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

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.1476.01 **Approved** 3rd May 1995

Technical Report

Distribution: W
Supersedes: -
Superseded by: -

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

ANSAware/RT 1.0 Manual

ANSAware/RT 1.0 Manual

APM.1476.01

3rd May 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

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

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

Contents

7 1 ANSAware/RT
7 1.1 Introduction
7 1.2 About this manual
8 1.3 ANSA, ANSAware and ANSAware/RT
8 1.4 New functionality in ANSAware/RT
8 1.5 Compatibility between ANSAware 4.1 and ANSAware/RT 1.0
8 1.5.1 Portability
9 1.5.2 Interworking
9 1.6 Requirements
9 1.6.1 ANSAware/RT platforms
10 1.6.2 System requirements
10 1.7 Release media
10 1.8 Support
11 1.9 Questions and Answers
11 1.9.1 About ANSAware/RT 1.0
12 1.9.2 Prerequisites
12 1.9.3 Compatibility
13 1.9.4 Performance and benchmarking
13 1.9.5 Future plans
13 1.9.6 Obtaining ANSAware/RT 1.0
13 1.9.7 Technical support services
14 1.10 ANSAware/RT version 1.0 release notes
14 1.10.1 Capsule Library
16 1.10.2 Application Libraries
17 1.10.3 PREPC
18 1.10.4 STUBC
18 1.10.5 Other changes

21 2 ANSAware/RT Installation and Configuration
21 2.1 Introduction
21 2.1.1 Goals
21 2.1.2 Overview
22 2.2 The generic distribution
22 2.2.1 The ANSAware/RT distribution tape
22 2.2.2 Installing the distribution
22 2.2.3 Building ANSAware/RT
22 2.3 Environment variables
23 2.4 Source code roadmap

27 3 An Open Architecture for Real-Time Processing
27 3.1 Introduction
27 3.2 Motivation
27 3.3 Scope

28	3.4	Issues for distributed real-time systems
28	3.4.1	Predictability
28	3.4.2	Programmer control
29	3.4.3	Timeliness
29	3.4.4	Mission orientation
29	3.4.5	Performance
29	3.5	An integrated system architecture
30	3.6	Technologies
30	3.6.1	Contributory technologies
31	3.6.2	Distributed system environments
32	3.6.3	Real-time distributed system environments
33	3.7	Application domain
37	**4**	**ANSAware/RT Design**
37	4.1	Introduction
37	4.2	Design Overview
37	4.3	Real-time programming model
38	4.4	Real-time communication
39	**5**	**Real-Time Programming Model**
39	5.1	Introduction
39	5.2	Distributed object execution
39	5.3	ANSA object execution
40	5.3.1	ANSA object execution model deficiencies for real-time applications
40	5.4	Real-time objects
42	5.5	Real-time object invocation
43	5.6	Scheduling
44	5.7	Priority scheduling
44	5.7.1	Priority management and priority inheritance
45	5.7.2	Resource allocation and task preemption
46	5.7.3	Dealing with priority inversion
47	5.8	Deadline scheduling
48	5.9	Other scheduling paradigms
48	5.10	Application controlled rendezvous
49	5.11	Summary
51	**6**	**Real-Time Communication System**
51	6.1	Introduction
51	6.2	Parallel protocol stacks
52	6.3	Timed RPC protocol
53	6.3.1	Discussion of problem
54	6.3.2	The protocol
55	6.3.3	Server deadline expiry
56	6.4	Decomposable RPC protocol
57	6.5	Summary
61	**7**	**ANSAware/RT Application Programming**
61	7.1	Introduction
61	7.2	ANSAware 4.1 programming interface
61	7.3	Attributes objects
61	7.3.1	Task attributes object
62	7.3.2	Entry attributes object

63	7.3.3	Thread attributes objects
63	7.4	Scheduling
63	7.4.1	Explicit task creation
64	7.4.2	Task scheduling policies
64	7.4.3	Scheduling Policy environment variable
65	7.4.4	Entry
65	7.4.5	Entry queuing and rendezvous policies
66	7.4.6	Rendezvous
66	7.4.7	Examples
67	7.5	Communications
67	7.5.1	QoS object
68	7.5.2	Explicit Binding
69	7.5.3	Invocations specifying in-band QoS
69	7.5.4	Clock synchronisation
70	7.5.5	Examples
73	**8**	**API manual pages**
91	**9**	**Tasking and Scheduling Implementation**
91	9.1	Introduction
91	9.2	Review of tasking in ANSAware 4.1
92	9.3	ANSAware/RT tasking
92	9.3.1	Global data protection
92	9.3.2	Thread private state
92	9.4	Stacked threads
93	9.5	Threads
93	9.6	Entry
94	9.7	Synchronous I/O
94	9.8	Communication tasks and system tasks
94	9.9	Other ANSAware/RT tasks
97	**10**	**Implementation of the Communication System**
97	10.1	Review of the ANSAware 4.1 communication system
97	10.1.1	Interface reference
97	10.1.2	MPS
97	10.1.3	EXecution protocols
98	10.1.4	Channel and session
98	10.1.5	Bindings
98	10.2	QoS and explicit binding in ANSAware/RT
99	10.3	State-based MPS
99	10.4	State-based EXecution protocol
99	10.5	TREX
100	10.6	In-band QoS
100	10.7	Session timeout recovery
101	10.8	Miscellaneous communications details
103	**11**	**Performance Measurements**
103	11.1	Basic performance
104	11.2	Distributed Hartstone performance
104	11.2.1	Communication latency
105	11.2.2	Priority queuing
106	11.2.3	Protocol preemptivity

107 11.2.4 Communication bandwidth
108 11.2.5 Result

ANSAware/RT Overview

1 ANSAware/RT

1.1 Introduction

This document forms the manual set of ANSAware/RT 1.0. It contains all information specific to the ANSAware/RT release and should be read in conjunction with the existing ANSAware 4.1 manual set, a postscript copy of which is included with the distribution.

This chapter provides an overview of the differences between ANSAware/RT 1.0 and ANSAware 4.1[1] and assumes the reader to be familiar with the latter.

1.2 About this manual

The ANSAware/RT manual is structured along the lines of the ANSAware 4.1 manual and comprises six logical parts:

- **An overview of ANSAware/RT** - a single chapter, §1 *ANSAware/RT*. This is intended as an overview of ANSAware/RT and of the manual itself in terms of providing the necessary background material. This should be read first by all readers
- **ANSAware/RT system manager's guide** - a single chapter, §2 *ANSAware/RT Installation and Configuration*. This is aimed at the system manager or individuals wishing to install and build ANSAware/RT and is primarily an installation guide with advice on the major platforms supported
- **ANSA real time architecture** - a single chapter, §3 *An Open Architecture for Real-Time Processing*. Provides a description of the extensions to the ANSA architecture for real-time programming. It is intended for system architects and as background material for application programmers and system engineers
- **ANSAware/RT design** - chapters: §4 *ANSAware/RT Design*, §5 *Real-Time Programming Model* and §6 *Real-Time Communication System*. Together these provide the rationale behind the design decisions and act as an introduction to the design. This part is likely to be of interest to both application programmers and systems engineers
- **ANSAware/RT application programming** - chapters: §7 *ANSAware/RT Application Programming* and §8 *API manual pages*. Together these are aimed at those wishing to write programs that will run over ANSAware/RT. This part describes the real-time extensions to both the programming model and the application programming interface
- **ANSAware/RT system programming** - chapters: , §9 *Tasking and Scheduling Implementation*, §10 *Implementation of the Communication*

1. ANSAware 4.1 is used in this document to refer to the generic 4.1 release and hence includes ANSAware 4.1.1.

System and §11 *Performance Measurements*. Together these provide detailed information about the implementation of the real-time extensions and are likely to be of most interest to systems engineers.

This manual is provided as a supplement to the existing ANSAware 4.1 manual set which is included with the ANSAware/RT release. It describes the new real-time facilities provided by ANSAware/RT and identifies the differences between ANSAware 4.1 and ANSAware/RT.

1.3 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, node and system management. ANSAware is currently in release 4.1.1

ANSAware/RT is an extension of ANSAware to address the requirements of real-time processing, such as predictability, programmer control, timeliness, and performance. ANSAware/RT provides these facilities by enabling access to the real-time properties of the underlying environment over which it runs and is consequently available only for operating systems that offer real-time facilities.[1] In particular, the implementation makes use of real-time POSIX threads (p-threads), a de-facto industry standard increasingly being provided by operating systems vendors.

ANSAware/RT 1.0 is not a replacement for ANSAware 4.1 as it is constrained to run on platforms that provide real-time facilities. It is intended for situations where there is a need to support applications with specific real-time requirements. ANSAware 4.1 runs over a much broader range of operating systems and is thus better suited to non real-time applications that must be distributed across a diverse set of platforms.

1.4 New functionality in ANSAware/RT

The following list identifies the major areas of new functionality:

- extended tasking system
- extended communication system
- extended application programming interface
- extended PREPC and IDL.

1.5 Compatibility between ANSAware 4.1 and ANSAware/RT 1.0

1.5.1 Portability

ANSAware 4.1 is the current release of ANSAware on which ANSAware/RT 1.0 is based. A design goal of ANSAware/RT 1.0 API was to provide real-time

1. For a detailed description of the platforms on which ANSAware/RT 1.0 is available, see §1.6.1 *ANSAware/RT platforms*.

functionality as an extension of the ANSAware 4.1 API and hence the majority of ANSAware 4.1 services will port to ANSAware/RT 1.0 without change.

However, in contrast to ANSware 4.1, ANSAware/RT adopts a synchronous I/O programming model which has made the previous asynchronous support unnecessary. As a consequence, the **pin(3)**, non-blocking keyboard library **kbinput(3)** and the X11 support libraries **ansaXt** and **ansaXtra** are not present in ANSAware/RT. Services using routines contained in these libraries (detailed in §1.10.2 *Application Libraries*) must be modified to use equivalent synchronous routines. In general these will be the native routines supplied by the underlying operating system.

Second, the PREPC invocation statement has been extended in a manner which is incompatible with a particular form previously available to ANSAware services. In particular, the qos parameter is used in ANSAware 4.1 specifically for controlling REX retry numbers, whilst in ANSAware/RT it has a more general role, and is itself a control record. Applications already using the qos parameter to control REX retry numbers will have to be modified to use the new QoS attribute object routines.

1.5.2 Interworking

Applications developed for ANSAware/RT 1.0 will interwork with applications running on ANSAware 4.1 provided they use compatible communications. Compatible communications supported by ANSAware/RT are the REX execution protocol over either UDP or IPC.

1.6 Requirements

1.6.1 ANSAware/RT platforms

As with ANSAware 4.1, ANSAware/RT has certain requirements on any *platform* on which it executes. For the most part, these are similar to those of ANSAware 4.1 but with the additional requirement that the underlying operating system must have real-time capability. In particular, ANSAware/RT tasking is implemented over POSIX threads (p-threads).

The following table lists the platforms on which ANSAware/RT is supported.

Table 1.1: Supported platforms

Platform	CPU	System	O/S	Libraries
alpha_osf_1.3.rt	alpha	DEC alpha	OSF1	-lpthreads -lmach -lc_r
alpha_osf_3.0.rt	alpha	DEC alpha	OSF1	-lpthreads -lmach -lc_r

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

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

1.6.1.1 *Disc requirements*

The following table lists the disk space requirements for ANSAware/RT for the supported platforms.

Table 1.2: Disk space requirements in megabytes

Platform	Distribution	Build Tree	Installation	Total
alpha_osf_1.3.rt	20	88	37	145
alpha_osf_3.0.rt	20	88	37	145

1.6.1.2 *Communication requirements*

ANSAware/RT uses only the internet family of network protocols.

1.6.2 System requirements

This section lists the system requirements, in terms of system software needed by ANSAware/RT.

1.6.2.1 *UNIX requirements*

ANSAware/RT should be compiled on DEC Alpha systems with the standard OSF1 compiler (cc) and using the standard make utility (/usr/bin/make). The compiler option -std1 is required to ensure ANSI compatibility and to define some additional conditional compilation macros (see the OSF1 man pages for details).

Results using other tools cannot be guaranteed and in particular, the GNU versions of the compiler (gcc) and make utility (gmake) are known to fail.

1.7 Release media

ANSA sponsors may access the released software by ftp.

The software and manuals are normally available on DEC TLZ tape (4.0mm DAT) written in tar format.

1.8 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 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) at the address inside the front cover of this manual. APM wish to be informed of all problems and difficulties encountered and, within the constraints of the ANSA work programme, 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.

Please e-mail ANSAware/RT bug reports to: **ansaware@ansa.co.uk**.

1.9 Questions and Answers

1.9.1 About ANSAware/RT 1.0

Q: *What is ANSAware?*

A: ANSAware is an example implementation of the ANSA architecture for open distributed processing. It provides a set of tools and libraries which allow programmers to distribute their applications in a heterogeneous system. ANSAware/RT is distributed in source form to allow experimentation and modification.

Q: *What is ANSAware/RT 1.0?*

A: ANSAware/RT 1.0 is the first release of an extended ANSAware intended to address the requirements of real-time processing, such as predictability, control, timeliness and performance. It is designed to provide these facilities by retaining the real-time properties of the underlying environment over which it runs and is consequently available only for operating systems that offer real-time facilities.

Q: *What are the new facilities of ANSAware/RT 1.0?*

A: The new facilities fall into the following major categories:

- extended tasking system
- extended communications system
- extended application programming interface
- extended PREPC and IDL.

Q: *Is ANSAware/RT 1.0 a replacement for ANSAware 4.1?*

A: No. ANSAware/RT runs only on real-time operating systems and is intended for applications with specific requirements for real-time functionality. ANSAware 4.1 runs over a much broader base of operating systems and is thus better suited to non real-time applications that must be distributed across a diverse set of platforms.

Q: *For which applications is ANSAware/RT suitable?*

A: ANSAware/RT is suitable for any distributed application where the limited range of platforms is not a problem. However, it is best suited to applications requiring explicit control over system resources in order to achieve predictable system performance. In addition, ANSAware/RT is targeted at applications providing supervisory control rather than low-level, synchronous sampled data loop functions, e.g. signal processing.

Q: How can I obtain more technical information about ANSAware/RT?

A: Contact APM Ltd, the address and telephone number of which appear on the front cover. Alternatively, send your request by e-mail to **ansaware@ansa.co.uk**.

1.9.2 Prerequisites

Q: What will I need to run ANSAware/RT 1.0?

A: You will need one of the supported hardware platforms: at release 1.0 this must be a DEC Alpha system running OSF1. Version 1.3 and 3.0 of OSF1 are supported. Your machine must have the standard OSF1 development tools and libraries installed in order to build ANSAware/RT.

Q: Which operating system and hardware platforms does ANSAware/RT 1.0 support?

A: Release 1.0 of ANSAware/RT supports only DEC Alpha systems running either version 1.3 or 3.0 of OSF1.

Q: Can I port ANSAware/RT 1.0 to other operating systems and hardware platforms?

A: ANSAware/RT is intended to be portable to other operating systems and hardware architectures. However, in addition to the requirements detailed for ANSAware 4.1, it should be noted that the operating system must be capable of providing real-time facilities. In particular, ANSAware/RT makes extensive use of POSIX threads (p-threads).

1.9.3 Compatibility

Q: Does ANSAware/RT 1.0 interoperate with ANSAware 4.1?

A: Yes, provided that compatible communications are used. Compatible communications are the REX execution protocol over either UDP or IPC.

Q: Are ANSAware 4.1 applications portable to ANSAware/RT 1.0?

A: A design goal of ANSAware/RT was to provide real-time functionality as an extension of the ANSAware 4.1 API and hence the majority of applications will port without change. However, there are a few exceptions and these are described in detail in §1.5.1 *Portability*.

Q: Which RPC protocols does ANSAware/RT 1.0 support?

A: ANSAware/RT 1.0 supports the existing REX protocol and a new timed variation, called TREX. The group execution protocol, GEX, is not supported at this release.

Q: Which transport protocols does ANSAware/RT 1.0 support?

A: ANSAware/RT supports UDP and IPC. It does not support TCP.

Q: Which network protocols does ANSAware/RT 1.0 support?

A: ANSAware/RT supports the internet family of network protocols.

Q: Which multicast protocols does ANSAware/RT 1.0 support?

A: ANSAware/RT does not support multicast, nor group transparency at this release.

1.9.4 Performance and benchmarking

Q: How does the performance of ANSAware/RT 1.0 compare with ANSAware 4.1?

A: The performance of ANSAware/RT is better than ANSAware 4.1, mainly due to the use of synchronous I/O operations which are more efficient. In addition, the performance of ANSAware/RT is further improved if the new TREX protocol is used.

1.9.5 Future plans

Q: How is ANSAware/RT expected to evolve?

A: ANSAware/RT is the first step on the road towards support for applications which implement large scale multi-media information services in heterogenous environments.

Q: Will ANSAware 4.1 continue to be supported?

A: Yes. ANSAware 4.1 addresses a different requirement, that of non real-time applications that must be distributed across a diverse set of platforms.

Q: Will ANSAware/RT support Microsoft Windows 3.x?

A: No. Microsoft Windows 3.x does not offer real-time facilities.

Q: Will ANSAware/RT support Microsoft Windows '95?

A: A port to Microsoft Windows '95 is considered to be relatively straightforward based on the anticipated facilities of the system. However, given that Windows '95 is not yet finalised, this cannot be guaranteed. It is not known at this time whether a port will be made available by APM.

Q: Which other operating systems and hardware platforms will ANSAware/RT support?

A: A release of ANSAware/RT over Microsoft Windows NT is planned. Ports to other suitable operating systems (e.g. Solaris) and support for the existing HP/RT or LynxOS ports is subject to availability of such machines at APM.

1.9.6 Obtaining ANSAware/RT 1.0

Q: How do I obtain ANSAware/RT 1.0?

A: ANSAware/RT is distributed by APM Ltd, the details of which appear on the front cover of this manual. Alternatively, send your request by e-mail to **ansaware@ansa.co.uk**.

Q: On which media is ANSAware/RT 1.0 available?

A: ANSAware/RT 1.0 is available on DEC TLZ (4.0mm DAT) tape, written in tar format.

1.9.7 Technical support services

Q: What technical support is provided for ANSAware/RT 1.0?

A: ANSAware/RT is distributed in source form to encourage as much self help as possible. However, limited support is provided by APM and an e-mail help-line is also available (see §1.8 *Support* for further details).

Q: Is training available for ANSAware/RT 1.0?

A: Please contact APM at the address on the inside of the front cover of this manual, or e-mail **apm@ansa.co.uk**, for information about training.

1.10 ANSAware/RT version 1.0 release notes

This section provides an overview of the differences between ANSAware 4.1 and ANSAware/RT. Further details can be found in subsequent chapters. The differences fall into the following categories:

- Capsule Library
- Application Libraries
- PREPC
- STUBC

1.10.1 Capsule Library

1.10.1.1 Environment variable

A new environment variable SCHEDPOLICY has been added to allow the selection of one of the scheduling policies supported by the underlying p-thread interface. Available options include SCHED_FIFO, SCHED_RR for real-time applications, SCHED_FG_NP, SCHED_BG_NP where throughput is the main criterion and the default SCHED_OTHER. For a full description of the meaning of the various policies see §7.4.2 *Task scheduling policies*.

1.10.1.2 Attributes objects

An attributes object is introduced 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. A full description of attributes objects is given in §7.3 *Attributes objects*.

Interfaces to task attribute objects are fully described in §8 *API manual pages* and briefly include:

- `ansa_taskattr_create` - creates a task attributes object containing default values for individual attributes
- `ansa_taskattr_delete` - deletes a task attribute object
- `ansa_taskattr_setinheritsched` - set how the scheduling policy is inherited
- `ansa_taskattr_setsched` - set the task scheduling policy
- `ansa_taskattr_setprio` - set the task priority
- `ansa_taskattr_setstacksize` - set the stack size for a task

There are corresponding routines to obtain the values of a task attributes object and these are prefixed with `ansa_taskattr_get`, e.g. `ansa_taskattr_getsched` would return the value for inheritance of the scheduling policy for a task.

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

- `ansa_entryattr_create` - create an entry attributes object with default values
- `ansa_entryattr_delete` - delete an entry attributes object
- `ansa_entryattr_setqueuing` - set the thread queuing policy

- `ansa_entryattr_setrendezvous` - set the task/thread rendezvous policy
- `ansa_entryattr_setceiling` - set the priority ceiling value
- `ansa_entryattr_setprio_min` - set the minimum priority value
- `ansa_entryattr_setprio_max` - set the maximum priority value
- `ansa_entryattr_getrendezvous` - get the task/thread rendezvous policy
- `ansa_entryattr_getceiling` - get the value for the priority ceiling
- `ansa_entryattr_getprio_min` - get the value for the minimum priority
- `ansa_entryattr_getprio_max` - get the value for the maximum priority

Routines related to thread attributes are:

- `ansa_threadattr_create` - create a thread attributes object with default values
- `ansa_threadattr_delete` - delete a thread attributes object
- `ansa_threadattr_setprio` - set the priority for a thread
- `ansa_threadattr_setdeadline` - set a deadline value for a thread
- `ansa_threadattr_getprio` - get the value of priority for a thread
- `ansa_threadattr_getdeadline` - get the deadline value for a thread

1.10.1.3 *Entry*

An entry is a new abstraction for a scheduling point by which different scheduling/processing concerns can be identified. Each capsule has a default entry to which all new created interfaces of the capsule are bound. Binding an interface to an entry results in all invocations on the interface being queued on the entry and later processed by the tasks allocated to the entry. Further details can be found in §7.4.4 *Entry* and §7.4.5 *Entry queuing and rendezvous policies*.

The interface to entry objects is as follows:

- `ansa_entry_create` - create an entry object
- `ansa_entry_bind` - binds an interface to an entry
- `ansa_entry_unbind` - causes an interface to revert to the default capsule entry
- `ansa_entry_close` - closes an entry

1.10.1.4 *Task creation*

Two additional interfaces are introduced to control ANSAware/RT task creation:

- `nucleus_tasks_onentry` - create a task to process threads on the specified entry
- `ansa_task_spawn` - spawn a new task with a separate thread of execution

1.10.1.5 *Thread creation*

An additional interface is added to spawn an ANSA thread on a specific entry:

- `instruct_Spawn_onentry` - spawn a thread on the specified entry

1.10.1.6 *QoS objects*

A QoS object consists of individual attribute values and is introduced to describe communication resource and performance constraints. The interface to QoS objects is briefly as follows and further details can be found in §7.5.1 *QoS object*:

- `ansa_endQoS_create` - create an endpoint QoS object with default values
- `ansa_endQoS_set`*`AnAttribute`* - set the individual parameter value for *`AnAttribute`* of an endpoint QoS object
- `ansa_invQoS_create` - create an in-band QoS object
- `ansa_invQoS_`*`setAnAttribute`* - set the individual parameter value for *`AnAttribute`* of an in-band QoS object

1.10.1.7 *Rendezvous*

ANSAware/RT extends the ANSAware 4.1 tasking system to allow stackable execution of threads. This permits a thread, while executing (has a task allocated), to wait at an entry to rendezvous and execute another thread (this can be an invocation).

The rendezvous function is:

- `ansa_rendezvous` - cause the calling task to rendezvous with a thread on the specified entry

1.10.1.8 *Clock*

The deadline associated with an invocation implies that both the server and the client share a common view of time. ANSAware/RT provides only minimum functions for clock reset and relies on an application to provide the appropriate clock synchronization service (as an ANSAware service, for example).

The ANSAware/RT nucleus provides two new functions:

- `system_readTime` - reads the current clock value of the capsule
- `system_resetTime` - reset the capsule clock according to the given parameter

1.10.2 Application Libraries

ANSAware/RT, in contrast to ANSAware 4.1, provides only a synchronous I/O model. As a result the following interfaces are no longer supported in ANSAware/RT:

- pin(3)
- non-blocking keyboard library, kbinput(3)
- X11 support libraries: ansaXt(3) and ansaXtra(3)

As a consequence the following routines are now obsolete:

```
pin_open
pin_close
locked_pin
locked_pin_close
```

```
aw_open
aw_close
aw_fgetc
aw_fgets
aw_restore_terminal

AnsaXtInit
AnsaXtCreateAppSession
AnsaXtDestroyAppSession
AnsaXtKillAppSession

XtAddTimeOut
XtAppAddTimeOut
XtRemoveTimeOut
ansa_XtDeleteFromAppContext
```

1.10.3 PREPC

1.10.3.1 Explicit binding

ANSAware/RT extends the PREPC language to add support for explicit binding operations to:

- associate QoS with communication endpoints
- control binding time

A server accomplishes explicit binding at service creation time, using

```
{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 construction 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.

A client performs explicit binding when it obtains a server interface reference `ir`, by using:

```
{} :: 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 existing ANSAware.

It is worth pointing out that the TREX protocol requires the explicit binding operation to 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’s explicit binding is destroyed and therefore related resources released when the server makes an explicit call

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

The corresponding client operation is

```
ir$Discard
```

1.10.3.2 *Invocations and the specification of in-band QoS*

The syntax of a PREPC invocation is extended to allow the association of 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.

1.10.3.3 *New exceptions*

ANSAware/RT can potentially generate additional exceptions (defined in the include file `status.h`) which applications should be prepared to handle where necessary.

The following new return codes may be returned when using the TREX execution protocol:

```
messageTooLong
clientTimeout
serverDeadlineExpire
```

Entry routines may return:

```
invalidEntryAttr
invalidCeiling
invalidEntry
```

Explicit binding operations may return:

```
invalidBinding
bindingExist
invalidQoS
noBinding
```

There are also two additional abort codes relating to ANSAware/RT's use of pthreads:

```
pthreadError
systemTimeout
```

1.10.4 STUBC

Although the stub compiler (STUBC) has been modified internally for ANSAware/RT, there are no user visible changes. In particular, there are no changes to the interface definition language (IDL).

1.10.5 Other changes

The existing REX execution protocol is supported and a new variant, timed REX (TREX) is introduced. TREX supports only the new explicit binding operations, i.e. interfaces created by implicit binding (the only mechanism available in ANSAware 4.1) cannot use TREX.

ANSAware/RT does not support the group execution protocol GEX in this release. Therefore there is no support for active replica interface group transparency.

This release of ANSAware/RT runs only over UDP or IPC. TCP is not supported.

ANSAware/RT System Manager’s Guide

2 ANSAware/RT Installation and Configuration

2.1 Introduction

This chapter contains information pertaining to the installation, configuration and maintenance of ANSAware/RT. In general, this is the same as that described for ANSAware 4.1 and hence this document only describes the differences.

2.1.1 Goals

The main goal of ANSAware/RT is 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:

- to be compatible with ANSAware 4.1
- to run over a de-facto industry standard: real-time POSIX threads
- to offer full p-thread real-time scheduling and threading capabilities
- to provide selective communication multiplexing by Quality of Service (QoS) specification and explicit binding operations
- to allow application controlled resource allocation
- to support a real-time programming model
- to be comparable to other distributed real-time system environments
- to permit interoperation between different real-time platforms
- to provide interoperation between different real-time and non-real-time platforms.

2.1.2 Overview

There is only one ANSAware/RT distribution available at Release 1.0, the generic one, although at this release it may be built only for the OSF1 Unix platform. This is because ANSAware/RT can inherently run only over operating systems that support real time facilities such as POSIX threads and there is currently a lack of commercially available platforms offering these facilities[1]. It is anticipated that further ports will be produced as suitable platforms become generally available.

1. ANSAware/RT has been successfully ported to HP/RT and LynxOS [APM.1207], but since these systems are not generally available, these ports are not being provided with this release.

2.2 The generic distribution

2.2.1 The ANSAware/RT distribution tape

ANSAware/RT is normally distributed on DEC TLZ (4.0mm DAT) tape, written in tar format.

2.2.2 Installing the distribution

Installing the distribution is the same procedure as for ANSAware 4.1 although only the generic distribution may be installed, i.e. there are no distributions for MS DOS or VMS.

2.2.3 Building ANSAware/RT

The only platforms supported at Release 1.0 are DEC alpha workstations running OSF1 at either release 1.3 (alpha_osf_1.3.rt) or release 3.0 (alpha_osf_3.0.rt).

When building ANSAware/RT, you will be asked to specify which message passing services (MPS) you require. At this release, only UDP and IPC are available. In contrast to ANSAware 4.1, TCP is not supported (although at this release it still appears in the list of options). If you wish to run applications that will communicate across different nodes, you must include UDP.

It is important not to use the alternative compiler option when building ANSAware/RT. In particular, the GNU compiler (gcc) is known to fail when compiling ANSAware/RT.

When compiling ANSAware/RT with the default compiler (as recommended), it is essential to specify the compiler option -**std1** in response to the build procedure’s prompt for additional compiler options. Failure to do so will cause the compilation to fail with errors referring to duplicate definitions in **AWstdio.h**.

2.3 Environment variables

In addition to the environment variables which may be used to control the behaviour of ANSAware 4.1, ANSAware/RT also recognises SCHEDPOLICY. This may be used to select the default task scheduling policy and may take the values:

- SCHED_FIFO - first in first out policy for real-time applications
- SCHED_RR - round robin policy for real-time applications
- SCHED_FG_NP - policy suitable for non-real-time applications that normally run in the foreground
- SCHED_BG_NP - policy suitable for non-real-time applications that normally run in the background
- SCHED_OTHER - the default time-sharing policy.

The first two are mapped onto the equivalent p-thread supported real-time scheduling policies.

2.4 Source code roadmap

The ANSAware/RT master tree is structured into a number of independent components as follows. Although the differences are comparatively minor, the entire structure is reproduced here for completeness.

- **master/util**: utilities
  - **config**: ansaimake configuration files
  - **prepc**: prepc source code
  - **stubc**: stubc source code
  - **imake**: ansaimake source code
  - **scripts**: package, ansaconfig and other shell scripts
  - **makedepend**: ansadepend source code
  - **sutils**: simple utility programs
- **master/include**: header files
  - **ansi**: header files for use with ANSI C
  - **lib**: header files for use with application libraries
  - **capsule**: platform independent header files
  - **capsule/CORE_unix**: UNIX specific header files
  - **capsule/stack**: platform specific stack handling header files
  - **config**: platform specific capsule configuration files
  - **machine**: machine or hardware specific header files
  - **opsys**: operating system specific header files
  - **stub**: header files for stub generation
  - **stub/MoveMacs**: machine specific header files for stub generation
  - **idl**: contains IDL files for ANSA services
- **master/src/ansa/capsule**: capsule libraries
  - **CORE_unix**: UNIX specific capsule code
- **master/src/ansa/test**: test programs for capsule libraries
- **master/src/ansa/trader**: trader sources
- **master/src/ansa/factory**: factory sources
- **master/src/ansa/nodemgr**: node manager source
- **master/src/ansa/reloc**: relocation service example
- **master/src/lib**: user libraries
  - **reloc**
  - **checkpt**
- **master/src/examples**: examples
  - **Alarm**
  - **BTest**
  - **Cksum**
  - **DHstone**: real-time distributed hartstone benchmark

- **DHstone_cb**: as above, communications bandwidth variation
- **DHstone_pp**: as above, protocol premptivity variation
- **DHstone_pq**: as above, priority queuing variation
- **Echo**
- **Except**
- **Mgmt**
- **Netinfo**
- **RealtimeEcho**: real time variation of Echo example
- **SBank**
- **Sample**
- **SlowInt**
- **Stubtest**
- **bindEcho**: explicit binding example
- **test1**
- **timerEcho**: real time variation of Echo example

- **master/contrib**: contributed applications: these are not produced or supported by APM.

The ANSAware/RT root directory contains a number of subdirectories which contain scripts and configuration files; these are as follows:

- **master**: the master source tree described above
- **ANSAware**: contains ANSAware/RT configuration files and platform specific utility scripts for ANSAware.sh
- **examples**: contains examples parameters used as input to ANSAware.sh
- **bin**: contains scripts used by ANSAware.sh and packages
- **awkscrs**: contains awk scripts used by ANSAware.sh and packages
- **dirs**: contains the list of directories describing the master source tree structure
- **packages**: contains package configuration files and package specific scripts
- **hardcopy**: contains an on-line, PostScript copy of this manual.

ANSA Real Time Architecture

3 An Open Architecture for Real-Time Processing

3.1 Introduction

This chapter is intended for systems architects and those requiring a high level description of the problem space and technology bases of real-time open distributed processing. An integrated system architecture is presented and the benefits of the architecture are described. The practical need and importance of the architecture is discussed along with the current technology trends in both distributed processing and real-time applications. The architecture is shown to target (though not exclusively) *supervisory control* as its applications.

3.2 Motivation

Distributed real-time processing makes unique demands on systems and their designs (e.g. predictability, programmer control, timeliness, mission orientation and performance, see §3.4 *Issues for distributed real-time systems*). Features supporting these demands 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 show how it is possible to extend a distributed system environment to support real-time applications and hence help avoid these problems.

3.3 Scope

The design of ANSAware/RT is based on 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 extensions necessary for distributed interactive multimedia applications.

The principal issues covered by the architecture 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 the ANSA architecture, because the latter has a generic engineering model. However, the architectural issues 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.

3.4 Issues for distributed real-time systems

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 utilises the resources within this environment. Such distributed real-time processing places a set of unique requirements on the system, 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.

3.4.1 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 the operations' timing requirements are satisfied. A fully predictable system can perform operations with guaranteed upper bounds on timing, independent of surrounding conditions. Conversely, anunpredictable 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.

3.4.2 Programmer control

Programmer control means that an application programmer has ultimate control of the behaviour of a system. This requirement derives 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 allowing programmer control over system behaviour is probably by providing a 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.

3.4.3 Timeliness

Real-time applications are different from the non real-time paradigm of computation in that they impose 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 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 of the latter is to allow an application to associate deadlines with real-time activities, and for the system to employ a deadline based scheduling policy to help meet the deadlines or to identify and cancel obsolete operations. Other more complicated functions include the description and enforcement of temporal relationships among related computational activities.

3.4.4 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 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.

3.4.5 Performance

Many real-time applications have stringent raw performance requirements. To support this, 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.

3.5 An integrated system architecture

The objective of this design is to provide an open real-time distributed system environment architecture. An important point 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 principal approach: ***system integration***. The importance and benefits of the approach are also identified.

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 *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, 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 reengineered 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 only to part of the 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.***

3.6 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 the art in distributed real-time system environment research and engineering, and the additional functions required in such an open, real-time, distributed architecture.

3.6.1 Contributory technologies

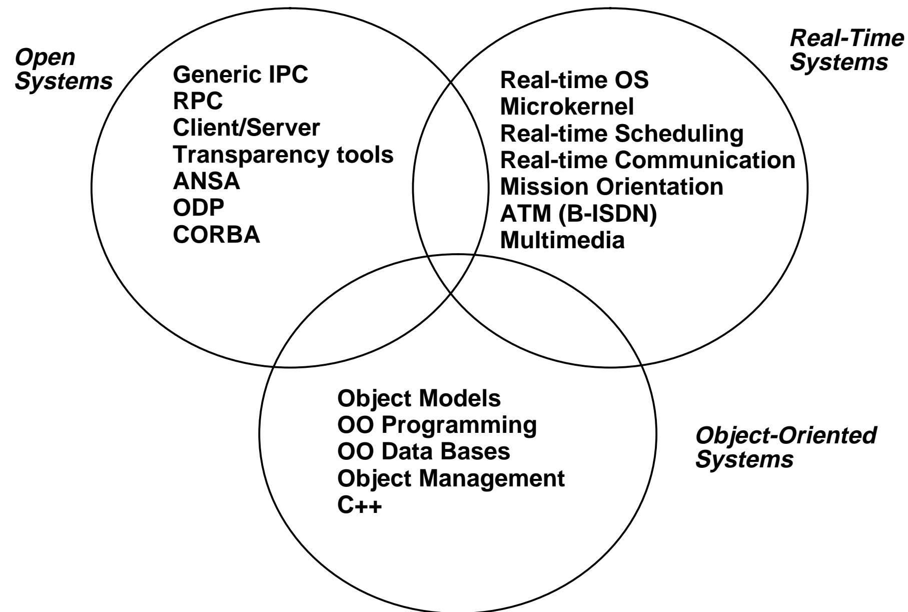
The fundamental contributory technologies are illustrated in Figure 3.1, which represents the integration of real-time systems, open systems and object oriented systems.

Real-time system technology provides the functionality of resource management for guaranteeing 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



3.6.2 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:

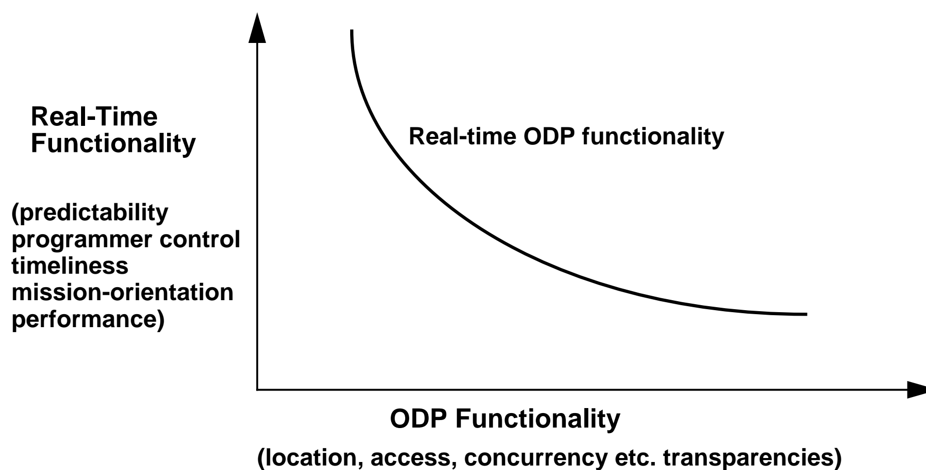
- location transparency --- masking the physical location of services
- access transparency --- masking differences in representation and operation invocation mechanism
- concurrency transparency --- masking overlapped execution
- replication transparency --- masking redundancy
- failure transparency --- 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.

3.6.3 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 access these services. Worse, a distributed system environment often masks the real-time features of base technologies. Therefore, one of the main aims of a real-time architecture is to extend the real-time features of base technologies to the level of the distributed system environment.

Figure 3.2: Real-Time ODP Functionality



One common misconception is that distributed system environments are unsuitable technology for real-time applications because RPC is often considered to be too slow. This is a misconception because the objective of real-time computing is to meet the timing requirements of an application, rather than simply 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. Current technology can already provide RPC calls whose latency is expressed in terms of milliseconds (as the required performance for the ***supervisory control*** targeted by the real-time ANSA architecture, see also §3.7 *Application domain*). Systems exist that can provide RPC calls whose performance is in the hundreds of microseconds range. [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 existing distributed system environments, 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

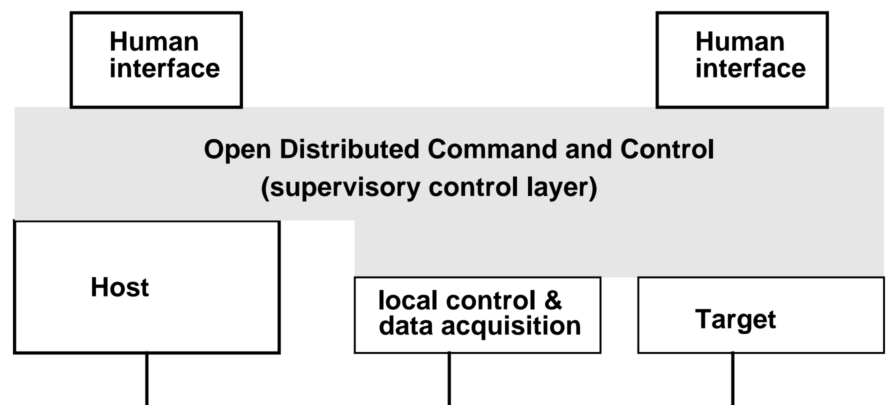
real-time distributed system environment functionality is given. The curve in the figure illustrates that real-time functionality and distributed systems environment functionality are often conflicting goals, which must be traded against one another. For example, most distribution transparencies (such as RPC protocols) are based on *time redundancy* technologies, which will need to be revised for real-time applications.

3.7 Application domain

Real-time systems span a wide range of application domains, including military, industry, commerce, medicine and so on.

The extensions to the ANSA architecture for real-time systems aim to support ***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 human interfaces; rather, it interacts with subsystems which provide these functions. The real-time response requirements of a supervisory control system are typically in the millisecond than either the microsecond or second ranges.

ANSAware/RT Design

4 ANSAware/RT Design

4.1 Introduction

This chapter, together with §5 *Real-Time Programming Model* and §6 *Real-Time Communication System*, provide the rationale behind the key design decisions for ANSAware/RT and explain the major problems. This part is intended as background for both application programmers and systems engineers.

4.2 Design Overview

The design of ANSAware/RT introduces:

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

The following sections provide a brief overview for each of the two major issues.

4.3 Real-time programming model

A real-time programming model provides the basic abstractions so that the 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 computational and engineering models. As in the existing ANSA architecture, 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 to be the most important system resources. Both static resource

allocation --- the allocation of system resources to interfaces --- and dynamic resource allocation --- the allocation of system resources to invocations are supported. *Predictability, programmer control* and *mission criticality* are the main concerns of the real-time programming model.

4.4 Real-time communication

Real-time applications present 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 ANSAware 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. The main benefit of this design derives from allowing 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 choice 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 resemble 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 common examples 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 avoids the additional burden of having to monitor and manage timing constraints by application programmers during remote calls.
- a decomposable RPC protocol to allow the synthesis of the protocol to provide different levels of invocation semantics (such as exactly-one, at-most-once), so that an application programmer can customize the system to application-specific requirements of functionality and performance. This work is targeted at new transportation protocols with QoS parameters in the operational interface.

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

5 Real-Time Programming Model

5.1 Introduction

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

5.2 Distributed object execution

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. Real-time distributed systems must provide support for the specialized requirements of real-time communication, tasking, scheduling, and control. These requirements must be explicitly addressed in an object execution model, if the object-oriented approach is expected to be applicable to a real-time world.

5.3 ANSA object execution

The ANSA Computational Model and ANSA Engineering Model together 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.

5.3.1 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.

5.4 Real-time objects

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

A real-time object, like any other ANSA 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.

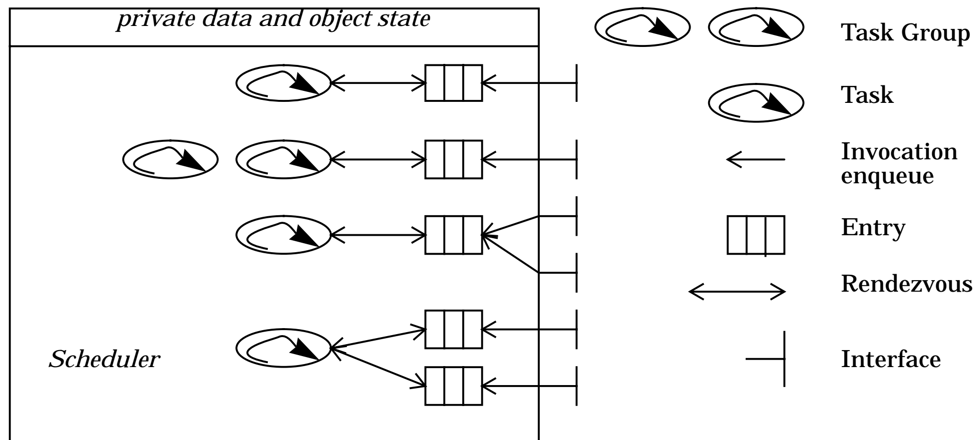
In Figure 5.1, a graphical illustration of a real-time object is given.

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

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

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

Figure 5.1: Real-time object illustration



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 §5.7.2 *Resource allocation and task preemption*).

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.

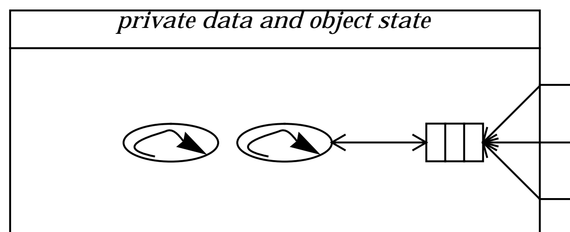
The user control over system tasking behaviour is further enhanced by the scheduling policy/mechanism separation used in the environment (detailed in §5.6 *Scheduling*).

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

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

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

Figure 5.2: Shared Single Entry (ANSAware) Configuration



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

Figure 5.3: Multiple Single Entries

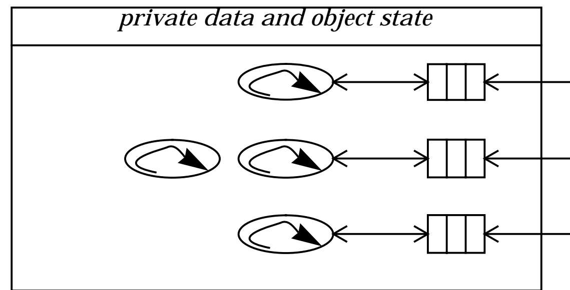
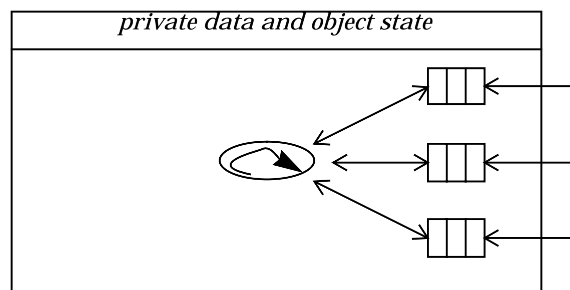


Figure 5.4: Single Task Multiple Single Entry



Another interesting simple form (Figure 5.4) is *Single Task Multiple Single Entry*, in which the single task decides at run-time which entry (interface) it would like to serve.

A combined configuration is illustrated in Figure 5.1. It contains the three simple configurations.

5.5 Real-time object invocation

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

To support mission-critical requirements, there must be some means to enable the urgency of a computational activity to be communicated between the nodes on which it executes. Such information is required 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 architecture by associating 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 on and becomes a property of the thread, which may then be used as a scheduling parameter by the server.

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 raise 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 emphasised 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.

5.6 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. This diversity has already lead to various real-time programming models for specific applications. Therefore, an ideal general purpose real-time support environment should provide multiple models of real-time programming. This is supported by multiple application-selectable scheduling policy modules on top of a shared set of scheduling mechanisms.

The system scheduling behaviour is defined in layers as:

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

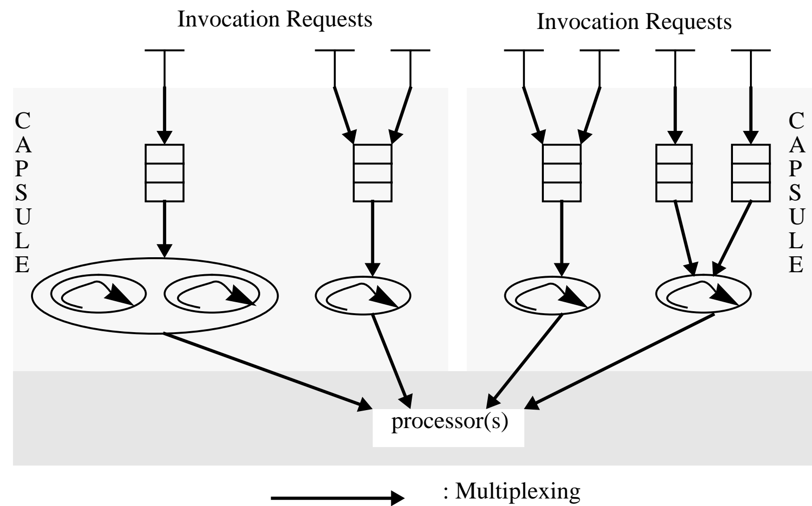
Task scheduling and thread scheduling are two separate, but related scheduling domains. Task scheduling is defined in the nucleus or the underlying operating system kernel. Thread scheduling is defined per entry. Task scheduling manages multiplexing of task executions over processor(s). Thread scheduling manages the multiplexing of requests (thread) over tasks. Figure 5.5 illustrates the structure of this multiplex.

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.

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

Figure 5.5: Threads, Tasks and Processor(s) Multiplexing



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 (assuming that the first thread in the queue is executed first). Such a policy may be a system defined one, like invocation priority based, invocation deadline based, or an application provided one.

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

5.7 Priority scheduling

This section discusses the mechanisms needed to provide a *priority based* scheduling model in the real-time ANSA architecture. 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* (§5.7.1 *Priority management and priority inheritance*) and *priority ceiling* (§5.7.3 *Dealing with priority inversion*) --- used when synchronization is enforced during a task and thread rendezvous --- is needed to understand how priority works on tasking.

5.7.1 Priority management and priority inheritance

A distinction is made between a task's ***static*** priority (that declared in its creation) and its ***dynamic*** priority (that is the static value potentially enhanced by a rendezvous or an explicit change of priority). It is the dynamic

priority that is used by the nucleus (or operating system) schedule to determine the current system-wide *urgency* of a task.

The tasking model is designed to support a structured approach to priority management. Statically, the different task/entry/interface configurations allow 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 can take into account the priority of an invocation, and use this priority as its dynamic priority. This is called ***priority inheritance***.

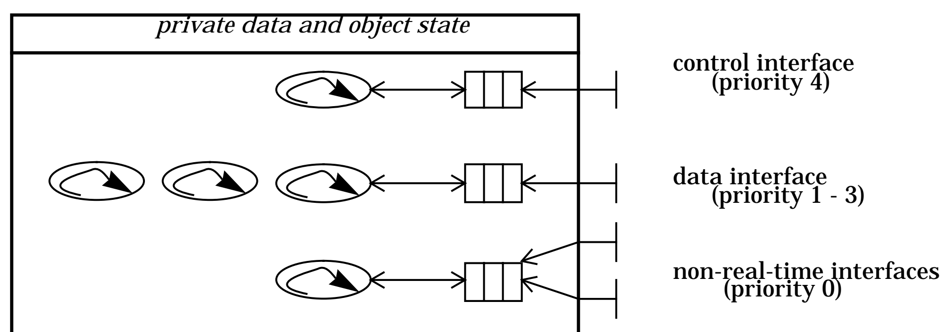
Two levels of priority inheritance schemes may be defined. They are called (basic) ***priority inheritance*** and ***transitive priority inheritance.*** In the first scheme, a serving task with a low priority raises its priority to the higher priority of an invocation request before it starts the service,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

5.7.2 Resource allocation and task preemption

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

In the real-time ANSA architecture, 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.

Figure 5.6: Layered Management of Priorities



One may allocate different levels of priorities to different real-time services, while priorities in one level may be used to identify the relative importance of an invocation among all the invocations on one interface. In Figure 5.6, 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 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.

5.7.3 Dealing with priority inversion

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

Figure 5.7: Priority Inversion in ANSAware/RT Objects

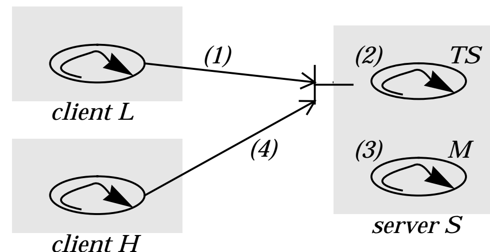


Figure 5.7 shows an example of priority inversion in real-time ANSA objects. Suppose there is a server object *S* with an interface *I* and client objects *L* and *H. L* is a low priority client --- it runs a low priority task which sends low priority invocations to *S. H* is a high priority client --- it runs a high priority task which sends high priority invocations to *S. S* has a task *TS* for serving invocations on *I.* Moreover, *S* has another middle priority task *M* running independently.

Priority inversion happens if the following sequence of actions appears:

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

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

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

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

5.8 Deadline scheduling

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

A simple deadline scheduling policy is to treat deadlines as priorities in thread queuing. An earlier deadline has higher priority than a late one. Let's call it *deadline based* thread scheduling. It is not assumed that the task scheduler (i.e. operating system scheduler) understands deadlines. The resultant behaviour is *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 requirement suggests that the RPC protocol --- the Remote EXecution protocol in ANSAware --- should take deadlines into account. Timed RPC is discussed in the next chapter.

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

One way to improve the robustness of a timed RPC protocol for real-time applications is to ask the scheduler to provide an early acknowledgement to the client. The server thread scheduler checks its local schedule information to decide if it is possible to execute a request within its deadline. The decision must take into consideration the invocation communication delay, the invocation demand of the processor, and the server load. If the acknowledgement is positive and received before a timeout value of the client, the client will wait for the final result. Otherwise, the client may consider the invocation unsuccessful and start to take necessary alternative actions. Although using the early acknowledgement does not actually increase the probability of invocation success, it will give the client more time to recover from the timing error.

5.9 Other scheduling paradigms

Priority and deadline scheduling can be combined to provide alternative scheduling models. One combination is *priority first, and then deadline based,* in which deadlines are only used to break the tie when two 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.

5.10 Application controlled rendezvous

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

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

The effect is that the thread waits to serve one request on any entry in the entry_set, provided that this occurs before the timeout expires.

The application controlled rendezvous model has the following characteristics:

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

- the application thread may perform its resource management when not responding to external requests. Therefore, it is possible to have interface specific tasks with pre-allocated resources and optimized synchronisation management.

5.11 Summary

This chapter has described a real-time programming model. Its scheduling flexibility has been demonstrated by its two-level scheduling multiplexing. Policy/mechanism separation is used to address the diversity of real-time programming. An integrated priority management scheme is introduced for preemption control.

6 Real-Time Communication System

6.1 Introduction

Real-time applications present more complicated functional requirements to the underlying communication systems than 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 making tradeoffs between functionality and performance. This work provides an opportunity for the inclusion of transport protocols which support QoS management.

6.2 Parallel protocol stacks

The main advantage of the existing 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 there is no interaction between the two modules 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 above problems are addressed by:

- redesigning the MPS interface as connection-based and 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
- extending the REX protocol to use this connection-based interface
- extending the programming interface so that applications have control over these connections.

Figure 6.1: Parallel Protocol Stack

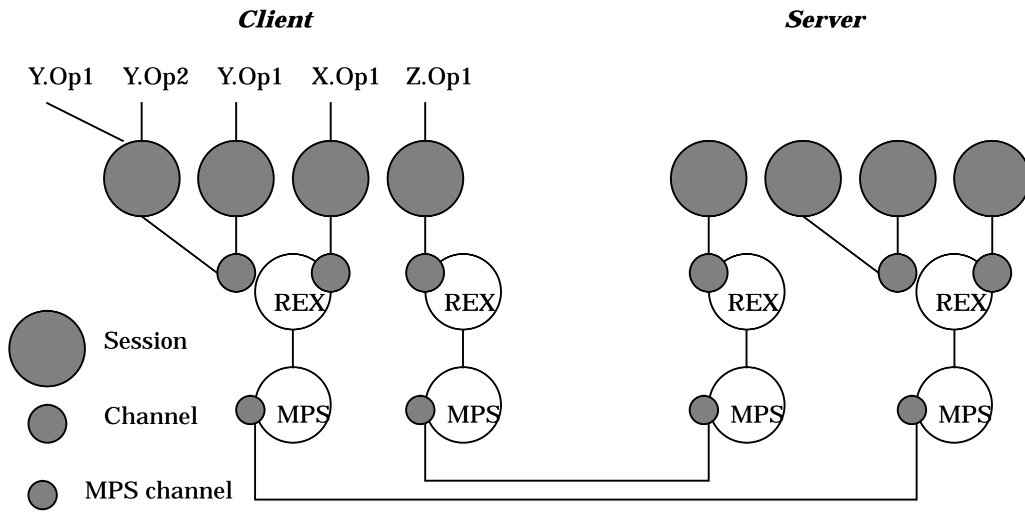
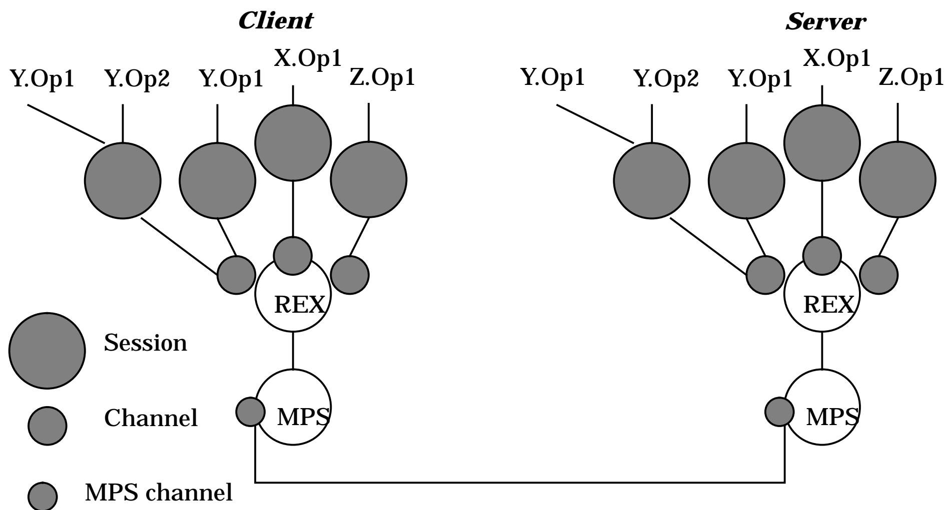


Figure 6.2: Multiplexing in ANSAware



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

6.3 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).

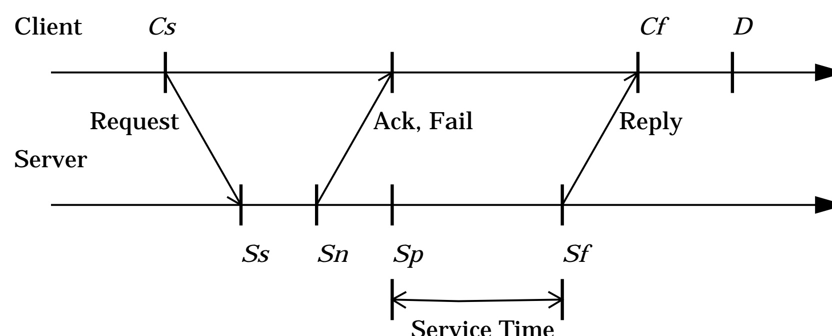
Invocations in ANSAware/RT can attach deadline constraints to their communication requests. Such TRPC calls raise the following three issues:

- 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 is approached by first making a strictly unsatisfiable definition, and then relaxing the definition to lead to a realistic solution.

The TRPC call can be defined as follows. At time *Cs*, the client sends a request with a deadline *D*, which is the latest time the client is willing to wait for successful invocation. At some time *Ss* the server gets the request; the server checks if the deadline can be met, and if it is unsatisfiable a fail acknowledgement is sent back at time *Sn*. Otherwise, the request is accepted and the request is processed at time *Sp*, and a reply is generated at time *Sf* This is illustrated in Figure 6.3.

Figure 6.3: Timed RPC Communication Sequence



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

6.3.1 Discussion of problem

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

1. to establish a bound on the time at which the delay in awaiting a TRPC call expires
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 a parameterised generic protocol, allowing different combinations of the parameters to represent different relaxed goals.

6.3.2 The protocol

Because using one deadline value to accomplish the two goals in a TRPC may result in incompatible situations, two arguments --- a *timeout* and a *deadline* --- are used instead. Each is aimed at one goal only. The timeout is used to specify the first goal --- how long the client is willing to wait for its result. It affects a client 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.

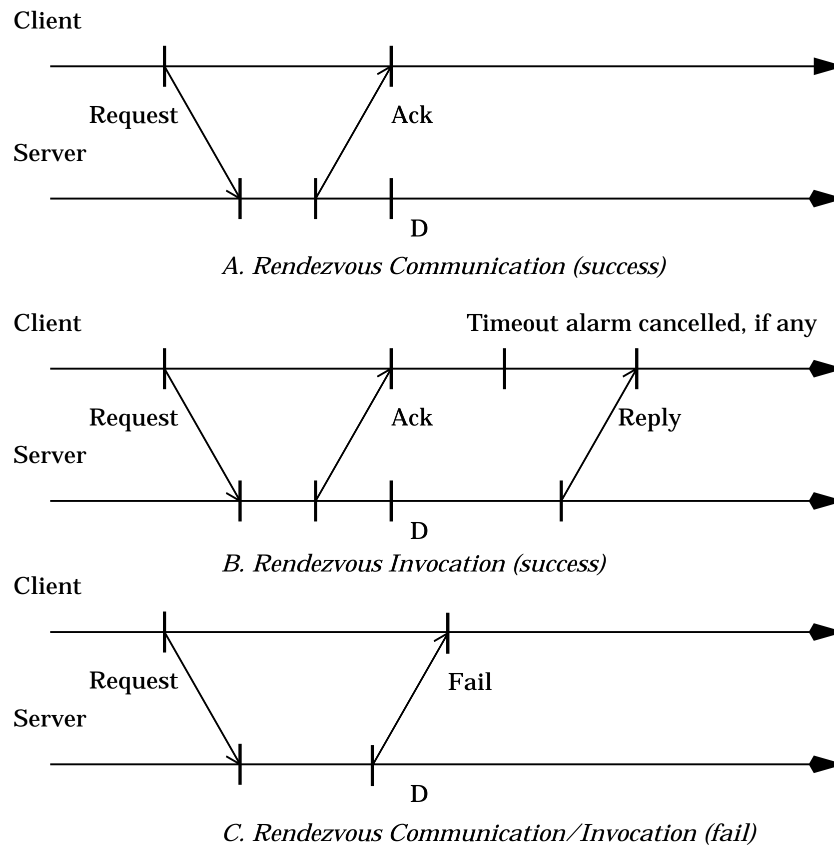
It should be pointed out that using two separate arguments does not solve the TRPC consistency problem. Rather, the two arguments give the problem a more realistic definition, allowing different relaxations be explored.

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 6.4: 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 6.4.

In summary, an invocation may be associated with an optional timeout, an optional deadline, and an optional deadline type. The collective choice of the three parameters determines the behaviour of 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 set the timeout and deadline to the same value. A timeout may be smaller than a deadline, to specify that an acknowledge should be returned earlier; it may be greater than a deadline, to allow the request to have a better chance of success.

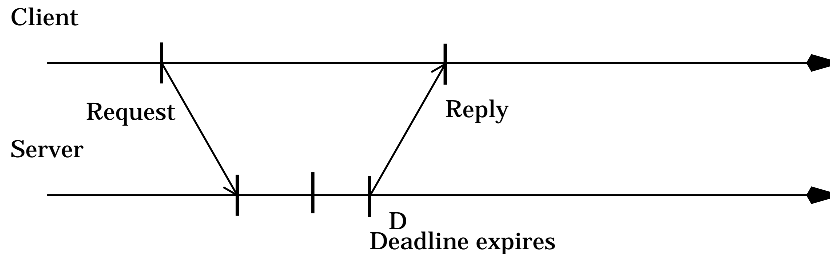
The default deadline type of an invocation deadline is *ServerDetermined* --- it depends on the scheduling policy used in the server to interpret the deadline, and has no effect on the communication protocol.

6.3.3 Server deadline expiry

There may be two types of deadline expiry at a server. One type is defined by the TRPC protocol, as illustrated by the rendezvous communications and

rendezvous invocations. The required semantics are enforced by the communication protocol.

Figure 6.5: Server Thread Deadline Expiry



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 termination. This is illustrated in Figure 6.5.

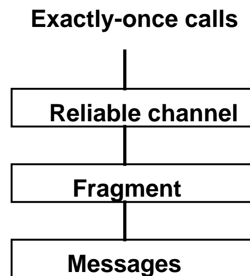
6.4 Decomposable RPC protocol

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 a 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, the alternative of an integrated design is more interesting for the purposes of better structure, flexibility and efficient coding. This introduces the requirement for a decomposable RPC protocol.

The ANSAware REX service provides exactly-once semantics of RPC calls. REX is decomposed into three layers as illustrated in Figure 6.6. The three layers share the same protocol data structure (sessions) and provide a single protocol service.

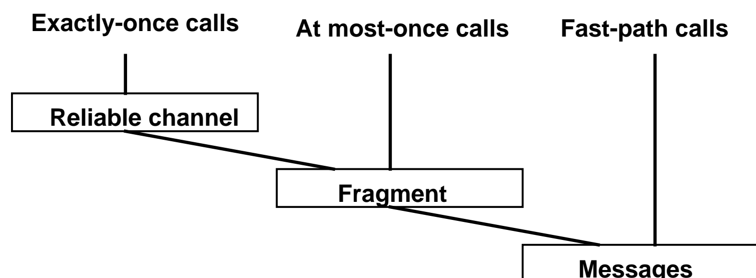
Figure 6.6: REX Functional Layers



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 6.7. In addition to the exactly-once service, two other services, the at-most-once service and fast-path services can be provided. The multi-transport service protocol is still one execution protocol in the ANSA sense. But it provides additional call semantics.

Figure 6.7: 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.

6.5 Summary

Real-time applications present more complicated functional requirements to the underlying communication systems. This chapter discussed 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 Application Programming

7 ANSAware/RT Application Programming

7.1 Introduction

This chapter is intended for applications programmers wishing to make use of the new real-time facilities and describes the extensions to the programming interface. It begins by reviewing the current ANSAware 4.1 programming interface and moves on to describe the changes and new interfaces.

7.2 ANSAware 4.1 programming interface

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

- interface instances, PREPC provides a generic statement for dynamically creating interface instances
- object invocation, PREPC provides a generic statement for object invocations.
- thread spawn, threads can be explicitly created by calling either a fork or a spawn function
- task creation, tasks can be added dynamically by calling the `nucleus_tasks` function

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 [APM92][APM.AR.01]).

7.3 Attributes objects

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

7.3.1 Task attributes object

A task attributes object is used to specify the properties of subsequently created tasks. The following attributes are applicable to tasks:

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

To create a task attributes object, the following function is used

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

This creates a task attribute object containing default values for each of the individual attributes.

To modify any attribute values in a task attributes object, use one of the following

```
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 the appropriate routine from the following list

```
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)
```

7.3.2 Entry attributes object

An entry attributes object is used to specify the properties of a subsequently created scheduling entry.

The following attributes apply to entries:

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

The routines to manipulate entry attributes objects are similar to those of the task attributes objects as already described.

```
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)
```

7.3.3 Thread attributes objects

Thread attributes objects are used to specify the properties of subsequently created threads. The properties that apply to threads are:

- Priority
- Deadline

The associated routines are as follows:

```
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)
```

7.4 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 in ANSAware/RT is provided by:

- task scheduling policies
- task priorities
- entry queuing policies
- entry priorities
- entry rendezvous policies
- allocation of tasks to entries
- explicit binding of service interfaces to entries.

The above seven forms of real-time functions allow for considerable control over application execution. At run time, the combination of these real-time features gives the application 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.

7.4.1 Explicit task creation

ANSAware/RT tasks can be spawned for two purposes:

- to process the requests on an entry

- to initiate an independent thread

In the first case, tasks are allocated to an entry by

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

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

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

7.4.2 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 overall 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 policies 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.

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

Detailed discussion of the relationships between scheduling policies, priority ranges and operating system processes can be found in a p-thread manual or in the POSIX standard [POSIX].

7.4.3 Scheduling Policy environment variable

The default task scheduling policy for tasks within a particular capsule may be specified by setting a new environment variable SCHEDPOLICY. This policy will apply to all tasks for which an explict scheduling policy is not specified by their creator. This environment variable may be set (and exported) before a capsule is created, or may be subsequently set by an application within the capsule.

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

7.4.4 Entry

An entry is a scheduling point by which different scheduling/processing concerns can be identified. Each capsule has a default entry to which all new created interfaces of the capsule are bound. Binding an interface to 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 a specific entry by

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

An interface can revert 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)
```

7.4.5 Entry queuing and rendezvous policies

ANSA threads can be generated in two ways:

- by an explicit spawn operation performed by an application

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

- implicitly by the infrastructure. For each RPC invocation, ANSAware/RT generates a corresponding thread on the entry to which the called interface is bound.

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

ANSAware/RT supports five thread queuing policies (with associated values for an entry attribute object shown in parentheses).

- first-come first-service (AW_E_FCFS): this is the default thread queuing policy
- fixed priority based (AW_E_PRI)
- earliest deadline based (AW_E_DEADLINE)
- priority first and then deadline based (AW_E_PRI_PLUS)
- deadline first and then priority based (AW_E_DEADLINE_PLUS)

Three task/thread rendezvous policies are supported:

- null (AW_R_NULL): the default policy

- priority inheritance (AW_R_PRI): the task inherits the priority of the thread it is serving
- priority ceiling (AW_R_CEILING): the task inherits the priority ceiling value of the entry

7.4.6 Rendezvous

ANSAware/RT also extends the ANSAware 4.1 tasking system to allow stackable execution of threads. This permits a thread, while it is executing (has a task allocated), to wait at an entry to rendezvous with 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)
```

7.4.7 Examples

The following examples are fragments of PREPC code intended to demonstrate how a combination of the scheduling interfaces already described, may be used to achieve specific behaviour. Complete PREPC examples, which may be compiled and run, can be found in the distribution under master/src/examples.

7.4.7.1 High priority server interface

The following code fragment demonstrates how a server might set up a high priority management interface by creating a new scheduling entry and assigning it a high priority task.

```
/* Create a scheduling entry with default attributes */

status = ansa_entry_create(mgmt_entry, ansa_entryattr_default);

/* Create our management interface (of type Mgmt) */

! { mgmt_ref } :: Mgmt$Create( CONCURRENCY_ONE )

/* Bind the new interface to the entry just created */

status = ansa_entry_bind(mgmt_ref, mgmt_entry);

/* Create a task attributes object and set up its values:
 * FIFO scheduling, high priority
 */

status = ansa_taskattr_create(&attr);
status = ansa_taskattr_setsched(&attr, AW_SCHED_FIFO);
status = ansa_taskattr_setprio(&attr, HIGH_PRIORITY);

/* Assign a task to the entry using the attributes object */

status = nucleus_tasks_onentry(mgmt_entry, attr, 1)
```

7.4.7.2 Task creation and rendezvous

The following example shows how a server might create an autonomous task to processes invocations on a specific entry.

```
void body(int argc, char *argv[], char *envp[])
{

     /* Create a scheduling entry with default attributes */

     status = ansa_entry_create(mgmt_entry,
          ansa_entryattr_default);

     /* Create our management interface (of type Mgmt) */

!    { mgmt_ref } :: Mgmt$Create( CONCURRENCY_ONE )

     /* Bind the new interface to the entry just created */

     status = ansa_entry_bind(mgmt_ref, mgmt_entry);

     /* Create a rendezvous task */

     status = ansa_task_spawn(rendezvous_task, arg,
          ansa_taskattr_default);
}

/* Define routine for the autonomous task to process */

void
rendezvous_task(int i)
{
     /* Loop forever, processing invocations on mgmt_entry */

     for {;;}
     {
          status = ansa_rendezvous(mgmt_entry, TIMEOUT);

          /* Do any of our own processing here... */

     }
}
```

7.5 Communications

Real-time applications made specific demands on the communication system in terms of timeliness requirements. These requirements are often referred to as a need for a specific Quality of Service (QoS). ANSAware/RT introduces support for the specification of communications QoS through an extension to the existing PREPC invocation statement and by means of a new explicit binding operation. The former is used to specify in-band QoS requirements for each invocation, whilst the latter supports endpoint QoS which applies to all invocations on a communications channel.

7.5.1 QoS object

A QoS object consists of individual attribute values and is introduced to describe communication resource and performance constraints. QoS objects are used by applications and are associated with the existing PREPC invocation statement and with the new explicit binding operation. Two categories of QoS object are defined in ANSAware/RT:

- endpoint QoS objects: to describe QoS constraints associated with a communication endpoint which could be either a server endpoint (socket) or a client endpoint (plug)
- in-band QoS object: to select the in-band QoS parameters of an established communication channel.

7.5.1.1 *Endpoint QoS*

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:

- EX protocol: this can be either the standard ANSAware REX protocol (compatible with ANSAware 4.1) or the new TREX protocol
- MPS protocol: ANSAware/RT supports either IPC or UDP (but not TCP)
- control parameters specific to an individual protocol. For example using UDP, an application can choose a specific port number, and spawn a specific p-thread for managing the communication on that port.

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)
```

7.5.1.2 *In-band QoS*

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

- for the REX protocol, these can be a timeout value and error retry number
- for the TREX protocol, these can be a 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.

Routines for manipulating the in-band QoS object are similar to those for the endpoint QoS object:

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

7.5.2 Explicit Binding

ANSAware/RT introduces the explicit binding operation to

- associate QoS with communication endpoints
- control binding time

A server accomplishes explicit binding at service creation time, using the PREPC statement

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

where {QoS} denotes an optional endpoint QoS object that must have been created by an earlier call to ansa_endQoS_create.

This operation creates a service instance and sets up the service communication endpoint with the required QoS constraint. The explicit binding operation is combined with service instance creation because the QoS constraint may affect the construction of the interface reference ir.

Without the QoS parameter, the creation operation will use the default implicit binding operation, which means that the capsule only ensures a minimum communication QoS (use the maximum amount of multiplexing, for example) for the service instance.

A client performs explicit binding having first obtained a server interface reference ir (using the trader import operation for example), by using the extended PREPC statement:

```
{} :: 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 existing ANSAware.

If the required QoS cannot be provided, the explicit binding operation will return a new exception noBinding.

It should be noted that the TREX protocol requires the explicit binding operation to be performed, before any interaction can take place. In other words, a real-time invocation cannot be performed before a real-time communication channel is explicitly established.

A server may destroy its explicit binding and release the associated resources by using the following PREPC statement

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

The corresponding client operation is

```
ir$Discard
```

7.5.3 Invocations specifying in-band QoS

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

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

where {QoS} is an optional in-band QoS object previously created by a call to ansa_invQoS_create.

This allows the association of invocation dependent control parameters, for example, priority or deadline.

7.5.4 Clock synchronisation

The deadline associated with an invocation implies that both server and client share a common view of time. As there are many clock synchronization mechanisms and systems, it would be inappropriate to build the clock synchronization mechanism into the capsule. Consequently, ANSAware/RT provides only the minimum functionality for clock reset and relies on

applications (by providing an ANSAware service, for example) to provide clock synchronization appropriate to their needs.

The nucleus provides two functions:

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

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

7.5.5 Examples

The following examples are fragments of PREPC code intended to demonstrate how a combination of the communications interfaces already described, may be used to achieve specific behaviour. Complete PREPC examples, which may be compiled and run, can be found in the distribution under master/src/examples.

7.5.5.1 Explicit binding and endpoint QoS

The following code fragment illustrates how a server may create an interface that will request a new socket using the new timed execution protocol (TREX).

```
void body(int argc, char *argv[], char *envp[])
{
     ansa_EndQoS qos;

     /* Create an endpoint QoS object */

     status = ansa_endqos_create(&qos);

     /* Set the protocol to be TREX */

     status = ansa_endqos_setprotocol(&qos, TREX_UDP);

     /* Make sure that we get a new transport connection*/

     status = ansa_endqos_setudp_new(&qos);

     /* Create the service interface with the specified QoS */

!    { ir } :: Service$Create( CONCURRENCY ) {qos}

     /* Export the service or whatever ... */
}
```

The client code for using this interface would be similar except that the binding is established using:

```
status = Service$Bind( ir ) {qos}
```

7.5.5.2 Invocations and in-band QoS

The following code fragment demonstrates how a deadline may be associated with an invocation.

```
void body(int argc, char *argv[], char *envp[])
{
     ansa_InvQoS invqos;

     /* Create an in-band QoS object */
```

```
        status = ansa_invqos_create(&invqos);

        /* Set a hard deadline value */

        status = ansa_invqos_setdeadline(&invqos, D_CallHard,
             DEADLINE_MILLISECS);

        /* Assuming we have already obtained an interface reference
         * (ir) for a service, an invocation of operation op1 is made
         * as follows.
         */

!       { result } <- ir$op1( arg ) {invqos}
}
```

8 API 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

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 format of time is a structure defined in capsule/timer.h and has the same semantics as in previous versions of ANSAware.

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

Endpoint QoS object

PURPOSE

To specify an endpoint 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

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

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

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

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

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 specified in microseconds althought the resolution is system dependant.

SEE ALSO

Entry.

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

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

Scheduling policy 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.

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.

ANSAware/RT System Programming

9 Tasking and Scheduling Implementation

9.1 Introduction

This chapter, together with §10 *Implementation of the Communication System*, are intended for systems engineers wishing to understand the implementation details of ANSAware/RT. In general these are described as changes from ANSAware 4.1 implementation, and although a brief review of the latter is included, some familiarity is assumed.

9.2 Review of tasking in ANSAware 4.1

ANSAware 4.1 threads represent paths 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 each communication endpoint, a time thread to execute time-related activities, and an initial 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:

- uses 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 is 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 in a reduction in size of ANSAware processes
- uses 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 switching and fast procedure execution (i.e. there is no need to pass context information through procedure parameters).

- shared data area may be accessed without a synchronization mechanism. The ANSAware task scheduler is non-preemptive, a task, while executing, will not relinquish control until it blocks or terminates. Therefore, on a single processor, exclusive access of the shared data is guaranteed and there is no need to prevent concurrent access by other threads.

9.3 ANSAware/RT tasking

In ANSAware/RT, each task is mapped onto a p-thread, and task scheduling is performed 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 is thus made redundant.

9.3.1 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.

9.3.2 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 4.1. Thread private state is used frequently in both application program and ANSAware operations.

In ANSAware/RT, global data needs to be protected by a synchronization mechanism. Therefore, ANSAware thread private data could introduce a significant performance overhead if no change to ANSAware was 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 via the `pthread_getspecific` procedure without a synchronization operation.

Thread private state is held as part of a task's private data area. When a task is created, 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.

9.4 Stacked threads

ANSAware 4.1 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 executing. This stackable thread mechanism is implemented by pushing the thread private state area onto the

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

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 restores the thread private state from stack when the new thread finishes.

9.5 Threads

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.

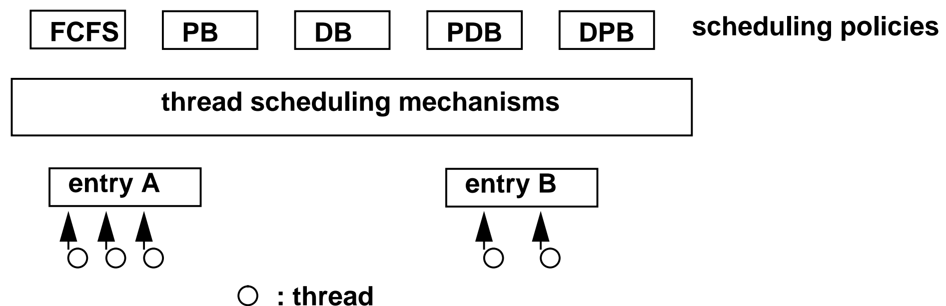
In ANSAware 4.1, a new thread is queued on the capsule-wide FCFS thread queue, waiting to be executed by a free ANSA system task.

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.

9.6 Entry

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

Figure 9.1: Thread scheduling: policies and mechanisms



A clea separation is made between policy and mechanism to ensure 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 9.1 illustrates the design.

Each entry is also associated with a rendezvous policy. Each 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.

9.7 Synchronous I/O

ANSAware 4.1 assumes a totally asynchronous I/O model because

- it allows tight coupling of communication scheduler and task scheduler for efficient scheduling of activities within ANSAware.
- it prevents the entire capsule from blocking as a result of a single thread performing a 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 the p-thread implementation of the tasking system, the asynchronous I/O model is no longer necessary because:

- task scheduling is done by the underlying operating system: there is no tight integration of tasking scheduling and communication scheduling
- a capsule will not block when a thread performs 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, the pin interface, the non-blocking keyboard input library and the X11 library are not present in ANSAware/RT and applications must be modified to use the equivalent synchronous I/O operations.

9.8 Communication tasks and system tasks

In ANSAware/RT 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 associated 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 9.2.

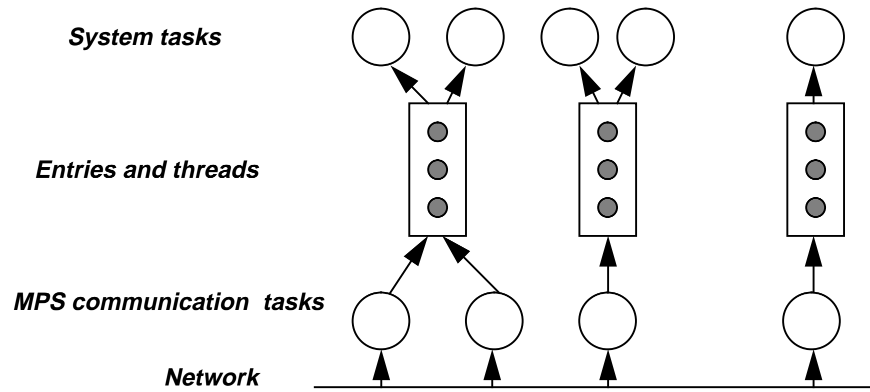
When a thread makes a synchronous invocation of a server, it blocks on a condition variable which is located in the task's private data area. When a reply arrives and is processed by a communication task, the condition variable is signalled, waking the calling thread.

9.9 Other ANSAware/RT tasks

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).

Figure 9.2: Communication tasks and system tasks



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

10 Implementation of the Communication System

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

10.1 Review of the ANSAware 4.1 communication system

The ANSAware 4.1 communication system implements four protocol layers:

- MPS: an interface to the transport protocols provided by the underlying operating system
- EXecution protocol: implements the invocation of ANSAware operations. ANSAware 4.1 supports the REX protocol 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 tasking and communication systems
- Stubs: for marshalling host language level variables into (and out of) linear communications buffers.

The ANSAware 4.1 communication design is optimised for efficient resource utilization by multiplexing the channels provided by each of the above to those of the next layer.

Real-time communication is not a concern of ANSAware 4.1.

10.1.1 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.

10.1.2 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 onto MPS endpoints whenever possible (on a capsule wide basis).

10.1.3 EXecution protocols

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

10.1.4 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 a 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.

10.1.5 Bindings

A service provider fabricates an *ifref* upon an interface creation operation. A client, having first obtained 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 MPS's in the server to form the interface reference.

Client side binding (the creation of a plug) is performed onthe 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 rebind.

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.

10.2 QoS and explicit binding in ANSAware/RT

QoS objects are introduced to express specific communication requirements 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 implicit binding, the *ifref* created by an explicit binding contains a single explicit address (deduced from the QoS), rather than the default list of possible addresses 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 to the plug cache, so that later calls on the interface are guaranteed an established channel.

10.3 State-based MPS

The MPS interface is extended to be state-based: 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 , client explicit binding and binding release. The original message *send* and *receive* operations are also extended to make use of the binding information.

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

10.4 State-based EXecution protocol

The EXecution protocol is modified both to be state-based and to use the state-based MPS. This allows extra binding information to be added 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 , client explicit binding and binding release. The original message *send* and *receive* operations are also extended to make use of the new binding information.

10.5 TREX

The generic design of the binding and state-based protocols allows the addition of new EX protocols. The timed RPC (see §6 *Real-Time Communication System*) 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 which significantly reduces the size of the protocol
- at-most-once semantics, no timeout controlled retry
- no security checking and hence no passing and checking of nonce
- smaller header size, resulting from the above three design decisions

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 the client and deadline expiry at the server

- 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.

10.6 In-band QoS

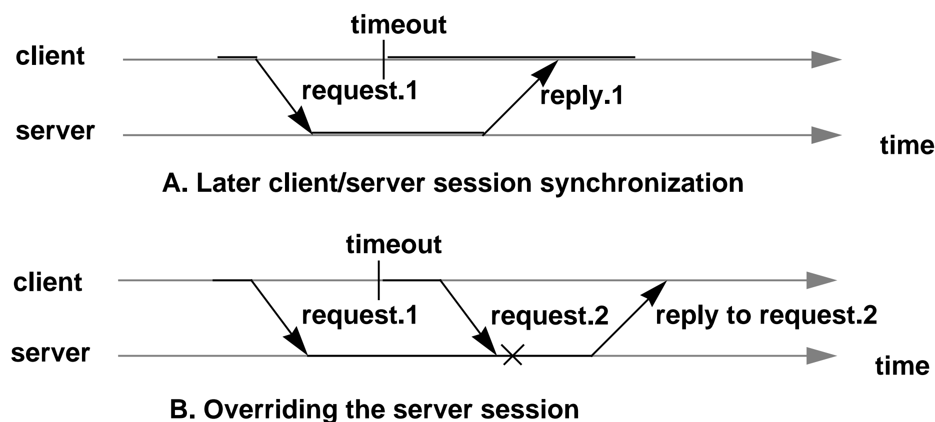
In-band QoS can be associated with each invocation to select the dynamic QoS parameters of a channel. In ANSAware/RT, if a channel uses TREX as its EX, the in-band QoS can select a priority, a timeout, a deadline and a deadline type.

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

10.7 Session timeout recovery

Client tiimeouts are a mixed blessing: the desired semantics of a timeout is for the client to resume control (so that the client can take some immediate recovery actions). However, the operation may continue in the server 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 adopted.

Figure 10.1: Session timeout recovery



The ANSAware/RT approach for session timeout recovery is illustrated in Figure 10.1. 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. This 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 undertaken then. The approach also allows the server end of a session to be aware that its client may time out, and that the client end of the session may be used for another invocation. A possible effect (caused by the ANSAware/RT 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. In this case, the reply to the original (now obsolete) invocation will

not be sent, the next reply on the session corresponding to the most recently received request.

10.8 Miscellaneous communications details

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

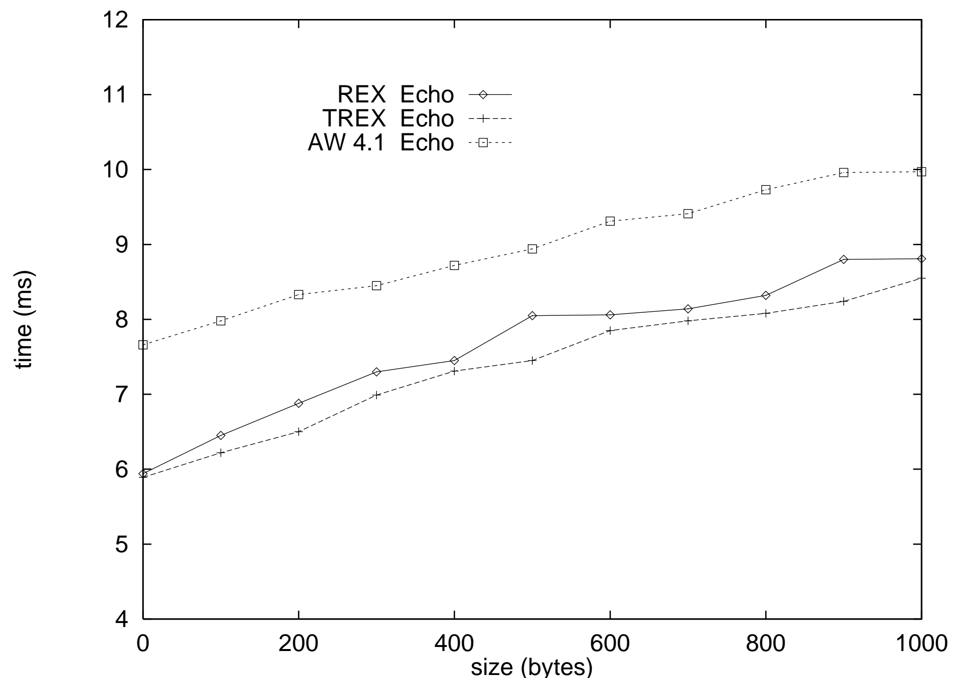
11 Performance Measurements

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.

11.1 Basic performance

Figure 11.1 shows the performance of ANSAware/RT 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 11.1: 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.

11.2 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 purpose 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.

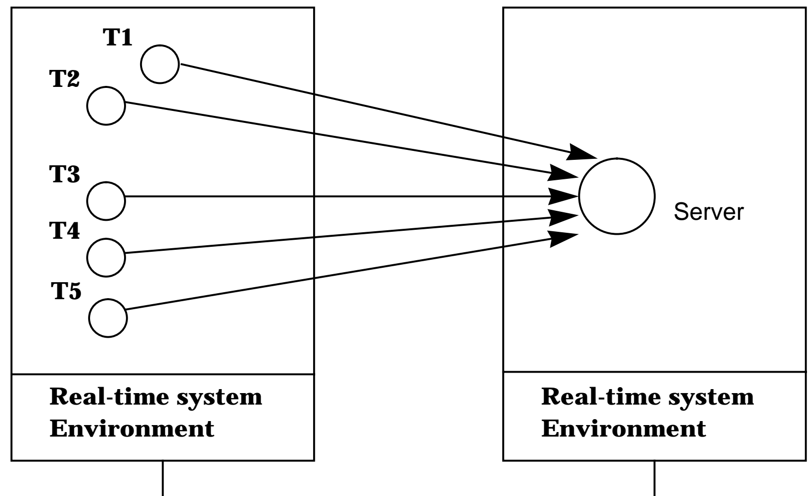
The DSHmc series is intended to stress the media contention algorithm. This series is not applicable to the test environment (the Ethernet hardware has no support of prioritised packets), and therefore is not described below.

11.2.1 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 based on 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 11.2 shows the structure of the DSHcl series task set. Table 11.1 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 computation time of the server combined with the time the request message is in transit causes deadlines to be missed.

Figure 11.2: 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 11.1: 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

11.2.2 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 based on 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 11.2 gives the timing requirements of the DSHpq series baseline test task set.

Table 11.2: DSHpq series task set

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

Table 11.2: DSHpq series task set

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

11.2.3 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 based on 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 11.3: N clients with multiple servers

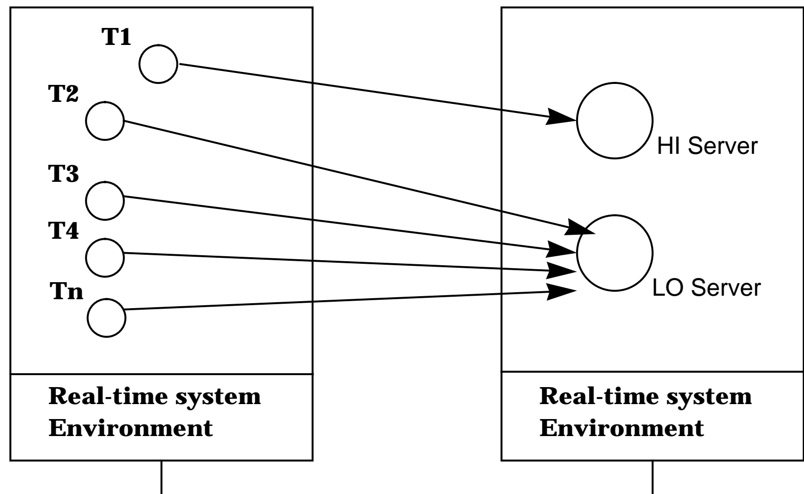


Figure 11.3 shows the structure of the DSNpp series task set. Table 11.3 gives the timing requirements of the baseline test task set.

Table 11.3: 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 11.3: DSHpp series task set

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

11.2.4 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 little computation time. Client tasks T1,..., Tn send requests to the server at the beginning of their periods and they consume little 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 11.4: 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 11.4: N clients with a single server

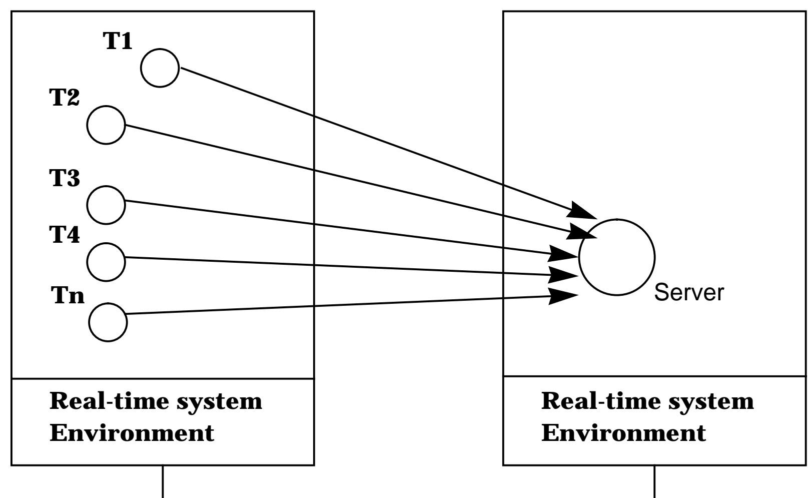


Figure 11.4 shows the structure of the DSHcb series task set. Table 11.4 gives the timing requirements of the baseline test task set.

11.2.5 Result

The benchmark results of ANSAware/RT over the OSF/1 kernel are presented in Table 11.5. Performance was measured using two DEC Alpha 3000/300 machines connected by 10 Mbps Ethernet.

Table 11.5: 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

As 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.

As might be expected, ANSAware/RT's performance is considerably better than the kernelized ARTS system, reflecting the superiority of a commercial real-time operating system, a much more powerful processor and a carefully tuned mechanism based on the practical experience of RIDE.

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.