ANSA Phase III

ANSAware/RT Version 1.0: Programming and System Overview

Guangxing Li

Abstract

This document explains how ANSAware 4.1 has been extended to support real-time computing. The major results are (1) an extended ANSAware that runs over a standard real-time environment (POSIX real-time thread package) and maintains the real-time properties of the environment; (2) the provision of selective communication multiplex by QoS specification and explicit binding operations.

This document summarises the programming aspects and   implementation techniques of ANSAware/RT, and gives some performance measurement results based on the Distributed Hartstone benchmark.

ANSAware/RT 1.0 is the baseline for future work on real-time performance management, streams, signals, QoS and explicit binding operations.

APM.1207.00.08	**Draft**	10th April 1995

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

Distribution:
Supersedes:
Superseded by:

ANSAware/RT Version 1.0: Programming and System Overview

ANSAware/RT Version 1.0: Programming and System Overview

Guangxing Li

APM.1207.00.08

10th April 1995

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

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

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

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

Contents

1 Introduction

1.1 Motivation

Distributed real-time computing is spreading. The need of a supporting system environment is inevitable. Even though the distributed computing environment and real-time computing environment are established vehicles, their integration is yet to be researched, because they seldom use compatible techniques.

This document explains how ANSAware (AW) 4.1 is extended to support real-time computing. The modified AW is referred to as ANSAware/RT Version 1.0 or ANSAware/RT in short.

1.2 Goals

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

- compatible with AW4.1
- running over a de-facto industry standard: real-time POSIX threads
- full p-thread real-time scheduling and threading capabilities
- selective communication multiplex by QoS specification and explicit binding operations
- application controlled resource allocation
- supporting the real-time programming model given in [APM.1222]
- comparable to other distributed real-time system environments
- interoperation between different real-time platforms
- interoperation between different real-time and non-real-time platforms

1.3 Audience and context

This document gives programming and system guidance for ANSAware/RT. The intended audience consists of system designers and implementors for distributed real-time system environments with an interest in the issues involved in extending AW. The audience is assumed to be familiar with AW, concurrent and real-time programming in general, p-thread package, and APM.1222 "Some Engineering Aspects of Real-Time".

1.4   Scope of extensions

[APM.1222] explains in detail the rational, concepts and mechanisms for a real-time ANSA system.

ANSAware/RT extends AW4.1 in the following aspects in terms of functional requirements:

- extended tasking system

  — entry: this is a new abstraction which represents a scheduling point. Functions related to entry including creation, release, allocation of tasks, association of interfaces etc. are provided

  — real-time scheduling: preemptive priority-based scheduling

  — multiple scheduling policies: the co-existing of real-time and non-real-time scheduling supports

  — real-time tasks: full real-time p-thread functions.

  — stackable threads: a thread may use its task resource to execute another thread

  — real-time threads: a thread may be associated with a priority and/or deadline

  — multiple thread scheduling policies based on policy/mechanism separation.

- extended communication system

  — multiple execution protocols

  — timed execution protocol: this is a new RPC protocol which understands priority, deadline and deadline types

  — stateful execution protocols and message passing protocols: this allows the preallocation of separate communication endpoints for different interfaces

  — selective multiplex of communication channels

- extended application programming interface

  — abstractions for access tasking resources

  — QoS objects, two kinds of QoS objects are introduced: one for the description of a communication end point, another for the description of in-band QoS requirement for an invocation.

  — explicit binding operations: to bind an interface with a communication channel of a specific QoS

  — invocations with QoS: the attachment of in-band QoS based on invocations.

- extended PREPC and IDL

1.5   Outline

The document is organized as follows:

- chapter 2 outlines the ANSAware/RT programming interface as extensions to AW 4.1

- chapter 3 discusses the extended tasking and scheduling

- chapter 4 presents the extended communication system
- chapter 5 shows the various experimentation and system performance result.
- chapter 6 presents the availability of the system.

2 Application programming interface

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

2.1 AW 4.1 programming interface

AW4.1 provides the following major functions for programming object, concurrency and resource management:

- interface instances, multiple instances of an interface type can be created dynamically

- thread spawn, threads can be created by either a fork or a spawn function

- task creation, tasks can be added dynamically via invoking the `nucleus_tasks` function

- object invocation, PREPC provides a generic statement for object invocations.

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

2.2 Environment variable

Real-time applications should set the SCHEDPOLICY environment variable one of the p-thread supported real-time scheduling policies, which can be SCHED_FIFO or SCHED_RR. Otherwise ANSAware/RT will choose the default time-sharing scheduling policy SCHED_OTHER for its task scheduling.

2.3 Tasking and scheduling

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

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

- choose the priority of each task

- control the allocation of entry or scheduling point --- select the basis where separate scheduling concerns are identified

- select how ANSA threads are queued on an entry for execution --- select a thread queuing policy for an entry

- control the allocation of tasks for each entry

- select how a thread's scheduling attributes may affect a task's scheduling attributes --- select a rendezvous policy of an entry

- control to which entry a service interface be bound.

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

For 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.

2.3.1   Attributes objects

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

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

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

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

- scheduling inheritance

- scheduling policy

- scheduling priority

- stack size

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

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

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

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

To delete a task attribute, use

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

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

The following entry attributes can be changed:

• thread queuing policy

• task/thread rendezvous policy

• priority ceiling value

• priority range values

Routines related to entry attributes are:

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

Routines related to thread attributes are:

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

2.3.2 Entry

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

Additional entry can be created by

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

An interface can bind to another entry by

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

An interface can resume to the default capsule entry by

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

An entry may be closed by

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

2.3.3   Task

ANSAware/RT tasks can be spawn for two purposes:

- to process the requests on an entry

- to initiate a separate execution thread

In the first case, tasks can be allocated by

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

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

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

2.3.4   Task scheduling policies

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

Essentially, there are two categories of tasks:

- time-sharing processing: used for interactive or background work with no critical time limits but a need for reasonable response time and high throughput.

- real-time processing: used for critical work that must be completed within a certain time period.

To control the scheduling policies for the two categories of work, appropriate policy parameter must be selected when creating a task. There are three scheduling policies supported by ANSAware/RT:

- AW_SCHED_OTHER: time-sharing scheduling

- AW_SCHED_FIFO: fixed-priority, first-in first-out preemptive scheduling

- AW_SCHED_RR: fixed-priority, round-robin preemptive scheduling.

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

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

2.3.5   Thread and entry scheduling policies

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

- by an explicit spawn operation

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

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

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

- thread queuing policy

- task/thread rendezvous policy

[APM.1222] defines five thread queuing policies:

- first-come first-service: this is the default thread queuing policy

- fixed priority based

- earliest deadline based

- priority first and then deadline based

- deadline first and then priority based

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

[APM.1222] defines five task/thread rendezvous policies:

- null: the default policy

- priority inheritance: the task inherits the priority of its serving thread

- priority ceiling: the task inheritance the priority ceiling value of the entry

- transitive priority inheritance

- deadline inheritance

- priority and deadline inheritance.

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

2.4   QoS objects

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

- endpoint QoS object: to describe QoS constraint associated with a communication endpoint which could be either a socket (a server endpoint) or a plug (a client endpoint). These QoS objects are used by binding operations to set up a communication channel between a client and its server.

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

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

- an EX protocol: this can be either the standard AW REX protocol or TREX protocol defined in [APM.1222].

- a MPS protocol: currently it can be either IPC or UDP.

- control parameters specific to an individual protocol. For example for UDP, an application can choose a specific port number, and spawn a specific p-thread for managing the communication on the port.

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

- for REX protocol, these can be a timeout value and an error retry number

- for TREX protocol, these can be a priority value, a timeout, a deadline type and a deadline value.

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

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

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

To setup the individual parameter value, use:

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

Similar routines exist for the in-band QoS object:

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

2.5 Binding

ANSAware/RT supports explicit binding operations for

- associating QoS with communication endpoints
- controlling the bind time

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

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

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

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

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

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

This operation will create a client communication endpoint, a plug, with the required QoS. The client can then use the `ir` to invoke the server as is the case without using the explicit binding operation.

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

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

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

The client site operation is

```
ir$Discard
```

2.6   Invocations

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

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

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

2.7   Rendezvous

ANSAware/RT also extends AW tasking system to allow stackable execution of threads. This permits a thread, while executing (holding a task), to wait at an entry to rendezvous and execute another thread (may be an invocation). The benefits 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)
```

2.8   Clock

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

The nucleus provides two functions:

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

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

3 Tasking and scheduling

3.1 AW tasking

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

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

AW 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 AW 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.

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

*   use global, continuous and extensible memory area to store the shared data structures holding the capsule state. AW increases memory for shared data structures in a dynamic manner but requires that the existing memory and the newly allocated memory be contiguous. This requirement has been achieved by copying the existing data to a new location where contiguous memory is available.In this way, memory space is allocated on demand, resulting downsize of AW process

*   use global variables to carry context information. The variables that form the context of an ANSA task are global variables which are shared by all ANSA tasks. Thus, context information is passed to all the procedures through global variables. This allows fast inter task context switch and fast procedure execution (i.e. there is no need to pass context information through procedure parameters)

*   shared data area can be accessed without a synchronization mechanism. AW task scheduler is non-preemptive, a task, while executing, will not relinquish control until it blocks or terminates. Therefore, it guarantees exclusive access of the shared data in a single processor environment, and there is no need for access protection of shared data.

3.2 ANSAware/RT task

In ANSAware/RT, each task is mapped into a p-thread, and task scheduling is done by the underlying operating system. All p-thread attributes also apply to tasks, allowing the exploitation of preemptive real-time scheduling, multiple scheduling policies, kernel supported synchronization objects, task private

data, task exception handling, task synchronous I/O etc. p-thread features. The original AW task schedule was made redundant.

3.2.1 Global data protection

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

3.2.2 Thread private state

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

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

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

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

3.3 Stackable task

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

ANSAware/RT introduces the dynamic rendezvous mechanism: a thread may rendezvous with another thread while execution. The required extensions are to allow a task to execute another thread when it is executing a thread.

This stackable task mechanism is implemented by pushing the thread private state area into the task's stack before executing the new thread (so that the new thread still can use the same task private data as its thread private state), and restore the thread private state from stack when the new thread finishes.

3.4 Thread

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

---

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

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.

3.5 Entry

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

Figure 3.1: Thread scheduling: policies and mechanisms
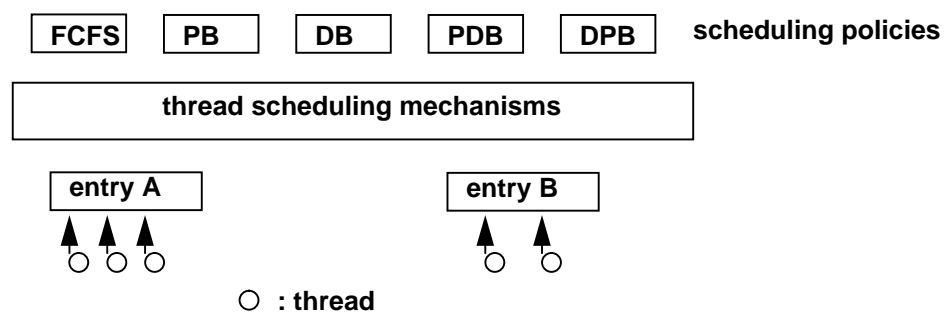
Policy/mechanism separation is used for efficient coding. A common set of thread queuing/dequeuing mechanisms is provided, and on top of the mechanisms a set of scheduling policy objects are placed. Figure 3.1 illustrates such a design.

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

3.6 Synchronous I/O

AW assumes a totally asynchronous I/O model because

• it allows the tight combination of communication scheduler and task scheduler for efficient AW activity scheduling

• it prevents a capsule from blocking because of an otherwise synchronous I/O operation.

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

The asynchronous I/O model in AW is supported by

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

- a non-blocking keyboard input library

- a library for supporting X11 applications.

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

- task scheduling is done by OS, there is no tight integration of tasking scheduling and communication scheduling

- a capsule will not block when a thread is doing a synchronous I/O.

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

3.7   Communication tasks and system tasks

Dedicated communication tasks are spawned to process incoming messages and the corresponding protocol by using of synchronous I/O operations. For each MPS endpoint (a socket), a task is spawned for handling messages from the endpoint. The communication task generates a thread corresponding to each invocation request. The thread is queued on an entry to which the called interface is bound. In this vein, the communication task is actually both a thread generator and a thread scheduler. The threads are executed by system tasks of an entry which are allocated by an application or the capsule. The scenario is shown in Figure 3.2.

Figure 3.2: Communication tasks and system tasks
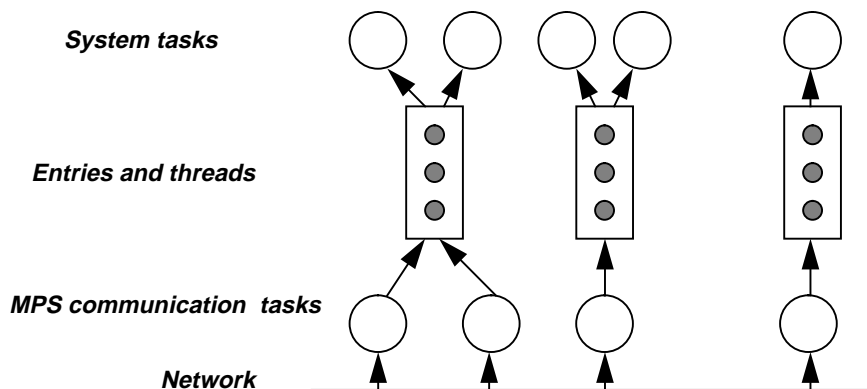
When a thread makes a synchronous invocation to a server, it blocks at a condition variable which is defined on a task's private data area. When a reply is back and processed by a communication task, the condition variable is signalled and therefore the calling thread is woken up.

3.8   Others

A timer task is spawned to process timing functions in the AW Timer Modular.

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

The organization and user interface of the timer task and signal task is similar to the one provided by [APM.TR.037].

4 Communication

4.1 AW communication system

AW communication system implements four protocol layers:

- MPS: an interface to the transport protocols provided by the underlying operating system

- EX: implements the invocation of ANSA operations. AW 4.1 supports the REX for point to point invocations and GEX for group invocations

- Channels/sessions: used to store the end-to-end state required for a remote invocation and to synchronise the execution of the tasking and the communication systems

- Stubs: marshal host language level variables into (and out of) linear communications buffers.

AW communication design is for efficient resource utilization by multiplexing the channels provided by each of these between those of the next layer.

Real-time communication is not a concern of current AW.

4.1.1 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 consists a channel id and the network address of the underlying MPS.

4.1.2 MPS

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

4.1.3 EX

The EX interface is also stateless and there is also no mechanism for QoS based selection/setup of a EX module.

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

4.1.4 Channel and session

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

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

4.1.5   Bindings

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

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

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

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 with in a capsule.

4.2   QoS and explicit binding

QoS objects are introduced to express different communication requirement and are used by explicit binding operations to create different communication channels.

When creating a service instance, a QoS object is allowed to 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* as the result.

In comparison with the implicit binding, the *ifref* created by an explicit binding contains only these information deduced from the QoS, rather than the default information that provides the maximum communicability and also the 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 site binding. The binding operation creates a plug and adds it in the plug cache, so that later calls on the interface are guaranteed an established channel.

4.3   Stateful MPS

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

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

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

4.4   Stateful EX

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

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

4.5   TREX

The generic design of the binding and state-full protocols allows the insertion of new EX protocols. The timed REX (TREX [APM.1222]) is implemented as one example.

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

- no fragmentation, this has significantly reduced the size of the protocol

- at-most-once semantics, no timeout controlled retry

- no security check, no passing and checking of nonce

- small header size, the result of the above three design choices

TREX also extends REX in the following aspects:

- the header of packages is expanded to include the information about the priority, deadline and deadline type

- extended session functions to process timeout at client side and deadline expires at server side

- extra message types for handling deadline exception and confirmation.

TREX supports only explicit binding operations, i.e. interfaces created by implicit binding operations will not be able to use TREX.

4.6   In-band QoS

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

4.7   Session overridden

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

Figure 4.1: Session timeout recovery

A. Later client/server session synchronization

B. Override the server session
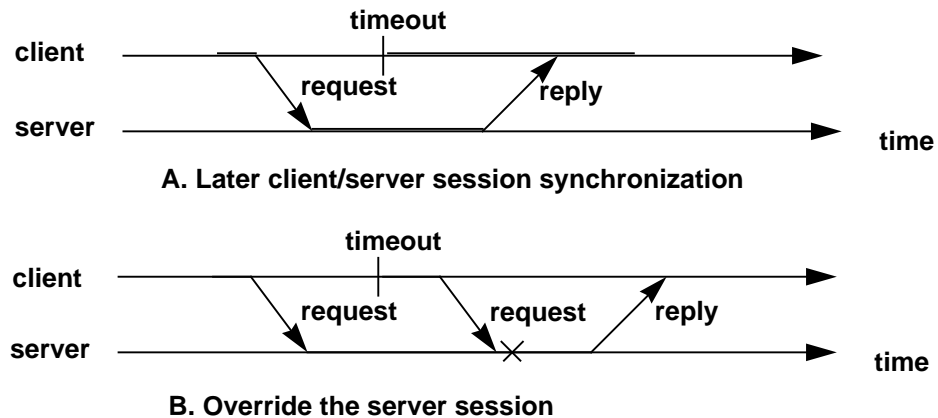
The ANSAware/RT approach for session timeout recovery is illustrated in Figure 4.1. With this approach, the client continues immediately after the timeout, and the client session is set to idle. No synchronisation packet exchange is initiated by the client. It allows the existence of inconsistency between a client session and its server side session. Should the server returns an obsolete result later, synchronisation of the client and server sessions are taken then. The approach also allows the server side session to be aware that its client side may timeout, and the client side session may be used for another invocation. A possible effect (caused by the ANSAware approach to session management) is that a later invocation from the same client side session may override a server side session representing an obsolete invocation.

4.8   Others

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

5 Performance results

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.

5.1 Basic performance

Figure 5.1 shows the performance of ANSAware/RT by using REX and TREX as two transportation protocols. For comparison, the AW.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 5.1: Basic RPC performance
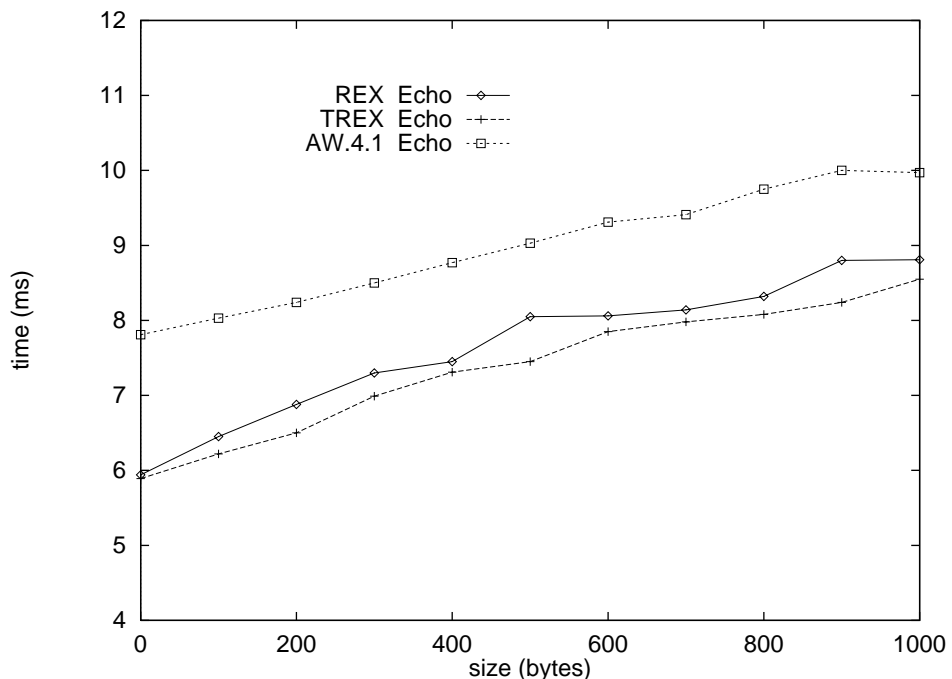
The performance improvement of ANSAware/RT over AW.4.1 seems because

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

The performance improvement of TREX over REX seems because

- TREX uses light weight mechanisms
- TREX uses shorter packet headers

5.2 Distributed hartstone performance

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

DHB was chosen to measure and evaluate ANSAware/RT real-time performance. The intention of DHB is to measure the real-time performance of the processor scheduling, the communication network scheduling and the 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, DSHpq, DSNpp, DSHcb and DSHmc series. The DSHmc series is intended to stress the media contention algorithm. This series is not applicable to our environment (the Ethernet hardware has no support of prioritised packets), and therefore is not described below.
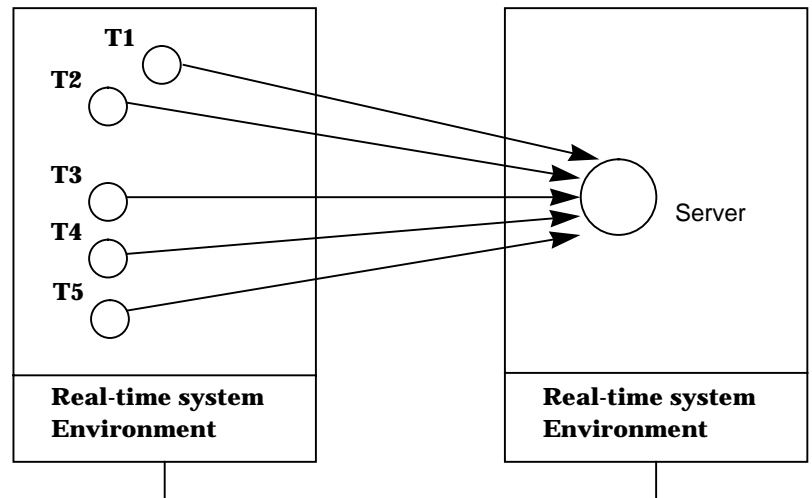
5.2.1 Communication latency

The DSHcl series is a Distributed, Synchronized, and Harmonic task set which tests the communication latency of the system. The base task set is patterned after the periodic, harmonic task set in the original Hartstone benchmark. The task set is extended to have a remote server, and each of the tasks T1,..., T5 sends a request to the server before consuming its own computation time. Figure 5.2 shows the structure of the DSHcl series task set. Table 5.1 gives the timing requirements of the DSHcl series baseline test task set.

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

The task workload is expressed in Kilo-Whetstone (KWS). The Whetstone calculation is the self-verifying version specified by [Curnow76]. A KWS is one execution of a mathematical library, which factors out the effect of a typical

Figure 5.2: Five clients with a single server

arithmetic computing. A task is required to execute a specific amount of KWS within its period.

Table 5.1: 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

5.2.2   Priority queuing

The DSHpq series task set is a Distributed, Synchronized, and Harmonic task set designed to test for priority queuing of communication packets. The base task set is patterned after the DSHcl series. It is quite similar to the DSHcl except for the difference in granularity. The fine-grained DSHcl uses shorter periods for the tasks and milliseconds to measure the server workload. The coarse-grained DSHpq uses longer periods for the tasks and KWS to measure server workload. Table 5.2 gives the timing requirements of the DSHpq series baseline test task set.

Table 5.2: DSHpq series task set

Task	workload (KWS)	Period (ms)
T1	1	160
T2	1	320
T3	2	640
T4	2	1280

Table 5.2: DSHpq series task set

Task	workload (KWS)	Period (ms)
T5	8	2560
T server	variable (KWS)	N/A

5.2.3   Protocol preemptivity

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

Figure 5.3: N clients with multiple servers

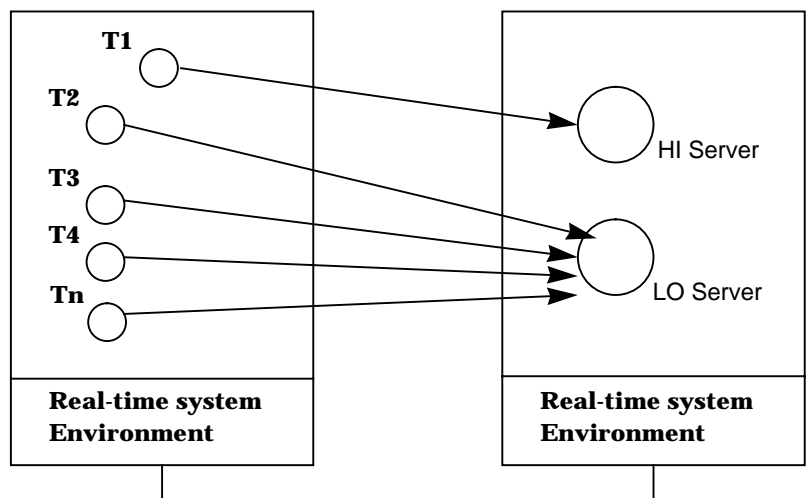Figure 5.3 shows the structure of the DSNpp series task set. Table 5.3 gives the timing requirements of the baseline test task set.

Table 5.3: 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
HI server	0	N/A	high
LO server	0	N/A	low

5.2.4   Communication bandwidth

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

Table 5.4: 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

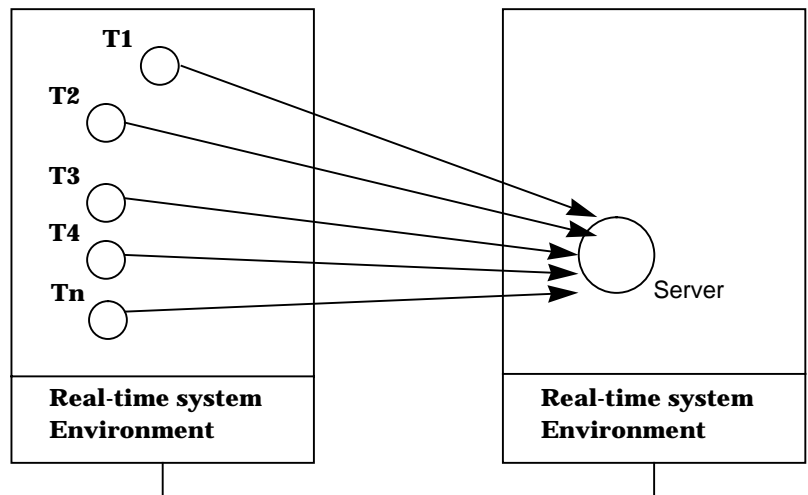**Figure 5.4: N clients with a single server**

Figure 5.4 shows the structure of the DSHcb series task set. Table 5.4 gives the timing requirements of the baseline test task set.

5.2.5   Result

The benchmark results of the ANSAware/RT system over the OSF/1 kernel are presented in Table 5.5. The ANSAware/RT performance was measured using two DEC Alpha 3000/300 machines connected by the 10 Mbps Ethernet.

Table 5.5: ANSAware/RT vs ARTS and RIDE performance

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

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

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

6 Availability

ANSAware/RT 1.0 is currently implemented over the DEC Alpha OSF 1 environment.

Source tree is available via contacting **apm@ansa.co.uk**. It uses the same installation and system build procedure as AW 4.1.

ANSAware/RT 1.0 has full interoperability with AW 4.1.

6.1 Compatibility

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

First, ANSAware/RT dose not support the asynchronous I/O operations of AW 4.1, this requires an application to change the operations to the equivalent synchronous ones.

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

7 Appendix A: ANSAware/RT on HP/RT

ANSAware/RT 1.0 was ported to HP/RT 1.1 to examine the portability of the system.

7.1 System environment

The system environment used for the porting consisted of a collection of HP workstations (as hosts) and target machines, all connected by a 10 Mbps Ethernet, at the BNR Harlow research centre. The workstations are HP 700 series machines running HP/UX 9.0, providing the tools and basic services for development. The targets are HP 700 machines running the HP/RT 1.1 real-time operating system, on which experimentation and evaluation were carried out.

7.2 Porting

The ANSAware 4.1.1 were first installed on the workstations to provide the trading service.

Separate system building procedures were used to create the tools (prepc and stubs) at the hosts first, and then to use them to compile target programs later. This is necessary because the tools and target programs use different compilers and different libraries.

The biggest problems were HP/RT's partial compliance with the POSIX real-time thread standard. For example HP/RT 1.1 has no support for thread-based exception handling, which makes thread releasing difficult. Various HP/RT dependent codes are used to replace the otherwise much more neat POSIX accepting codes.

The overall effort of the porting and performance measurement was 7.5 days.

7.3 Performance

The basic performance is shown in Figure 7.1. The Distributed Hartstone performance is shown in Table 7.1.

Table 7.1: ANSAware/RT on HP/RT performance

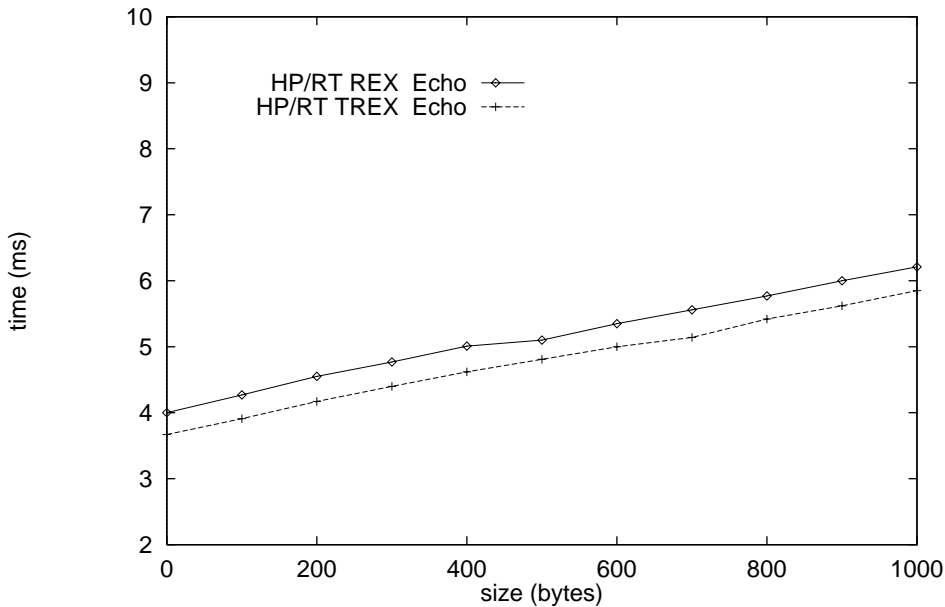| Series	ARTS (Sun 3/140)	ANSAware/RT (HP 700)	ANSAware/RT (DEC Alpha 3000/300)
DSHcl	35 ms	36 ms	41 ms
DSHpq	18 KWS	725 KWS	2010 KWS
DSHpp	(13) 20 tasks	15 tasks	105 tasks

Table 7.1: ANSAware/RT on HP/RT performance

Series	ARTS (Sun 3/140)	ANSAware/RT (HP 700)	ANSAware/RT (DEC Alpha 3000/300)
DSHcb	14 tasks	23 tasks	23 tasks

Figure 7.1: HP/RT Basic RPC performance

The basic performance shows that HP 700 HP/RT's communication performance is better than that of the DEC Alpha OSF1 3000/300's.

The DSHcl performance shows that ANSAware/RT(HP 700) is better than ARTS in real-time communication latency, but less efficient than ANSAware/RT(Alpha 3000/300).

The DSHpq performance shows that ANSAware/RT(HP 700) is better than ARTS in priority queuing of messages, but less capable than ANSAware/RT(Alpha 3000/300). This reflects the differences of processor power of the three systems.

The DSHpp performance shows that ANSAware/RT(HP 700) is about the same as ARTS in handling protocol preemption, but less effective than ANSAware/RT(Alpha 3000/300). This might suggest that HP/RT's thread preemptivity has some potential to improve.

The DSHcb performance shows that ANSAware/RT(HP 700) is better than ARTS in providing real-time communication bandwidth, and is about the same as ANSAware/RT(Alpha 3000/300). This is in consistent with the basic performance result: even through HP 700's processor is less powerful than that of Alpha 3000/300, its communication performance is better than the later.

7.4   Acknowledgement

Martin Howard at BNR Harlow research centre helped the author in the whole process of the porting.

8   Appendix B: ANSAware/RT on LynxOS

ANSAware/RT 1.0 was ported to LynxOS to examine the portability of the system and interoperability of different real-time platforms by using of ANSAware/RT.

8.1   System environment

The system environment used for the porting consistsedof a group of UNIX workstations and a LynxOS machine, all connected by a 10 Mbps Ethernet at APM. The workstations are DEC Alpha 3000/300 series machines running OSF/1 1.3, providing the basic services for development. The LynxOS machine runs LynxOS 2.2 and has a 80486 processor.

8.2   Porting

ANSAware 4.1 was first installed and ported to the LynxOS to provide the installation and system building tools for ANSAware/RT.

The major issues that arose during porting were:

*   LynxOS doesn't provide the standard UNIX interfaces as assumed by the ANSAware installation and system build scripts

*   The ANSAware standard marshalling library for 80486 machines doesn't work for LynxOS

*   LynxOS library maintainer (`ar`) and link editor (`ld`) have unnecessary constraints on the C `main` procedure

*   LynxOS timer system calls have bugs

*   LynxOS p-threads has no support for thread-based exception handling

The overall effort of the porting and performance measurement was two people for ten days.

8.3   Performance

The basic performance is shown in Figure 8.1, this shows only the "loopback" RPC performance of LynxOS.

The Distributed Hartstone performance is shown in Table 8.1. This performance measurement is done by using a Alpha 3000/300 as a server machine and the LynxOS machine as a client machine. It demonstrates the interoperability of different real-time platforms using ANSAware/RT. No changes were made to ANSAware/RT, because the ANSAware/RT priority and deadline programming model has been carefully designed to overcome interoperability problems between p-thread systems. For example, the minimum scheduling priority of the LynxOS real-time scheduler is 0, while it

is 16 for the Alpha OSF/1. Minor changes have been made to the Distributed Hartstone package to explicit use this ANSAware/RT feature.

Table 8.1: ANSAware/RT on LynxOS performance

Series	ARTS (Sun 3/140)	ANSAware/RT/LynxOS (80486 and Alpha 3000/300)	ANSAware/RT/Alpha (Alpha 3000/300)
DSHcl	35 ms	41 ms	41 ms
DSHpq	18 KWS	1376 KWS	2010 KWS
DSHpp	(13) 20 tasks	57 tasks	105 tasks
DSHcb	14 tasks	24 tasks	23 tasks

Figure 8.1: LynxOS (80486) Basic RPC performance (same machine)
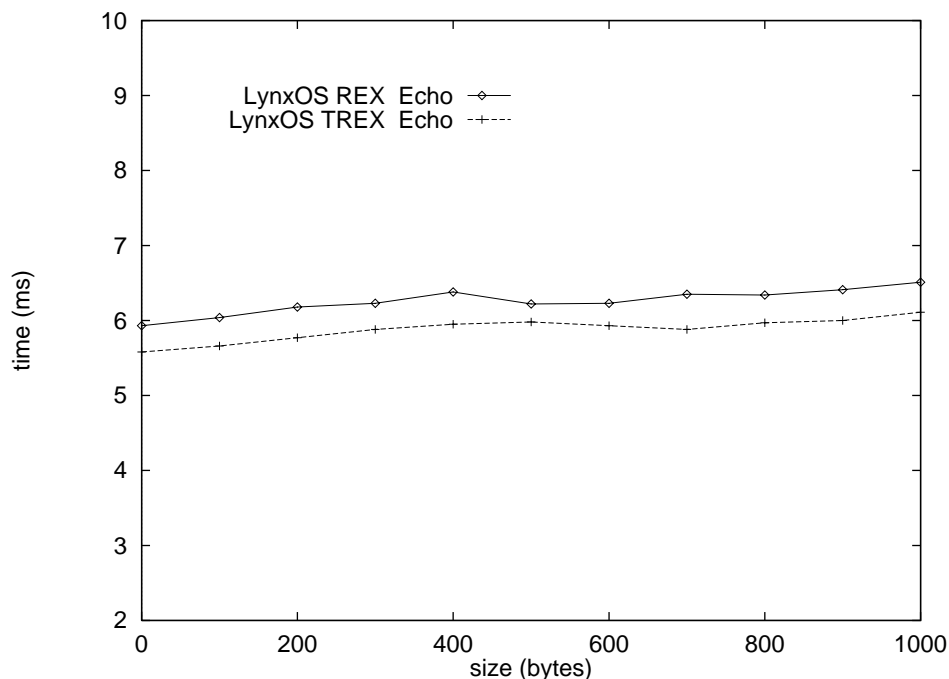
The Distributed Hartstone performance shows the combined performance of a LynxOS 80486 and an OSF/1 DEC Alpha 3000/300 is about the same as that of two OSF/1 DEC Alpha 3000/300 machines in communication latency and communication bandwidth tests. In the priority queuing and protocol preemption tests, the LynxOS/Alpha performance is in between the OSF/1 Alpha performance and ARTS Sun 3/140 performance.

8.4   Acknowledgement

The LynxOS port is done jointly by the author and Malcolm W. Vanston-Rummey of GPT Ltd. The LynxOS machine was provided by GPT Ltd.

References

[APM92]

    APM Ltd., ANSAware Version 4.1 Manual, Architecture Projects Management Ltd., Cambridge U.K., May 1992.

[APM.TR.037]

    C Nicolaou, ANSAware Use of DCE/POSIX Threads and RPC, Architecture Projects Management Ltd., Cambridge U.K., February 1993.

[APM.1222]

    G Li, Some Engineering Aspects of Real-Time, Architecture Projects Management Ltd., Cambridge U.K., May 1994.

[Curnow76]

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

[Kamenoff91]

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

[Li93]

    G Li, Supporting Distributed Real-time 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.

[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.