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

Distributing Real-Time Objects: Some Early Experiences

Guangxing Li

Abstract

A paper to submit for International Workshop on Object-Oriented Real-Time Dependable Systems.

This article discusses the extension of a distributed object-based paradigm - the ANSA Architecture and its example implementation ANSAware - for real-time applications. It reviews some of early experiences in the design, implementation and performance evaluation of two ANSA-based real-time system environments (RIDE and RAW) .

APM.1231.01 **Approved** 25th October 1994

External Paper

Distribution:
Supersedes:
Superseded by:

Distributing Real-Time Objects: Some Early Experiences

Distributing Real-Time Objects: Some Early Experiences

Guangxing Li

APM.1231.01

25th October 1994

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

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

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

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

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

Contents

1 Distributing real-time objects: some early experiences

1.1 Introduction

Neither object orientation, nor distributed computing and nor real-time computing are new ideas, but their integration has never been easy, or yet to be researched. This article discusses the extension of a distributed object-based paradigm - the ANSA Architecture [Herbert94] and its example implementation ANSAware [APM92] - for real-time applications.

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

Distributed real-time processing places unique requirements on systems and their designs (e.g. predictability, programmer control, timeliness, mission orientation and performance) [Li94]. Such features are yet to be provided by the current computing environments.

This paper reviews some of early experiences in the design, implementation and performance evaluation of two ANSA-based system environments for distributed real-time applications. The focus is on what are the required engineering mechanisms rather than how they can be used by application programmers, which can be found in [Li93]. It is difficult to deal with such a wide-ranging subject in a short article; readers who would like to read more, are invited to contact the author.

This paper is organised as follows. Section 2 gives a short introduction about ANSA. Section 3 discusses the rational of what are the important aspects of real-time objects. Section 4 presents some of the engineering designs required to extend ANSA for real-time systems. Section 5 briefs an implementation of a real-time ANSA system (named RIDE) over a microkernel and ATM environment. Section 6 briefs an implementation of another real-time ANSA system (named RAW) over a standard-based commercial real-time POSIX environment. Section7 gives the performance evaluation of the two systems by using of the Distributed Hartstone benchmark [Mercer90]. Section 8 gives a summary.

1.2 ANSA

ANSA is an Architecture for Open Distributed Processing (ODP), which provides new ways of thinking about the design and construction of object oriented client/server distributed systems. ANSA uses five complementary projection (i.e. enterprise, information, computational, engineering and technology) models to describe architectural components, among which the computational model and engineering model are most relevant to this work. ANSA has been a significant input to the joint ISO/IEC and ITU-T development of a Reference Model of ODP [ISO93].

1.2.1 ANSA object model

The core of the ANSA computational model (ACM) is the use of *objects* as units of distribution for management and replacement. An object has one or more *interfaces* that are the points of provision and use of services. Interfaces are first class entities in their own right and references to them may be freely passed around the system.

An interface contains a set of named *operations* (i.e. procedures or methods). Interfaces have the usual remote procedure call style of interaction: operations are invoked with a set of arguments and a response is returned. Arguments and results to invocations consist of references to other interfaces. The effect of an interaction is that the client and server share access to the argument and result interface. This model makes each interface an abstract data type. ANSA also has a stream interface, the discussion of which is beyond this paper.

1.2.2 ANSA engineering model and ANSAware

The ANSAware is an implementation of the ANSA computational model and an example of the ANSA engineering model (AEM).

The AEM provides a framework for the specification of mechanisms to support distribution of application programs that conform to ACM. The main concepts of the AEM may be summarised: (1) *transparency mechanisms* provide a uniform interface for distributed applications that address the problems and benefits of distribution. (2) *nucleus* provides minimal and sufficient support for the implementation of distribution. It encapsulates all of the heterogeneity of processor and memory architecture. (3) *capsule*: the collection of computational objects (in engineering form), transparency mechanisms and nucleus forming a virtual node of a network. (4) *thread*: a sequence of instructions modelling a computational model activity within a capsule. It represents a unit of potentially concurrent activity that can be evaluated in parallel with other threads, subject to synchronization constraints. (5) *task* a virtual processor which provides a thread with the resources (e.g. a stack) it requires to progress. Tasks[1] provide the resources for real concurrency. An ANSA task is conceptually equivalent to an operating system *thread*. (6) *interface reference*: an identifier which contains sufficient information to allow the holder (the client) to establish communication with the interface denoted by the reference (the server). (7) *channel*: the abstraction for initiating

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

operations to a specific remote interface and for receiving invocations on a specified interface. The initiating side (client) end-point of a channel is called a *plug*. The receiving side (server) end-point is called a *socket*. (8) *interpreter*: a portion of the nucleus. It can be viewed as defining an instruction set for a distributed abstract machine. It interprets inter-object interactions (invocations), performs all argument and result processing, and links threads to sessions (a session is a cache of a plug or a socket) and transfers buffers between them.

1.3 Distributing real-time objects

The essence of a real-time object model is to provide the basic abstractions so that stringent timing constraints of real-time activities are respected (guaranteed at best). The main 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. In most high level languages, this dependency is considered as non-essential detail that is to be hidden from the programmer. As a result the performance of software implemented in these languages becomes sensitive to system resource allocation strategies (in a dynamic system, this means performance depends on system load), and outside the control of individual programmers. More complex resources such as the communication subsystem of distributed systems further accentuate the problem with the introduction of (sometimes distributed) resource allocation algorithms which are usually inaccessible to the application programmer.

Object interdependence can be classified into two categories: *static* interdependence --- the structural relationships between objects, and *dynamic* interdependence --- the interactions (execution views) between objects. Many useful results are known about the static relationships between distributed objects. 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, the *passive object* model, and the *actor object* model.

For real-time applications, this execution aspect is of vital importance --- it has fundamental impact on the *predictability* of computational activities. Real-time object execution models are required to address not only how the computational activities are carried out, but also how shared resources are used (i.e. the manner in which contention for system resources is resolved taking into account timing constraints of real-time activities). The latter issue is often neglected and considered irrelevant engineering detail in non-real-time computing. Distributed real-time systems must provide support for the specialized requirements of real-time communication, tasking, scheduling, and control. These requirements must be explicitly addressed in an object execution model, if the object-oriented approach is expected to be applicable to a real-time world.

1.4 Towards a real-time object model

The collective effect of ACM and AEM defines the ANSA Object Execution Model (AOEM). AOEM is designed for object distribution, but not for real-time applications. It lacks real-time predictability in the following sense:

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

The Real-Time ANSA Object Execution Model (RTAOEM) provides additional abstractions and mechanisms for extending AOEM for real-time. There are two major goals for RTAOEM:

- runs over a standard real-time environment without destroying their real-time properties.
- maintains a reasonable high-level abstraction of an object model

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

RTAOEM has two parts: a tasking model and a communication model.

1.4.1 A real-time tasking model

1.4.1.1 Real-time objects

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

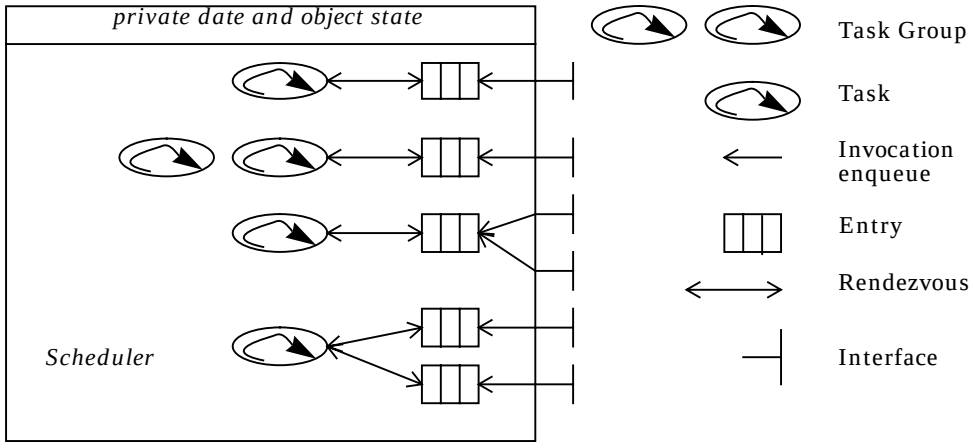
A real-time object is composed of data, one or more tasks of execution, and a set of interfaces. A new abstraction, *scheduling entry,* or shortly *entry,* is introduced as the basic mechanism for real-time scheduling.

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

In Figure 1.1, a graphical illustration of a real-time object is given.

Flexible tasking is based on the entry abstraction. System tasks may be allocated for each individual entry. The tasks allocated are dedicated to execute the threads on the entry. When executing a thread, a task is also allowed to *rendezvous* with other entries dynamically. A *rendezvous* of a task with an entry means that the task waits to accept and execute one thread on

Figure 1.1: Real-time object illustration



the entry. Different control parameters may be selected for each entry to choose a thread enqueue policy, a task/entry rendezvous policy, and to enforce concurrency controls.

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

The choice to allocate a new entry for some interfaces reflects the need to separate these interfaces from others for the purpose of resource management. The number of tasks allocated to an entry not only enforces the real concurrency allowed for the execution of threads on the entry, but also affects the real-time scheduling properties, for example, *preemptivity*. The flexibility for allowing a task to rendezvous with an entry enables an application to have complete control over its virtual processor(s) based on its knowledge of the system state. These resource management activities can all be done dynamically, increasing the flexibility and usefulness in an open dynamic environment.

1.4.1.2 Real-time object invocation

RTAOEM allows the association of an optional *priority* (criticality) and/or *deadline* with each invocation. As the invocation crosses the physical boundary and becomes a thread in the called object, this priority and/or deadline is passed and becomes a property of the thread, which may then be used as a scheduling parameter on the server site.

1.4.1.3 Scheduling

The main goal of the real-time tasking design is to allow the maximum control of scheduling at the application level. Care has been taken to achieve the balance between flexible and deterministic scheduling. A policy/mechanism separation approach has been taken to address the diversity of real-time programming. 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. Preemption is used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behaviour of the task execution. 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. The primary function performed by multiplexing is the sharing of processor resources, which is similar to the multiplexing in communications systems and protocols for sharing communication resources. The use of separate entries to process requests on separate interfaces offers a number of (potential) advantages (1) allows the use of a specific scheduling policy (thread scheduling policy) suitable for each interface or each interface class, (2) allows the possibility of using interface specific tasks to serve requests, and thus allows for more efficient resource utilisation, (3) separate entries may be processed in parallel, thus increasing performance, (4) allows the possibility of end-to-end scheduling and guarantees, (5) preserves the modularity and separation of service interfaces.

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

Some typical thread enqueue policies are (1) first come first service, (2) priority based, (3) deadline based, (4) priority and deadline based.

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. Some typical task/thread rendezvous policies are (1) null --- the priority/deadline of a thread has no effect on the serving task, (2) priority inheritance, (3) transitive priority inheritance, (4) priority ceiling and (5) deadline inheritance.

Detailed examinations of some typical real-time scheduling schemes, such as priority based scheduling and deadline based scheduling, can be found in [Li93].

1.4.2 A real-time communication model

Real-time applications present more complicated functional requirements to the underlying communication systems. RTAOEM provides abstractions to express the individual QoS constraints for a communication channel and the selection of in-band QoS parameters of a communication channel.

The following abstraction is provided in RTAOEM

- the allocation of a separate communication channel for each client/server binding
- the association of a communication QoS to a channel
- the association of an in-band QoS to an invocation.

The in-band QoS object supports the specification of a priority, a timeout, a deadline, a deadline type, and any other QoS parameters a communication channel may support, this depends on the individual network system.

Priority and deadline are used to convey the urgency of a real-time activity across a network. The combination of a timeout, a deadline and a deadline type can be used to bound the expected execution time of an invocation, which is further discussed in section 1.4.2.2.

From engineering point of view, the following mechanisms are required:

- a parallel protocol stack
- a timed RPC protocol

1.4.2.1 A parallel protocol stack

A parallel communication protocol stack allows the preallocation of communication resources (a separate channel, for example) and the removal of layered multiplexing. The main gain is that it allows the application to explore the communication QoS that the low-level operating environment can provide. For example, in an ATM environment, a channel (or a virtual circuit) is allowed to associate with various transportation QoS, such as jitter, delay, priority etc. Even in an ordinary operating environment, this design allows the choose of communication protocols, such as TCP, UDP, IPC etc.

1.4.2.2 A timed RPC protocol

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

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

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

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

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

The design of a TRPC is complicated by the fact that the end-to-end delay of messages can be arbitrary or even infinite (messages can get lost). It can be shown that the two goals are not mutually compatible. In its simplest form, in which each request takes zero service time, the TRPC problem is equivalent to the *timed synchronous communication* problem [Lee90]. In the case of message loss, timed synchronous communication is the well known *Two Generals* problem, in which the two generals are trying to agree upon *a common time* of attack before a deadline but can only communicate via unreliable messengers. Such a protocol does not exist.

The design of a TRPC is further complicated by the fact that making the decision of whether a request is schedulable at the server side is often

unattainable --- a guarantee scheduler often makes many of the impossible assumptions such as that the invocation service time is known, operations are independent etc.

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

The deadline type parameter is introduced to choose how the deadline can be used in the server side of a call. It can be used to control the latest start or latest finish time of an invocation.

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

1.5 RIDE

RTAOEM was first implemented as RIDE [Li93] in the Cambridge Distributed Systems Environment. It has a real-time microkernel named WANDA, and runs ANSAware 3.0. The environment is composed of interconnected 680x0, VAX, ARM and MIPS machines over an ATM and Ethernet network.

RIDE explores several real-time scheduling issues:

- preemptive nucleus: just like a real-time operating system, the ANSAware nucleus was redesigned and implemented as preemptive. This involves the introducing of fine-grained locking and elimination the use of global context informations.
- using of kernel threads: ANSA tasks are mapped to WANDA kernel threads to use the WANDA real-time thread scheduling facilities.
- user-level ANSA real-time thread scheduling
- ANSA/WANDA cooperative supports of priority inheritance.

RIDE explores the following real-time communication issues:

- extending the ANSAware communication interface to provide a connection oriented communication paradigm to take advantage the ATM connection-oriented real-time transportation interface. This permits an application to allocate an individual communication stack for each ANSA communication channel.
- state-full ANSA transportation and message passing protocols, this is the required nucleus support for connection-oriented communication channels. These protocols support the establishment of end-to-end virtual circuits with some known transportation properties.
- a decomposable RPC protocol which allows the synthesis of the protocol to provide different levels of invocation semantics (such as exactly-one, at-most-once) and different timing semantics (the interpretation of priority and deadline) so that an application programmer can customize the

system to application-specific requirements of functionality and performance. The decomposable protocol provides the combined effect of a normal RPC protocol the timed RPC protocol in an integrated way.

1.6 RAW

RAW is a generalization of RIDE to run over a standard real-time environment. RAW also uses binding and QoS concepts to provide a more general vehicle for resource management and requirement specification. Binding is the process (operation) by which an activity in one object establishes the ability to invoke operations at an interface to some other object. A binding establishes and controls the communication sessions involving multiple objects so that their interactions are possible.

RAW is implemented over a networked DEC/Alpha OSF1 workstations environment. RAW extends RIDE in the following aspects:

- compatible with a new version of ANSAware AW4.1
- running over a de-facto industry standard: real-time Posix threads
- full pthread real-time scheduling and threading capabilities
- selective communication multiplex by QoS specification and explicit binding operations, which is more general than a connection-oriented communication interface
- multiple RPC protocols. RAW uses different protocols for real-time and non-real-time data transportation. This allows the exploration of network and application dependent transportation mechanisms
- in-band QoS association based on invocations and communication channels.

1.7 Performance evaluation

There has been a growing interest in defining standard synthetic benchmarks for real-time computing systems. The Hartstone Benchmark (HB) [Weiderman89], Distributed Hartstone Benchmark (DHB) [Mercer90] and Hartstone Distributed Benchmark (HDB) [Kamenoff91] are three examples of this effort. 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 is chosen to measure and evaluate RIDE and RAW performance. The intention of the 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:

- DSHcl, a Distributed, Synchronized, and Harmonic task set which tests the communication latency of the system.
- DSHpq, a Distributed, Synchronized, and Harmonic task set designed to test for priority queuing of communication packets.
- DSNpp, a Distributed, Synchronized, and Non-harmonic task set designed to test the degree of preemptability of the protocol engines.
- DSHcb, a Distributed, Synchronized, and Harmonic task set which tests the communication bandwidth.
- DSHmc series, a task set for measuring media contention, which does not apply to the Ethernet.

To achieve comparable results, we executed DHB on both RIDE and RAW by using of a 10 Mbit/Second Ethernet and two workstations. In the RIDE test, two DEC Firefly multiprocessor workstations were deployed (but only one processor was used in the experimentation for each machine). In the RAW test, two DEC Alpha 3000/300 workstations were used.

To make a comparison, the relevant performance of the ARTS distributed real-time operating system is also given in Table 1.1. The ARTS performance is copied from [Mercer90] which was measured by using SUN3/140s and a private 10 Mbit/Second Ethernet.

Table 1.1: RIDE, RAW vs ARTS performance

Series	ARTS	RIDE	RAW
DSHcl	35 ms	26 ms	39 ms
DSHpq	18 KWIPS	16 KWIPS	2010 KWIPS
DSHpp	(13) 20 tasks	18 tasks	105 tasks
DSHcb	14 tasks	15 tasks	23 tasks
DSHmc	NA	NA	NA

Comparison of the performance of RIDE and ARTS is, however, not as simple as it looks. The ARTS system uses kernel supported objects, object invocations, and preemptive protocol processing; while RIDE uses a relatively heavyweight user level RPC mechanism. In RPC systems, the marshalling and un-marshalling of arguments, the overhead of an RPC protocol, the multiplexing of a required operation within an interface, and the demultiplexing of replies for clients are time consuming. Taking these into account, it is reasonable that RIDE is 9 ms less efficient in the DSHcl series test (which tests communication latency). On the other hand, RIDE performs as well as ARTS in the DSHpq, DSNpp and DSHcb series tests. That is, RIDE can achieve about the same performance as ARTS in the priority queuing of communication packets, in the preemptability of the protocol engine, and in the provision of communication bandwidth.

The much better performed RAW 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.

1.8 Summary

The paper reviews some of the main features of the ANSA-based real-time object model RTAOEM, and two of its prototyping systems RIDE and RAW.

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

The performance of the two prototype implementation is compared to that of some typical systems by using of the Hartstone and Distributed Hartstone Benchmarks, and has shown that the design is viable.

References

[APM92]

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

[Gopinath93]

P Gopinath and T Bihari, Concepts and Examples of Object-Oriented Real-Time Systems, In Readings in Real-Time systems, Y H Lee and C M Krishna ed., pp 123-136, IEEE CS Press, June 1993.

[Herbert94]

A Herbert, An ANSA Overview, IEEE Network, pp 18-23, January 1994.

[ISO93]

ISO/IEC 10746-3, ITU-TS Recommendation X.903: Basic Reference Model of Open Distributed Processing: Prescriptive Model, (2nd CD draft) June 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.

[Li94]

G Li and J Bacon, Supporting Distributed Real-Time Objects, to appear in Proceedings of the Second Workshop on Parallel and Distributed Real-Time Systems, Cancun, Mexico, April, 1994.

[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.

[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.