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

An Overview of Real-Time ANSAware 1.0

Guangxing Li

Abstract

Distributed computing and real-time computing are well established areas of research, but their integration is yet to be studied because they seldom use compatible techniques. This paper provides an overview of an ANSA based distributed computing environment (named ANSAware/RT) for open, distributed real-time computing. The focus of this article is the engineering mechanisms necessary to support real-time processing. ANSAware/RT incorporates real-time tasks and communication channels as its basic programming components. It synthesises aspects of resource requirements, allocation and scheduling into an object-based programming paradigm.

This is an external paper for Distributed Systems Engineering Journal.

APM.1285.01 **Approved** 8th March 1995
External Paper

Distribution:
Supersedes:
Superseded by:

An Overview of Real-Time ANSAware 1.0

Guangxing Li

Architecture Projects Management Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD, U.K.

Abstract: Distributed computing and real-time computing are well established areas of research, but their integration is yet to be studied because they seldom use compatible techniques. This paper provides an overview of an ANSA based distributed computing environment (named ANSAware/RT) for open, distributed real-time computing. The focus of this article is the engineering mechanisms necessary to support real-time processing. ANSAware/RT incorporates real-time tasks and communication channels as its basic programming components. It synthesises aspects of resource requirements, allocation and scheduling into an object-based programming paradigm.

1 Introduction

Open Distributed Processing (ODP) is concerned with the use of commodity technology, such as OSF DCE [20] or implementations of OMG CORBA [19], to build integrating applications that link together existing applications, databases, control systems and users.

Real-time processing is concerned with the timeliness of computing activities. Real-time processing places unique requirements on distributed systems including predictability, programmer control, timeliness, mission orientation and performance [13].

The need for an ODP architecture to include real-time processing is driven by two technology trends:

- *general purpose distributed computing environments are evolving towards real-time systems.* For example, the advances in digital communication networks and in personal computer workstations allow the generation, communication and presentation of real-time voice and video media simultaneously. Many non-real-time systems are being corrected to real-time multimedia processing [12]. This trend requires distribution and real-time control functionality to be combined and for real-time features to be intrinsic elements of normal system services, rather than as special add-ons
- *real-time applications are evolving towards large distributed systems.* One-million-line real-time software systems in telecomms, manufacturing, transportation and other application areas are becoming common today [6]. Such systems are large by any standard and inevitably distributed. Therefore, in addition to the problems associated with real-time operation, they are subject to all of the problems of any large software system, such as maintainability, evolution and distribution. There is an increasing need to adopt open architectural standards.

An ODP architecture with real-time extensions provides the capability to treat all forms of real-time objects as *first class citizens*. That is, operations and

mechanisms provided for existing non-real-time components can be applied to, and used by, real-time components. The provision of a uniform system architecture facilitates increased productivity, especially for applications which offer combinations of distributed and real-time functionality: e.g. multimedia conferencing, distributed control etc. It allows existing distributed system mechanisms (such as trading, security, monitoring, replication, location, migration and federation) to be applied to real-time components. It also allows evolution of systems from the development of individual real-time systems, to groups of real-time systems and then to *enterprise-wide* command and control real-time systems.

Current reference models for open distributed processing, including ISO RM-ODP [9], OSI Management, OMG Object Management Architecture [19] and ANSA [8], make no mention of real-time issues. Current standards for open distributed processing, such as the OSF's DCE and the OMG's Common Object Request Broker Architecture (CORBA) do not address real-time or performance management requirements. As relatively new technologies, attention has focused entirely on functionality; real-time issues are not addressed.

This paper provides an overview of the design, implementation and performance evaluation of the real-time ANSAware (ANSAware/RT or shortly AW/RT) 1.0 to address the problem of real-time computing in the ODP domain. AW/RT is based on the ANSA architecture [8] and its example implementation ANSAware 4.1 [2]. The focus of this article is the engineering mechanisms; other issues such as architecture and application programming interfaces can be found in [14, 16].

This paper is organised as follows. Section 2 gives a short introduction to ANSA. Section 3 discusses the important aspects of real-time objects. Sections 4-6 present the engineering designs required to extend ANSA for real-time systems. Sections 7-10 outline the implementation techniques used for AW/RT in comparison with ANSAware. Section 11 gives a performance evaluation of the system by using of the Distributed Hartstone Benchmark [18]. Section 12 gives a summary and finally section 13 discusses related work.

2 ANSA

ANSA is an Architecture for ODP, which provides new ways of thinking about the design and construction of object oriented distributed systems. ANSA uses five complementary models (enterprise, information, computational, engineering and technology) to describe architectural components, of which the computational model and engineering model are most relevant to this work. The overall ANSA framework has heavily contributed to the joint ISO/ITU Reference Model of ODP (RM-ODP) [9].

2.1 ANSA object model

The ANSA computational model (ACM) uses *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 service. 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) which defines its type. 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 data and (possibly) references to other interfaces. The effect of an interaction is that the client and server share access to the argument and result interfaces. This model makes each interface an abstract data type. ANSA also has a stream interface for representing arbitrary communication flows, which is beyond the scope of this paper.

2.2 ANSA engineering model and ANSAware

The ANSA Engineering Model (AEM) provides a framework for the specification of mechanisms to support distribution of application programs that conform to ACM. ANSAware (AW) is an example implementation of AEM. AW is a suite of software for building ODP systems, providing a basic platform and software development support in the form of program generators and system management applications. AW provides a uniform view of a multi-vendor world, allowing system builders to link together distributed components into network wide applications. It historically preceded, and accurately predicted the need for both the DCE and CORBA.

The main components of AEM are:

- *transparency mechanisms* which automate aspects of distribution such as object migration and object replication
- *nucleus* which provides minimal resource management support for the implementation of distribution. It encapsulates all of the heterogeneity of processor and memory architecture
- *capsules* which are collections of application objects, transparency mechanisms and nucleus objects forming a virtual node of a network
- *threads:* sequences of instructions modelling a computational model activity within a capsule. A thread represents a unit of potentially concurrent activity that can be evaluated in parallel with other threads, subject to synchronization constraints
- *tasks:* virtual processors which provide threads with the resources (e.g. stacks) they require to progress. Tasks[1] provide the resources for real concurrency. An ANSA task is conceptually equivalent to an operating system *thread*
- *interface reference*s: identifiers which contains sufficient information to allow their holder (client) to establish communication with the interface denoted by the reference (server)
- *channels:* resources (e.g. stubs and protocol drivers) required to enable end to end communication. The initiating side (typically a client) end-point of a channel is called a *plug*. The receiving side (typically a server) end-point is called a *socket*
- *binders:* components to support binding: the process by which an activity in one object establishes the ability to invoke operations at an interface to

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

some other object. Binding establishes and controls the communication channels between objects so their interactions are possible.

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 ideally). The main difficulty is that the actual timing characteristics of software are determined not only by the raw processor speed, but also by competition for scarce resources. In most high level languages, this dependency is considered as non-essential detail to be hidden from the programmer. As a result the performance of software implemented in these languages becomes sensitive to 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
- *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 [4], the *passive object* model [1], and the *actor object* model [3].

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). 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 the object execution model, if it is expected to be applicable to a real-time world.

4 ANSAware/RT design

Collectively, the ACM and AEM define the ANSA object execution model. This model is designed for object distribution, but not for real-time applications. It lacks real-time predictability in the following sense:

- it multiplexes both tasks and communication channels whenever possible
- both thread/task scheduling and communication scheduling are implicit

- no abstraction is provided to express urgency and resource requirement for application programmers.

A real-time object model can be obtained by extending the execution model with explicit resource allocation, real-time scheduling and real-time communication support. The AW/RT real-time object model has two parts: a tasking model and a communication model.

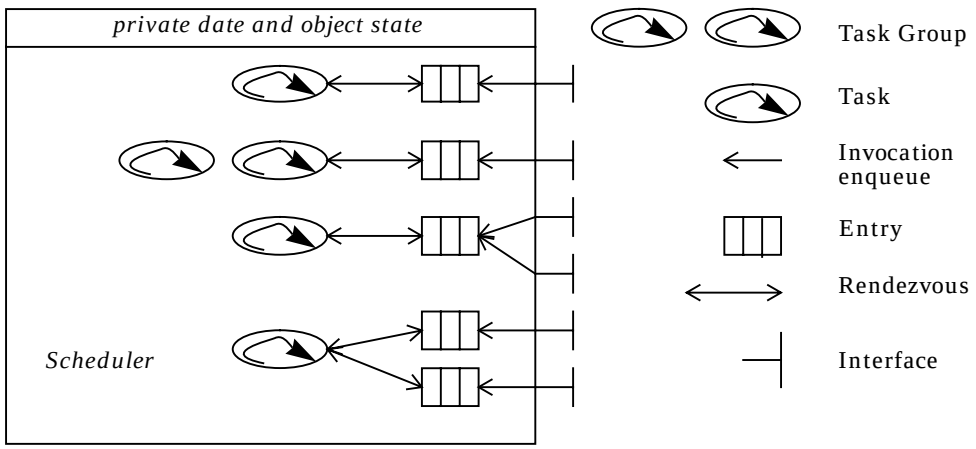
5 ANSAware/RT tasking model

5.1 Real-time objects

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 and resource allocation.

An entry is a thread queue with a record of control data. An entry may be created, and interfaces may be bound to it dynamically. When an interface is bound to an entry, each operation request on the interface will be transferred (by the infrastructure) to a thread enqueued on the entry. Any thread representing a computational activity is also spawned on an entry. The entry is an engineering concept which is confined within a capsule. Figure 1 gives a graphical illustration of a real-time object.

Figure 1: Real-time object illustration



Tasks may be allocated for each individual entry to execute its threads. 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 the entry. Different control parameters may be selected for each entry to choose a thread queuing 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 separate 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* and *priority inversion* [17]. 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.

5.2 Real-time object invocation

AW/RT allows the association of an optional *priority* (criticality) and/or *deadline* with each invocation. As the invocation crosses from one object to another, this priority and/or deadline is passed and becomes a property of the executing thread on the server site.

5.3 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. Scheduling is defined in layers as:

- thread scheduling --- by a rendezvous scheduler for each entry
- task scheduling --- by a nucleus scheduler for 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 can be used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behaviour of the task execution. 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

- it allows the use of different scheduling policies at each interface or interface class,
- it allows the possibility of using interface specific tasks to serve requests, and thus allows for more efficient resource utilisation,
- separate entries may be processed in parallel, thus increasing performance,
- it allows the possibility of end-to-end scheduling and guarantees,
- it preserves the modularity and separation of 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 examination of some typical real-time scheduling schemes, such as priority based scheduling and deadline based scheduling, can be found in [17].

6 ANSAware/RT communication model

Real-time applications present more complicated functional requirements to the underlying communication systems. AW/RT provides abstractions to express the individual Quality of Service (QoS) constraints for a communication channel and the selection of in-band QoS parameters of an invocation.

The following abstractions are provided in AW/RT

- the allocation of a separate communication channel (with an individual QoS) for each client/server binding by an explicit binding operation, allowing the control of communication multiplexing and time of binding
- the association of in-band QoS with each 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, depending on the individual network's capabilities.

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 (see section 6.2).

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

- a parallel protocol stack
- a timed RPC protocol

6.1 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 a choice of communication protocols, such as TCP, UDP, IPC etc.

6.2 A timed RPC protocol

Arbitrary delays associated with synchronous invocations 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.

Invocations in AW/RT can attach deadline constraints to their communication requests. Such 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 that 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:

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

The design of a timed RPC 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 problem is equivalent to the *timed synchronous communication* problem [11]. In the case of message loss, timed synchronous communication is the well known *Two Generals* problem [11], 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 timed RPC is further complicated by the fact that making the decision of whether a request is schedulable at the server side is often *unattainable* [23] --- a guarantee implies constraints 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 timed RPC 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 how long the client is willing to wait for its result. It affects a client side of the protocol only. The deadline specifies the time within which the request should be executed on the server. It affects the server side of the protocol only.

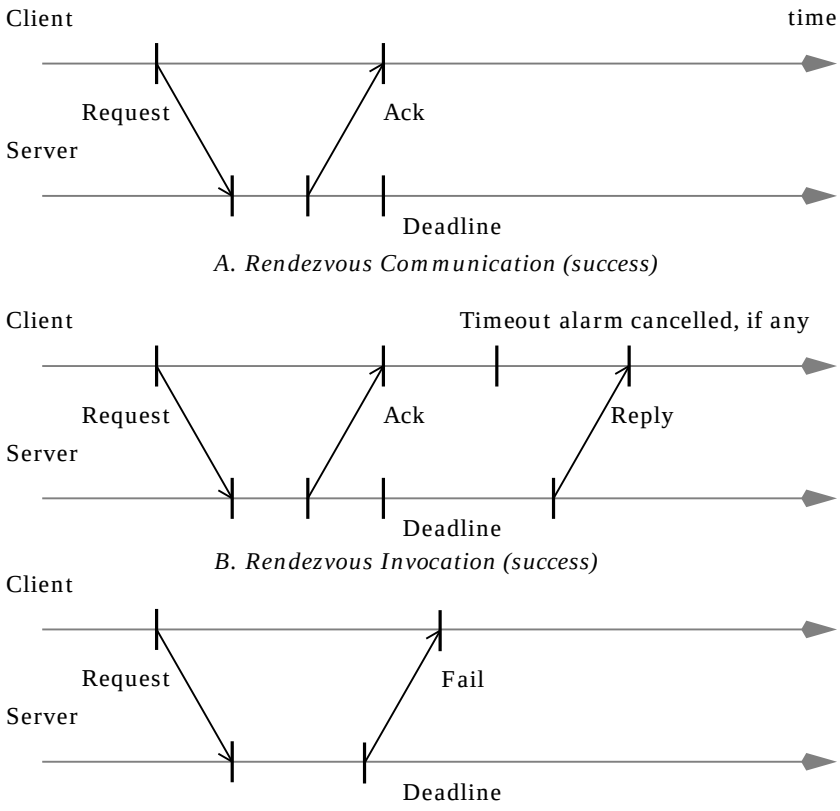
It should be pointed out that using the two separate arguments does not solve the consistency problem. Rather, the two arguments give the problem a more realistic definition, allowing different relaxations be explored.

The first relaxation is using a *timeout* to enforce the client's *absolute* deadline. The client decides that the request is unsuccessful if it does not get a reply/ acknowledgement from the server by the timeout. There is a possibility for *inconsistent decisions* --- the client believes the request is failed, while the server knows the request is successful. Deadlines may or may not be used in this situation. The timeout expiration presents the client an exception situation of *don't know*. It is up to the client to take further rescue actions.

The second relaxation is using a deadline to specify a client's objective time value by which the request should be finished. Whether this deadline can be

guaranteed or not is purely a matter of server scheduling and message passing delays. In this relaxation, the client waits until a reply/acknowledgement is received from the server. Therefore, the client deadline is not *absolute*. This relaxation allows a client and its server to reach a consistent decision.

Figure 2: Rendezvous communication/invocation interaction



The second relaxation can be further extended by relaxing the meaning of a deadline. Instead of bounding the finishing time of a request, a deadline can be used to bound the start time of a request in the server --- to bound the start time by which the request must rendezvous with a server task. If the rendezvous is issued before the deadline, then the request is successful and a success acknowledgement is sent back to the client, otherwise the request is cancelled and a fail acknowledgement is returned. At the client side, there are two possible actions to be taken when it receives a success acknowledgement. One is that the client thinks the request is finished, and control is returned so that it can continue. This is defined by the *RendezvousCommunication* deadline type. Another is that the client cancels its timeout, if any, and waits until a reply is returned later by the server. This is defined by the *RendezvousInvocation* deadline type. The two resulting interaction patterns are illustrated in figure 2.

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 timed RPC protocol. The result of such a timed RPC call can be a *timeout* --- possibly an inconsistent state, a *success* or a *failure*.

An example of object invocation using timed RPC in AW/RT is shown as following. In the invocation, the deadline type is *RendezvousCommunication*, the deadline value is 100 ms, the timeout value is 200 ms. If the timeout expires, an operation specific exception handler will be called which may take whatever necessary actions for the error recovery.

```
ansa_InvQoS qos;
ansa_invqos_setdeadline(&qos, 100);
ansa_invqos_setdeadlineType(&qos, rendezvousCommunication);
ansa_invqos_settimeout(&qos, 200);
{result} <- ifref$op(parameters) Signal clientTimeout {qos}
```

7 ANSAware/RT implementation

AW/RT was first implemented as RIDE [17] in the Cambridge University Distributed Systems Environment. It used an experimental real-time microkernel named WANDA, and ANSAware 3.0. The environment was composed of interconnected 680x0, VAX, Acorn RISC Machine, and MIPS machines over a mixed ATM and Ethernet network.

AW/RT described in this paper is an evolution of RIDE to run over a standard real-time environment. AW/RT also uses binding and QoS concepts taken from the RM-ODP to provide a more general vehicle for resource management and requirement specification.

AW/RT has been ported to the DEC/Alpha OSF1 operating system, the HP/RT target system and LynxOS. AW/RT 1.0 achieved the following design goals

- portability and interworking with a new version of ANSAware (4.1)
- running over a de-facto industry standard: real-time POSIX threads (pthreads)
- full pthread real-time scheduling and threading capabilities
- selective communication multiplexing by QoS specification and explicit binding operations
- application controlled resource allocation
- multiple RPC protocols: different protocols are used for real-time and non-real-time interactions
- interoperation between different real-time platforms while retaining their real-time features.

In the following sections, ANSAware is explained first and then the AW/RT tasking and communication system.

8 ANSAware

Logically, AW starts with several threads and one or more tasks. There is a receiver thread for receiving messages at 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 by a coroutine. Additional tasks may be created to provide extra physical concurrency. Tasks are shared by all threads. Threads waiting to execute are queued on one FCFS queue. The AW scheduler in the nucleus 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 concurrency:

- use of global, continuous and extensible memory areas to store the shared data structures holding the system 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
- use of 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 are accessible without requiring a synchronization mechanism since the AW scheduler is non-preemptive. A task, while execution, will not relinquish control until it blocks or terminates. Therefore, it is guaranteed exclusive access of the shared data in a single processor environment, and there is no need for access protection of shared data.

The AW communication system implements four protocol layers:

- Message passing protocol (MPS): an interface to the transport protocols provided by the underlying operating system
- Execution protocol (EX): implement the invocation of ANSA operations. AW 4.1 supports the Remote Execution Protocol (REX) for point to point invocations and the Group Execution Protocol (GEX) for invocations between replicated clients and servers
- 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.

The communication system supports an implicit binding model which was designed to have good scaling characteristics and to optimise the usage of resources. It uses maximum multiplexing to efficient resource management and provides only one Quality of Service.

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 name and the network address of the underlying 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 endpoints whenever possible (in a capsule wide basis).

The EX interface is also stateless and there is also no mechanism for QoS based selection/setup of an EX module. REX is designed for asynchronous communication optimized for either low-latency or high throughput. REX provides a rate-based fragmentation service. The execution reliability semantics of REX calls are exactly-once in the absence of total communication failure.

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 *ifrefs*. Clients send/receive invocation requests/replies through plugs. Servers receive/transmit invocation requests/replies over sockets. The channel name provides an extra layer of demultiplexing on top of the MPS address, so that the capsule can locate the right dispatcher for a specific interface. There is no interaction between the channel and MPS modules when channels are created and destroyed; the two are independent of each other.

A service provider fabricates an *ifref* by an interface creation operation, which can then be passed to a potential client. A client holding the *ifref* must then bind to the service in order to communicate with it. Client side binding (the creation of a plug) is performed on the first invocation of a service; the first invocation is detected by the absence of the *ifref* from the *ifref* to plug cache. The bind operation which allocates a new plug also adds the plug to the cache. Removing an *ifref* from the cache will force a rebinding operation. At no stage is there any interaction between the binding process and the EX and MPS modules; it is assumed that all communications between channels is multiplexed over a single MPS address within a capsule.

9 ANSAware/RT tasking

In AW/RT, each task is mapped into a pthread, and task scheduling is done by the underlying operating system. All pthread 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. pthread features. Thus the original AW task scheduler is made redundant.

9.1 Global data protection

Because of the real concurrency and preemptive nature of the pthread 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 AW/RT operation, a task must first acquire the lock, then operate on the shared data, and finally release the lock when finished.

9.2 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, therefore it would be

expensive to leave these state still in the global data area in AW/RT which need to be protected by a synchronization mechanism for each access. The solution adopted is to use pthread per-thread state to store AW thread private state (rather than using the global area). Such state information are then accessible by using of the `pthread_getspecific` procedure without a synchronization operation. The thread private state is actually part of a task private data area. When a task is created, it allocates a private state area as pthread per-thread data, and part of this area is used as thread private state when the task is executing a thread.

9.3 Stacked threads

AW threads are non-stacked: a task will not execute another thread before it finishes the current one. AW/RT introduces the dynamic rendezvous mechanism: a thread may rendezvous with another thread while executing. The required extensions are to allow a task to execute another thread when it is executing a thread. This stacked thread mechanism is implemented by pushing the thread private state area into the task's stack before executing the new thread (so that the new thread 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.

9.4 Threads

Threads are created in two cases: (1) an application may create new threads for additional concurrency; (2) a communication task may create one additional thread for each RPC request from a client. In AW, a new thread is queued on the capsule-wide FCFS thread queue, waiting to be executed by a free ANSA system task. In AW/RT, a new thread is queued on an entry instead of the capsule FCFS queue. In case (1), the application gives an additional entry argument when a new thread is created. In case (2), the binding between an interface and an entry determines on which entry the new thread should be queued.

9.5 Entry

Each entry is associated with a thread queue and a thread queuing policy. Policy/mechanism separation is used for efficient coding. A common set of thread queuing/dequeuing mechanisms are provided, and on top of the mechanisms a set of scheduling policy objects are imposed. Figure 3 illustrates the design.

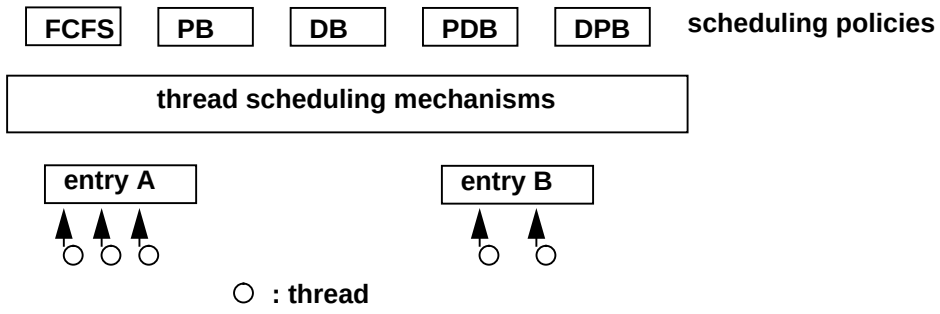
Each entry is also associated with a rendezvous policy. Each policy provides two functions: `rendezvous_inheritance` and `rendezvous_deinheritance`. The `rendezvous_inheritance` function is executed before a task executes a thread so that the task can take the thread scheduling parameters into consideration.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

9.6 Synchronous I/O

AW assumes a totally asynchronous I/O model because

- it allows the tight combination of communication event processing and task scheduling for efficiency

Figure 3: Thread scheduling: policies and mechanisms



- 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 UNIX signal like programming interface called *pin*. 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 pthread implementation of the tasking system, the asynchronous I/O model is no longer necessary because

- task scheduling is done by the operating system; 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

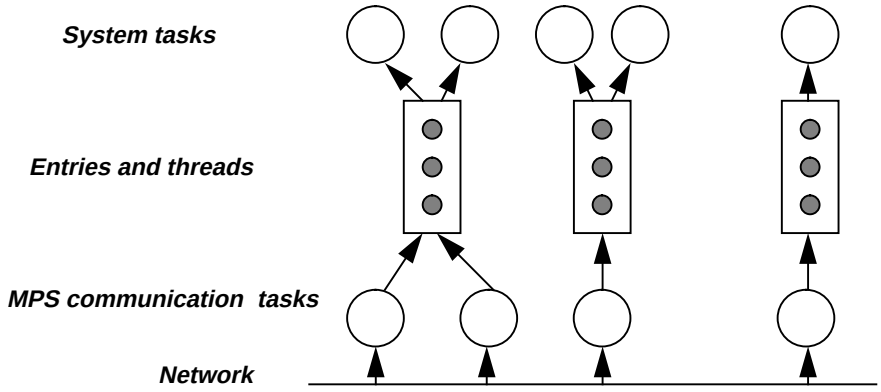
In other words, AW/RT does not need to assume the asynchronous I/O model, and a complete synchronous I/O model is more natural and easy to programming. Therefore, AW/RT removes the pin interface, the non-blocking keyboard input library and the X11 library. The application programmer can use the equivalent synchronous I/O operations supported by the pthread interface.

9.7 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 message passing service endpoint (i.e. a socket), a task is spawned to handle messages from it. 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. Thus, the communication task is both a thread generator and a thread scheduler. The threads are executed by tasks at an entry allocated by an application. The scenario is shown in figure 4.

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 returned and has been processed by a communication task, the condition variable is signalled and therefore the calling thread is woken up.

Figure 4: Communication tasks and system tasks



10 ANSAware/RT communication system

10.1 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 function to fabricate the required *ifref*. The explicit binding function calls the corresponding explicit binding operations in each protocol layer (EX and MPS). The binding function at each protocol layer interpret the relevant QoS attributes, setups the protocol related binding states, and return data that may be used to build the *ifref* as the result.

In comparison with implicit binding, the *ifref* created by an explicit binding contains only that data deduced from the QoS, rather than the default data that provides the maximum communicability and also the maximum multiplexing.

Client side explicit binding is performed by a binding function, which is also associated with a QoS object. This function, like the server side binding function, 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 procedure to the relevant QoS and the *ifref* to create the client site binding. The binding function finishes by creating a plug and adding it in the plug cache, so that later calls on the interface are guaranteed an established channel.

10.2 State-full message passing protocol

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

The MPS interface is extended with three functions: server explicit binding function, client explicit binding function and binding release function. The original message *send* and *receive* operations are also extended to make use of the binding data structure. A default binding is established at MPS

initialisation time to be used as the default communication channel for the implicitly bound interfaces.

10.3 State-full execution protocol

EX is modified to use the state-full MPS, and is itself redesigned to state-full as well. This allows the addition of extra binding data to include EX dependent data and state. For example, the binding data 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. Like MPS, the EX interface is extended with three similar binding functions.

10.4 Timed remote execution protocol

The generic design of the binding and state-full protocols allows the insertion of new EX protocols. The timed RPC is implemented as one example.

Timed Remote Execution Protocol (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 integrity 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 to implement the semantics of TRPC:

- the header of requests 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 bindings, i.e. interfaces created by implicit binding operations will not be able to use TREX.

10.5 In-band QoS

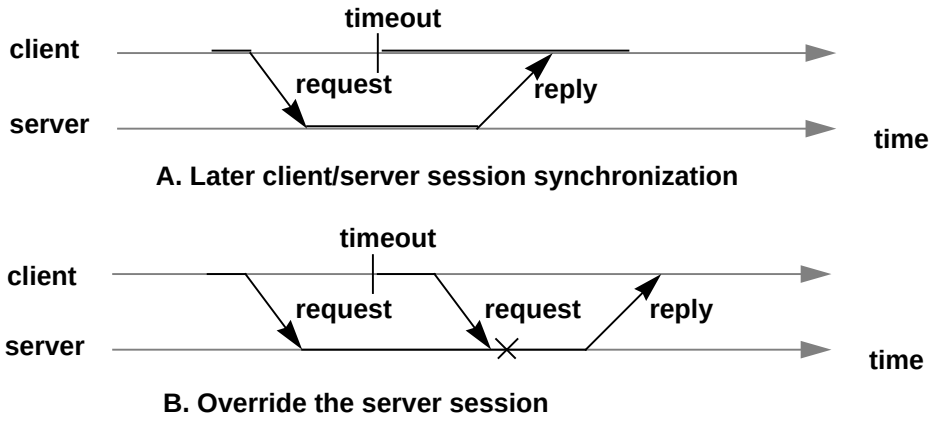
In-band QoS is allowed to be associated to each invocation to select the dynamic QoS parameters of a channel. Currently, if a channel uses TREX as its EX, the in-band QoS can select a priority, a timeout, a deadline and a deadline type.

10.6 Session overridden

Client timeouts are a mixed blessing: the desired semantics requires that when a timeout expires the client should resume control (e.g. to take some immediate recovery actions). However, the operation is still carried on at the server side and an extra packet exchange is required to resynchronise the client and server. 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 as illustrated in figure 5.

The client continues immediately after the timeout, and the client session is set to *idle*. No synchronisation packet exchange is initiated by the client, thus

Figure 5: Session timeout recovery



allowing an inconsistency between a client session and its server session. Should the server returns an obsolete result later, resynchronisation of the client and server (sessions) are taken at that time. The approach also allows the server session to be aware that its client may timeout, and that the client session may be used for another invocation. Thus, a later invocation from the same client may override an obsolete invocation at a server.

11 Performance evaluation

There are several standard synthetic benchmarks for real-time computing systems, including Hartstone Benchmark (HB) [22], Distributed Hartstone Benchmark (DHB) [18] and Hartstone Distributed Benchmark (HDB) [10]. 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 AW/RT 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 based on a typical client/server interaction model. The task workload is expressed in Kilo-Whetstone (KWS) or milliseconds. A KWS is one execution of a mathematical library, which factors out the effect of a typical arithmetic computing. Each experiment starts with five periodic client tasks and one or two server tasks. Each task is required to execute a specific amount of workload within its period. The experiment

continues with added workload until any task's deadline cannot be met. The five sets of experiments are:

- DSHcl, a Distributed, Synchronized, and Harmonic task set which tests the communication latency of the system. The server computation time is increased in milliseconds to squeeze the time left for message passing
- DSHpq, a Distributed, Synchronized, and Harmonic task set. The server computation time is increased in KWS to measure how well the system does at priority queuing of communication packets
- DSNpp, a Distributed, Synchronized, and Non-harmonic task set. The number of low-priority client tasks are increased to test the degree of preemptability of the protocol engines
- DSHcb, a Distributed, Synchronized, and Harmonic task set. The number of high-priority client tasks is increased to test the bandwidth for real-time communications
- DSHmc series, a task set for measuring media contention, which does not apply to the Ethernet.

Table 1: RIDE, AW/RT vs ARTS performance

Series	ARTS SUN 3/140	RIDE DEC Firefly	AW/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 achieve comparable results, DHB was executed on AW/RT by using of a 10 Mbps Ethernet and two DEC Alpha 3000/300 workstations. The relevant performance of the ARTS distributed real-time operating system and RIDE system are also given in table 1. The ARTS performance is copied from [18] which was measured by using SUN3/140s and a private 10 Mbps Ethernet.

Comparison of the performance of AW/RT, 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 AW/RT and RIDE use 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 performance of AW/RT reflects the combined 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. Unfortunately, we have been unable to find figures that enable us to compare all three systems in the same platform.

12 Summary

The paper reviews the main features of the real-time ANSAware 1.0. AW/RT provides a framework to facilitate the enforcement of stringent timing constraints found in distributed real-time applications. The AW/RT design 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 AW/RT implementation is compared to that of some typical systems by using of the Distributed Hartstone Benchmarks, and has shown that the design is viable.

13 Related work

The general discussion of an open system architecture for real-time processing can be found in [14]. The description of the AW/RT programming interfaces can be found in [16]. Comparison of the two ANSA based real-time environments AW/RT and RIDE is described in [15]. Current research at CNET [7] and Lancaster University [5] are all converging on a common architecture for distributed multimedia and real-time processing relevant to the AW/RT system.

AW/RT has been influenced by ARTS [21] in the design of its priority scheduling models. AW/RT also shares a few common features with ARTS in the handling of object invocations. However, the object models of the two systems differ substantially. Compared to AW/RT, ARTS is a relatively low-level mechanism and interoperation between different real-time platforms is not being addressed. AW/RT objects have the flexibility to group operations as interfaces and the flexibility to create interface instances dynamically. This is further enhanced with entries, by which dynamic resource allocation and management is possible. On the communication side, AW/RT has a timed RPC protocol which provides richer timing semantics than the simple time fence protocol of ARTS; AW/RT also allows the preallocation of communication resources (channels) to interfaces whereas ARTS can only associate priority to messages.

Acknowledgement

I would like to acknowledge the contribution of my colleagues in the ANSA core team, particularly Andrew Herbert, John Warne and Youcef Laribi for their valuable comments and suggestions. I am also indebted to Martin Howard of BNR and Malcolm W. Vanston-Rummey of GPT for their assistance when the system was ported to HP/RT and LynxOS respectively. Finally, thanks to the anonymous reviewers who also helped to improve this paper.

Reference

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

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

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

[4] A P Black et al., Distributed and Abstract Types in Emerald, In IEEE Transactions on Software Engineering, 12(12), December 1986.

[5] G Coulson et al. Extensions to ANSA for Multimedia Computing, Computer Networks and ISDN Systems, Vol. 25, 1992.

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

[7] L Hazard et al. Towards the Integration of Real Time and QoS Handling in ANSA Architecture, ANSA Phase 3 Project Report CNET/RC.ARCADE.01, June 1993.

[8] A Herbert, An ANSA Overview, IEEE Network, January 1994.

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

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

[11] I Lee and S B Davidson, A Performance Analysis of Timed Synchronous Communication Primitives, IEEE Transactions on Computers, Vol. 39, No. 9, September 1990.

[12] W H Leung et. al., A Software Architecture for Workstations Supporting Multimedia Conferencing in Packet Switching Networks, IEEE JSAC, April 1990.

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

[14] G Li and D Otway, An Open Architecture for Real-Time Processing, in ICL Technical Journal, November 1994.

[15] G Li, Distributing Real-Time Objects: the ANSA Approach, in Proceedings of IEEE CS 1st Workshop on Object-Oriented Real-Time Dependable Systems, Dana Point, California, October 1994.

[16] G Li, Real-Time ANSAware Version 1.0: Programming and System Overview, APM document 1207, Architecture Projects Management Ltd., Cambridge U.K., May 1994.

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

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

[19] OMG, Object Management Architecture Guide, OMG TC Document 92.11.1, 1992.

[20] OSF, Introduction to OSF DCE, Cambridge, MA, 1992.

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

[22] 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.

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