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

ANSAware/RT 1.0 Manual

anon

Abstract

This document forms the manual set of ANSAware/RT 1.0. It contains all information specific to the ANSAware/RT release. For all non-real time specific information see the ANSAware 4.1 manual set.

APM.1441.00.03 **Draft** 22nd March 1995

Briefing Note

Distribution:
Supersedes:
Superseded by:

ANSAware/RT 1.0 Manual

ANSAware/RT 1.0 Manual

anon

APM.1441.00.03

22nd March 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

ANSAware/RT Overview

1 ANSAware/RT

Distributed computing systems are increasingly seen as important enablers for new business opportunities. New products and services are emerging in local and wide area networks such as the information superhighway.

Advances in digital communication networks and in personal workstations allow simultaneous processing of real-time data, voice, and video. There is strong demand for real-time functionality, provided 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 common [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 constraints can be addressed in the context of emerging software practice, addressing scale, evolution and distribution.

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. Even though distributed computing environments and real-time computing are established techniques, their integration is a new challenge, because they seldom use compatible mechanisms.

ANSAware/RT is an implementation of the ANSA architecture for distributed real-time systems. It extends ANSAware 4.1 [APM92] (the ANSA distributed computing environment) with compatible real time technology.

1.1 ANSA, ANSAware and ANSAware/RT

ANSA is an architecture for Open Distributed Processing. ANSAware is an example implementation of that architecture. It provides a set of tools and libraries which allow programmers to distribute their applications in a heterogeneous system. It includes a set of basic services in support of dynamic binding, relocation, and node and system management. ANSAware is currently in release 4.1.

ANSAware/RT extends ANSAware for real-time processing.

1.2 Overview of documentation

This manual contains the documentation of ANSAware/RT specific topics. This chapter contains overview material, release notes, platform and support information. Chapters 2-7 describe the design for ANSAware/RT and chapters 8-13 describe the implementation, complete with manual pages.

For installation, system management and all other non-real time distributed systems and application programming issues please refer to the ANSAware 4.1 manual set, which is included with the ANSAware/RT release.

1.3 ANSAware/RT version 1.0 release notes

The main goal was to implement an *extended* ANSAware that runs over a standard real-time environment and *retains* the real-time properties of the environment. Specifically, the goal can be divided into the following items:

- compatible with ANSAware 4.1
- running over a de-facto industry standard: real-time POSIX threads
- full p-thread real-time scheduling and threading capabilities
- selective communication multiplex by QoS specification and explicit binding operations
- application controlled resource allocation
- supporting a real-time programming model (described in chapter 6)
- comparable to other distributed real-time system environments
- interoperation between different real-time platforms
- interoperation between different real-time and non-real-time platforms.

1.3.1 Functional extensions

In terms of functional requirements, ANSAware/RT extends ANSAware 4.1 in the following ways:

- extended tasking system
  - — entry: this is a new abstraction which represents a scheduling point. Functions related to entry including creation, release, allocation of tasks, association of interfaces etc. are provided
  - — real-time scheduling: preemptive priority-based scheduling
  - — multiple scheduling policies: the co-existing of real-time and non-real-time scheduling supports
  - — real-time tasks: full real-time p-thread functions.
  - — stacked threads: a thread may use its task resource to execute another thread
  - — real-time threads: a thread may be associated with a priority and/or deadline
  - — multiple thread scheduling policies based on policy/mechanism separation
- extended communication system
  - — multiple execution (RPC) protocols
  - — timed execution protocol: this is a new RPC protocol which understands priority, deadline and deadline types
  - — state-full execution protocols and message passing protocols: this allows the pre-allocation of separate communication endpoints for different interfaces
  - — selective multiplex of communication channels
- extended application programming interface
  - — abstractions for access tasking resources

  - — QoS objects, two kinds of QoS objects are introduced: one for the description of a communication end point, another for the description of in-band QoS requirement for an invocation
  - — explicit binding operations: to bind an interface with a communication channel of a specific QoS
  - — invocations with QoS: the attachment of in-band QoS based on invocations
- extended PREPC and IDL.

1.4 Compatibility between ANSAware 4.1 and ANSAware/RT 1.0

1.4.1 Portability

ANSAware 4.1 is the current released ANSAware. ANSAware/RT 1.0 is based on ANSAware 4.1. ANSAware/RT 1.0 API is an extension of the ANSAware 4.1 API, therefore all ANSAware 4.1 services will port to ANSAware/RT 1.0 with minimum changes.

Two changes need to be made to run ANSAware 4.1 applications over ANSAware/RT.

First, ANSAware/RT dose not support the asynchronous I/O operations of ANSAware 4.1. This requires a change to the equivalent synchronous I/O operations.

Second, the PREPC qos parameter in an object invocation statement is used in ANSAware 4.1 just for controlling REX retry numbers, while in ANSAware/RT it has a more general rule, and is itself a control record.

1.4.2 Interworking

Applications developed for ANSAware/RT 1.0 will interwork with applications running on ANSAware 4.1.

1.5 Requirements

ANSAware/RT runs over DEC Alpha/OSF1, versions 1.3 and 3.0.[1]

1.6 Release media

ANSA sponsors may access the released software by ftp.

The software and manuals are normally available on DEC TLZ tape cartridge.

1.7 Support

ANSAware software is being made available in source form to enable recipients to undertake experiments and porting. It is not guaranteed to be free of defects although it has been extensively tested with all the facilities which the ANSAware/RT implementation team have available. ANSAware/RT

1. It has been successfully ported to HP/RT and LynxOS [APM.1207], but due to a lack of testing facilities, these ports cannot be made available with this release.

is not supported as a warranted software product and any or all parts of ANSAware/RT may change in future versions. The ANSA architecture will continue to evolve and these changes will be reflected in the design and implementation of ANSAware and ANSAware/RT.

If you experience any trouble with porting or using ANSAware/RT please contact Architecture Projects Management Ltd. (APM). APM wish to be informed of all problems and difficulties encountered and, within the constraints of the ANSA project, will give advice on how to deal with them.

If your use of ANSAware/RT is governed by a Software Licence, then this will tell you in detail the nature and extent of any support that you are entitled to receive.

APM will be pleased to receive any extensions, improvements, new ports or additional services for incorporation in future releases, at its discretion. Conditionally compilable sources are preferred.

ANSAware bug reports can be filed on: **ansaware@ansa.co.uk**.

ANSAware/RT Design

2 ANSA Real Time Architecture

2.1 Motivation

Distributed real-time processing places unique requirements on systems and their designs (e.g. predictability, programmer control, timeliness, mission orientation and performance, see Chapter 3). Such features do not exist in today's computing environments.

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.

The ANSA real-time architecture and ANSAware/RT design shows how it is possible to extend a distributed system environment to support real-time applications and hence help avoid these problems.

2.2 Design scope

In the design for a real-time ANSA, we propose an integrated architecture for distributed real-time systems. It contains engineering aspects of a distributed environment applied to real-time applications. Rather than being focused on narrow subsystems or algorithms, the perspective and scope of the design is the entire system environment. The emphasis is on an engineering design that stands on both *current* and future technologies. The design does not afford full coverage of all engineering aspects of a distributed real-time environment. Streams, explicit binding and a synchronous programming model are some of the planned extensions necessary for distributed interactive multimedia applications.

The principle issues covered by this design 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
- distributed 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.

2.3 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 constrain open system architecture.
- explains how real-time technologies are integrated with open systems.

2.4 Outline of the design manual

The design manual is structured as follows:

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

3 An Open Architecture for Real-Time Processing

3.1 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 most of today's computing environments, and must be added explicitly.

3.1.1 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.

3.1.2 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.

3.1.3 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. 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 relationships among related computational activities.

3.1.4 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 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.

3.1.5 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 popular layered design for non-real-time applications. Also, real-time applications often require a trade off between modularity, flexibility and functionality to maximize performance.

3.2 An integrated system architecture

The objective of this design is to provide 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 and distributed control. Increased integration allows existing distributed system 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 media 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 special add-ons***
- ***Real-time applications are evolving towards large distributed systems.*** One-million-line real-time software systems are becoming 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 and distribution.***

3.3 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.

3.3.1 Contributory technologies

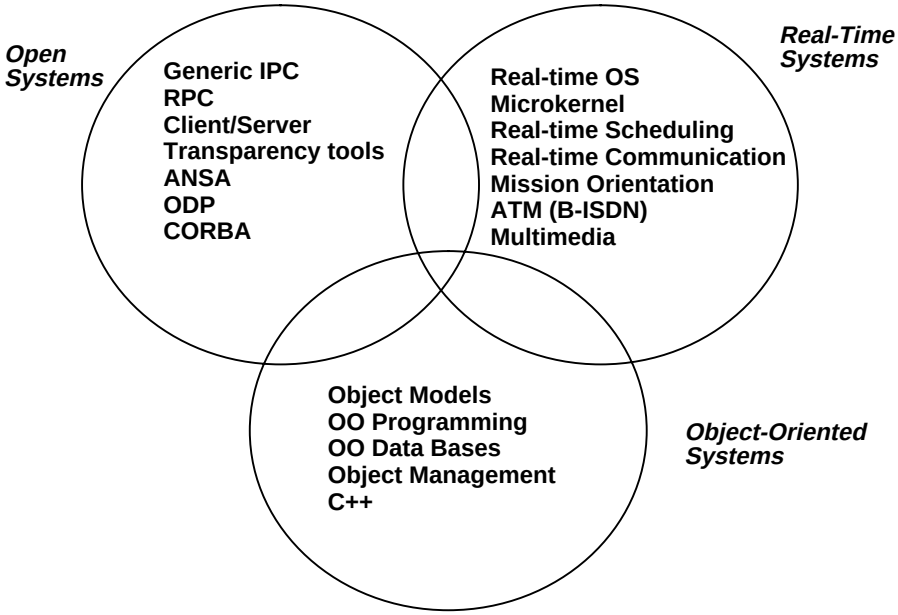
The fundamental contributory technologies are illustrated in Figure 3.1. It represents the integration of real-time systems, open systems and object oriented systems.

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

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

Object oriented technology provides the functionality for software reuse and maintenance.

Figure 3.1: Contributory Technologies



3.3.2 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.

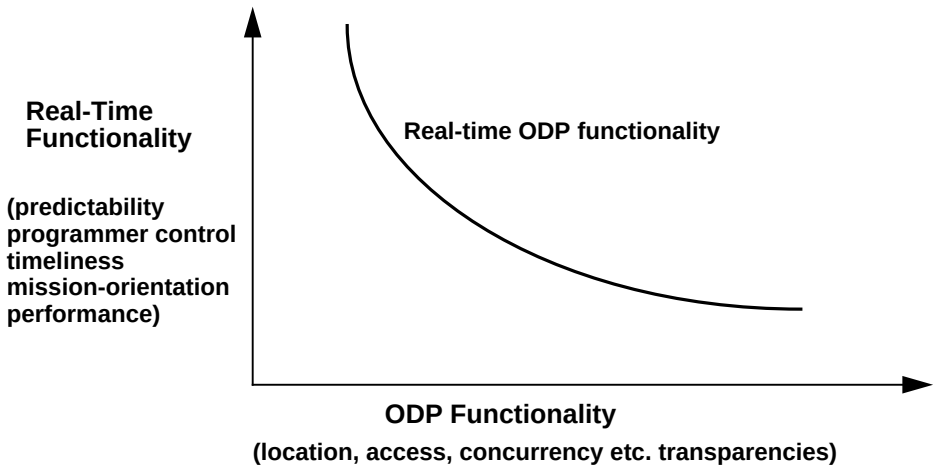
It is now recognised[APM.TR.33] that distribution transparency can be broken down into a number of individual transparency issues:

- location transparency --- masking the physical location of services
- access transparency --- masking 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 technological boundaries.

3.3.3 Real-time distributed system environments

Despite the relative maturity of distributed system practice, real-time distributed systems remain 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 masks the real-time features of base technologies. Therefore, one of the main aims of this work is to extend the real-time features of base technologies to the distributed system environment level.

Figure 3.2: Real-Time ODP Functionality



One common misconception is perhaps that distributed system environments are 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 misconception 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 progresses, 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 3.4). 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 3.2, a graphical illustration of the real-time distributed system environment functionality is given. The curve in the figure illustrates that real-time functionality and distributed systems environment functionality are often conflicting goals, which must be traded against one another. For example, most distribution transparencies (such as RPC

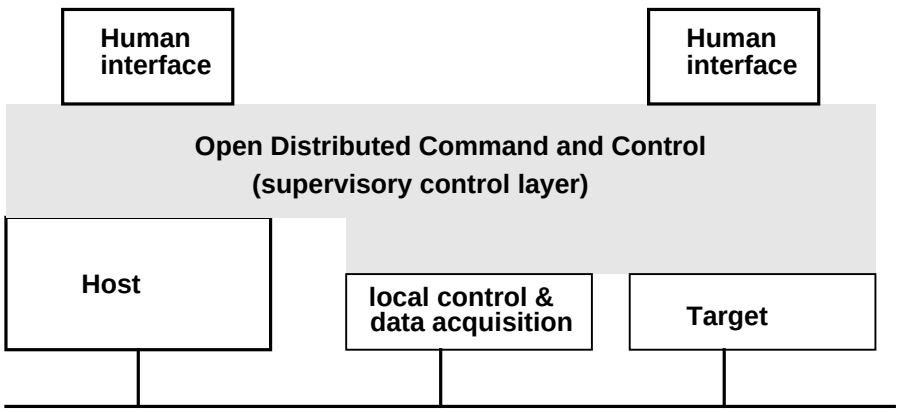
protocols) are based on *time redundancy* technologies, such technologies need to be revised for real-time applications.

3.4 Target

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

The scope of this design 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 3.3: Supervisory Control



Supervisory control is a middle-level function (see Figure 3.3), 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.

3.5 Summary

This chapter has examined the problem space and technology bases of real-time open distributed 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.

4 ANSA Computational Model and Engineering Model

4.1 Introduction

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

4.2 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 ANSA Computational Model [APM.AR.01]. This section briefly summarises the overall concepts of the ANSA Computational Model.

The key ANSA Computational Model 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 ANSA Computational Model 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.

4.3 Engineering Model

The ANSA Engineering Model provides a framework for the specification of mechanisms to support distribution of application programs that conform to ANSA Computational Model. The details of ANSA Engineering Model can be found in [ISO/IEC95]. The ANSA Engineering Model contains a number of sub-components and supports a number of application-level components as shown in Figure 4.1.

- **Transparency Mechanisms** provide a uniform interface for distributed applications. They 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 ANSA Engineering Model 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.

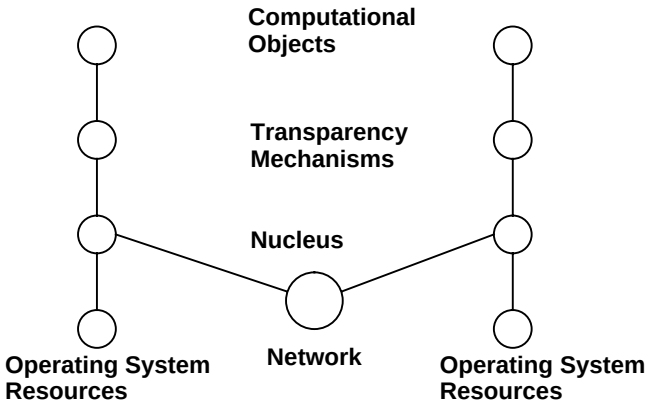
The main concepts of the ANSA Engineering Model may be summarised as follows:

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

Figure 4.1: ANSA Engineering Model



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 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 one interface reference always refer to one interface instance

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*. There is a one-to-one correspondence between channels and interface instances. 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

1. ANSA threads are cheap resources (each requires less than one hundred bytes of memory); whereas ANSA tasks are expensive resources (each requires several kilo-bytes 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.

session, thread and task state changes to complete the execution of each instruction.

5 ANSAware/RT Design Overview

The areas of ANSAware/RT 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 chapters 6 and 7.

5.1 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 ideally). 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 programming systems, 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 systems 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 ANSAware/RT 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.

5.2 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, TPC etc.
- a timed RPC protocol to allow the association of deadlines with invocations. ANSAware 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, i.e. 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.

6 Real-Time Programming Model Design

6.1 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 allocation, scheduling mechanisms and policies, and the application's control over scheduling.

6.2 Distributed object execution

The use of an object-oriented data model and the client-server execution model make the distribution of data and 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 [APM.TR.18]. 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.

6.3 ANSA object execution

Collectively, the ANSA Computational Model and ANSA Engineering Model define the ANSA object execution model. The ANSA object execution model 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.

6.3.1 ANSA object execution model deficiencies for real-time applications

The ANSA Object Execution Model model is designed for object distribution, but not for real-time applications. It lacks real-time predictability in the following sense:

- it multiplexes both tasks and communication channels whenever possible
- both thread/task scheduling and communication scheduling are implicit
- no abstraction is provided to express urgency and resource requirement for application programmers.

6.4 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 ***entry***, is introduced as the basic mechanism for real-time scheduling.

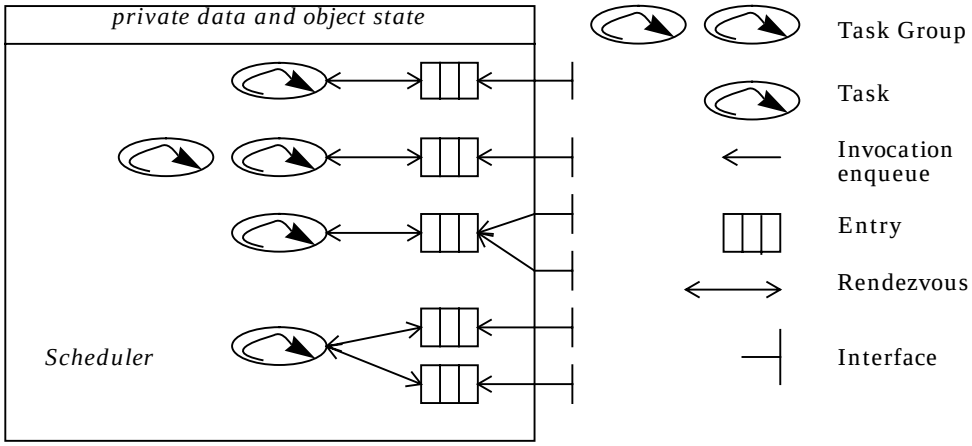
An entry is a thread queue with a record of control data. An entry may be created dynamically, and interfaces of an object may be bound to it. When an interface is bound to an entry, each operation request on the interface will be transferred (by the infrastructure) to a thread enqueued on the entry.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

In Figure 6.1, 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,

Figure 6.1: Real-time object illustration



whether a task can rendezvous with a specific entry, and what kinds of resource scheduling policies are used.

The ability to allocate a new entry for some interfaces reflects the need to separate such 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 6.7.2).

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 6.6).

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

Figure 6.2: Shared Single Entry (ANSAware) Configuration

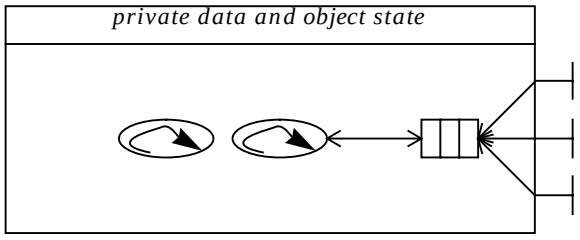


Figure 6.3: Multiple Single Entries

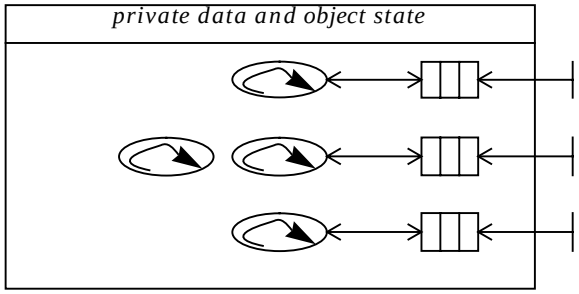
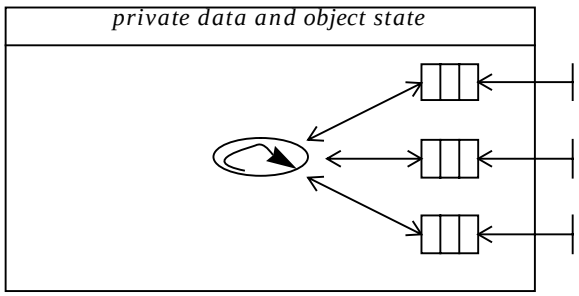


Figure 6.4: Single Task Multiple Single Entry



Another simple form (Figure 6.3) is *Multiple Single Entries*, in which each interface has its own entry.

Another interesting simple form (Figure 6.4) 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 6.1. It contains the three simple configurations.

6.5 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:

- it allows extra flexibility in conjunction with the server scheduler, in determining how the invocation is to be processed
- it allows a low-priority invocation to be sent from a high-priority task without having to enhance the server (thread) task's priority
- 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 system resource management is also determined by the scheduling policy (the interpretation of the scheduling parameters) and the resources allocated to the service. This is further explained in the following sections.

6.6 Scheduling

The main goal of the real-time ANSA tasking design is to allow maximum control of scheduling at the application level. Care has been taken to achieve a 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:

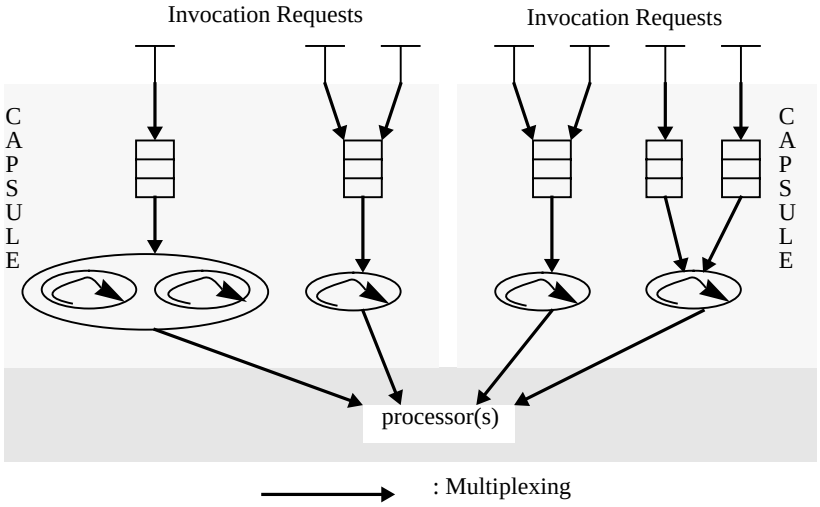
- thread scheduling --- the rendezvous scheduler on each entry
- task scheduling --- the nucleus scheduler on tasks.

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 6.5 illustrates the structure of this multiplex.

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

- it allows the use of a specific scheduling policy (thread scheduling policy) suitable for each interface or each interface class
- it 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
- it allows the possibility of end-to-end scheduling and guarantees
- preserves the modularity and separation of service interfaces.

Figure 6.5: Threads, Tasks and Processor(s) Multiplexing



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.

6.7 Priority scheduling

This section discusses the mechanisms needed to provide a *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 [Lehockzy89] 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 6.7.1) and *priority ceiling* (section 6.7.3) --- used when the enforced synchronization during a task and a thread rendezvous --- is needed to understand how priority works on tasking.

6.7.1 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 a 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.

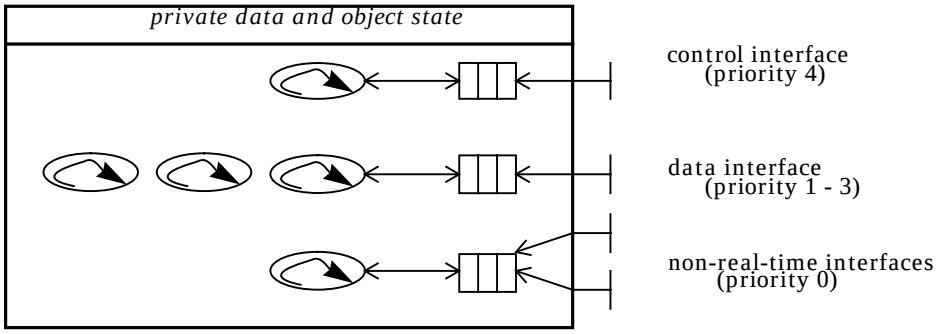
6.7.2 Resource allocation and task preemption

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 for 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 an invocation priority dynamically --- 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.

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 6.6, three entries are allocated: one for non-real-time interfaces, one for a real-time data interface, and one for a real-time control interface. 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 queuing 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

Figure 6.6: Layered Management of Priorities



system, the three serving tasks may provide two preemption possibilities among themselves with the priority inheritance mechanism: an invocation with 2 preempts an invocation with priority 1, and later the invocation with priority 2 is preempted by an invocation with priority 3. 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.

6.7.3 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 challenges of a static priority based programming model.

Figure 6.7: Priority Inversion in ANSAware/RT Objects

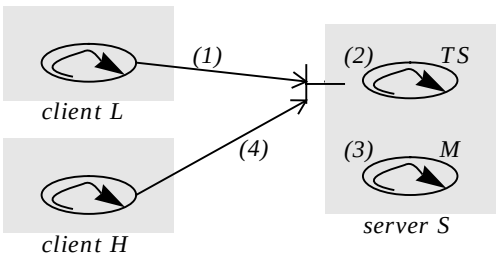


Figure 6.7 shows an example of priority inversion in real-time ANSA objects. Suppose there is a server object *S* with an interface *I* and client objects *L* and *H*. *L* is a low priority client --- it runs a low priority task which sends low priority invocations to *S*. *H* is a high priority client --- 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.

6.8 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 *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 previous section. It should be pointed out that deadline based scheduling only provides 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

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.

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.

6.9 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 threads 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.

6.10 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.

6.11 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.

7 Real-Time Communication System Design

7.1 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 an opportunity for the inclusion of transport protocols which support QoS management.

7.2 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, in particular the two level modules have no interactions when channels are created and destroyed. The 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), this inevitably introduces non-determinism.

Detailed discussions of the adverse effects, 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 the MPS interface as connection-based, and to maintain 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 REX to use this connection-based interface
- extend the programming interface so that applications have control over these connections.

Figure 7.1: Parallel Protocol Stack

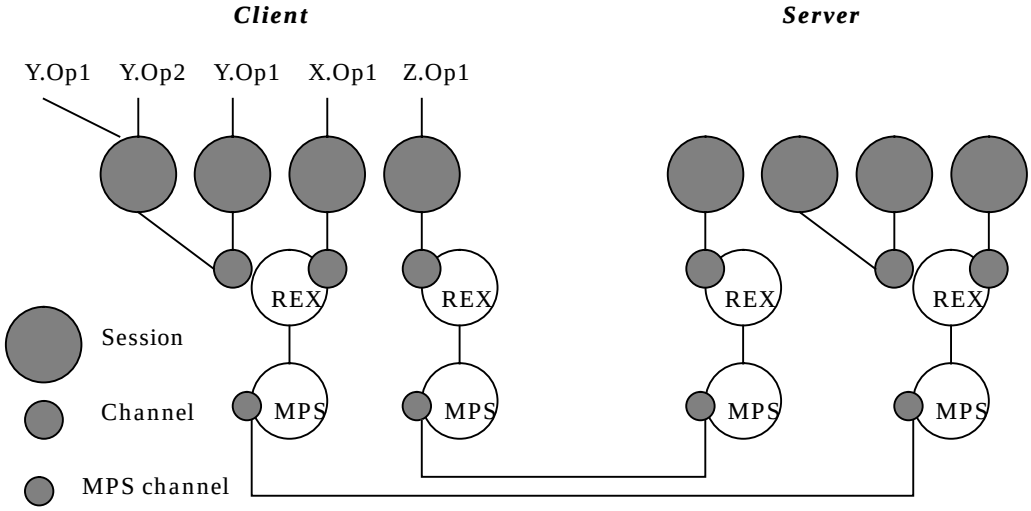
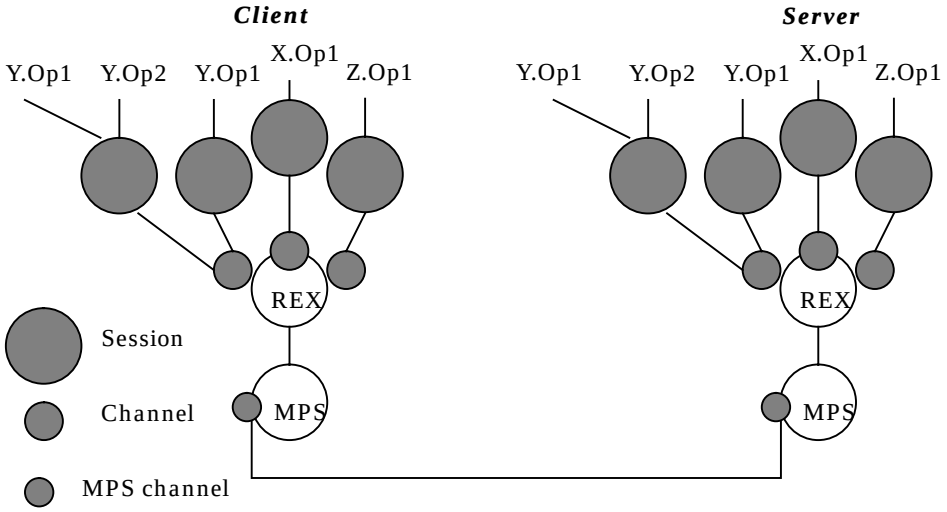


Figure 7.2: Multiplexing in ANSAware



The result is a parallel communication protocol stack, as illustrated by Figure 7.1, which is in contrast with the original ANSA multiplexing structure for a server/client interaction as illustrated in Figure 7.2.

7.3 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).

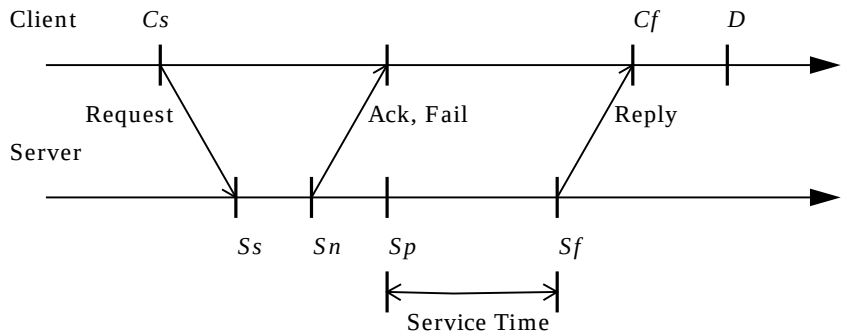
Invocations in ANSAware/RT can attach deadline constraints to their communication requests. Such TRPC calls raise the following three issues:

- 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. 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 exists no clear definition of TRPC yet when examined in a distributed setting. The interpretations applied significantly affect the implementation. The problem will be approached by first making a strictly unsatisfiable definition, and then relax 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 7.3.

Figure 7.3: 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.

7.3.1 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* [Xu93] --- 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 design 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.

7.3.2 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 time the request should be executed on the server. It affects the server side of the TRPC protocol only.

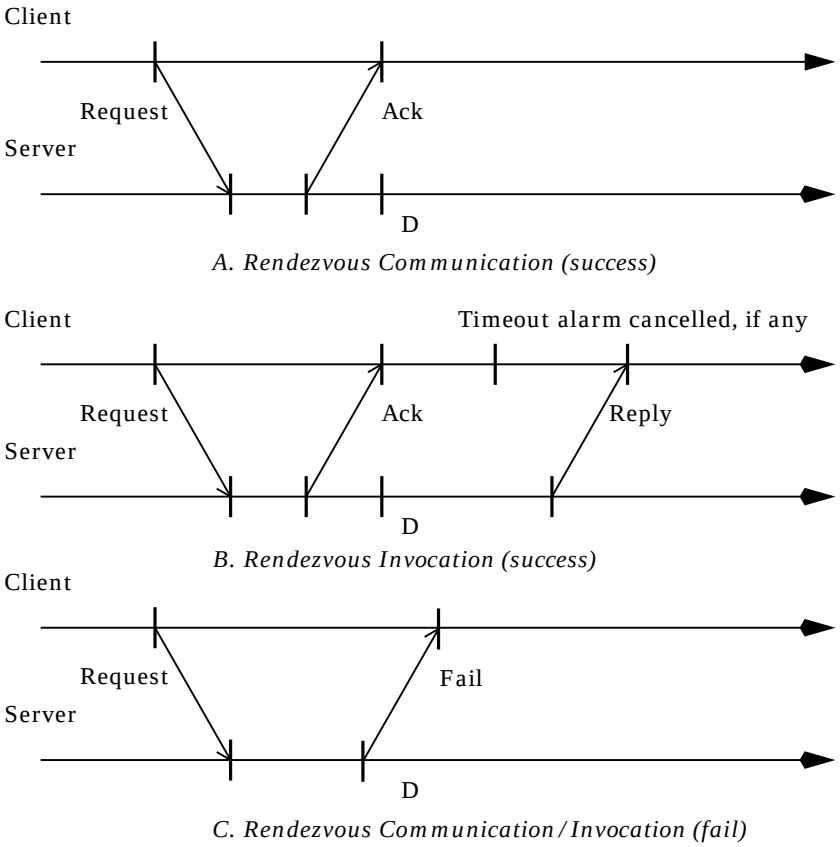
It should be pointed out that using the two separate arguments does not solve the TRPC consistency problem. Rather, the two arguments give the problem a more realistic definition, allowing different relaxations be explored.

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 the 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 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.

Figure 7.4: Rendezvous Communication/Invocation Interaction



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 7.4.

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.

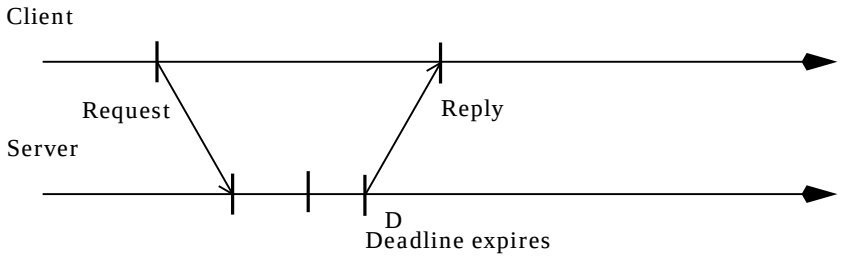
The default deadline type of an invocation deadline is *ServerDetermined* --- it depends on the scheduling policy used in the server to interpret the deadline, and has no effect on the communication protocol.

7.3.3 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 7.5: 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 7.5.

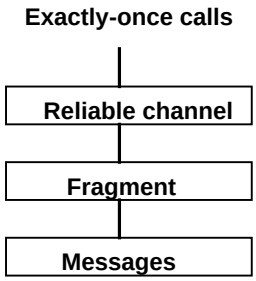
7.4 Towards a decomposable RPC protocol

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. ANSAware/RT 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 7.6. The three layers shares the same protocol data structure (sessions) and provide just one protocol service.

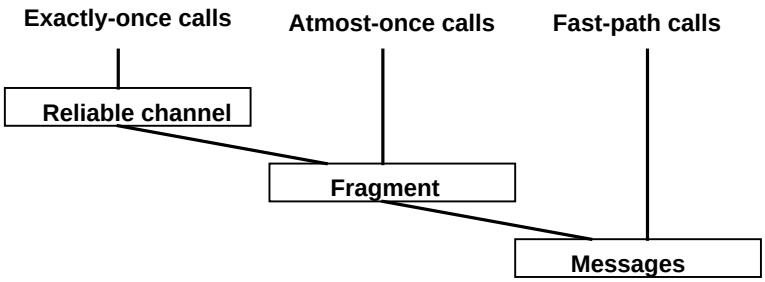
Figure 7.6: REX Functions Layers



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 three layers of REX can be reassembled to provide a multiple service interface. The transport protocol looks like Figure 7.7. In addition to the exactly-once service, two other services, the at-most-once service and fast-path services can be provided. The multi-transport service protocol is still one execution protocol in the ANSA sense. But it provides additional call semantics.

Figure 7.7: 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 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.

7.5 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.

ANSAware/RT Implementation

8 ANSAware/RT Implementation Overview

8.1 Goals

The main goal is to implement an *extended* ANSAware 4.1 that runs over a standard real-time environment and *retains* the real-time properties of the environment. Specifically, the goal can be divided into the following sub-goals:

- compatible with ANSAware 4.1
- running over a de-facto industry standard: real-time POSIX threads
- full p-thread real-time scheduling and threading capabilities
- selective communication multiplex by QoS specification and explicit binding operations
- application controlled resource allocation
- supporting the real-time programming model given in chapter 6 and 7
- comparable to other distributed real-time system environments
- interoperation between different real-time and non-real-time platforms

8.2 Scope of extensions

ANSAware/RT extends ANSAware 4.1 in the following aspects in terms of functional requirements:

- extended tasking system
  - — entry: this is a new abstraction which represents a scheduling point. Functions related to entry including creation, release, allocation of tasks and association of interfaces are provided
  - — real-time scheduling: preemptive priority-based scheduling
  - — multiple scheduling policies: the co-existing of real-time and non-real-time scheduling supports
  - — real-time tasks: full real-time p-thread functions
  - — stacked thread system: a thread may use its task resource to execute another thread
  - — real-time threads: a thread may be associated with a priority and/or deadline
  - — multiple thread scheduling policies based on policy/mechanism separation
- extended communication system
  - — multiple execution protocols
  - — timed execution protocol: this is a new RPC protocol which understands priority, deadline and deadline types

  - state-full execution protocols and message passing protocols: this allows the preallocation of separate communication endpoints for different interfaces
  - selective multiplex of communication channels
- extended application programming interface
  - abstractions for access tasking resources
  - QoS objects, two kinds of QoS objects are introduced: one for the description of a communication end point, another for the description of in-band QoS requirement for an invocation.
  - explicit binding operations: to bind an interface with a communication channel of a specific QoS
  - invocations with QoS: the attachment of in-band QoS based on invocations.
- extended PREPC and IDL

8.3 Outline of the implementation manual

The implementation section of this manual is organized as follows:

- chapter 9 outlines the ANSAware/RT programming interface as extensions to ANSAware 4.1
- chapter 10 discusses the extended tasking and scheduling implementation
- chapter 11 presents the implementation of the extended communication system
- chapter 12 shows the results of system performance tests
- chapter 13 presents the manual pages for the functions which are not part of ANSAware 4.1.

9 Extensions to the Application Programming Interface

This chapter first briefly describes current the ANSAware 4.1 programming interface, and then proceeds to discuss the ANSAware/RT programming interface extensions.

9.1 ANSAware 4.1 programming interface

ANSAware 4.1 provides the following major functions for programming object, concurrency and resource management:

- interface instances, multiple instances of an interface type can be created dynamically
- thread spawn, threads can be created by either a fork or a spawn function
- task creation, tasks can be added dynamically via invoking the `nucleus_tasks` function
- object invocation, PREPC provides a generic statement for object invocations.

The above functions are extended by ANSAware/RT. Other functions such as factory, trading, relocation, notification, exception handing etc. are retained in ANSAware/RT, and therefore not mentioned further (see ANSAware 4.1 documents).

9.2 Environment variable

Real-time applications should set the SCHEDPOLICY environment variable as one of the p-thread supported real-time scheduling policies, which can be SCHED_FIFO or SCHED_RR. For non-real-time applications and where throughput is a main criterion, SCHEDPOLICY can be set as either SCHED_FG_NP or SCHED_BG_NP. Otherwise ANSAware/RT will choose the default time-sharing scheduling policy SCHED_OTHER for its task scheduling.

9.3 Tasking and scheduling

The ability to control scheduling is an important requirement for real-time application designers. Real-time applications must be able to control scheduling in order to service external events (which can be an invocation, for example) in a timely and predictable manner. Control over scheduling takes several forms in ANSAware/RT:

- select how the operating system scheduler selects tasks (which are mapped to a p-thread each) to run --- select a task scheduling policy

- choose the priority of each task
- control the allocation of entry or scheduling point --- select the basis where separate scheduling concerns are identified
- select how ANSA threads are queued on an entry for execution --- select a thread queuing policy for an entry
- control the allocation of tasks for each entry
- select how a thread's scheduling attributes may affect a task's scheduling attributes --- select a rendezvous policy of an entry
- control which entry a service interface will be bound to.

The above seven forms of real-time functions allow for a great amount control over application execution. At run time, the combination of these real-time features gives the user control over system CPU resources.

For a p-thread system, only priority-based scheduling is supported, which implies the same for ANSAware/RT task scheduling. Task scheduling policies work in conjunction with priority levels. A global priority range applies to all task scheduling policies, but each policy has an associated priority range. Tasks are allowed to change both scheduling policies and priorities depending on application needs.

9.3.1 Attributes objects

An attributes object is used to describe a ANSAware/RT task, thread or entry. This description consists of the individual attribute values that are used to create a task, thread or entry. An attributes object is analogous to a type definition in a programming language; it describes details of the object to be created.

To create a task attributes object, the following function can be used

```
ansa_Status ansa_taskattr_create (ansa_TaskAttr *attr)
```

This routine creates a task attribute object containing default values for individual attributes. The following attributes can be changed:

- scheduling inheritance
- scheduling policy
- scheduling priority
- stack size

To modify any attribute values in a task attributes object, use

```
ansa_Status ansa_taskattr_setinheritsched (ansa_TaskAttr *attr,
                                           ansa_Integer inherit)
ansa_Status ansa_taskattr_setsched (ansa_TaskAttr *attr,
                                    ansa_Integer scheduler)
ansa_Status ansa_taskattr_setprio (ansa_TaskAttr *attr,
                                   ansa_Integer prio)
ansa_Status ansa_taskattr_setstacksize (ansa_TaskAttr *attr,
                                        ansa_Integer stacklen)
```

To obtain an attribute value in a task attribute object, use

```
ansa_Integer ansa_taskattr_getinheritsched(ansa_TaskAttr attr)
ansa_Integer ansa_taskattr_getsched (ansa_TaskAttr attr)
ansa_Integer ansa_taskattr_getprio (ansa_TaskAttr attr)
ansa_Integer ansa_taskattr_getstacksize (ansa_TaskAttr attr)
```

To delete a task attribute, use

```
ansa_Status ansa_taskattr_delete (ansa_TaskAttr *attr)
```

Similar routines exist to control the creation, manipulation and deletion of entry attributes objects.

The following entry attributes can be changed:

- thread queuing policy
- task/thread rendezvous policy
- priority ceiling value
- priority range values

Routines related to entry attributes are:

```
ansa_Status ansa_entryattr_create (ansa_EntryAttr *attr)
ansa_Status ansa_entryattr_delete (ansa_EntryAttr *attr)
ansa_Status ansa_entryattr_setqueuing (ansa_EntryAttr *attr,
                                       ansa_Integer queuing)
ansa_Status ansa_entryattr_setrendezvous( ansa_EntryAttr *attr,
                                         ansa_Integer rendezvous)
ansa_Status ansa_entryattr_setceiling (ansa_EntryAttr *attr,
                                       ansa_Integer ceiling)
ansa_Status ansa_entryattr_setprio_min (ansa_EntryAttr *attr,
                                        ansa_Integer prio)
ansa_Status ansa_entryattr_setprio_max (ansa_EntryAttr *attr,
                                        ansa_Integer prio)
ansa_Integer ansa_entryattr_getrendezvous (ansa_EntryAttr attr)
ansa_Integer ansa_entryattr_getceiling (ansa_EntryAttr attr)
ansa_Integer ansa_entryattr_getprio_min (ansa_EntryAttr attr)
ansa_Integer ansa_entryattr_getprio_max(ansa_EntryAttr attr)
```

Routines related to thread attributes are:

```
ansa_Status ansa_threadattr_create (ansa_ThreadAttr *attr)
ansa_Status ansa_threadattr_delete (ansa_ThreadAttr *attr)
ansa_Status ansa_threadattr_setprio (ansa_ThreadAttr *attr,
                                     ansa_Cardinal   prio)
ansa_Status ansa_threadattr_setdeadline (ansa_ThreadAttr *attr,
                                         ansa_Cardinal   deadline)
ansa_Integer ansa_threadattr_getprio (ansa_ThreadAttr attr)
ansa_Integer ansa_threadattr_getdeadline (ansa_ThreadAttr attr)
```

9.3.2 Entry

An entry is a scheduling point by which different scheduling/processing concerns can be identified. Each capsule has a default entry by which all new created interfaces of the capsule are bound. Binding an interface with an entry results in all invocations on the interface being queued on the entry and later processed by the tasks allocated to the entry.

Additional entries can be created by

```
ansa_Status ansa_entry_create (ansa_Entry *entry,
                               ansa_EntryAttr attr)
```

An interface can bind to another entry by

```
ansa_Status ansa_entry_bind (ansa_InterfaceRef *ref,
                             ansa_Entry *entry)
```

An interface can resume to the default capsule entry by

```
ansa_Status ansa_entry_unbind (ansa_interfaceRef *ref)
```

An entry may be closed by

```
ansa_Status ansa_entry_close (ansa_Entry *entry)
```

9.3.3 Task

ANSAware/RT tasks can be spawned for two purposes:

- to process the requests on an entry
- to initiate a separate execution thread

In the first case, tasks can be allocated by

```
ansa_Status nucleus_tasks_onentry (ansa_Entry *entry,
                                   ansa_TaskAttr attr,
                                   ansa_Cardinal extratasks)
```

In the second case, a task (with its own ANSA thread) can be created by

```
ansa_Status ansa_task_spawn ( void (*proc)(),
                              long  arg,
                              ansa_TaskAttr attr)
```

9.3.4 Task scheduling policies

Task scheduling policies are introduced to give flexibility and control in determining how work is performed so that an application can balance the work with the behaviour of a capsule.

Essentially, there are two categories of tasks:

- time-sharing processing: used for interactive or background work with no critical time limits but a need for reasonable response time and high throughput.
- real-time processing: used for critical work that must be completed within a certain time period.

To control the scheduling policies for the two categories of work, appropriate policy parameters must be selected when creating a task. ANSAware/RT supports three scheduling policies:

- AW_SCHED_OTHER: time-sharing scheduling
- AW_SCHED_FIFO: fixed-priority, first-in first-out preemptive scheduling
- AW_SCHED_RR: fixed-priority, round-robin preemptive scheduling.

They are mapped to the equivalent p-thread scheduling policy SCHED_OTHER, SCHED_FIFO and SCHED_RR respectively.

Detailed discussion of the relations between scheduling policies, priority ranges and operating system processes can be found in a p-thread manual.

9.3.5 Thread and entry scheduling policies

ANSA thread (or thread) can be generated in two cases:

- by an explicit spawn operation

```
ansa_Status instruct_Spawn_onentry (ansa_Dispatch dispatch,
                                    ansa_Entry    *entry,
                                    ansa_BufferLink buffer,
                                    ansa_ThreadAttr attr)
```

- for each invocation, ANSAware/RT generates a representing thread on the entry to which the called interface is bound. This is a implicit operation done by the infrastructure.

Threads are queued on entries. Each entry has two scheduling attributes which can be controlled by an application:

- thread queuing policy
- task/thread rendezvous policy

Chapter 6 defines five thread queuing policies:

- first-come first-service: this is the default thread queuing policy
- fixed priority based
- earliest deadline based
- priority first and then deadline based
- deadline first and then priority based

They are supported by ANSAware/RT and can be chosen by an entry attribute value of AW_E_FCFS, AW_E_PRI, AW_E_DEADLINE, AW_E_PRI_PLUS or AW_E_DEADLINE_LIUS.

Chapter 6 defines five task/thread rendezvous policies:

- null: the default policy
- priority inheritance: the task inherits the priority of its serving thread
- priority ceiling: the task inheritance the priority ceiling value of the entry
- transitive priority inheritance
- deadline inheritance
- priority and deadline inheritance.

ANSAware/RT supports the first three rendezvous policies which can be selected by AW_R_NULL, AW_R_PRI, AW_R_CEILING.

9.4 QoS objects

A QoS object consists of individual attribute values and is introduced to describe communication resource and performance constraints. Two categories of QoS objects are defined in ANSAware/RT:

- endpoint QoS object: to describe QoS constraints associated with a communication endpoint which could be either a socket (a server endpoint) or a plug (a client endpoint). These QoS objects are used by

binding operations to set up a communication channel between a client and its server.

- in-band QoS object: to select the in-band QoS parameters of an established communication channel. These QoS objects are associated with individual invocations.

ANSAware has two layers of communication protocols: the execution (EX) protocol layer and the message passing service (MPS) protocol layer. An endpoint QoS object has attributes to select:

- an EX protocol: this can be either the standard ANSAware REX protocol or TREX protocol defined in chapter 7
- a MPS protocol: ANSAware/RT supports IPC or UDP
- control parameters specific to an individual protocol. For example for UDP, an application can choose a specific port number, and spawn a specific p-thread for managing the communication on the port.

An in-band QoS object can be associated with an invocation to select the channel related control parameters:

- for REX protocol, these can be timeout value and error retry number
- for TREX protocol, these can be priority value, timeout, deadline type and deadline value.

In future implementations, it is expected that the in-band QoS object may allow the selection of the in-band QoS parameters associated with a real-time MPS service.

To create an endpoint QoS object and set up the default values of its parameters, use:

```
ansa_Status ansa_endQoS_create (ansa_EndQoS *qos)
```

To setup the individual parameter value, use:

```
ansa_Status ansa_endQoS_setAnAttribute (ansa_EndQoS *qos,
                                        a_Type      a_Value)
```

Similar routines exist for the in-band QoS object:

```
ansa_Status ansa_invQoS_create (ansa_InvQoS *qos)
ansa_Status ansa_invQoS_setAnAttribute (ansa_InvQoS *qos,
                                        a_Type      a_Value)
```

9.5 Binding

ANSAware/RT supports explicit binding operations to

- associate QoS with communication endpoints
- control binding time

Server site explicit binding is accomplished at service creation time, use

```
{ir} :: Type$Create(concurrency) {QoS}
```

This operation creates a service instance and sets up the service communication endpoint with the required QoS constraint. This binding

operation is combined with service instance creation because the QoS constraint may affect the formation of the interface reference `ir`.

Without the QoS parameter, the creation operation will use the default implicit binding operation, which means the capsule only ensures a minimum communication QoS (use multiplex as much as possible, for example) for the service instance.

Client site explicit binding is accomplished when the client gets hold of a server interface reference `ir`, and use

```
{} :: ir$Bind() {QoS}
```

This operation will create a client communication endpoint, a plug, with the required QoS. The client can then use the `ir` to invoke the server as is the case in ANSAware.

It is worth point out that the TREX protocol requires explicit binding operation be initiated, before any further interaction can take place. In other words, a real-time invocation cannot be initiated before a real-time communication channel is explicitly set up.

The server site explicit binding is destroyed and therefore related resources released when the server makes an explicit call

```
{} :: Type$Destroy(ir)
```

The corresponding client site operation is

```
ir$Discard
```

9.6 Invocations

Each invocation can be associated with an optional in-band QoS object, which may be used to control the semantics of communication. The invocation syntax is

```
{results} <- ir$operation(arguments) signals {QoS}
```

This allows the association of priority, deadline etc. invocation dependent control parameters.

9.7 Rendezvous

ANSAware/RT also extends ANSAware 4.1 tasking system to allow stackable execution of threads. This permits a thread, while execution (held a task), to wait at an entry to rendezvous and execute another thread (this can be an invocation). The benefit is that it allows an application to schedule thread execution based on its runtime knowledge.

The rendezvous function is

```
ansa_Status ansa_rendezvous (ansa_Entry *entry,
                             ansa_Cardinal timeout)
```

9.8 Clock

The deadline associated with an invocation implies both the server and the client share the common view of time. As there are many clock synchronization mechanisms and systems, it would be inappropriate to build the clock synchronization mechanism within the capsule. ANSAware/RT provides only minimum functions for clock reset and relying on an application to provide the appropriate clock synchronization service at application level (as a normal ANSA service, for example).

The nucleus provides two functions:

```
void system_readTime (ansaTime *time)
void system_resetTime (ansaTime time)
```

The first function reads the current clock value of the capsule and the second resets the clock according to a given parameter.

10 Tasking and Scheduling Implementation

10.1 ANSAware tasking

ANSAware threads represent points of execution and provide the notion of logical concurrency. ANSAware tasks represent the resources required (stacks) to execute an ANSAware thread and provide the actual concurrency.

Logically, ANSAware starts with several threads and one or more tasks. There is a receiver thread for receiving messages on the communication endpoints, a time thread to execute time-related activities, and an application program thread to execute the user program code.

ANSAware tasks are user level entities implemented through a coroutine package. Additional tasks may be created to provide extra physical concurrency. Tasks are shared by all threads. All threads waiting to execute are queued on one FCFS queue (named entry in ANSAware/RT). The ANSAware nucleus scheduler assigns free tasks to execute queued threads. The scheduler is non-preemptive and is only entered when the current thread/ task blocks or terminates. If a thread/task is resumed, the scheduler will return control to it.

ANSAware takes several advantages of the coroutine nature of its tasking package:

- use capsule-wide global, continuous and extensible memory area to store the shared data structures holding the capsule state. ANSAware increases memory for shared data structures in a dynamic manner but requires that the existing memory and the newly allocated memory be contiguous. This requirement has been achieved by copying the existing data to a new location where contiguous memory is available.In this way, memory space is allocated on demand, resulting downsize of ANSAware processes
- use capsule-wide global variables to carry context information. The variables that form the context of an ANSA task are global variables which are shared by all ANSA tasks. Thus, context information is passed to all the procedures through global variables. This allows fast inter task context switch and fast procedure execution (i.e. there is no need to pass context information through procedure parameters).
- shared data area is accessible without a synchronization mechanism. ANSAware task scheduler is non-preemptive, a task, while execution, will not relinquish control until it blocks or terminates. Therefore, it guarantees exclusive access of the shared data in a single processor environment, and there is no need for access protection of shared data.

10.2 ANSAware/RT tasking

In ANSAware/RT, each task is mapped into a p-thread, and task scheduling is done by the underlying operating system. All p-thread attributes also apply to

tasks, allowing the exploitation of preemptive real-time scheduling, multiple scheduling policies, kernel supported synchronization objects, task private data, task exception handling, task synchronous I/O etc. p-thread features. The original ANSAware task schedule was made redundant.

10.2.1 Global data protection

Because of the real concurrency[1] and preemptive nature of the p-thread system, synchronisation is needed to ensure safe access to shared data. A pessimistic synchronisation approach is taken: all data structures are protected by a single lock. To perform any ANSAware/RT operation, a task must first acquire the lock, then operate on the shared data, and finally release the lock when finished.

10.2.2 Thread private state

Each thread has a few private state variables, such as `exception_code`, `exception_state`, `memory_list` etc. These thread private state variables are stored in the global data area in ANSAware. Thread private state is used frequently in both application program and ANSAware operations.

In ANSAware/RT, global data area needs to be protected by a synchronization mechanism. Therefore, the ANSAware thread private data may introduce a significant performance overhead if no change to ANSAware is made.

The solution adopted is to use p-thread per-thread state to store ANSAware thread private state (rather than using the global area). Such state information is then accessible by using the `pthread_getspecific` procedure without a synchronization operation.

The thread private state are actually part of a task private data area. When a task is crated, it allocates a private state area as p-thread per-thread data, and part of this area is used as thread private state when the task is executing a thread.

10.3 Stacked threads

ANSAware threads are non-stackable: a task will not execute another thread before it finishes the current one.

ANSAware/RT introduces a dynamic rendezvous mechanism: a thread may rendezvous with another thread while itself is executing. This stackable thread mechanism is implemented by pushing the thread private state area into the task's stack before executing the new thread (so that the new thread can still use the same task private data as its thread private state), and restore the thread private state from stack when the new thread finishes.

10.4 Thread

Threads are created in two cases: (1) an application may create new threads for additional concurrency; (2) a communication task may create one additional thread for each RPC request from a client. In ANSAware, a new

1. p-threads can be executed in parallel, for example, in a multiprocessor environment.

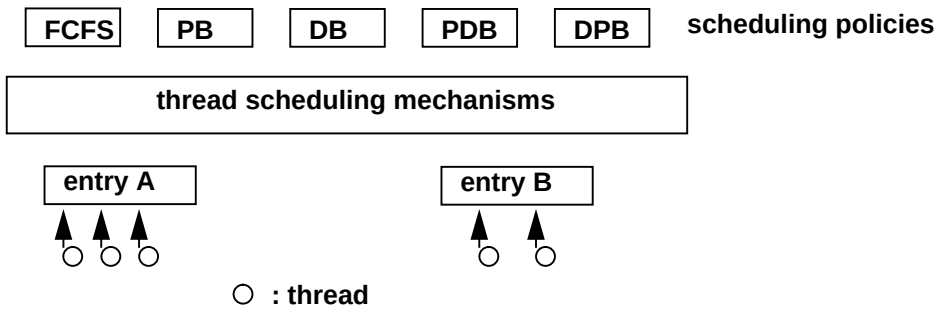
thread is queued on the capsule-wide FCFS thread queue, waiting to be executed by a free ANSA system task.

In ANSAware/RT, a new thread is queued on an entry instead of the capsule FCFS queue.In case (1), the application gives an additional entry argument when a new thread is created. In case (2), the binding between an interface and an entry determines on which entry the new thread should be queued.

10.5 Entry

Each entry is associated with a thread queue and a thread queuing policy.

Figure 10.1: Thread scheduling: policies and mechanisms



Policy/mechanism separation is used for efficient coding. A common set of thread queuing/dequeuing mechanisms is provided, and on top of the mechanisms a set of scheduling policy objects are placed. Figure 10.1 illustrates such a design.

Each entry is also associated with a rendezvous policy. Each such policy provides two functions: `rendezvous_inheritance` and `rendezvous_deinheritance`. The `rendezvous_inheritance` function is executed before a task executes a thread so that the task can take the thread scheduling parameters into consideration. For example, it allows the task to inherit the thread's priority. The `rendezvous_deinheritance` function is executed after a task finishes the execution of a thread to eliminate any scheduling effect on the task caused by the `rendezvous_inheritance` function.

10.6 Synchronous I/O

ANSAware assumes a totally asynchronous I/O model because

- it allows the tight combination of communication scheduler and task scheduler for efficient ANSAware activity scheduling
- it prevents a capsule from blocking because of an otherwise synchronous I/O operation.

The asynchronous I/O approach separates out the indication that data is available from the actual reading of the data.

The asynchronous I/O model in ANSAware is supported by

- a pin(3) programming interface. An application can register an interrupt handler to be invoked when input occurs on a pin and that handler is then able to spawn a thread to read any input data
- a non-blocking keyboard input library
- a library for supporting X11 applications.

With p-thread implementation of the tasking system, the asynchronous I/O model is no longer necessary because

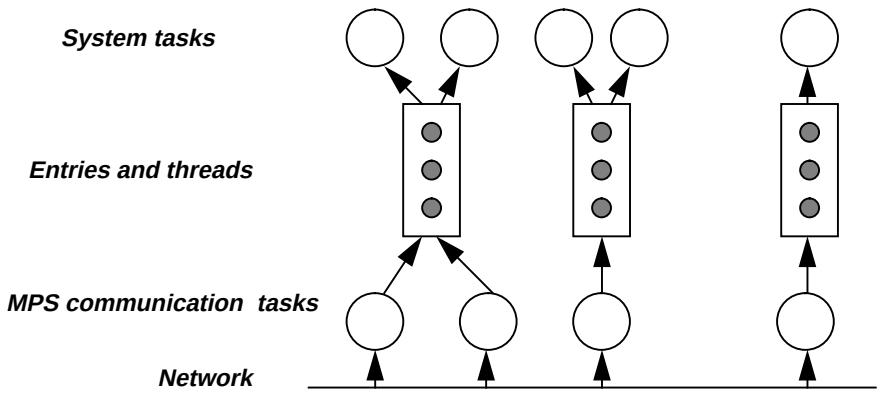
- task scheduling is done by OS, there is no tight integration of tasking scheduling and communication scheduling
- a capsule will not block when a thread is doing a synchronous I/O

In other words, ANSAware/RT does not need to assume the asynchronous I/O model, and a complete synchronous I/O model is more natural and easy to program. Therefore, ANSAware/RT removes the pin interface, the non-blocking keyboard input library and the X11 library, and assumes the application programmer will access the equivalent synchronous I/O operations supported by the p-thread package (see also section 1.4.1).

10.7 Communication tasks and system tasks

Dedicated communication tasks are spawned to process incoming messages and the corresponding protocol by using synchronous I/O operations. For each MPS endpoint (a socket), a task is spawned to handle messages from the endpoint. The communication task generates a thread corresponding to each invocation request. The thread is queued on an entry to which the called interface is bound. In this vein, the communication task is actually both a thread generator and a thread scheduler. The threads are executed by system tasks of an entry which are allocated by an application or by the capsule. The scenario is shown in Figure 10.2.

Figure 10.2: Communication tasks and system tasks



When a thread makes a synchronous invocation to a server, it blocks at a condition variable which is defined on a task's private data area. When a reply comes back and is processed by a communication task, the condition variable is signalled and therefore the calling thread is woken up.

10.8 Others

A timer task is spawned to process timing functions of the ANSAware timer module.

A signal task is spawned to process asynchronous interrupts (such as control-C).

The organization and user interface of the timer task and signal task is similar to the one provided by [APM.TR.37].

11 Implementation of the Communication System

This chapter describes the modification and extensions to the ANSAware 4.1 communication system.

11.1 ANSAware communication system

ANSAware communication system implements four protocol layers:

- MPS: an interface to the transport protocols provided by the underlying operating system
- EX: implement the invocation of ANSA operations. ANSAware 4.1 supports the REX for point to point invocations and GEX for group invocations
- Channels/sessions: used to store the end-to-end state required for a remote invocation and to synchronise the execution of the tasking and the communication systems
- Stubs: marshal host language level variables into (and out of) linear communications buffers.

ANSAware communication design was optimised for efficient resource utilization by multiplexing the channels provided by each of these between those of the next layer.

Real-time communication is not a concern of ANSAware.

11.1.1 Interface reference

An interface reference (*ifref*) contains sufficient information to allow the holder (client) to establish communication with the interface denoted (server). An interface reference has a set of address records, each of which in turn contains a channel id and the network address of the underlying MPS.

11.1.2 MPS

The MPS interface is stateless and defines an unreliable datagram service. There is no mechanism for QoS based selection/setup of a MPS module. High-level protocols multiplex MPS endpoint whenever possible (on a capsule wide basis).

11.1.3 EX

The EX interface is also stateless. There is no mechanism for QoS based selection/setup of an EX module.

REX is designed for asynchronous communication optimized for either low-latency or high throughput. REX provides a rate-based transportation service. The execution reliability semantics of REX calls are exactly-once in the absence of total communication failure.

11.1.4 Channel and session

Channels (i.e. sockets and plugs) are used to store static communication information; sessions duplicate channel state and store additional dynamic information for each invocation.

There is an one-to-one correspondence between channels and *ifref*s. Clients send/receive invocation requests/replies through plugs. Servers receive/ transmit invocation requests/replies over sockets. The channel id provides an extra layer of demultiplexing on top of the MPS address, so that the capsule can locate the right dispatcher for a specific interface.

There is no interaction between the channel and MPS modules when channels are created and destroyed; the two are independent of each other.

11.1.5 Bindings

A service provider fabricates an *ifref* upon an interface creation operation. A client holding the *ifref* must then bind to the service in order to communicate with it.

The *ifref* is created by an implicit binding operation at the server side. The binding operation allocates a socket and concatenates it with the default communication addresses (address hint) supported by all MPSes in the server to form the interface reference.

Client side binding (the creation of a plug) is performed the first invocation of a service; the first invocation is detected by the absence of the *ifref* from the *ifref* to plug cache. Removing an *ifref* from the cache will force a rebound.

The cache provides a mapping from an *ifref* to a plug. The bind operation which allocates a new plug also adds the plug to the cache.

At no stage is there any interaction between the binding process and the EX and MPS modules; it is assumed that all communications between channels is multiplexed over a single MPS address within a capsule.

11.2 QoS and explicit binding

QoS objects are introduced to express specific communication requirement and are used by explicit binding operations to create specific communication channels with appropriate characteristics.

When creating a service instance, a QoS object can be associated with the interface creation operation. The operation uses an explicit binding operation to fabricate the resulting *ifref*. The explicit binding operation calls the corresponding explicit binding operations in each protocol layer (EX and MPS). The binding operation at each protocol layer interprets the relevant QoS attributes, sets up the protocol related binding states, and returns a result that may be used to build the *ifref*.

In comparison with the implicit binding, the *ifref* created by an explicit binding contains only these information deduced from the QoS, rather than the default information that is generated from maximum multiplexing.

Client side explicit binding is performed by an explicit binding operation, which is also associated with a QoS object. The binding operation, like the server side explicit binding operation, calls the explicit binding operations at each protocol layer with the *ifref* and the QoS object as arguments. The

explicit binding operation at each protocol layer executes a complimentary operation to the relevant QoS and the *ifref* to create the client binding. The binding operation creates a plug and adds it in the plug cache, so that later calls on the interface are guaranteed an established channel.

11.3 State-full MPS

The MPS interface is extended to be state-full: it supports a connection-oriented communication paradigm. The connection is encapsulated as binding information of a channel. Each channel is associated with a binding data structure which represents end-to-end state establishment with some known channel specific properties (deduced from a QoS object).

The MPS interface is extended with three operations: server explicit binding operation, client explicit binding operation and binding release operation. The original message *send* and *receive* operations are also extended to make use of the binding informations.

A default binding is established at MPS initialisation time to be used as the default communication channel for the implicitly bound interfaces.

11.4 State-full EX

EX is modified to use the state-full MPS, and itself is redesigned to be state-full as well. This allows the addition of extra binding information to the binding created by MPS to include EX dependent data and state. For example, the binding contains extra information about the header size of an EX protocol which can be used by MPS to fetch the correct EX-dependent packet headers.

The EX interface is extended with three operations: server explicit binding operation, client explicit binding operation and binding release operation. The original message *send* and *receive* operations are also extended to make use of the binding information.

11.5 TREX

The generic design of the binding and state-full protocols allows the insertion of new EX protocols. The time RPC (see chapter 7) is implemented as one example. It is called TREX because it is a modification of REX.

TREX is a cut-down version of REX as follows:

- no fragmentation, this has significantly reduced the size of the protocol
- at-most-once semantics, no timeout controlled retry
- no security check, no passing and checking of nonce
- small header size, the result of the above three design choices

TREX also extends REX in the following ways:

- the header of packages is expanded to include information about priority, deadline and deadline type
- extended session functions process timeout at client side and deadline expiry at server side
- extra message types for handling deadline exception and confirmation.

TREX supports only explicit binding operations, i.e. interfaces created by implicit binding operations cannot use TREX.

11.6 In-band QoS

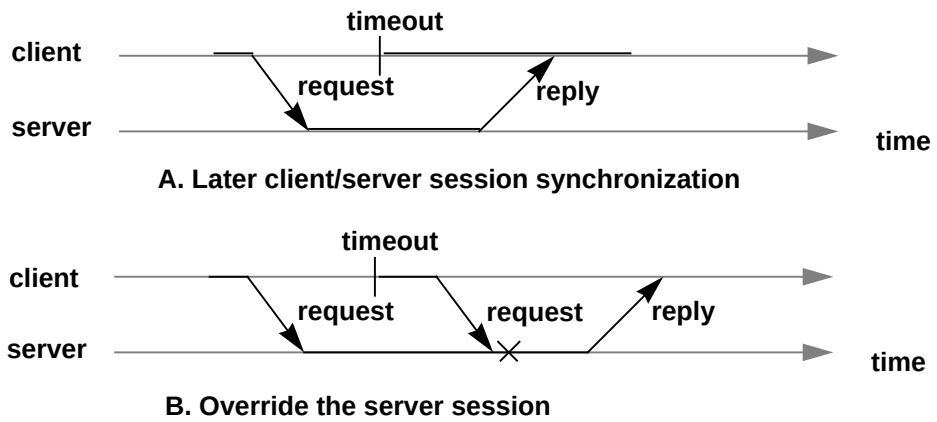
In-band QoS can be associated to each invocation to select the dynamic QoS parameters of a channel. Currently, if a channel uses TREX as its EX, the in-band QoS can select a priority, a timeout, a deadline and a deadline type.

In-band QoS are associated with sessions by the interpreter at the client side, and by the TREX at the server side.

11.7 Session overridden

Timeouts at the client sides are a mixed blessing: the desired semantics of a timeout is the client should resume control (so that the client can take some immediate recovery actions). However, the operation is still carried on at the server side and extra packet exchange is required to synchronise the client and server sessions. If the packet exchange takes place at the timeout expiry time, the extra overhead of synchronisation may lead to uncertain timeout semantics. Therefore, an alternative approach is pursued.

Figure 11.1: Session timeout recovery



The ANSAware/RT approach for session timeout recovery is illustrated in Figure 11.1. With this approach, the client continues immediately after the timeout, and the client session is set to idle. No synchronisation packet exchange is initiated by the client. It allows the existence of inconsistency between a client session and its server side session. Should the server return an obsolete result later, synchronisation of the client and server sessions are taken then. The approach also allows the server side session to be aware that its client side may time out, and the client side session may be used for another invocation. A possible effect (caused by the ANSAware approach to session management) is that a later invocation from the same client side session may override a server side session representing an obsolete invocation.

11.8 Others

REX is adjusted to make its QoS support explicit, which includes the retry number and timeout value.

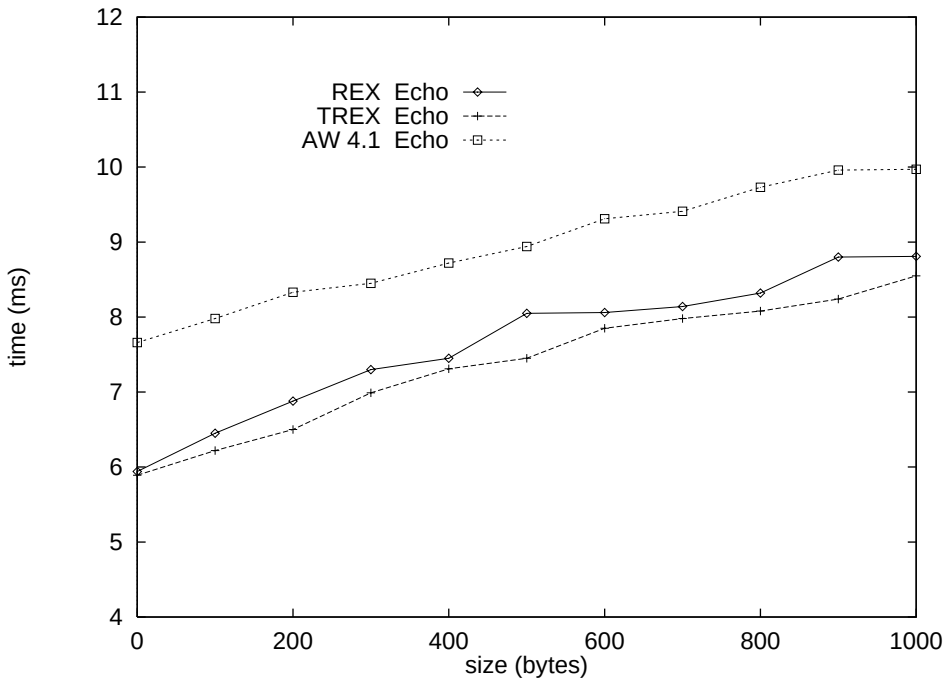
12 Performance Measurements

Two sets of performance measurements have been taken: the basic performance for simple RPC calls and the synthetic performance for the integrated effect of real-time scheduling and communication.

12.1 Basic performance

Figure 12.1 shows the performance of ANSAware/RT by using REX and TREX as two transportation protocols. For comparison, the ANSAware 4.1 performance (without using kernel p-threads) is also given in the figure. All experiments were run between lightly loaded DEC Alpha 3000/300 workstations connected by 10 Mbps Ethernet. All measurements are for an *echo* operation which sends and receives *n* bytes of data.

Figure 12.1: Basic RPC performance



The performance improvement of ANSAware/RT over ANSAware 4.1 can be explained by:

- synchronous I/O operations are more efficient than asynchronous I/O operations
- ANSAware/RT fixed a few memory management bugs of ANSAware 4.1

The performance improvement of TREX over REX can be explained by:

- TREX uses light weight mechanisms

- TREX uses shorter packet headers.

12.2 Distributed hartstone performance

There are several standard synthetic benchmarks for real-time computing systems, including Hartstone Benchmark (HB) [Weiderman89], Distributed Hartstone Benchmark (DHB) [Mercer90] and Hartstone Distributed Benchmark (HDB) [Kamenoff91]. The HB is a set of timing requirements for testing a system's ability to handle hard real-time applications. It is specified as a set of tasks with well-defined workload and timing constraints. It is a benchmark for single processor machines. The DHB and HDB are both extensions of HB for distributed real-time systems. They are designed to give figures of merit for the complex end-to-end scheduling and timing behaviour of the system. In comparison, the HDB gives a broader definition and merit of real-time distributed systems' behaviour, while the DHB has a concrete definition of the series of tests.

DHB was chosen to measure and evaluate ANSAware/RT real-time performance. The intention of DHB is to measure the real-time performance of processor scheduling, communication network scheduling and coordination between these scheduling domains. It is argued that since more sophisticated scheduling algorithms may require more overhead for low-level operations, a system which offers better schedulability for its applications and thus better overall performance may not have the best times for low-level operations. A system which is leaner and faster in terms of low-level operations may not be capable of scheduling a task set to meet all of its deadlines. DHB is thus designed to factor all of these attributes into the overall evaluation of a system.

DHB defines five sets of experiments. They are:

- DSHcl: communication latency
- DSHpq: priority queuing
- DSNpp: protocol preemptivity
- DSHcb: communication bandwidth
- DSHmc: media contention.

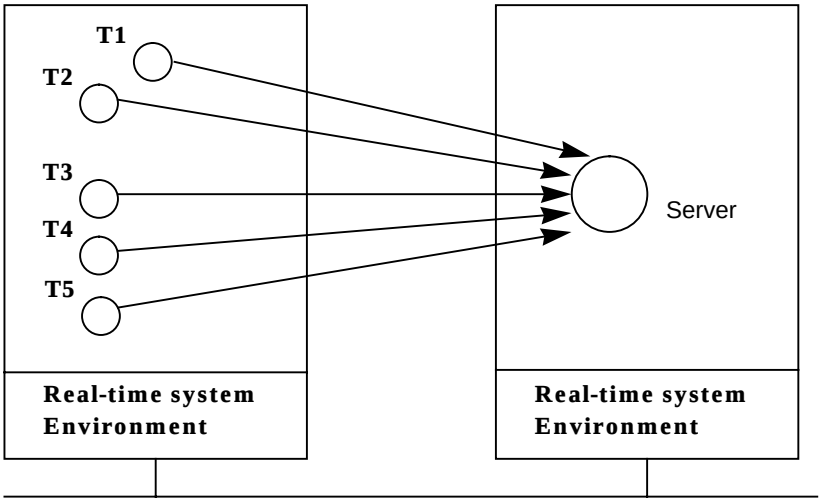
The DSHmc series is intended to stress the media contention algorithm. This series is not applicable to our environment (the Ethernet hardware has no support of prioritised packets), and therefore is not described below.

12.2.1 Communication latency

The DSHcl series is a Distributed, Synchronized, and Harmonic task set which tests the communication latency of the system. The base task set is patterned after the periodic, harmonic task set in the original Hartstone benchmark. The task set is extended to have a remote server, and each of the tasks T1,..., T5 sends a request to the server before consuming its own computation time. Figure 12.2 shows the structure of the DSHcl series task set. Table 12.1 gives the timing requirements of the DSHcl series baseline test task set.

The computation time of the server is increased in milliseconds to gradually squeeze the tasks until the size of the server combined with the time the request message is in transit causes deadlines to be missed.

Figure 12.2: Five clients with a single server



The task workload is expressed in Kilo-Whetstone (KWS). The Whetstone calculation is the self-verifying version specified by [APM92]. A KWS is one execution of a mathematical library, which factors out the effect of typical arithmetic computing. A task is required to execute a specific amount of KWS within its period.

Table 12.1: DSHcl series task set

Task	workload (KWS)	Period (ms)
T1	1	80
T2	1	160
T3	2	320
T4	2	640
T5	8	1280
T server	variable (ms)	N/A

12.2.2 Priority queuing

The DSHpq series task set is a Distributed, Synchronized, and Harmonic task set designed to test for priority queuing of communication packets. The base task set is patterned after the DSHcl series. It is quite similar to the DSHcl except for the difference in granularity. The fine-grained DSHcl uses shorter periods for the tasks and milliseconds to measure the server workload. The coarse-grained DSHpq uses longer periods for the tasks and KWS to measure server workload. Table 12.2 gives the timing requirements of the DSHpq series baseline test task set.

Table 12.2: DSHpq series task set

Task	workload (KWS)	Period (ms)
T1	1	160
T2	1	320

Table 12.2: DSHpq series task set

Task	workload (KWS)	Period (ms)
T3	2	640
T4	2	1280
T5	8	2560
T server	variable (KWS)	N/A

12.2.3 Protocol preemptivity

The DSNpp series task set is a Distributed, Synchronized, and Non-harmonic task set designed to test the degree of preemptability of the protocol engines. The base task set is patterned after the periodic, non-harmonic task set in the Hartstone benchmark. The base task set contains two remote servers; the high priority server can preempt the low priority server. The client task set is composed of one high priority (high frequency) task and a variable number of low priority (low frequency) tasks. The high priority task T1 sends a request to the high priority server at the beginning of its period. Each of the low priority tasks T2,..., Tn sends a request to the low priority server at the beginning of its period and before consuming its own computation time. The number of low frequency tasks is increased gradually until the first deadline is missed.

Figure 12.3: N clients with multiple servers

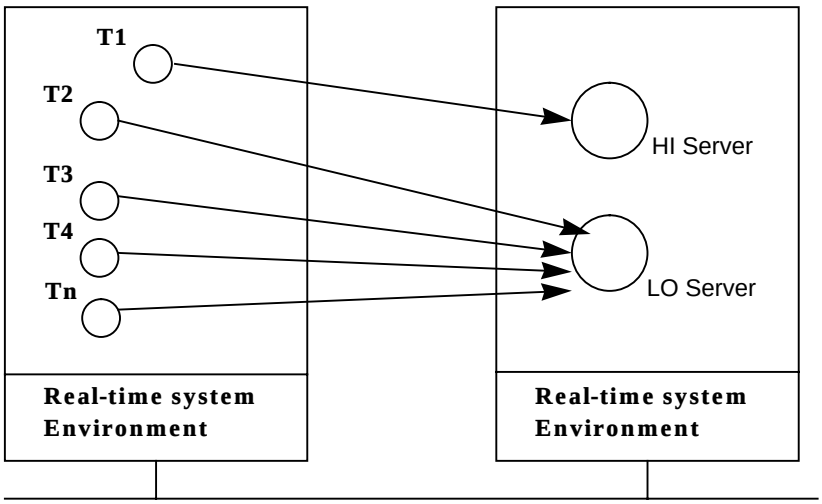


Figure 12.3 shows the structure of the DSNpp series task set. Table 12.3 gives the timing requirements of the baseline test task set.

Table 12.3: DSHpp series task set

Task	workload (KWS)	Period (ms)	Priority
T1	1	50	high
T2	1	5120	low
...	1	5120	low
Tn	1	5120	low

Table 12.3: DSHpp series task set

Task	workload (KWS)	Period (ms)	Priority
HI server	0	N/A	high
LO server	0	N/A	low

12.2.4 Communication bandwidth

The DSHcb series task set is a Distributed, Synchronized, and Harmonic task set which tests the communication bandwidth. The task set contains a remote server that consumes no computation time. Client tasks T1,..., Tn send requests to the server at the beginning of their periods and they consume no computation time. The number of high priority tasks is increased, which increases the load on the communication subsystem, until the first deadline is missed.

Table 12.4: DSHcb series task set

Task	workload (KWS)	Period (ms)	Priority
T1	0	80	high
T2	0	80	high
...	0	80	high
Tn-4	0	80	high
Tn-3	0	160	high-1
Tn-2	0	320	high-2
Tn-1	0	640	high-3
Tn	0	1280	high-4
T server	0	N/A	N/A

Figure 12.4: N clients with a single server

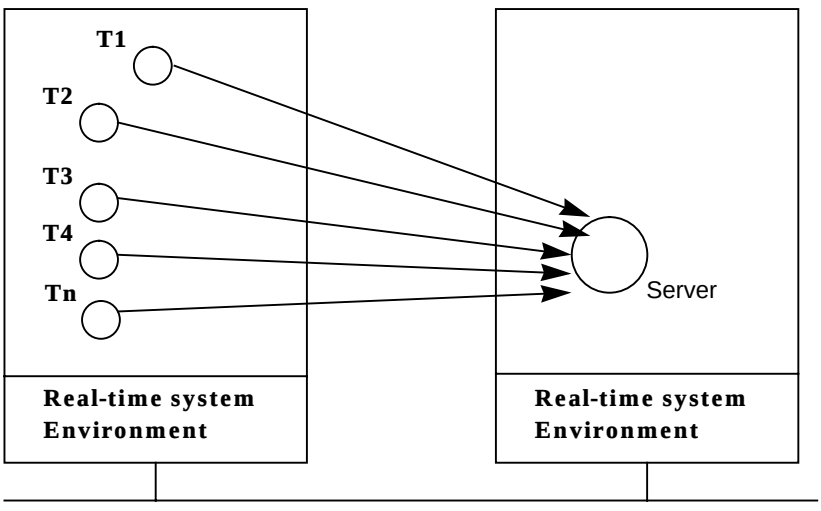


Figure 12.4 shows the structure of the DSHcb series task set. Table 12.4 gives the timing requirements of the baseline test task set.

12.2.5 Result

The benchmark results of the ANSAware/RT system over the OSF/1 kernel are presented in Table 12.5. The ANSAware/RT performance was measured using two DEC Alpha 3000/300 machines connected by the 10 Mbps Ethernet.

Table 12.5: ANSAware/RT vs ARTS and RIDE performance

Series	ARTS (Sun 3/140)	RIDE (DEC Firefly)	ANSAware/RT (DEC Alpha 3000/300)
DSHcl	35 ms	26 ms	41 ms
DSHpq	18 KWS	16 KWS	2010 KWS
DSHpp	(13) 20 tasks	18 tasks	105 tasks
DSHcb	14 tasks	15 tasks	23 tasks

To make a comparison, the relevant performance of the ARTS distributed real-time operating system [Tokuda89] and an earlier implementation of the ANSAware 3.0 based real-time system RIDE [Li93] are also given in the table.

It comes as no surprise to see that ANSAware/RT performs much better than the kernelized ARTS system, which reflects the combination effects of the superiority of a commercial real-time operating system, a much more powerful processor and a carefully tuned mechanisms based on the practical experience of RIDE.

13 Manual Pages

This chapter contains manual page entries for the programming interfaces supported by ANSAware/RT as extensions of ANSAware 4.1. The ANSAware 4.1 programming interfaces are not included.

NAME

Channel QoS object

PURPOSE

To specify a channel QoS requirement.

SYNOPSIS

```
#include “qos.h”

ansa_Status ansa_endqos_create (ansa_EndQoS *qos);

ansa_Status ansa_endqos_setudp_prio (
      ansa_EndQoS *qos,
      ansa_Integer pri);

ansa_Status ansa_endqos_setprotocol (
      ansa_EndQoS *qos,
      ansa_Integer protocol);

ansa_Status ansa_endqos_setudp_new (ansa_EndQoS *qos);
```

DESCRIPTION

The **ansa_endqos_create** function must be called before an **ansa_EndQoS** object can be used in an explicit binding operation. This function sets up the default values of the QoS requirement. The **ansa_endqos_setprotocol** function selects a particular RPC protocol, which can be **REX_IPC**, **REX_UDP**, **TREX_IPC** and **TREX_UDP**. The **ansa_endqos_setudp_new** function requires a new UDP socket for a REX_UDP or TREX_UDP protocol.

The **ansa_endqos_setudp_prio** function sets up the priority of the listening thread of the UDP socket. This in effect sets up the scheduler priority for the interfaces bound to this socket.

SEE ALSO

PREPC explicit binding operations.

NAME

Invocation QoS object

PURPOSE

To specify an invocation QoS requirement.

SYNOPSIS

```
#include “qos.h”

ansa_Status ansa_invqos_create (ansa_InvQoS *qos);

void ansa_invqos_setprio(
      ansa_InvQoS *qos,
      int prio);

void ansa_invqos_settimeout(
      ansa_InvQoS *qos,
      int val);

void ansa_invqos_setdeadline(
      ansa_InvQoS *qos,
      ansa_DeadlineType type,
      int val);
```

DESCRIPTION

The **ansa_invqos_create** function must be called before an **ansa_InvQoS** object can be used in an invocation. This function sets up the default values of the QoS requirement: a default timeout value, a priority value, and a default deadline type value (with an indefinite deadline value).

The **ansa_invqos_setprio** function sets up the priority of the invocation. The **ansa_invqos_settimeout** function sets up the timeout value (in milliseconds). The **ansa_invqos_setdeadline** function sets up the deadline type and deadline value (in milliseconds). The deadline type can be **D_CallSoft** (a soft deadline that will not raise an exception if the deadline cannot be meet, this is the default deadline type value), **D_CallHard** (a hard deadline that will raise an exception if the server cannot meet the deadline when the invocation finish), and **D_RenHard** (a hard deadline that will raise an exception if the server cannot start processing the request before the deadline, the request is not executed in this case).

The relative time value of the deadline will be automatically transferred to the global time value at the client side and back to relative time value at the server side by the **TREX** protocol.

SEE ALSO

PREPC invocation extension.
Clock setup.

NAME

PREPC explicit binding operations

PURPOSE

To bind an interface with a channel with a specific QoS.

SYNOPSIS

```
#include “qos.h”
ansa_EndQoS qos;
! {ifref} :: TypeName$Create(concurrency, args) {qos}
! {} :: TypeName$Destroy(ifref)
! {} :: TypeName$Bind(ifref) {qos}
! ifref$Discard
```

DESCRIPTION

The **TypeName$Create** statement is used to create an interface instance with a given QoS. This is the server side explicit binding operation. The created instance can be deleted, and therefore the allocated resources released, by the **TypeName$Destroy** operation.

Client side explicit binding is done by the **TypeName$Bind** operation, and the binding can be released by the **ifref$Discard** operation.

The ANSAware default implicit binding operations still work, which mean an interface instance can be created by the **TypeName$Create** operation without the qos parameter, and a client can call the interface without a prior explicit **TypeName$Bind** operation.

SEE ALSO

Channel QoS object.

ANSAware PREPC.

NAME

PREPC invocation extension

PURPOSE

To make an invocation with a specific QoS.

SYNOPSIS

```
#include “qos.h”
ansa_InvQoS qos;
! {results} <- ifref$op(arguments) exceptions {qos}
```

DESCRIPTION

The **ifref$op** statement is used to make an invocation on an interface instance with a given QoS. The channel must already have been set up by the explicit binding operations if real-time QoS is used.

The exceptions include two new cases: **clientTimeout** and **serverDeadlineExpire**, if the **TREX** protocol is used.

SEE ALSO

Invocation QoS object.

ANSAware exception handling.

ANSAware PREPC.

NAME

Task attributes object

PURPOSE

To specify real-time attributes for tasks.

SYNOPSIS

```
#include “task.h”
ansa_Status ansa_taskattr_create (ansa_TaskAttr *attr);
ansa_Status ansa_taskattr_delete(ansa_TaskAttr *attr);
ansa_Status ansa_taskattr_setprio(ansa_TaskAttr *attr,int pri);
int ansa_taskattr_getprio(ansa_TaskAttr *attr);
ansa_Status ansa_taskattr_setsched(ansa_TaskAttr *attr,int sched);
int ansa_taskattr_getsched(ansa_TaskAttr *attr);
ansa_Status ansa_taskattr_setstacksize(
      ansa_TaskAttr *attr,int size);
int ansa_taskattr_getstacksize(ansa_TaskAttr *attr);
int ansa_sched_policy();
```

DESCRIPTION

These functions are to create and set up real-time attributes for tasks. The **sched** attribute can be **AW_SCHED_FIFO**, **AW_SCHED_RR** and **AW_SCHED_OTHER** which correspond to SCHED_FIFO, SCHED_RR and SCHED_OTHER policies of the pthreads, respectively. For each policy, there is a priority range can be applied: { **AW_PRI_FIFO_MIN**, **AW_PRI_FIFO_MAX** }, { **AW_PRI_RR_MIN**, **AW_PRI_RR_MAX** } and { **AW_PRI_OTHER_MIN**, **AW_PRI_OTHER_MAX** }.

The **ansa_sched_policy** function returns the scheduling policy for system tasks, which can be set up by the environment variable **SCHEDPOLICY**.

SEE ALSO

Real-time task.

Entry.

Environment variable.

ANSAware task and thread.

NAME

Entry attributes object

PURPOSE

To specify real-time attributes for entry.

SYNOPSIS

```
#include “entry.h”

ansa_Status ansa_entryattr_create (ansa_EntryAttr *attr);

ansa_Status ansa_entryattr_delete (ansa_EntryAttr *attr);

ansa_Status ansa_entryattr_setqueuing (
      ansa_EntryAttr *attr, int policy);

int ansa_entryattr_getqueuing(ansa_EntryAttr attr)

ansa_Status ansa_entryattr_setrendezvous (
      ansa_EntryAttr *attr, int policy));

int ansa_entryattr_getrendezvous(ansa_EntryAttr attr)

ansa_Status ansa_entryattr_setprio_min (
      ansa_EntryAttr *attr, int prio);

int entry_attr_getpri_min(ansa_EntryAttr attr)

ansa_Status ansa_entryattr_setprio_max (
      ansa_EntryAttr *attr, int prio);

int ansa_entryattr_getprio_max(ansa_EntryAttr attr)

ansa_Status ansa_entryattr_setprio_ceiling (
      ansa_EntryAttr *attr, int prio));

int ansa_entryattr_getprio_ceiling(ansa_EntryAttr attr)
```

DESCRIPTION

These functions allow the creation of entry attributes object and set up of individual attribute values. The **ansa_entryattr_setqueuing** function sets up the thread enqueue policy, which can be **AW_E_NULL**, **AW_E_PRIORITY**, and **AW_E_DEADLINE**. The **ansa_entryattr_setrendezvous** function sets up the task/thread rendezvous policy, which can be **AW_R_NULL**, **AW_R_PRIORITY**, and **AW_R_CEILING**.

SEE ALSO

Entry.

Real-time task.

Real-time thread.

NAME

Thread attributes object

PURPOSE

To specify real-time attributes for threads.

SYNOPSIS

```
#include “thread.h”

ansa_Status ansa_threadattr_create (ansa_ThreadAttr *attr);

ansa_Status ansa_threadattr_setprio(
      ansa_ThreadAttr *attr,int pri);

int ansa_threadattr_getprio(ansa_ThreadAttr attr);

ansa_Status ansa_threadattr_setdeadline(
      ansa_ThreadAttr *attr,int deadline);

int ansa_threadattr_getdeadline(ansa_ThreadAttr attr);
```

DESCRIPTION

These functions are to create and set up real-time attributes for threads. The deadline value is in milliseconds.

SEE ALSO

Real-time task.

Real-time thread.

Entry.

ANSAware task and thread.

NAME

Real-time task

PURPOSE

To create tasks for an entry.

SYNOPSIS

```
#include “task.h”
#include “entry.h”

ansa_Status task_Make(
     ansa_Entry *entry,
     ansa_TaskAttr attr,
     ansa_Cardinal extraTasks);
```

DESCRIPTION

The **task_Make** function creates extraTasks number of tasks on the entry.

SEE ALSO

Task attributes object.

Entry.

NAME

Real-time thread

PURPOSE

To create real-time threads for an entry.

SYNOPSIS

```
#include “thread.h”
#include “entry.h”


ansa_ThreadId thread_dispatch (
      ansa_ThreadId parent,
      ansa_Dispatch dispatcher,
      ansa_ChannelId socket,
      ansa_BufferLink buffer,
      void (*epilogue)(),
      ansa_Entry *entry,
      ansa_ThreadAttr attr);
```

DESCRIPTION

The **thread_dispatch** function creates a thread on a given entry if the socket value is zero, otherwise the thread will be spawned on the entry bounded by the socket.

SEE ALSO

Thread attributes object.

Entry.

NAME

Entry

PURPOSE

To create a scheduling point, i.e. entry.

To bind an interface to an entry.

SYNOPSIS

```
#include “entry.h”

ansa_Status ansa_entry_create (
      ansa_Entry *entry, ansa_EntryAttr attr);

ansa_Status ansa_entry_bind (
      ansa_InterfaceRef *ref, ansa_Entry *entry);

ansa_Status ansa_entry_unbind (ansa_InterfaceRef *ref);

ansa_Status ansa_entry_close (ansa_Entry *entry);
```

DESCRIPTION

The **ansa_entry_create** function creates a new entry. The **ansa_entry_bind** function binds an interface instance to an entry. The **ansa_entry_unbind** function binds the interface instance back to the default system entry. The **ansa_entry_close** function closes an entry.

SEE ALSO

Entry attributes object.

Real-time task.

Real-time thread.

NAME

Rendezvous

PURPOSE

To rendezvous with an entry at run time.

SYNOPSIS

```
ansa_Status ansa_rendezvous (
      ansa_Entry *entry, ansa_Cardinal timeout);
```

DESCRIPTION

The **ansa_rendezvous** function allows a thread to rendezvous with an entry at run time. The timeout is in microseconds.

SEE ALSO

Entry.

NAME

Clock setup

PURPOSE

To provide the basic mechanism for global clock synchronization.

SYNOPSIS

```
void system_readTime (ansa_Time *time);
void system_resetTime (ansa_Time time);
```

DESCRIPTION

The **system_readTime** returns the current clock value for this nucleus. The **system_resetTime** resets the clock of this nucleus according to the new time.

The nucleus time service works on a virtual time basis:

```
nucleus time = local machine time + offset
```

The **system_resetTime** function adjusts the offset value to get a global nucleus time.

SEE ALSO

ANSAware/RT timed RPC protocol.

NAME

Semaphore

PURPOSE

To provide the traditional semaphore synchronization mechanisms.

SYNOPSIS

```
#include “entry.h”
ansa_Status ansa_sem_init (ansa_Semaphore *sem, int c);
ansa_Status ansa_sem_destroy (ansa_Semaphore *sem);
ansa_Status ansa_semP (ansa_Semaphore *sem);
ansa_Status ansa_semV (ansa_Semaphore *sem);
ansa_Status ansa_semTimeP (
      ansa_Semaphore *sem, ansa_Cardinal timeout);
```

DESCRIPTION

The timeout value is in microseconds.

SEE ALSO

ANSAware synchronization mechanisms.

NAME

Environment variable

PURPOSE

To select the scheduling policy for system tasks.

SYNOPSIS

DESCRIPTION

The **SCHEDPOLICY** environment variable is used to select the system tasks' scheduling policy. By default it is **SCHED_OTHER**, and can be set as **SCHED_FIFO** or **SCHED_RR**.

SEE ALSO

Task attributes object.

References

[APM92]

APM Ltd., ANSAware Version 4.1 Manual, APM Ltd., Cambridge CB3 0RD, U.K., May 1992.

[APM.AR.01]

O Rees, The ANSA Computational Model, AR 01, APM Ltd., Cambridge, CB3 0RD, U.K., 1993.

[APM.TR.33]

A Herbert, The Challenge of ODP, TR 33, APM Ltd., Cambridge, CB3 0RD U.K., 1993 Also Appeared as an Invited Paper for the Berlin ODP Conference, October 1991.

[APM.TR.18]

A Herbert, Distributing Objects, TR 18, APM Ltd., Cambridge, CB3 0RD, U.K.,1993.

[APM.TR.37]

C Nicolaou, ANSAware Use of DCE/POSIX Threads and RPC, TR 37, APM Ltd., Cambridge, CB3 0RD, U.K., February 1993.

[APM.1207]

G Li, Real-Time ANSAware Version 1.0: Programming and System Overview, APM document 1207, APM Ltd., Cambridge CB3 0RD, U.K., May 1994.

[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, 12(12), December 1986.

[Curnow76]

H J Curnow and B A Wichmann, A Synthetic Benchmark, Computer Journal, 19(1):48-49, January, 1976.

[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., 123-136, IEEE CS Press, June 1993

[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.

[Kamenoff91]

N I Kamenoff and N H Weiderman, Hartstone Distributed Benchmark: Requirements and Definitions, Proc. of Twelfth IEEE Real-Time Systems Symposium, 1991.

[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 Realtime Computing, PhD thesis, University of Cambridge Computer Laboratory, Technical Report 322, August 1993.

[Mercer90]

C W Mercer and Y Ishikawa and H Tokuda, Distributed Hartstone: A Distributed Real-Time Benchmark Suite, International Conference on Distributed Computing Systems, 1990.

[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.

[Tennenhouse89]

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

[Tokuda89]

H Tokuda and C W Mercer, ARTS: A Distributed Real-Time Kernel, Operating Systems Review, 23(3), July 1989.

[Weiderman89]

N Weiderman, Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications, Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-89-TR-23, June, 1989.

[Xu93]

J Xu and D L Parnes, On Satisfying Timing Constraints in Hard-Real-Time Systems, IEEE Transactions on Software Engineering, 19(1), January 1993.