Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Some Engineering Aspects of Real-Time

Guangxing Li

Abstract

Advances in digital communication networks and in personal workstations are beginning to allow the simultaneous processing of real-time data, voice, and video. There is a great demand to provide real-time functionality as standard system services, not as features added as afterthoughts. At the same time, the size of real time systems is increasing: one-million-line real-time software systems are becoming common today. Such systems are very large and distributed by nature. There is an increasing need to adopt an open architectureal approach so that real-time system engineering can be addressed not only with real-time constraints, but also with other software practise constraints such as scale, evolution, distribution etc.

Distributed real-time processing places unique requirements on systems and their designs (e.g. predictability, programmer control, timeliness, mission orientation and performance). Such features do not exist in today's computing environments.

This document proposes an integrated architecture for distributed real-time systems. It describes some engineering designs of a distributed environment for real-time applications.

APM.1222.02 **Approved** 11th April 1995

Technical Report

Distribution:
Supersedes:
Superseded by:

Some Engineering Aspects of Real-Time

Some Engineering Aspects of Real-Time

Guangxing Li

APM.1222.02

11th April 1995

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	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Overview

1.1 Motivation

Advances in digital communication networks and in personal workstations are beginning to allow the simultaneous processing of real-time data, voice, and video. There is a great demand to provide real-time functionality as standard system services, not as features added as afterthoughts. At the same time, the size of real time systems is increasing: one-million-line real-time software systems are becoming common today [Gopinath93]. Such systems are very large and distributed by nature. There is an increasing need to adopt an open architectural approach so that real-time system engineering can be addressed not only with real-time constraints, but also with other software practise constraints such as scale, evolution, distribution etc.

Distributed real-time processing places unique requirements on systems and their designs (e.g. predictability, programmer control, timeliness, mission orientation and performance). Such features do not exist in today's computing environments.

This document proposes an integrated architecture for distributed real-time systems. It describes some engineering aspects of a distributed environment for real-time applications. The perspective and scope of this research is the entire system environment, rather than being focused on the more narrow subsystem or algorithms. The stress is on the engineering design that would stand on both the *current* and future technologies. This document *is not* a full coverage of all engineering aspects of a distributed real-time environment.

The aim of this document is to set up a start point for the development of a detailed design and implementation of an ANSA based real-time platform [Li94].

1.2 Scope of the document

It is the stringent timeliness and performance natures that are the primary source of problems posted by real-time applications. The services provided by existing distributed system environments predate the present concerns of real-time applications and provide insufficient and inappropriate services for supporting real-time applications. For example, current standards for distributed processing, such as the OSF DCE, OMG CORBA and ISO RM-ODP make no mention of real-time issues. This report shows how it is possible to extend a distributed system environment to support real-time applications and hence help avoid these problems.

The principle issues covered by this document are:

- real-time system environment characteristics, i.e. the problems to be addressed

- the uniform treatment of real-time computing and non-real-time computing i.e. system integration
- technology bases i.e. the relevant technologies for the system design and implementation
- environment i.e. the services and scope of the proposed system
- target i.e. the possible application areas of the architecture
- distributing real-time objects
- real-time programming models
- real-time communications.

The design is presented as an extension of ANSA, because it has generic engineering model. The architectural issues of this work are also applicable to other distributed system environments such as OSF DCE and OMG CORBA, where computational and engineering issues are blurred, and where the internal structure is monolithic.

1.3 Benefits

The benefits of the work are several:

- permits the application of open system architecture to real-time systems.
- identifies how and where real-time applications may constraint open system architecture.
- explains how real-time technologies are integrated with open systems.

1.4 Audience

The reader is assumed to be familiar with the ANSA Computational Model and Engineering Model. A brief overview of the two models are included in chapter 3 to improve the readability of this document. It is also assumed the reader is familiar with the basic concepts of real-time and distributed systems, which can be found in [Wai93] or [Li93].

1.5 Outline of the document

The document is structured as follows:

- chapter 2 discusses the architectural issues of a real-time open system.
- chapter 3 briefs the ANSA computational model and engineering model.
- chapter 4 outlines the areas of engineering high level design.
- chapter 5 presents the distributed real-time programming model.
- chapter 6 presents some of the real-time communication designs.
- chapter 7 gives the summary.

2 Towards an Open Architecture for Real-Time

2.1 Problems

Consider a distributed real-time computing environment, in which autonomous machines communicate via various shared communication media. Processing requests can originate at any node in this environment. The actual processing of the requests makes use of the resources within this environment. Such distributed real-time processing requests place a set of unique requirements including *predictability*, *programmer control*, *timeliness*, *mission orientation*, and *performance*. These features do not exist in today's computing environments, and must be addressed by future systems research.

2.1.1 Predictability

Predictability is the tendency of a system to perform a set of operations in a well-defined, or *determined* fashion, so that each of these operations' timing requirements are satisfied. A fully predictable system can perform operations with guaranteed upper bounds, independent of surrounding conditions. Conversely, a fully non-predictable system is one in which operation times have no guaranteed upper bound. Predictability applies to every level of the components of a real-time distributed system environment. Such an environment must provide a certain degree of predictability, even though it is not always possible to be fully predictable, to support any useful real-time performance guarantee.

2.1.2 Programmer control

Programmer control means an application programmer has ultimate control of the behaviour of a system. This feature comes from the fact that many real-time applications are embedded systems (which are often static systems, and therefore it is possible to control the systems' behaviour) and that real-time applications have immense behaviour diversity (therefore it is impossible to use one fixed system behaviour for many real-time applications). The simplest method of programmer control on system behaviour is probably the choice of priorities for real-time tasks. By allowing a user to indicate the relative priorities of tasks, the programmer can affect throughput and/or responsiveness goals for the system on a much finer granularity than by a *best-effort* approach. A programmer may also be allowed to select the scheduling policy, pre-allocation of system and application resources to critical services and so on.

2.1.3 Timeliness

Real-time applications are different from the no-real-time paradigm of computation in that they impose strict requirements on the timing behaviour of the system. The correctness of a real-time system depends not only on the functional behaviour of the system, but also depends on the temporal behaviour as well. A real-time system environment must provide mechanisms which take these time related issues into account and must help application programs to meet these time constraints. A simple example is to allow an application to associate deadlines with real-time activities, and the system employs a deadline based scheduling policy to help the deadlines be met or to identify and cancel obsolete operations. Other more complicated functions include the description and enforcement of temporal relations among related computational activities.

2.1.4 Mission orientation

Mission orientation means that an entire distributed computer system is dedicated towards accomplishing a specific purpose through the cooperative execution of one or more application programs distributed across its nodes. In the real-time sense, mission orientation also means ***mission critical*** --- the degree of mission success is strongly correlated with the extent to which the overall system can achieve the maximum dependability regarding real-time constraints. In its simplest form, mission orientation requires that a priority or deadline associated with a mission has global meaning when it spans over the network. More generally, global importance and urgency characteristics are propagated through the system, for use in resolving contention over system resources according to application defined policies.

2.1.5 Performance

Many real-time applications have stringent raw performance requirements. The optimized integration of application software and its supporting environment is desirable. This is in contrast with the popular layered design for non-real-time applications. Also, real-time applications often require trading off modularity, flexibility and functionality to maximize performance.

2.2 An integrated system architecture

The objective of this research is the provision of an open real-time distributed system environment architecture. An important issue is that such an open system environment cannot be designed by considering only component design issues. An integrated system design philosophy is required. This section discusses the principle approach --- ***system integration***. The importance and benefits of the approach are also briefly highlighted.

The system integration approach provides the ability to treat all forms of real-time objects or data as *first class citizens* in a system environment. That is, operations and mechanisms provided for existing non-real-time components can be applied to, and used by, real-time objects. The provision of a uniform system environment will increase productivity, especially for the creation of applications which offer combinations of distributed and real-time functionality: e.g. multimedia conference, distributed control. Increased integration allows existing distributed system environment mechanisms to be applied to real-time components (such as trading, security, monitoring, replication, location, migration and federation). The aim is also to allow evolution of the architecture from the development of individual control systems, to groups of control systems and then to the *enterprise-wide* command and control systems.

Two technology trends exhibit the importance of system integration:

- ***General purpose distributed computing environments are evolving towards real-time systems***. For example, the advances in digital communication networks and in personal computer workstations are beginning to allow the generation, communication and presentation of real-time voice and video medium simultaneously. Many non-real-time systems have been disembowelled to extend their use to real time [Leung90]. Many UNIX systems, for example, are used for real-time control because of their rich programming tools, despite their unsuitability for such applications. ***There is a great demand to provide real-time functionality as normal system services, rather than as later added on features***
- ***Real-time applications are evolving towards large distributed systems.*** One-million-line real-time software systems are become common today [Gopinath93]. Such systems are large by any standard and are distributed by nature. Therefore, in addition to the problems associated with real-time operation, such applications are subject to all of the problems of any large software system, such as maintainability and distribution. Furthermore, in many real-time applications, tight real-time constraints may apply to only part of the whole system. For example, it is estimated that only 10 to 30 percent of a typical vehicle control software system is directly related to actual real-time control of the vehicle. ***There is an increasing need to adopt an open and architectural approach so that real-time software engineering can be addressed not only with real-time constraints, but also with other software practise constraints such as evolution, scale, distribution etc.***

2.3 Technologies

This section is structured as follows:

- a description of the fundamental contributory technologies
- a review of functions in an open distributed system environment
- a brief description of the current state of art of the distributed real-time system environment research and engineering, and the additional functions required in such an open, real-time, distributed architecture.

2.3.1 Contributory technologies

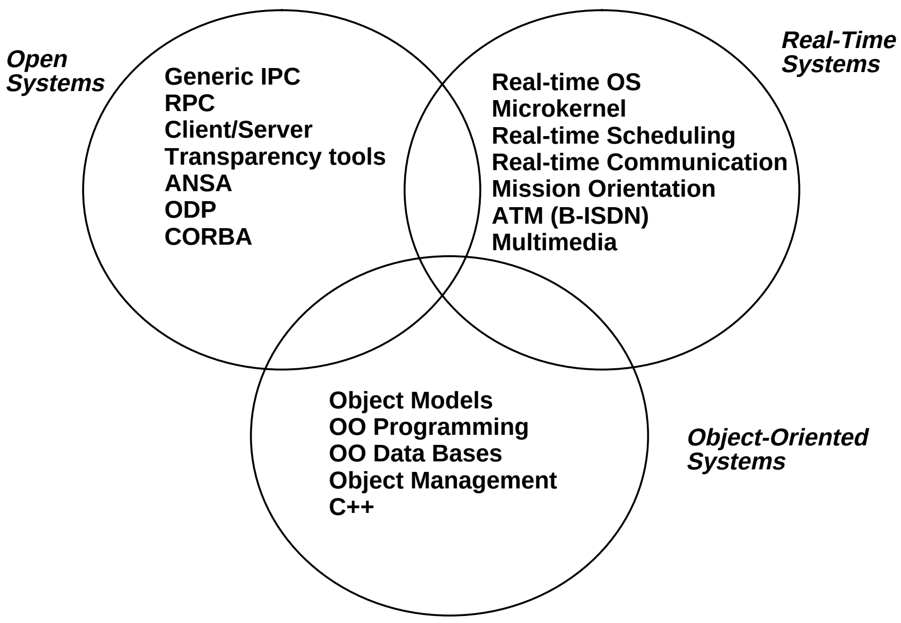
The fundamental contributory technologies are illustrated in Figure 2.1. It represents the integration of real-time systems, open systems and object oriented systems.

The real-time system technology provides the functionality of resource management for guaranteeing the stringent time-constrained computing activities.

The open system technology provides the functionality for distribution, evolution, heterogeneity, federation and scale.

The object oriented technology provides the functionalists for software reuse and maintenance.

Figure 2.1: Contributory Technologies



2.3.2 Distributed system environments

A distributed system environment is a run-time system that provides a set of abstractions and tools to support the writing of programs in a distributed environment. The effect of using a distributed system environment is that applications are automatically supported by a run-time environment which incorporates a set of ***distribution transparency*** mechanisms. These shield application designers and users from the technological complexities involved in distributed application programs. Remote Procedure Call (RPC) and client-server interactions are widely accepted as distributed system environment technical apparatus.

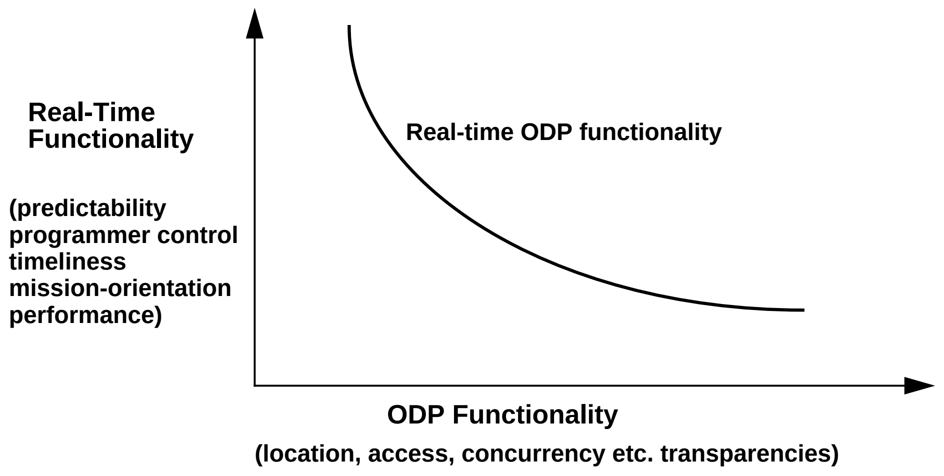
It is now recognised [Herbert93] that distribution transparency can be broken down into a number of individual transparency issues:

- location transparency --- masking off the physical location of services
- access transparency --- masking any differences in representation and operation invocation mechanism
- concurrency transparency --- masking overlapped execution
- replication transparency --- masking redundancy
- failure transparency --- masking recovery of services after failures
- resource transparency --- masking changes in the representation of a service and the resources used to support it
- migration transparency --- masking movement of a service from one application to another
- federation transparency --- masking administrative and technology boundaries.

2.3.3 Real-time distributed system environments

Despite the relative maturity of distributed system environment research, real-time distributed system environment remains a neglected, if not unaddressed, topic. The result is that even if base technologies (such as microkernel, ATM networks etc.) can provide real-time services, a distributed system environment provides no corresponding abstractions to use these services. Even worse, a distributed system environment often mask off the real-time features of base technologies. Therefore, one of the main aim of this work is to extend the real-time features of base technologies to the distributed system environment level.

Figure 2.2: Real-Time ODP Functionality



One common misconcept is perhaps that distributed system environment is not the suitable technology for real-time applications because RPC (as one of the main technique basis of distributed system environment) is often criticized for providing poor performance or is not fast enough. This is a misconcept because the objective of real-time computing is to meet the timing requirements of an application, rather than being fast. The most important property of a real-time system is ***predictability***. On the other hand, fast is a relative term. As technology progress, there will be faster and faster RPC systems. Even now it is not difficult to provide milliseconds level RPC calls (as the required performance for the ***supervisory control*** targeted by our architecture, see also section 2.4). For example, there are already reports of systems that can provide hundreds of microseconds level RPC calls. [Biagioni93] [Johnson93]. Fast computing is helpful in meeting stringent timing constraints, but fast computing alone does not bring real-time properties.

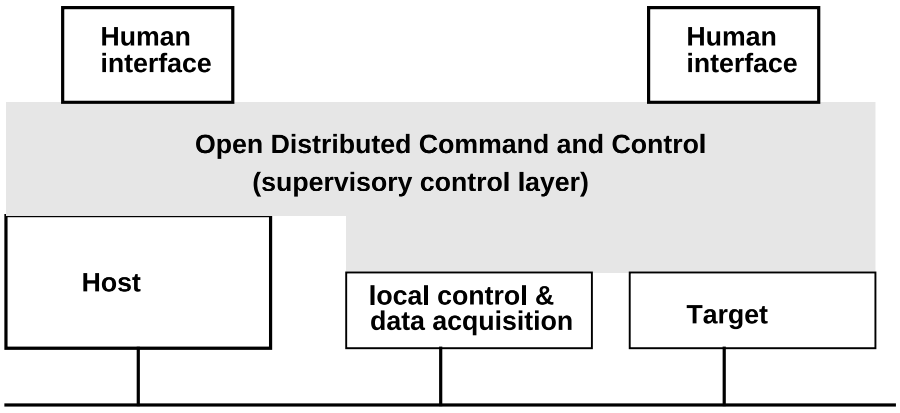
A real-time system must be able to handle time-constrained processing of requests. A real-time distributed system environment adds another dimension to the problem of distributed system environment, because the concern is now not only with the functional correctness, but also with the timeliness of the results produced. In Figure 2.2, a graphical illustration of the real-time distributed system environment functionality is given. The curve in the figure illustrates that the real-time distributed system environment functionalists are the trade-off of the real-time functionality and distributed system environment functionality. This reflects the fact that real-time functionality and distributed system environment functionality are often conflicting goals. For example, most distribution transparencies (such as RPC protocols) are based on *time redundancy* technologies, such technologies need to be revised for real-time applications.

2.4 Target

Real-time systems span a wide variety of field of applications, including military, industry, commerce, medicine and so on. This indicates a wide spectrum of possible problems.

The scope of this research for real-time applications is ***supervisory control*** [Northcutt88] as opposed to low-level, synchronous sampled data loop functions like sensor/actuator feedback control, signal processing, priority interrupt processing and so on.

Figure 2.3: Supervisory Control


Human interface
Human interface
Open Distributed Command and Control (supervisory control layer)
Host
local control & data acquisition
Target

Supervisory control is a middle-level function (see Figure 2.3), above the local control and data acquisition functions and below the human interface management functions. This type of system does not do much direct polling of sensors and manipulation of actuators, nor does it provide extensive man machine interfaces; rather, it deals with subsystems which provide these functions. The real-time response requirements of a supervisory control system are closer to the millisecond than either the microsecond or second ranges.

2.5 Summary

This chapter has examined the problem space and technology bases of real-time open distribued processing. An integrated system architecture is suggested and the benefits of the architecture are presented. The practical need and importance of the architecture is discussed along with the current technology trends in both distributed processing and real-time applications. It also suggests that the architecture may target (not exclusively) *supervisory control* as its applications.

3 ANSA Computational Model and Engineering Model

3.1 Introduction

This chapter provides a brief overview of ANSA Computational Model (ACM) and ANSA Engineering Model (AEM) to improve the readability of this document. Readers familiar with ACM and AEM may skip this chapter.

3.2 Computational Model

A computational model is a framework for describing the structure, specification and execution of programs. The principle behind and the concepts underlying the ANSA architecture are articulated via the ACM [Rees93]. This section briefly summarises the overall concepts of the ACM.

The key ACM concepts are:

(Computational) Object: a unit of program modularity state and operations for initializing, accessing and updating that state. Object state may contain references to the interfaces of itself and other objects.

Interface: a view of an object as an abstract service. An interface is specified as a set of operations together with synchronization and ordering constraints on the use of these operations.

Operation: part of an interface. An operation has a *signature* and a body which defines the effect and outcome from an *invocation* of the operation.

Signature: a specification of the name of an operation, the number and interface types of the argument parameters and, optionally, a set of *terminations* which specify the possible outcomes from the operation.

Activity: the agency by which computations make progress. An activity may pass from one object to another by the first *invoking* an operation on an interface of the second. Activities may split into parallel sub-activities and later recombine. New activities can be initiated to proceed in parallel, these may be able to communicate with other activities but are not dependent upon their initiating activity.

Termination: the specification of a set of possible outcomes from *invocations* of an operation. A termination has a name and specifies the *interface types* of the result parameters from an outcome with that name.

Interface type: the signature of the operations in an interface of the type.

(Operation) Invocation: the execution of the body of an operation defined by a reference to an interface and an operation name in a context established by the referenced interfaces and a set of arguments.

Server: in the context of an invocation, the object which provides the interface containing the operation being invoked.

Client: in the context of an invocation, the object from which the invocation was initiated.

The ACM is in two parts:

- **an interaction model** defines permitted forms of interaction and a type scheme within which potential interactions are to be classified. The interaction model consists of an invocation scheme and a type scheme.
- **a construction model** defines elements from which the interacting objects may be constructed.

The invocation scheme defines how clients may use interfaces provided by servers. Two kinds of operation, *interrogation* (call) and *announcement* (cast), are permitted. Invocation of an interrogation is a synchronous request/ response style. Invocation of an announcement is an asynchronous request only style, a new activity is created in the server and the invoking activity continues in the client.

The type scheme provides a set of types into which interfaces are classified and defines a relation over interface types that allows the detection of the possibility of interaction errors before the interaction commences.

The ANSA construction model provides the elements necessary to construct objects that conform to the ANSA interaction model.

3.3 Engineering Model

The AEM provides a framework for the specification of mechanisms to support distribution of application programs that conform to ACM. The details of AEM can be found in [ISO/IEC95]. The AEM contains a number of sub-components and supports a number of application-level components as shown in Figure 3.1.

- **Transparency Mechanisms** provide a uniform interface for distributed applications that address the problems and benefits of distribution. The transparency mechanisms communicate with one another via the nucleus and the network to achieve the desired transparency.
- **Nucleus** is the part of the AEM which provides minimal and sufficient support for the implementation of distribution. It encapsulates all of the heterogeneity of processor and memory architecture. The Nucleus itself is not distributable.

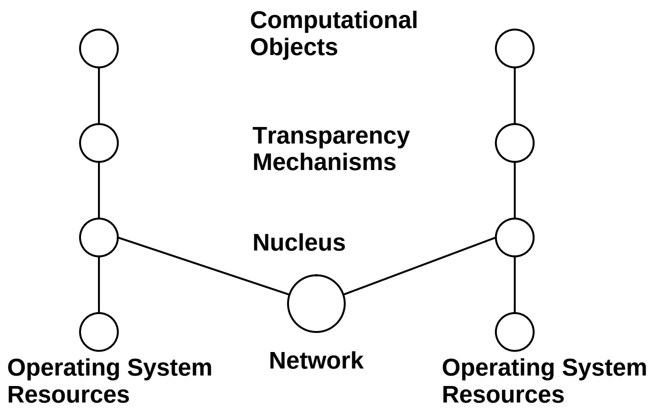
The main concepts of the AEM may be summarised:

Capsule: the collection of computational objects (in engineering form), transparency mechanisms and nucleus forming a virtual node of a network. It can be seen as the abstraction of an address space in a local operating system to provide the unit of protection and failure atomicity.

Thread: a sequence of instructions modelling a computational model activity within a capsule. It represents a unit of potentially concurrent activity that can be evaluated in parallel with other threads, subject to synchronization constraints.

Task: a virtual processor which provides a thread with the resources (e.g. a stack) it requires to progress. Tasks[1] provide the resources for real

Figure 3.1: ANSA Engineering Model



concurrency. An ANSA task is conceptually equivalent to an operating system *thread*.

Interface Reference: an interface reference is an identifier which contains sufficient information to allow the holder (the client) to establish communication with the interface denoted by the reference (the server). Interfaces have types (corresponding to their code component) which may be instantiated multiple times with different state (corresponding to their data component). Such instantiations are called *interface instances*, and interface references always refer to interface instances.

Channel: the abstraction for initiating operations to a specific remote interface and for receiving invocations on a specified interface. The initiating side (client) end-point of a channel is called a *plug*. The receiving side (server) end-point is called a *socket*. Channels are asymmetric in that a channel may have many clients (plugs) bound to it, but only one server (socket).

- **Binder**: a component to support binding: the process by which an activity in one object establishes the ability to invoke operations at an interface to some other object. Binding establishes and controls the communication channels between objects so their interactions are possible.

  Interpreter: a portion of the nucleus. It can be viewed as defining an instruction set for a distributed abstract machine. It interprets inter-object interactions (invocations), performs all argument and result processing, and links threads to sessions (a session is a cache of a plug or a socket) and transfers buffers between them. It also provides the necessary session, thread and task state changes to complete the execution of each instruction.

1. ANSA threads are cheap resources (each requires less than one hundred bytes of memory); whereas ANSA tasks are expensive resources (each requires several kilobytes of memory). In a distributed application there may be many threads (e.g. 100’s or 1000’s); it is important only to allocate a task to execute a thread when there is a processor available to run it.

4 Areas of Work

The areas of engineering design work include:

- a real-time programming model
- a real-time communication system.

The following sections provide a brief overview for each of the two major issues. More details can be found in later chapters.

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). A serious difficulty is that the actual timing characteristics of software are determined not only by the raw processor speed, but also by the sharing policy for scarce resources. For example, the real-time response of a time-shared system depends heavily on the processor scheduling policy of its operating system. In most high level languages, this dependency is considered as non-essential detail that is to be hidden from the programmer. As a result the performance of software implemented in these languages becomes sensitive to system resource allocation strategies (in a dynamic system, this means performance depends on system load), and outside the control of individual programmers. More complex resources such as the communication subsystem of distributed systems further accentuate the problem with the introduction of (sometimes distributed) resource allocation algorithms which are usually inaccessible to the application programmer.

The real-time programming model developed in this document 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. This section outlines some mechanisms for providing such functions within an RPC communication infrastructure. Three extensions aimed at making the ANSA communication system more suitable for real-time applications are identified. These extensions are:

- a parallel communication protocol stack to allow the preallocation of communication resources 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 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 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. The solution to the timed RPC presented in this document 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 transportation 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, timeliness and performance are the main concerns of the real-time communication system.

5 A Real-Time Programming Model

5.1 Introduction

This chapter discusses some 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.

5.2 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. [Herbert93a] [Blair92]. 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 [Black86], the *passive object* model [Allchin83], and the *actor object* model [Attoui91].

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.

5.3 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.

5.3.1 ANSA object execution model deficiencies for real-time applications

To be more specific, ANSAware is used as an example to detail the AOEM. In ANSAware, 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 ANSAware 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 run time knowledge of resource usage.

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.

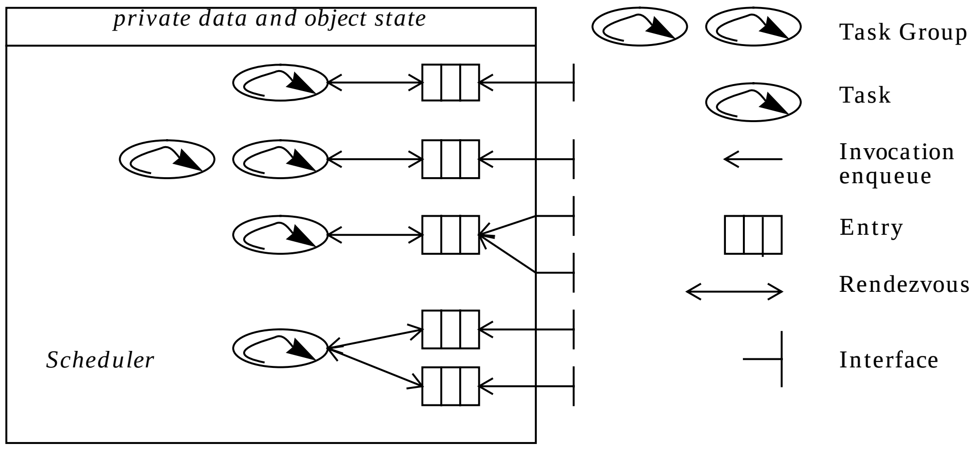
5.4 Real-time objects

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, ***scheduling entry*** or shortly ***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 5.1: Real-time object illustration



In Figure 5.1, a graphical illustration of a real-time object is given.

System tasks may be allocated for each individual entry. The tasks allocated are dedicated to execute the threads on the entry. A thread is also allowed to *rendezvous* with other entries dynamically. A ***rendezvous*** of a thread with an entry means that the thread 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 thread/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 5.7.2).

The flexibility for allowing a thread 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 environment (detailed in section 5.6).

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 5.2) is *Shared Single Entry* configuration, in which all interfaces share a single entry with all tasks serving all incoming requests on all interfaces.

Figure 5.2: Shared Single Entry (ANSA) Configuration

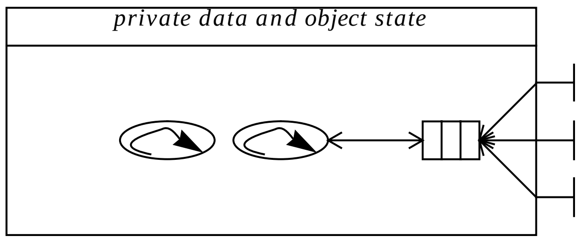


Figure 5.3: Multiple Single Entries

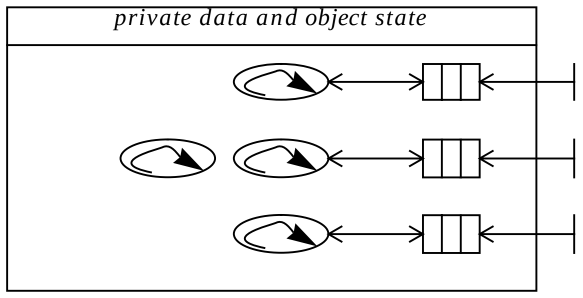
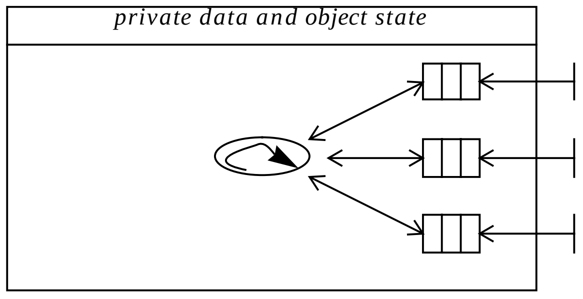


Figure 5.4: Single Task Multiple Single Entry



Another simple form (Figure 5.3) is *Multiple Single Entries*, in which each interface has its own entry.

Another interesting simple form (Figure 5.4) 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 5.1. It contains the three simple configurations.

5.5 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 can be 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.

5.6 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 is supported by the multiple application-selectable scheduling policy modules on top of a shared set of scheduling mechanisms.

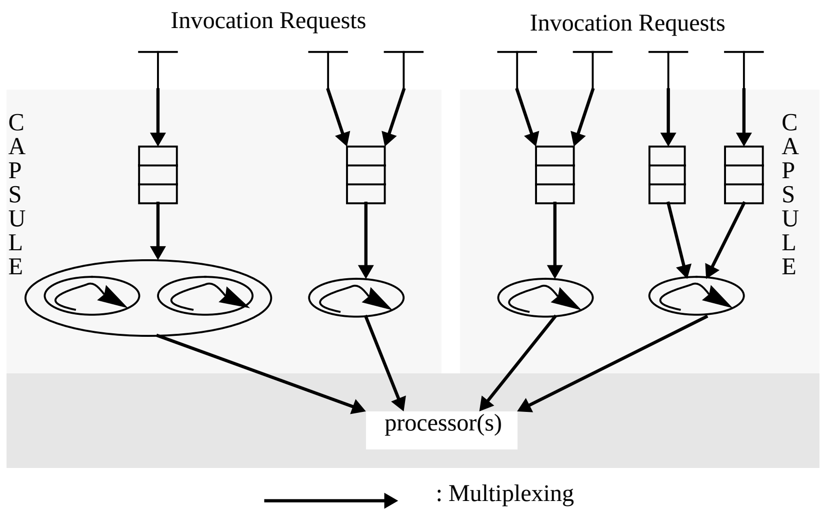
The system scheduling behaviour is defined in layers as:

- 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 5.5 illustrates the structure of this multiplex.

Figure 5.5: 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 [Nicolaou91]. 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.

5.7 Priority scheduling

This section discusses the mechanisms needed to provide the *priority based* scheduling model in the real-time ANSA framework. Priority based scheduling is the most popular (and perhaps more important, supported) real-time scheduling method [Ada9X93] [POSIX]. There are well-known analytic methods [Lehockzy87] to decide the schedulability of a set of periodic or aperiodic tasks.

While priority is a well defined and generally applicable notion, its role in task scheduling needs to be carefully examined. A clear definition of the *priority inheritance* (section 5.7.1) and *priority ceiling* (section 5.7.3) --- used when the enforced synchronization during a task and a thread rendezvous --- is needed to understand how priority works on tasking.

5.7.1 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***.

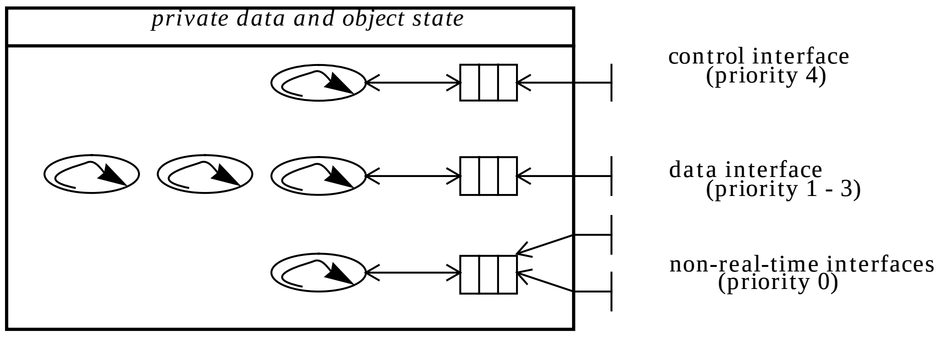
Two levels of priority inheritance schemes may be defined. They are called (basic) ***priority inheritance*** and ***transitive priority inheritance.*** In the first scheme, a serving task with a low priority raises its priority to the higher priority of an invocation request before it starts the service, 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.

5.7.2 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 5.6: Layered Management of Priorities


private data and object state
control interface (priority 4)
data interface (priority 1 - 3)
non-real-time interfaces (priority 0)

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 5.6, 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.

5.7.3 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 5.7: Priority Inversion in Real-Time ANSA Objects

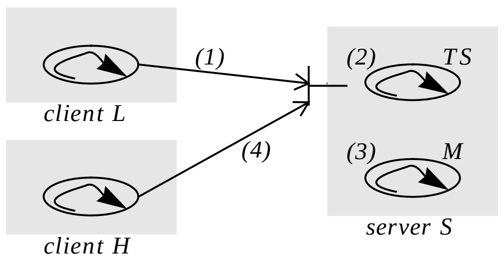


Figure 5.7 shows an example of priority inversion in real-time ANSA objects. Suppose there is a server object *S* with an interface *I* and client objects *L* and *H*. *L* is a low priority client --- it runs a low priority task which sends low priority invocations to *S*. *H* is a high priority client --- 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.

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.

5.8 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. Deadline guarantee is discussed in more detail in [Li93].

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 the next chapter.

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 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.

5.9 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* [Miller90], 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.

5.10 Application controlled rendezvous

In addition to allocating system task(s) on an entry for serving requests, a thread is also allowed to rendezvous with entries at run-time. The interface may be as follow:

```
Rendezvous(entry_set, timeout)
```

The effect is that the thread 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 Rendezvous statement ensures that only one request is executed in the accepting thread (with the service task). Other requests are queued to be processed later
- the application thread may perform its own synchronisation. This may help improve resource usage by synchronizing before a request starts executing, and not after
- the application thread may initiate object invocations like other client tasks
- the application thread 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.

5.11 Summary

This chapter has described a real-time programming model. 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 programming model described in this chapter is a conceptual model, which has not included an application programming interface (API). The API for real-time ANSAware 1.0 can be found in [Li94].

6 A Real-Time Communication System

6.1 Introduction

Real-time applications present much more complicated functional requirements to the underlying communication systems than the non-real-time ones. This chapter discusses some designs for providing such functions within an RPC communication infrastructure. The facilities discussed are:

- a parallel protocol stack for the preallocation of communication resources and the removal of layered multiplexing. This allows the application to explore network QoS support
- a timed RPC protocol for the association of deadlines with invocations
- a decomposable RPC protocol for the tradeoffs between functionality and performance. This work provides the necessary insight for the design of new transportation protocols with QoS parameters.

6.2 Towards a parallel protocol stack

The main advantage of the ANSAware 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 [Tennenhouse89].

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.

Figure 6.1: Parallel Protocol Stack

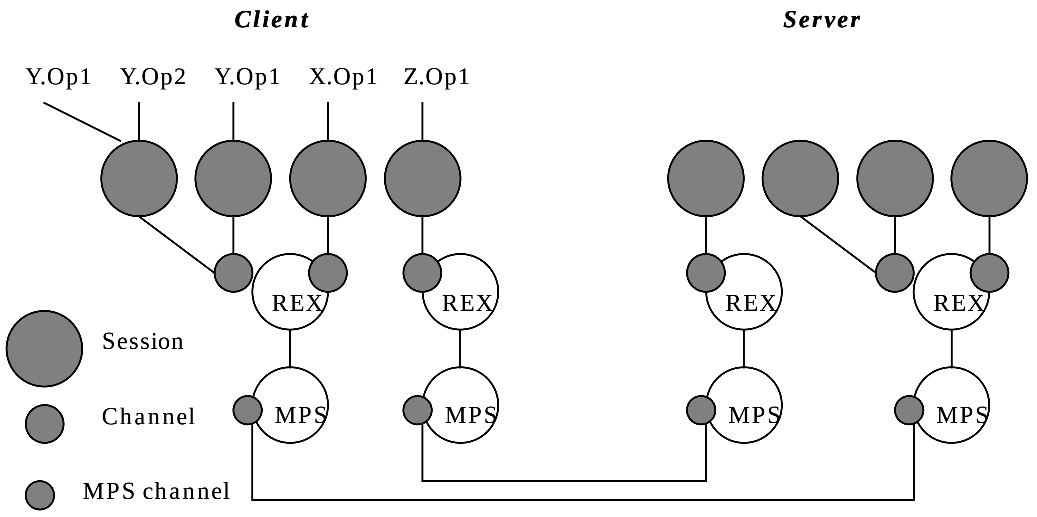
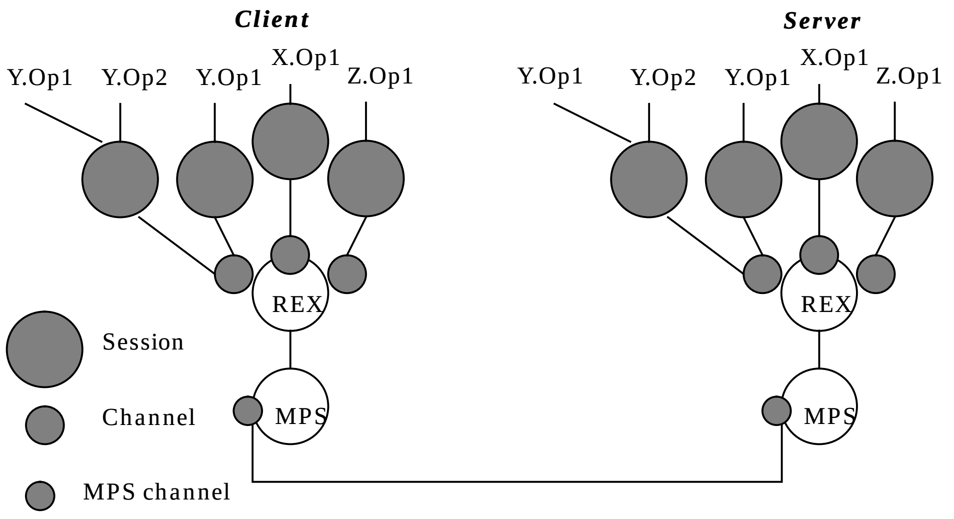


Figure 6.2: Multiplexing in the Testbench



The result is a parallel communication protocol stack, as illustrated by Figure 6.1, which is in contrast with the original ANSA multiplexing structure for a server/client interaction as illustrated in Figure 6.2.

6.3 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).

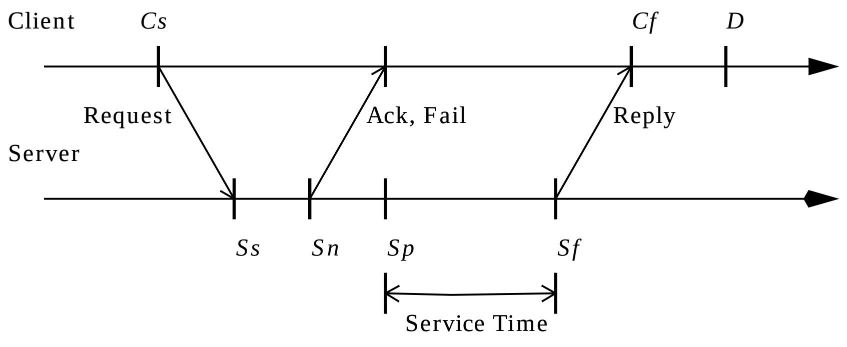
Invocations in real-time ANSA can attach deadline constraints to their communication requests. Such TRPC calls raise the following three issues:

- 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 sense of time --- the deadline. It is thus necessary to assume a common sense of time is provided by the infrastructure between a client and a server
- 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 6.3.

Figure 6.3: 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.

6.3.1 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 [Lee90]. 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.

6.3.2 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.

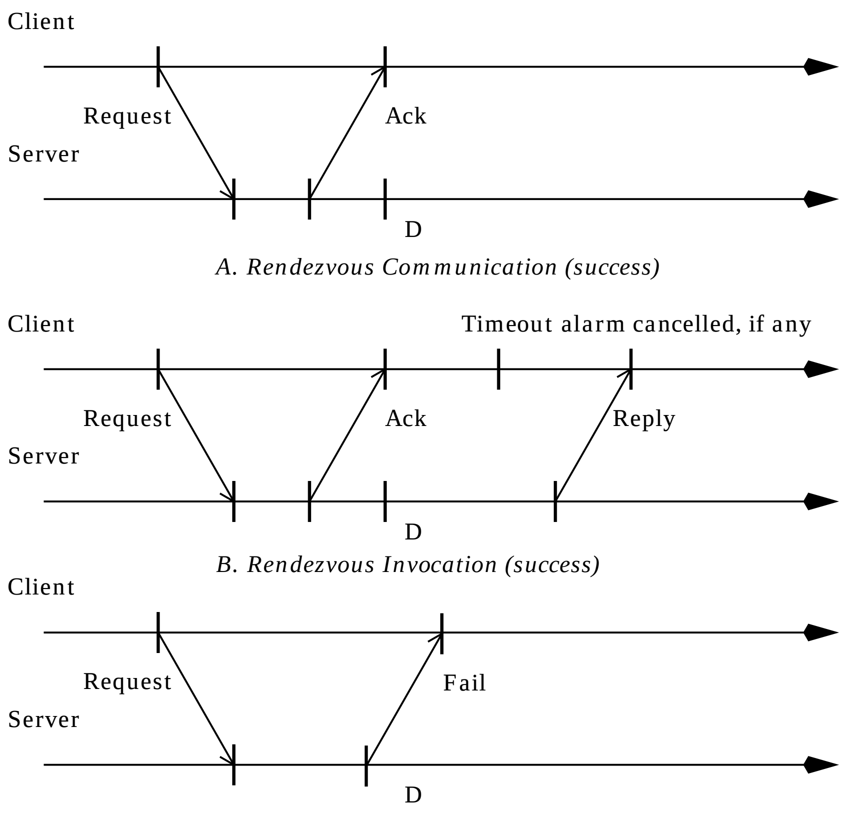
It should be pointed out that using the two separate arguments does not solve the TRPC consistency problem. Rather, the two arguments give the problem a more realistic definition, allowing different relaxations be explored.

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 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

Figure 6.4: Rendezvous Communication/Invocation Interaction



A. Rendezvous Communication (success)

B. Rendezvous Invocation (success)

C. Rendezvous Communication/Invocation (fail)

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 6.4.

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*.

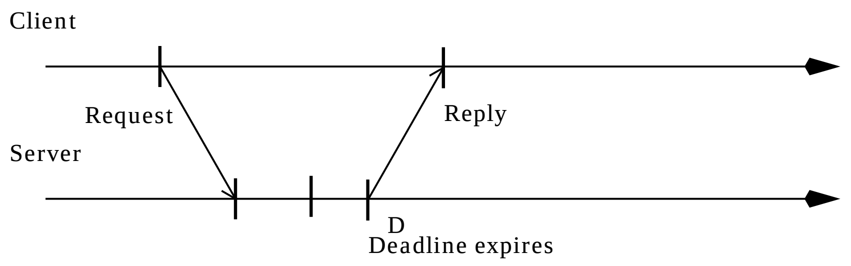
Obviously, it is not necessary to choose the timeout and deadline the same value. A timeout may be smaller than a deadline, to specify that an acknowledge should be returned earlier; it may be greater than a deadline, to allow the request to have a better chance of success.

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.

6.3.3 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 6.5: 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 6.5.

6.4 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 6.6. The three layers are layered functions sharing the same protocol data structure --- sessions, and to provide just one protocol service.

Figure 6.6: REX Functions Layers

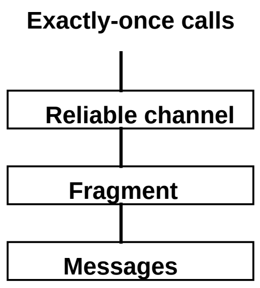
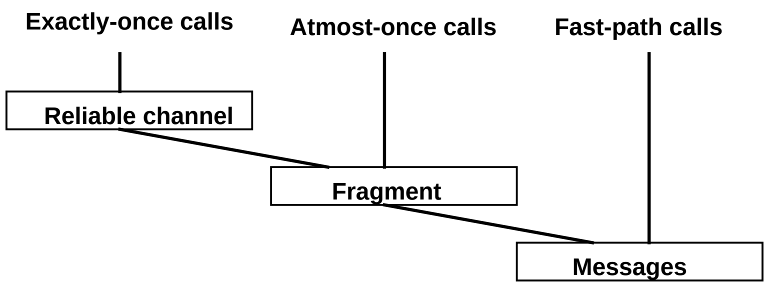


Figure 6.7: A Decomposable Protocol



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 6.7. 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.

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 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.

6.5 Summary

Real-time applications present more complicated functional requirements to the underlying communication systems. This chapter discussed some mechanisms for providing such functions within the ANSAware RPC communication system. The facilities examined are:

- a parallel protocol stack
- a timed RPC protocol
- a decomposable RPC protocol.

7 Summary

7.1 Major results

This document presents an engineering framework for open, distributed real-time applications. The major results of this work are:

- the identification of the practical need and importance of an open, distributed real-time architecture
- the definition of a set of requirements for the design of an open, distributed real-time architecture
- the development of a framework for the uniform treatment of real-time computing and non-real-time computing i.e. system integration
- the identification of a set of technology bases
- the development of a real-time programming model and solutions for some general real-time problems
- the development of a RPC based real-time communication system.

7.2 Conclusions

The major conclusions of this work are:

- real-time issues can be addressed in an *OPEN* architecture
- the ANSA computational model can be extended to cater for real-time programming
- a real-time open system engineering model is viable
- real-time technologies can be integrated with open architecture.

7.3 To be addressed issues

It is anticipated that the ANSA performance model and infrastructure will cover the following important aspects:

- a Quality of Service framework
- an explicit binding framework
- a performance transparency architecture
- a stream interface.

The engineering design for these ares relies on the forthcoming computational model design and the resource management programming model design, and is therefore left not to start until the two high level design finish.

7.4 Acknowledgments

The author wishes to thank Leon Bouwmeester, Nigel Edwards, Andrew Herbert, Nicola Howarth, Rob van der Linden, Dave Otway, Jean-Bernard Stefani, and Francis Wai for their review and feedback on the document.

References

[Ada9X93]

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

[Allchin83]

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

[Attoui91]

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

[Biagioni93]

E Biagioni, E Copper, and R Sansom, Designing a Practical ATM LAN, IEEE Network, March 1993.

[Black86]

A P Black et al., Distributed and Abstract Types in Emerald, IEEE Transactions on Software Engineering, 13(1), January 1987.

[Blair92]

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.

[Gopinath93]

P Gopinath and T Bihari, Concepts and Examples of Object-Oriented Real-Time Systems, In Readings in Real-Time systems, Y H Lee and C M Krishna ed., IEEE CS Press, June 1993

[Herbert93]

A Herbert, The Challenge of ODP, TR 33, APM Ltd., Poseidon House, Castle Park, Cambridge U.K., 1993 Also Appeared as an Invited Paper for the Berlin ODP Conference,   October 1991.

[Herbert93a]

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

[Ishikawa90]

Y Ishikawa, H Tokuda, and C W Mercer, Object-Oriented Real-Time Language Design: Constructs for Timing Constraints, In OOPSLA/ECOOP'90, Ottawa, October 1990.

[ISO/IEC95]

ISO/IEC 10746-3, ITU-TS Recommendation X.903: Reference Model of Open Distributed Processing: Architecture, January 1995.

[Johnson93]

D B Johnson and W Zwaenepoel, The Peregrine High-performance RPC System, Software---Practice and Experience, 23(2), February 1993.

[Lee90]

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

[Lehockzy89]

J P Lehockzy, L Sha and Y Ding, The Rate Monotonic Scheduling Algorithm --- Exact Characterization and average-case Behaviour, Proc. of Tenth IEEE Real-Time Systems Symp., 1989.

[Leung90]

W H Leung et. al., A Software Architecture for Workstations Supporting Multimedia Conferencing in Packet Switching Networks, IEEE JSAC, 8(3):380-390, April 1990.

[Li93]

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

[Li94]

G Li, ANSAware/RT Version 1.0: Programming and System Overview, APM document 1207, APM Ltd., Poseidon House, Castle Park, Cambridge, CB3 0RD, 1994.

[Miller90]

F W Miller, Predictive Deadline Multi-Processing, Operating Systems Review, vol. 24, no. 4, 1990.

[Northcutt88]

J D Northcutt. Mechanisms for Reliable Distributed Real-Time operating Systems: The Alpha Kernel, Orlando FL: Academic Press, 1987.

[POSIX]

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

[Rees93]

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

[Stankovic88]

J A Stancovic, Misconceptions about Real-Time Computing: A Serious Problem for the Next Generation, IEEE Computer, 21(10), October 1988.

[Tennenhouse89]

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

[Wai93]

F Wai, D Otway, N Howarth and A Herbert, A Performance Framework, APM document 1051, APM Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993.