where and when preemption is needed. Priority inheritance provides a complementary mechanism to allow a serving task to use dynamically an invocation priority --- preemption happens if there is a serving task available and the invocation priority is higher than a current running task. This tasking model prompts a layered management of priorities as illustrated by the following example.

Figure 4.6: Layered Management of Priorities



One may allocate different levels of priorities to different real-time services, while priorities in one level may be used to identify the relative importance of an invocation among all the invocations on one interface. In Figure 4.6, three entries are allocated to serve non-real-time interfaces, a real-time data handling interface, and a real-time control handling interface separately. They are named as *n-entry, d-entry,* and *c-entry* respectively. In the n-entry, a task of priority 0 is allocated (assuming the smaller priority value means a lower priority), a FCFS thread enqueue policy is used, and therefore invocation

priorities are masked, and have no effects on the scheduling activities. Priorities 1 to 3 are assigned to the d-entry, on which three tasks of initial priority 1 are allocated. Invocations on the d-entry may thus have a priority range 1 to 3. In a single processor system, the three serving tasks may provide two preemption possibilities among themselves with the priority inheritance mechanism: a 2 priority invocation preempts a 1 priority invocation, and later the 2 priority invocation is preempted by a 3 priority invocation. A task of priority 4 is assigned to the c-entry. It is guaranteed that any invocation on the d-entry will preempt any running thread on the n-entry, while any invocation on the c-entry will preempt any running thread on either the n-entry or the d-entry.

4.4.3 Dealing with priority inversion

Priority inversion is the phenomenon where a higher priority activity (task) is forced to wait for the execution of a lower priority activity (task). The duration of such priority inversion must be bounded to satisfy the deadline constraint of the higher priority activity. The technique for bounding such priority inversion is one of the main design challenge of a static priority based programming model.

Figure 4.7: Priority Inversion in Real-Time ANSA Objects

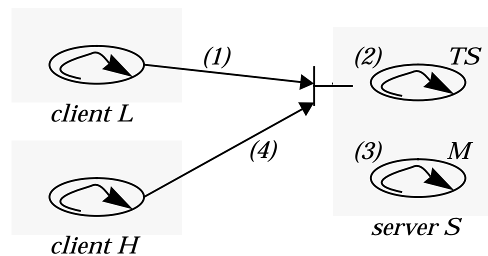


Figure 4.7 shows an example of priority inversion in real-time ANSA objects. Suppose there is a server object *S* with an interface *I* and client objects *L* and *H*. *L* is a low priority client --- it runs a low priority task which sends low priority invocations to *S*. *H* is a high priority client --- it runs a high priority task which sends high priority invocations to *S*. *S* has a task *TS* for serving invocations on *I*. Moreover, *S* has another middle priority task *M* running independently.

Priority inversion happens if the following sequence of actions appears:

1. *L* sends a low priority invocation to *S;*
2. *TS* begins processing *L*'s request with the low priority;
3. *M* starts running, preempting *TS*;
4. *H* sends a high priority invocation to *S*, and has to wait until *M* finishes.

There are three possible solutions to the priority inversion problem. If the operations provided by the interface allow concurrent access, a group of tasks may be allocated for the interface. By using (basic) priority inheritance, an alternative task inherits *H*'s priority so that it can preempt *M*.

If the operations provided by the interface do not allow concurrent access, such as in a monitor or critical-section interface, transitive priority inheritance can be used. In the example, after (4), *TS* may inherit the high priority, so that it can preempt *M*. *H* waits only a minimum period of time till *TS* finishes one operation.

Transitive priority inheritance is difficult to implement[1]. An alternative approach is ***priority ceiling***. Each entry may be associated with a fixed priority ceiling value, which specifies an upper-bound priority that applies to all the invocations on the interfaces bound to the entry. While a task is executing a thread on the entry, its priority is raised to the ceiling priority. If an invocation has a higher priority than the ceiling priority, it is rejected. Priority ceiling is easy to implement, but may introduce some unnecessary blocks. For example, in step (2) *TS* will be executed with the high priority; it unnecessarily blocks *M* if *H* does not call *S* during *TS*'s execution. In this sense, priority ceiling is a pessimistic technique for bounding priority inversion. Fortunately, operations implemented by a critical-section interface are often short. Therefore priority ceiling is still an attractive technique, even though it is pessimistic.

4.5 Deadline scheduling

A deadline value associated with an invocation specifies a bound on the completion time of the requested operation. By assigning deadline values with invocations, the problem of satisfying timing constraints becomes one of scheduling processes to meet deadlines, or *deadline scheduling.*

A simple deadline scheduling policy is to treat deadlines as priorities in thread queuing. An earlier deadline has higher priority than a late one. Let's call it *deadline based* thread scheduling. It is not assumed that the task scheduler (i.e. operating system scheduler) understands deadlines. The resultant behaviour is a *non-preemptive earliest deadline first* execution of invocations.

Preemption is possible if the task scheduler provides an earliest deadline first preemptive scheduling service and serving tasks are allowed to inherit thread deadlines. Under these conditions, deadlines can be handled exactly as priorities as defined in the last section. It should be pointed out that deadline based scheduling provides only a deterministic scheduling approach. It provides no guarantees for satisfying deadlines.

As deadlines impose timing constraints directly to invocations, a late result produced by a server task has little or no meaning. This timeliness requirement suggests that the RPC protocol --- the Remote EXecution protocol in the ANSA system --- should take deadlines into account. Timed RPC is discussed in Section 4.11.

One way to improve the robustness of a timed RPC protocol for real-time applications is to ask the scheduler to provide an early acknowledgement to the client. The server thread scheduler checks its local schedule information to decide if it is possible to execute a request within its deadline. The decision must take into consideration the invocation communication delay, the invocation demand of the processor, and the server load. If the

1. To implement transitive priority inheritance, the infrastructure needs to maintain the dynamic task/thread relations and requires special operating system supports for transitive priority inheritance operations

acknowledgement is positive and received before a timeout value of the client, the client will wait for the final result. Otherwise, the client may consider the invocation unsuccessful and start to take necessary alternative actions. Although using the early acknowledgement does not actually increase the probability of invocation success, it will give the client more time to recover from the timing error.

4.6 Other scheduling paradigms

Priority and deadline scheduling can be combined to provide alternative scheduling models. One combination is *priority first, and then deadline based,* in which deadlines are only used to break the tie when two thread have the same priority. This could apply in multi-media information systems, for example, priorities being used to identify information importance and deadlines being used to identify the relative order of frames in media streams (media interleaving).

Another combination is *deadline first and then priority based* [Miller 90], in which deadlines are used as first scheduling criteria, but in the case of unsatisfied deadline, priorities are used instead for scheduling. This allows function priorities to be attached while at the same time, achieving the high throughput property of a deadline based scheduling algorithm.

4.7 Application controlled rendezvous

In addition to allocating system task(s) on an entry for serving requests, the real-time ANSA also allows tasks (when serving threads) to rendezvous with entries at run-time. The interface is as follow:

```
Accept(entry_set, timeout)
```

The effect is that the task waits for at most timeout to serve one request on any entry in the entry_set.

The application controlled rendezvous model has the following characteristics:

- clients do not see any difference from the standard object invocation semantics.
- the Accept statement ensures that only one request is executed in the accepting task (the server task). Other requests are queued, until the server task executes a subsequent Accept statement.
- the application task may perform its own synchronisation. This may help improve resource usage by synchronizing before a request starts executing, and not after.
- the application task may initiate object invocations like other client tasks.
- the application task may perform its resource management when not responding to external requests. Therefore, it is possible to have interface specific tasks with pre-allocated resources and optimized synchronisation management.

4.8 Summary of the real-time programming model

This section has described the real-time programming model of the real-time ANSA. Its scheduling flexibility has been demonstrated by its two-level scheduling multiplexing. Policy/mechanism separation is used to address the diversity of real-time programming. An integrated priority management scheme is introduced for preemption control. The application controlled rendezvous is shown to be a powerful mechanism for resource management and synchronisation.

4.9 Real-time communications

Real-time applications present more complicated functional requirements to the underlying communication systems. We have identified three mechanisms for providing such functions within an RPC communication infrastructure. The facilities provided include:

- a parallel protocol stack for the preallocation of communication resources and the removal of layered multiplexing.
- a timed RPC protocol for the association of deadlines with invocations.
- a decomposable RPC protocol for the tradeoffs between functionality and performance.

4.10 Towards a parallel protocol stack

The main advantage of the ANSA communication system design is its efficient resource utilization. The price, however, is the heavy use of multiplexing. This raises the following problem for real-time applications:

- there is no association between the (interface level) channels and Message Passing Service (MPS) channels, and the two level modules have no interactions when channels are created and destroyed; the two are independent of one another. The end result is that even through it is possible to distinguish interfaces providing real-time services from those providing non-real-time services at a high level, communication to/from these interfaces may share the same MPS communication channel (such as a connection or virtual circuit), which inevitably introduces non-determinism.

Detailed discussions of the adverse effect, known as *performance cross-talk*, of multiplexing several channels onto a single channel can be found in [Tennenhouse 89].

The problem can be overcome as follows:

- redesign MPS interface as connection-based, it maintains simple states of its channels. If the operating system can provide a connection-based service, a MPS connection is directly mapped on to an operating system IPC socket.
- extend the EXecution Protocol to use this connection-based interface,
- extend the programming interface so that applications have control over these connections. For example, if the low-level IPC connections allow QoS associations, the ANSA interface-level channels should have correspondent abstractions for the QoS attachment.

Figure 4.8: Parallel Protocol Stack

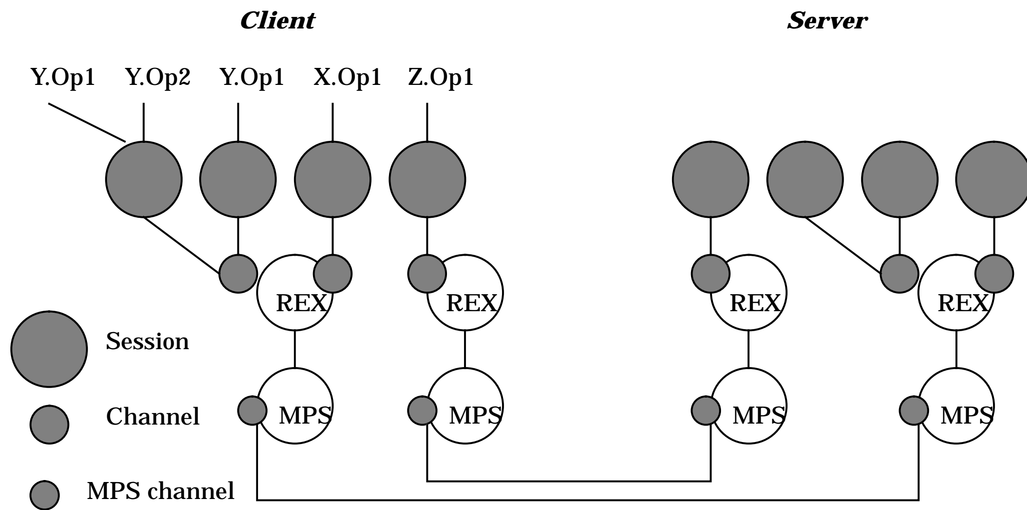
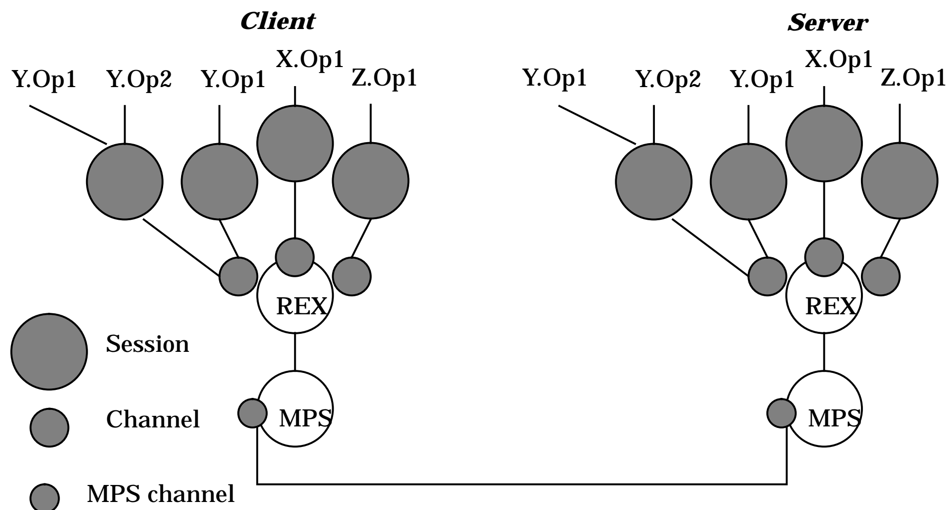


Figure 4.9: Multiplexing in the Testbench



The result is a parallel communication protocol stack, as illustrated by Figure 4.8, which is in contrast with the original ANSA multiplexing structure for a server/client interaction as illustrated in Figure 4.9.

4.11 Towards a timed RPC protocol

Arbitrary delays associated with synchronous invocation cannot be tolerated due to the time-dependent nature of real-time applications. A dependable protocol is desirable to provide a timeliness service for real-time RPC, or timed RPC (TRPC).

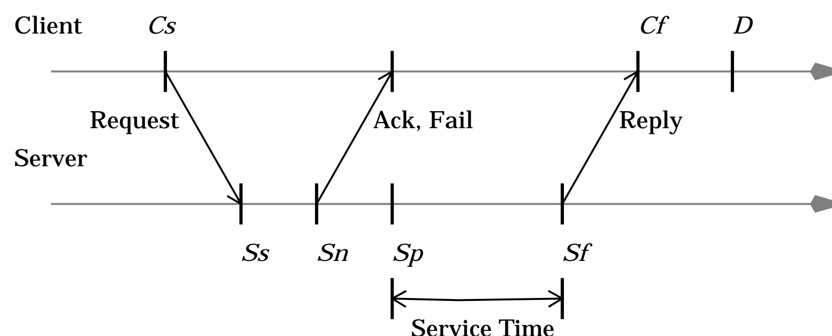
Invocations in real-time ANSA can attach deadline constraints to their communication requests. Such TRPC calls raise the following three issues:

- the management of time in a networked environment. Intuitively, a deadline is an upper bound, which is placed on the time duration for the invocation to occur. Therefore both the server and client must have the same global sense of time --- the deadline. It is thus necessary to assume a global sense of time is provided by the infrastructure.
- the interpretation of deadlines.
- a communication protocol to implement reasonable meanings of deadlines.

To the author's knowledge, there is no clear definition of TRPC yet when examined in the distributed setting. The interpretations applied significantly affect the implementation. The problem will be approached by first making a strictly unsatisfiable definition, and then relaxing the problem to lead to realistic solutions.

The TRPC call can be defined as follows. At time *Cs*, the client sends a request with a deadline *D*, which is the latest time the client is willing to wait for successful invocation. At some time *Ss* the server gets the request; the server checks if the deadline can be met, and if it is unsatisfiable a fail acknowledgement is sent back at time *Sn*. Otherwise, the request is accepted and the request is processed at time *Sp*, and a reply is generated at time *Sf* This is illustrated in Figure 4.10.

Figure 4.10: Timed RPC Communication Sequence



The problem is to design a nontrivial protocol (one which allows the possibility of success) which guarantees the client and server will meet a deadline, and agree on whether or not the request is successful. In other words, a TRPC protocol should enable a client and its server to arrive at a consistent state --- they agree on whether the invocation should be continued, or failed (the invocation is cancelled) and alternative actions should be taken.

4.11.1 Discussion of problem

There are two goals one might try to accomplish with the deadline of a TRPC:

- Goal 1: to establish a bound on the time at which the delay in awaiting a TRPC call expires.
- Goal 2: to establish a bound on the time at which a TRPC call is either scheduled to execute and finish or is unschedulable and cancelled.

The design of a TRPC is complicated by the fact that the end-to-end delay of messages can be arbitrary or even infinite (messages can get lost). It can be shown that the two goals are not mutually compatible. In its simplest form, in which each request takes zero service time, the TRPC problem is equivalent to the *timed synchronous communication* problem [Lee 90]. In the case of message loss, timed synchronous communication is the well known *Two Generals* problem, in which the two generals are trying to agree upon *a common time* of attack before a deadline but can only communicate via unreliable messengers. Such a protocol does not exist.

The design of a TRPC is further complicated by the fact that making the decision of whether a request is schedulable at the server side is often *unattainable* --- a guarantee scheduler often makes many of the impossible assumptions such as that the invocation service time is known, operations are independent etc.

The intention of this work is to develop a protocol for TRPC that works in reasonable environments. Therefore, an upper bound on message delivery and a guarantee scheduler cannot be assumed. Instead, various *relaxations* of the problem are investigated, this yields to a parametrised generic protocol, allowing different combinations of the parameters to represent different relaxed goals.

4.11.2 The protocol

Because using one deadline value to accomplish the two goals in a TRPC may result in incompatible situations, two arguments --- a *timeout* and a *deadline* --- are used instead. Each is aimed at one goal only. The timeout is used to specify the first goal --- how long the client is willing to wait for its result. It affects a client side of the TRPC protocol only. The deadline is used for the second goal --- within which the request should be executed on the server. It affects the server side of the TRPC protocol only.

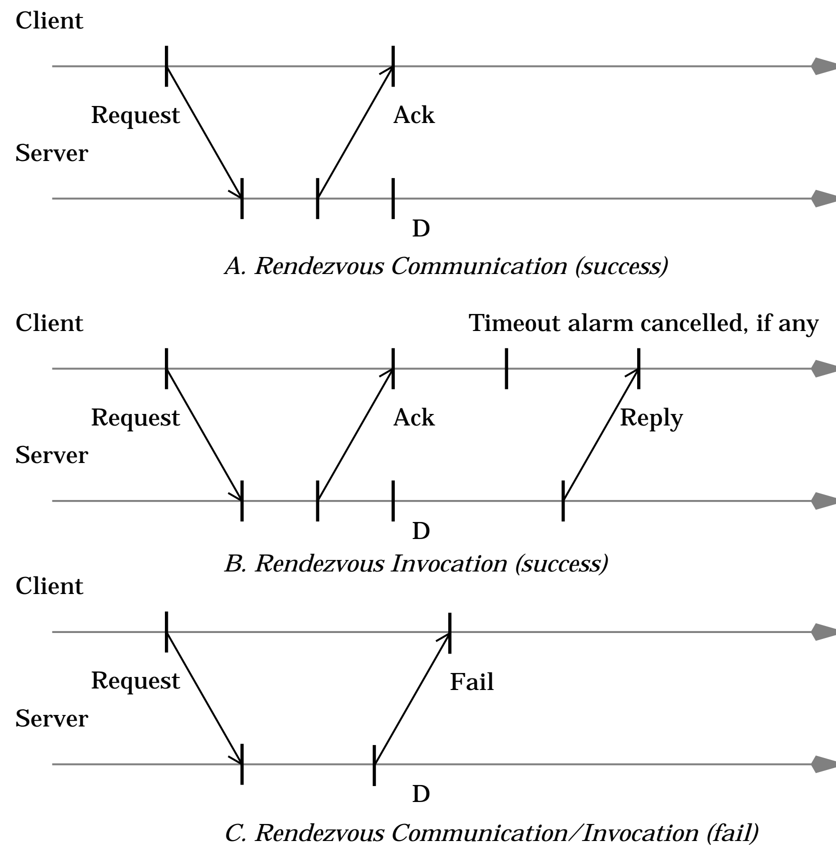
It should be pointed out that using the two separate arguments does not solve the TRPC consistency problem. Rather, the two arguments give the problem a more realistic definition, allowing different relaxations be explored.

The first relaxation is using a *timeout* to enforce the client's *absolute* deadline. The client decides that the request is unsuccessful if it does not get a reply/acknowledgement from the server by the timeout. There is a possibility for *inconsistent decisions* --- the client believes the request is failed, while the server knows the request is successful. Deadlines may or may not be used in this situation. The timeout expiration presents the client an exception situation of *don't know* It is up to the client to take further rescue actions.

The second relaxation is using a deadline to specify a client's objective time value by which the request should be finished. Whether this deadline can be guaranteed or not is purely a matter of server scheduling and message passing delays. In this relaxation, the client waits until a reply/acknowledgement is received from the server. Therefore, the client deadline is not *absolute*. This relaxation allows a client and its server to reach a consistent decision.

The second relaxation can be further extended by relaxing the meaning of a deadline. Instead of bounding the finishing time of a request, a deadline can be used to bound the start time of a request in the server --- to bound the start time by which the request is rendezvoused with a server task. If the

Figure 4.11: Rendezvous Communication/Invocation Interaction



A. Rendezvous Communication (success)

B. Rendezvous Invocation (success)

C. Rendezvous Communication/Invocation (fail)

rendezvous is issued before the deadline, then the request is successful and a success acknowledgement is sent back to the client, otherwise the request is cancelled and a fail acknowledgement is returned. At the client side, there are two possible actions to be taken when it receives a success acknowledgement. One is that the client thinks the request is finished, and control is returned so that it can continue. This is defined by the *RendezvousCommunication* deadline type. Another is that the client cancels its timeout, if any, and waits until a reply is returned later by the server. This is defined by the *RendezvousInvocation* deadline type. The two resulting interaction patterns are illustrated in Figure 4.11.

In summary, an invocation may be associated with an optional timeout, an optional deadline, and an optional deadline type. The collective choice of the three parameters determines the behaviour the TRPC protocol. The result of such a TRPC call can be a *timeout* --- possibly an inconsistent state, a *success* or a *failure*.

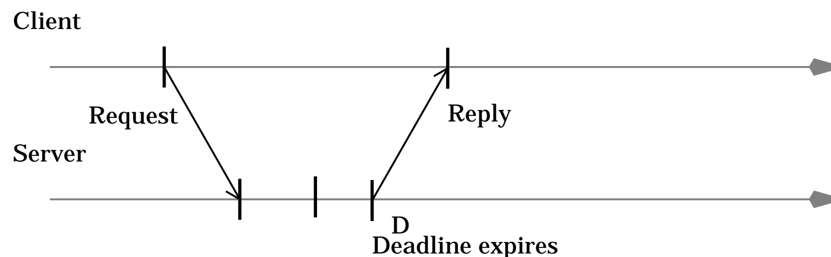
Obviously, it is not necessary to choose the timeout and deadline the same value. A timeout may be smaller than a deadline, to specify that an acknowledge should be returned earlier; it may be greater than a deadline, to allow the request to have a better chance of success.

The default deadline type of an invocation deadline is *ServerDetermined* --- it depends on the scheduling policy used in the server to interpret the deadline, and has no effect on the communication protocol.

4.11.3 Server deadline expiry

There may be two types of deadline expiry at a server side. One type is defined by the TRPC protocol, as illustrated by the rendezvous communications and rendezvous invocations. The required semantics are enforced by the communication protocol.

Figure 4.12: Server Thread Deadline Expire



Another type of deadline expiry may be caused by the tasking components. An active thread serving an invocation may be notified of a deadline expiry signal

--- if the operating system scheduler understands deadlines. If the service routine is designed to accept and handle the signal, a deadline exception may be raised. This deadline exception, however, is different from the one processed by the TRPC protocol. The active thread itself detects the deadline expiry, and may therefore cancel its execution and returns a special value *deadline-exception* to the client. This kind of interaction does not require special TRPC protocol support, as the deadline-exception is just a special value of reply. This is illustrated in Figure 4.12.

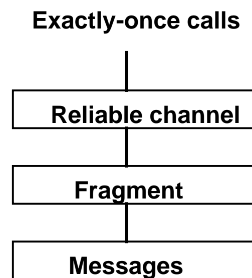
4.12 Towards a decomposable RPC protocol

An RPC protocol is normally required to provide *exactly-once* call semantics. The exactly-once protocol is used to ensure that calls are executed once and only once in the absence of crashes or prolonged communication failure, in order to preserve the local procedure call semantics for the client. Probes, acknowledgements and retransmissions are used for error-detection and error-recovery in such protocols. Error detection and error recovery both introduce significant performance overheads.

For real-time applications probes and retransmissions are not normally suitable techniques for error control, and exactly-once semantics are sometimes not a desired feature because retransmitted data or control information could be a *late* message, and have little meaning in real-time sense. Alternative *light-weight* protocols with *at-most-once* semantics are desirable instead. Real-time ANSA is assumed to operate in a system which may consist of a mixture of real-time and non-real-time applications, therefore both the exactly-once and the at-most-once semantics are desirable. It is possible to implement the two protocols separately, but because the two protocols share many similarities, alternative integrated design is more interesting for the purposes of better structure, flexibility and efficient coding. This raises the desire to design a decomposable RPC protocol.

The ANSA REX service provides exactly-once semantics of RPC calls. REX can be decomposed into three layers as illustrated in Figure 4.13. The three layers are layered functions sharing the same protocol data structure --- sessions, and to provide just one protocol service.

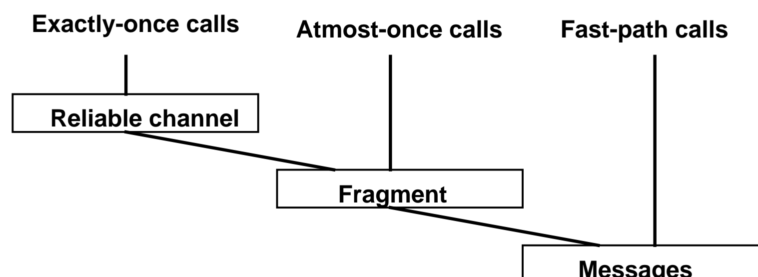
Figure 4.13: REX Functions Layers



The *message* layer uses the underlying MPS service to provide a simple unreliable, unfragmented message passing service. This layer sends/receives messages not larger than a single MPS packet size. The *fragmentation* layer provides unreliable, but persistent (recovery from dropped fragment) transmission of large messages.The *reliable-channel* layer provides reliable transmission of large messages (recovery from lost and duplicated messages).

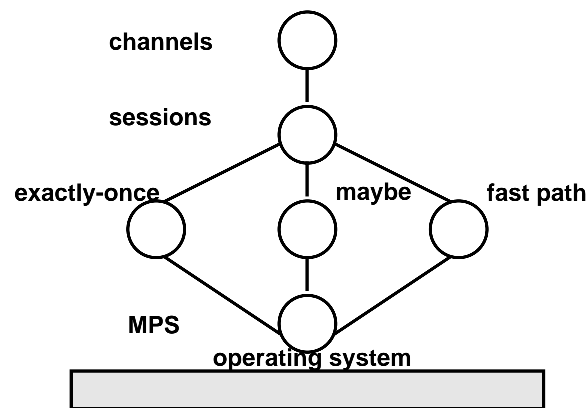
The REX three layers can be reassembled to provide a multiple service interface. The transportation protocol looks like Figure 4.14. The communication system is illustrated in Figure 4.15. In addition to the exactly-once service, two other services, the at-most-once service and fast-path services can be provided. The multi-transportation service protocol is still one execution protocol in the ANSA sense. But it provides additional call semantics.

Figure 4.14: A Decomposable Protocol



The fast-path service is designed to execute operations within the *critical data path* of the RPC system. It is assumed that the request is independent of other invocations (no resource sharing with others and no nested invocations), and both the request and result fit in one single MPS packet. Under these

Figure 4.15: The Communication System



conditions, the server can execute the request within a communication task (thread), allowing significant performance improvement by saving the cost of thread dispatches and task context switches.

The decomposable RPC protocol is our first step in the design of an integrated protocol for supporting various transportation QoS at RPC level.

4.13 Summary

The real-time programming model provides a framework to facilitate the enforcement of stringent timing constraints found in distributed real-time applications. The model incorporates tasks and communication channels (the two most important resources in real-time distributed computing) as its basic programming components. It synthesises aspects of resource requirements, resource allocation and resource scheduling into an object-based programming paradigm. Predictability, user control and mission criticality are the main characteristics of the model.

The real-time communication system examines various approaches for updating current ANSA RPC communication system for real-time applications.

The parallel protocol stack provides the basis for using real-time network technologies.

The development of the timed RPC protocol contributes a capability to associate timing constraints with object invocations. Using the timed RPC protocol allows a programmer to express and enforce reasonable timing requirements (representing different tradeoffs between consistency and strictness) on a per-call basis.

The decomposable protocol provides the necessary insight for the future development of RPC based real-time transportation protocols.

5 A model of real-time QoS

This chapter explains how the real-time programming model fits our general binding/QoS framework.

5.1 QoS domains

The real-time programming model can be summarised by three QoS domains:

- session QoS domain: it allows the association of priority, deadline etc. with object invocations. The binding operation of this domain is done implicitly while an invocation is started. There is no need for the explicit definition of the BIND operation in the domain.
- tasking QoS domain: it allows the specification of task resource requirement and management.
- communication QoS domain: it allows the specification of communication resource requirement and management.

The three domains are shown in Figure 5.1, 5.2 and 5.3.

Figure 5.1: Session QoS domain

```
Session : QoSDOMAIN =
BEGIN
 DeadlineTye: KEYWORD = {rend_communication, rend_invocation};
 Deadline: KEYWORD = Timer;
 Timeout: KEYWORD = Timer;
 Priority: KEYWORD = INTEGER;
END.
```

5.2 Binders

The three QoS domains are supported by three resource managers:

- tasking manager: manager task/entry resources for the task QoS domain.
- communication manager: managing transportation resources for the communication QoS domain
- session manager: managing session resources for the invocation QoS domain. The session manager has some private interfaces with both tasking manager and communication manager for managing session resources.

Figure 5.2: Communication QoS

```
MPS: QoSDOMAIN =
BEGIN
    Protocol: KEYWORD = {TCP, UDP, IPC, MSNL};
    Address: KEYWORD = String;
    Rate: KEYWORD = INTEGER;
    Bandwidth: KEYWORD = INTEGER;
END.

REX: QoSDOMAIN =
BEGIN
    Protocol: KEYWORD = {exact-once, atmost-once, timed};
    Rate: KEWORD = INTEGER;
    Retries: KEYWORD = INTEGER;
END.

TRANSPORTATION: QoSDOMAIN =
NEEDS MPS, REX;
BEGIN
    Type: KEYWORD = {socket, plug};
    Channel: KEYWORD = {shared, not-share};
    Mps: KEYWORD = MPS.K;
    Rex: KEYWORD = REX.K;
    T_binding: INTERFACE =
    BEGIN
      Result: TYPE = {ok, fail, retry};
      UnBind: OPERATION [] RETURNS [Result];
    END.
    BIND: OPERATION [if_ref: InterfaceRef] QOS
           RETURNS [T_binding];
END.
```

Figure 5.4 shows the BIND operations for each of these domains.

Figure 5.3: TASK QoS domain

```
ENTRY: QoSDOMAIN =
BEGIN
     Id: KEYWORD = INTEGER;
     Type: KEYWORD = {shared, not-share};
     EnqueuePolicy: KEYWORD = {FCFS,PB,DB,PDB,DPB};
     RendezvousPolicy: KEYWORD = {Null,PI,DI,CEILING,PDI};
     Concurrency: KEYWORD = INTEGER;
END.

TASK: QoSDOMAIN =
BEGIN
     Type: KEYWORD = {rt_periodic, rt_sporidic, non-rt};
     Tasks: KEYWORD = INTEGER;
     Period: KEYWORD = INTEGER;
     Priority: KEYWORD = INTEGER;
     Stack: KEYWORD = INTEGER;
END.

TASKING: QoSDOMAIN =
NEEDS ENTRY, TASK;
BEGIN
     Entry: KEYWORD = ENTRY.K;
     Task: KEYWORD = TASK.K;

     T_binding: INTERFACE =
     BEGIN
          Result: TYPE = {ok, fail, retry};
          UnBind: OPERATION [] RETURNS [result];
          ReBind: OPERATION [] QOS RETURNS [Result];
     END.

     BIND: OPERATION [if_ref: InterfaceRef] QOS
           RETURNS [T_binding];

END.
```

Figure 5.4: BIND Operations

```
-- Communication BIND

{socket} <- transportation_binder$BIND (svr_if)
                           ((Channel == not_share) and
                           (Mps.Protocol == TCP) and
                           (Rex.Rate == 1000))

{plug} <- transportation_binder$BIND (clt_if)
                           (Rex.Protocol == atmost-once)

-- Tasking BIND

{entry} <- entry_binder$BIND (svr_if)
                           ((Entry.type == not_share) and
                           (EnqueuePolicy == FCFS) and
                           (Task.Tasks == 2))
-- Session BIND

{result} <- clt_if$operation(arguments) (Priority == 3)
```

6 Related work

6.1 IMAC

The Integrated Multimedia Application Communication (IMAC) architecture [Nicolaou 91] provides a framework which facilitates the construction of multimedia applications.

IMAC is based on the ANSA architecture and has been implemented as an extension to the ANSA Testbench. IMAC provides a mechanism for the specification of communication oriented QoS on a per-invocation basis. Interface operations may specify a set of QoS options with which they are prepared to be invoked. The QoS options are expressed as constraints on the underlying communication system. A method of mapping from application level QoS to communication level QoS is provided.

IMAC QoS extensions are based on the ANSA implicit binding model, i.e. QoS constraints are mainly associated with sessions. It does not address QoS domain issues and explicit binding issues.

6.2 Lancaster work

The Basic Service Platform (BSP) [Coulson 93] developed at Lancaster University provides an ANSA based platform for the design and implementation of distributed multimedia applications. Up to now, the platform has been mainly concentrated on engineering extensions of ANSAware for multimedia transportation and synchronization. Special rate based transportation services, synchronization services, and device management services are developed as a set of special ANSA interfaces. BSP work has resulted in numerous insight on engineering issues.

BSP adopts a typical API approach for QoS: QoS specification and management are provided by a set of standard interfaces. Explicit binding is introduced in an ad hoc manner for some special interfaces that provide bounded invocations. BSP does not address the full range of explicit binding and QoS issues.

6.3 OSI QoS framework

The ISO/IEC OSI QoS framework [ISO/OSI/QoS 93] concentrates primarily on QoS for OSI communications. The framework defines terminology and concepts for QoS and provides a model which identifies objects of interest to QoS. The QoS associated with objects and their interactions is described through the definition of a set of QoS characteristics. A model is defined which provides unifying concepts for the management of QoS.

The framework is intended for existing or planned OSI standards and may be applicable to other fields.

The OSI QoS framework is complementary to our work: it defines a detailed example of OSI QoS domain that may fit into our general QoS/binding model. The OSI QoS framework does not address any binding issues.

Note: Editorial: defining OSI QoS framework using our QoS language may be a future project.

6.4 CNET work

CNET [Hazard et al. 93] is working on a global architecture for handling problems raised by QoS handling and real time. CNET work adopts a strict approach for separation of computational and engineering concerns: the programmer is provided with means to express QoS in an abstract way, and the engineering infrastructure translates the abstract QoS expressions into precise mechanisms. The main computational elements introduced are an extension to ANSA interface typing and subtyping rules for the specification of QoS constraints and an extension to the notion of binding for the dynamic establishment of QoS constraints.

CNET work proposes a formal specification language QL based on a real time logic model as its QoS language. It also proposes to use synchronous languages to express the control part of real time applications, which has inspired another relevant work at APM.

CNET work has provided many useful insight to our work in the pursue of a generic binding framework and language based approach to QoS. CNET work, in comparison to our work, lacks a systematic treatment of binding. The CNET QoS language QL is not efficient in expressing resource constraints, and there is no mention how QoS can be managed dynamically.

Note: Editorial: future work is proposed to collaborate with CNET team to merge the two binding/QoS architecture. It would also be an interesting project to extending our QoS constraint language to cover the QL components.

7 Conclusion

This document presents a framework for handling the explicit binding and QoS issues raised in the ANSA/ODP architecture. The feasibility of the framework is demonstrated by its capability for the accommodation of a real-time programming model.

The major results of this document are:

- the development of a framework and a basic set of requirements for explicit binding operations.
- the development of a set of requirements for QoS specifications, and a simple QoS language that demonstrates how the binding framework and QoS specification are organized.
- the definition of a real-time programming model and solutions for some general real-time problems.
- the integration of the real-time programming model with the binding/QoS framework as an example of real-time QoS.

The main attributes of the QoS model are separation and integration, which provides a realistic approach for the description of resource management activities and allows the evolution of technologies.

The real-time programming model provides a framework to facilitate the enforcement of stringent timing constraints found in distributed real-time applications. The model incorporates tasks and communication channels (the two most important resources in real-time distributed computing) as its basic programming components. It synthesises aspects of resource requirements, resource allocation and resource scheduling into an object-based programming paradigm.

It is believed that the binding/QoS framework is a generic one, i.e. it should be applicable to other application areas.

7.1 Acknowledgment

We wish to thank Francis Wai, xxx for their review and feedback on the document.

References

[Ada9X 93]

Ada 9X Documents, Ada 9X Project Report, Real-Time Systems Annex, Office of the Under Secretary of Defense for Acquisition, US Department od Defense, February 1993.

[Allchin 83]

J E Allchin and M S Mc Kendry, Synchronization and Recovery of Actons, In Proc. of Second Symp. on Priniciples of Distributed Computing, August 1983.

[Attoui 91]

A Attoui and M Schneider, An Object Oriented Model for Parallel and Reactive Systems, In IEEE Real-Time Systems Symposium, December 1991.

[Black 86]

A P Black et al, Distributed and Abstract Types in Emerald, IEEE Transactions on Software Engineering, 12(12), December 1986.

[Blair 92]

G S Blair and R Lea, The Impact of Distribution on the Object-Oriented Approach to Software Development, IEE/BCS Software Engineering Journal, 7(2), March 1992.

[Coulson 93]

G Coulson, Multimedia Application support in Open Distributed Systems, Ph.D. Thesis, Lancaster University Computing Department, April, 1993.

[Hazard et al. 93]

L Hazard, F Horn, and J B Stefani, Toward the Integration of Real Time and QoS handling in ANSA Architecture, CNET Report CNET.RC.ARCADE.01, June 1993.

[Herbert 91]

A Herbert, Engineering Model: Conceptual Framework, RC.282, APM Ltd., Cambridge U.K., 1991

[Herbert 93]

A Herbert, Distributing Objects, TR 18, APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993.

[Herbert et al. 93]

A Herbert et al., ORB Interoperability TR 43, APM Ltd, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993.

[ISO/ODP 93]

ISO/IEC JTC1/SC21/WG7, Reference Model for Open Distributed Processing: Part 1, Working Document 21/7/N755, August, 1993.

[ISO/OSI/QoS 93]

ISO/IEC JTC1/SC21/WG1, Quality of Service Framework, Working Draft #2, July, 1993.

[Lehockzy 89]

J P Lehockzy, L Sha, and Y Ding, The Rate Monotonic Scheduling algorithm -- Exact Charicterization and Average-case Behaviour, In Proc. of Tenth IEEE real-Time Systems Symposium, December, 1989.

[Lee 90]

I Lee and S B Davidson, A Performance Analysis of Timed Synchronous Communication Primitives, IEEE Transactions on Computers, 39(9):1117--1131, September 1990.

[Li and Otway 93]

G Li and D Otway, An Open Architecture for Real-time: Engineering Aspects. RC 1072, APM Ltd., Cambridge U.K., October 1993 .

[Li 93a]

G Li , Supporting Distributed Realtime Computing, PhD thesis, University of Cambridge Computer Laboratory, Technical Report 322, August 1993.

[Miller 90]

F W Miller, Predicative Deadline Multi-Processing, Operating Systems Review, 24(4):52-62, October, 1990.

[Moitra 86]

A Moitra, Scheduling of Hard Real-Time Systems, In Foundations of Software Technology and Theoretical Computer Science, LNCS 241, pp 352-381, Springer-Verlay, 1986.

[Nicolaou 91]

C Nicolaou, A Distributed Architecture for Multimedia Communication Systems, PhD thesis, University of Cambridge Computer Laboratory, Technical Report 220, May 1991.

[POSIX]

POSIX, IEEE POSIX Std 10003.4 (Draft 13), September 1992.

[Rees 93]

O Rees, The ANSA Computational Model, TR 01, APM Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993.

[Tennenhouse 89]

D L Tennenhouse, Layered Multiplexing Considered Harmful, In Protocols for High Speed Networks, IFIP WG.1/6.4 Workshop, May 1989.