Technical Report

UCAM-CL-TR-322
ISSN 1476-2986

Number 322

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Supporting distributed realtime computing

Guangxing Li

December 1993

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted August 1993 by the author for the degree of Doctor of Philosophy to the University of Cambridge, King's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

List of Tables

List of Figures

Glossary

ANSA Advanced Networked Systems Architecture

ACM ANSA Computational Model

AEM ANSA Engineering Model

AOEM ANSA Object Execution Model

ATM Asynchronous Transfer Mode

BNF Backus-Naur Form

DCE Distributed Computing Environment

DHB Distributed Hartstone Benchmark

DSE Distributed System Environment

FCFS First Come First Service

GEX Group EXecution Protocol

HB Hartstone Benchmark

HDB Hartstone Distributed Benchmark

IDL Interface Definition Language

IPC Inter Process Communication

ISO International Standards Organisation

KWIPS Kilo-Whetstone Instructions Per Second

LAN Local Area Network

LANCE The Am7990 Local Area Network Controller for Ethernet

LWC Light-Weight Channel

MIPS Millions of Instructions Per Second

MPS Message Passing Service

MSNL Multi-Service Network Level

ODP Open Distributed Processing

OSI Open Systems Interconnection

OSF Open Software Foundation

PCP Priority Ceiling Protocol

PIP Priority Inheritance Protocol

POSIX (IEEE Standard) Portable Operating System Interface for Computer Environments

PREPC PREProcessor of C

QoS Quality of Service

REX Remote EXecution Protocol

RPC Remote Procedure Call

TA Timed Automata

TCP Transmission Control Protocol

TREX Timed Remote EXecution Protocol

TRPC Timed Remote Procedure Call

UI Unix International

Chapter 1

Introduction

This dissertation is concerned with the design and construction of a distributed system environment for supporting realtime applications. The perspective and scope of this research is the entire system environment, rather than being focussed on the more narrow subsystems or algorithms. The contributions range from high-level programming abstractions down to an operating system kernel interface through the detailed engineering tradeoffs required to create, implement, and integrate the mechanisms within the environment.

1.1 Problem

Computers have been used for realtime systems for almost 50 years [90]. However, it is only recently that computer research institutions are becoming interested in realtime computing, realizing the significance of realtime systems and their increasing practical importance. Realtime systems engineering still faces many challenges [94]: current systems concepts and functions are unfavourable for the development of a general and consistent framework for realtime systems engineering. The realtime problem domain has also been further complicated by the rapid spread of distributed computing.

There are specific functional, economical and technology reasons for the use of distributed computing in realtime contexts. An application may be inherently spatially dispersed, and its realtime performance requirements do not permit the latency of the requisite communications which would be needed between these locations and a centralized computing facility. Other functional reasons for distribution are reliability and availability. Enhanced reliability and continued availability are better achieved by distribution — replication and partition of both data and function — rather than by centralization. Economically, the rapid increases in microprocessor performance and decreases in cost make it more cost-effective to have many small computers working together in place of one large computer of equivalent power. On the technology aspect, there is an increasing interest to equip modern workstations with more realtime devices. For example, the Cambridge Desk Area Network [47] uses a switching fabric to connect multimedia devices (video cameras and audio microphones) within a workstation environment. Such systems exhibit physically distributed realtime computing properties as well. It also raises the potential for distributed realtime services, such as multimedia conferencing.

Consider a distributed computing environment, in which autonomous machines communicate via various shared communication media. Processing requests can originate at any node in this network. The actual processing of the requests makes use of the resources within this environment. Such distributed realtime processing requests place a set of unique requirements including *predictability, user control, timeliness, mission orientation, and performance.* These features do not exist in today's computing environments, and must be addressed by future systems research.

Predictability is the tendency of a system to perform a set of operations in a well-defined, or *determined* fashion, so that each of these operations' timing requirements are satisfied. A fully predictable system performs operations in the same amount of time, every time, independent of surrounding conditions. Conversely, a fully nondeterministic system is one in which operation times have no guaranteed upper bound. Predictability applies to every level of the components of a realtime distributed system environment. Such an environment must provide a certain degree of predictability, even though it is not always possible to be fully predictable, to support any useful realtime performance guarantee.

User Control means a user has ultimate control of the behaviour of a system. This feature comes from the fact that many realtime applications are embedded systems (which are often static systems, and therefore it is possible to control the systems' behaviour) and that realtime applications have immense behaviour diversity (therefore it is impossible to use one fixed system behaviour for many realtime applications). The simplest method of user control on system behaviour is probably the choice of priorities for realtime tasks. By allowing a user to indicate the relative priorities of tasks, the user can affect throughput and/or responsiveness goals for the system on a much finer granularity than by a "do the best you can overall" approach. Users may also be allowed to select the scheduling policy, preallocation of system and application resources to critical services and so on.

Timeliness Realtime applications are different from the no-realtime paradigm of computation in that they impose strict requirements on the timing behaviour of the system. The correctness of a realtime system depends not only on the functional behaviour of the system, but also depends on the temporal behaviour as well. A realtime system environment must provide mechanisms which take these time related issues into account and must help application programs to meet these time constraints. A simple example is to allow an application to associate deadlines with realtime activities, and the system employs a deadline based scheduling policy to help the deadlines be met or to identify and cancel obsolete operations. Other required functions include the description and enforcement of temporal relations among related computational activities.

Mission Orientation means that an entire distributed computer system is dedicated towards accomplishing a specific purpose through the cooperative execution of one or more application programs distributed across its nodes. In the realtime sense, mission orientation also means **mission critical** — the degree of mission success is strongly correlated with the extent to which the overall system can achieve the maximum dependability regarding realtime constraints. In its simplest form, mission orientation requires that a priority or deadline associated with a mission has

global meaning when it spans over the network. More generally, global importance and urgency characteristics are propagated through the system, for use in resolving contention over system resources according to application defined policies.

Performance Realtime applications have stringent raw performance requirements. The optimised integration of application software and its supporting environment is desirable. This is in contrast with the popular layered design for non-realtime applications. Also, realtime applications often require trading off modularity, flexibility and functionality to maximize performance.

1.2 Scope

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

The scope of this research for realtime applications is *supervisory control* [86] as opposed to low-level, synchronous sampled data loop functions like sensor/actuator feedback control, signal processing, priority interrupt processing and so on. Supervisory control is a middle-level function, above the sampled data loop functions and below the human interface/management functions. This type of system does not do much direct polling of sensors and manipulation of actuators, nor does it provide extensive man/machine interfaces; rather, it deals with subsystems which provide these functions. The realtime response requirements of a supervisory control system are closer to the millisecond than either the microsecond or second ranges. Some treatments of interrupt processing (specifically, the communication interrupt processing for multi-media data) in the Computer Laboratory can be found in [23] and [61].

1.3 The Proposed System

This dissertation describes RIDE[1] — a distributed system environment developed for the task of programming and executing large realtime applications. Predictability, user control, timeliness, mission orientation, and efficiency are important attributes of the system.

The approach taken is to view RIDE as functional extensions and elaborations of a non-realtime system environment. This approach permits reuse of a large amount of existing knowledge, infrastructure and software for writing distributed applications. It is also manageable within the work for one dissertation.

The RIDE design is based on an existing distributed environment, namely the Advanced Networked Systems Architecture (ANSA) [5]. The implementation is based on the Cambridge Systems Environment. It has an experimental microkernel WANDA, and runs ANSA Testbench 3.0. The environment is composed of networked 680x0, VAX, ARM, and MIPS machines.

RIDE inherits many of the advantages of modularity, configurability, maintainability, and

[1]Real tIme Distributed system Environment

openness of the ANSA architecture. In the long term, the architectural concepts explored in RIDE are designed to be incorporated into the ANSA architecture itself as part of the ANSA phase III workprogramme.

1.4 Outline of the Dissertation

The dissertation is structured as follows:

Chapter 2 discusses the research background and addressed issues.

Chapter 3 outlines the RIDE realtime programming model.

Chapter 4 presents the RIDE communication system.

Chapter 5 discusses an approach for the description of temporal relations.

Chapter 6 presents some of the implementation details.

Chapter 7 shows various experimentations and the system performance.

Chapter 8 reviews related research.

Chapter 9 outlines the conclusions and future work.

Chapter 2

Background and Issues

This research aims to gain new experience of distributed realtime systems environment in a modern context. Specifically, microkernel and distributed system environment technologies have been chosen as the design and implementation basis, due to their acceptance as the enabling technology in modern operating system practice [39, 50].

This chapter first reviews the current work on microkernels, distributed system environments, realtime scheduling and realtime communication; and then discusses the issues addressed by this dissertation.

2.1 Microkernel Architecture

Microkernel architecture has been the subject of operating system research for the last decade, illustrated by such projects as: Amoeba [27], Chorus [34], Mach [30], the V-system [15], QNX [53] and WANDA [23]. Microkernel architecture is an approach for operating system implementation, which structures an operating system as a modular set of system servers sitting on top of a minimal microkernel, rather than using the traditional monolithic structure. This approach promises to help meet systems and platform builders' needs to support sophisticated and varied environments that can cope with growing complexity and new architectures. One of the main aims is the ability to integrate realtime applications, with new hardware technologies and distributed environments, all within an *open system* environment.

A microkernel provides generic services independent of a particular operating system, such as fast context switches, realtime scheduling, and memory management. A microkernel also provides a simple Interprocess Communication (IPC) facility that allows system servers to call each other and exchange data independently of specific system configurations.

2.1.1 Microkernel and Realtime Systems

From the viewpoint of realtime applications, microkernels have three (potentially) important features — *performance, scale* and *preemptivity.*

Performance. The stress on minimal generic low-level services, rather than a full operating system Application Programming Interface (such as UNIX), enables a microkernel to be executed in an efficient and *policy-free* way, that delivers nearly the full, device-level performance of the underlying hardware.

Scale. The combination of the generic services of a microkernel forms a standard base which can support all other system-specific functions. These system-specific functions can then be configured into appropriate system servers managing the other physical and logical resources of a computer system. By including or excluding various resource managers (for example, a file system) either statically or at run time, microkernel architecture can be *scaled down* for single board computer-based targets, or *scaled up* to encompass many processors connected by various networks.

Preemptivity. Time-consuming operating system services are performed by system servers. These server tasks can be scheduled at *client-driven* priorities, such that high priority user level tasks can preempt operating system work done on behalf of other, lower priority, user tasks. This permits high priority realtime activities with fast response time.

2.2 Current Microkernels

This section presents a brief review of the Chorus, Mach and QNX microkernels. They represent the state of the art of microkernel research.

2.2.1 Chorus

The Chorus architecture is based on a minimal realtime distributed *Nucleus* that integrates distributed processing and communication at the lowest level. The Nucleus provides generic mechanisms to sets of independent servers called *subsystems*, which coexist on top of the Nucleus. The Nucleus is divided into four major components:

- a **realtime multi-tasking executive.** It controls the allocation of local processors, manages priority-based preemptive scheduling of Chorus threads, and provides primitives for fine grain synchronization of, and low-level communication between, threads.
- a **virtual memory manager.** It is responsible for managing memory requirements of the system.
- a low level hardware **supervisor** which dynamically dispatches external events such as interrupts, traps and exceptions to dynamically defined routines or ports.
- an **IPC manager.** It provides the global communication services (exchange of messages through ports).

The Chorus/Mix subsystem has been provided as a server above the Nucleus to provide system builders with a standards-based (UNIX System V), realtime and distributed UNIX environment.

2.2.2 Mach

The Mach microkernel provides facilities for the management of CPU, communication, virtual memory and devices. The key features are:

- **Task and Thread Management.** A task is a passive resource abstraction, consisting of an address space and communication access. Computation within a task is performed by one or more threads sharing resources in the address space. Threads are scheduled to processors, and may run in parallel on a multiprocessor. Two classes of scheduling policies are provided: fixed priority and timesharing. The schedulability of tasks, their threads, and processors can be controlled by user level programs.
- **IPC** is via the port mechanism: a communication channel implemented as a message queue. All services, resources, and facilities within the Mach kernel, as well as those exported by particular Mach tasks or servers, are represented as ports and are manipulated by sending messages to these ports. Ports are protected by a capability mechanism.
- **Memory Object Management.** The address space of a task is represented as a set of mappings from linear addresses to offsets within Mach memory objects. The Mach virtual memory service manages physical memory as a cache of the contents of memory objects. Memory object backing storage is implemented by a user level server.
- **Device Support.** Mach provides low-level device support. Each device is represented as a port to which messages can be sent to transfer data or control the device.

Traditional operating systems are implemented as Mach applications. Some example applications are 4.3BSD UNIX, MS-DOS, and the OSF/1.

Realtime Mach

Realtime Mach [101] has been developed by the Advanced Realtime Technology group at Carnegie Mellon university, based on the experience of the ARTS [100] distributed realtime kernel. The objective of the realtime Mach project has been to develop a realtime version of Mach that can support a predictable realtime computing environment and to develop an associated realtime tool set. Realtime Mach has the following realtime features:

- a **Realtime Thread Model.** The realtime-thread model supports a predictable realtime scheduler and provides a uniform interface to both realtime and non-realtime threads. Apart from priority, timing attributes may be associated with a realtime thread, which include deadline, deadline type (hard or soft deadline), worst case execution time, and periodic properties (start time, period, phase offset etc.).
- an **Integrated Time-Driven Scheduler.** The scheduler has two layers — policy and mechanism. A scheduling policy is a self contained object, and can be associated with a processor or a processor set. Different policies can be assigned to different

processor sets. Apart from providing traditional realtime scheduling policies such as fixed priority, the scheduler also provides rate monotonic scheduling policies (see Section 2.5). The mechanism layer manages the actual thread context switch within the kernel.

- **Realtime Thread Synchronization.** The synchronization mechanism in realtime Mach is based on mutual exclusion using a lock variable. The lock acquire and release operations provide a priority inheritance mechanism (see Section 2.5) in order to avoid the unbounded priority inversion problem.
- a **Memory Resident Object Manager.** This mechanism eliminates the unpredictable page fault handing delay associated with the Mach memory object management by using *eager* evaluation rather than the Mach *lazy* evaluation technique.

2.2.3 QNX

The QNX architecture, like Chorus and Mach, is composed of a small microkernel surrounded by a team of cooperating processes that provide higher-level operating system services. Realtime performance has been a main driving force for the development of QNX. The QNX microkernel implements four services:

- **IPC.** The microkernel provides process to process synchronous message-passing mechanisms. Message queues are implemented as servers based on this lower-level service. Processes can request that messages be delivered in priority order, and that process execution proceeds at the priority of the highest-priority blocked process waiting for service.
- **Process Management.** The process scheduling primitives conform to the IEEE POSIX 1003.4 (realtime thread) draft standard [59]. QNX provides preemptive, prioritised context switching with round-robin and FIFO scheduling.
- **Low-level Network Communication.** Low-level network communication is provided by an optional kernel-resident process, the network manager. When present, the network manager provides the microkernel with the facilities needed to move messages to and from other microkernels on a network, transparently to other processes.
- **Interrupt Dispatch.** QNX allows user processes to connect a handler within a user process to an interrupt vector within the kernel. The connected handler can then be called by the kernel in response to physical interrupts. Such interrupt handlers can be dynamically added and removed from a running system.

2.3 Distributed System Environment

A distributed system environment (DSE) is a run-time system that provides a set of abstractions and tools to support the writing of programs in a distributed environment. The effect of using a DSE is that applications are automatically supported by a run-time environment which incorporates a set of *distribution transparency* mechanisms. These shield

application designers and users from the technological complexities involved in distributed application programs. Remote Procedure Call (RPC) [12] and client-server interactions are widely accepted as DSE technical apparatus.

It is now recognised [50] that distribution transparency can be broken down into a number of individual transparency issues:

- location transparency — masking off the physical location of services.
- access transparency — masking any differences in representation and operation invocation mechanism.
- concurrency transparency — masking overlapped execution.
- replication transparency — masking redundancy.
- failure transparency — masking recovery of services after failures.
- resource transparency — masking changes in the representation of a service and the resources used to support it.
- migration transparency — masking movement of a service from one application to another.
- federation transparency — masking administrative and technology boundaries.

DSE's have been a central subject of distributed system research for the last ten years. Many DSE's have been proposed to handle various aspects of distribution transparency. Some are dedicated systems to extend low-level operating system abstractions (such as message passing) with support for distributed objects, as illustrated by COOL [68] and MachObjects [44]. Some are special systems for a particular application area like fault-tolerance or persistence, as illustrated by the ISIS [11] and OPERA [32] projects. Others are more general systems that handle many transparency issues; these are illustrated by the OSF DCE [38], UI Atlas [56] and the ANSA systems.

2.3.1 Realtime Distributed System Environment

Despite the relative maturity of DSE research, realtime DSE remains a neglected, if not unaddressed, topic. The result is that even if low-level microkernels provide realtime services, a DSE provides no corresponding abstractions to use these services. Even worse, a DSE often mask off the realtime features of microkernels. This is unfortunate because realtime performance has been one of the two main driving forces (along with distribution) behind microkernel architecture research. The main aim of this research is therefore to extend the realtime features of a microkernel to the DSE level.

One common misconception is perhaps that DSE is not the suitable technology for realtime applications because RPC (as one of the main technique basis of DSE) is often criticised for providing poor performance or is not fast enough. This is a misconcept because the objective of realtime computing is to meet the timing requirements of an application, rather than being fast. The most important property of a realtime system is *predictability* (see Section 1.1). On the other hand, fast is a relative term. As technology progresses, there

will be faster and faster RPC systems. Even now it is not difficult to provide milliseconds level RPC calls (as the required performance for the *supervisory control* targeted by RIDE, also see Section 1.2). For example, there are already reports of systems that can provide hundreds of microseconds level RPC calls [10] [63]. Fast computing is helpful in meeting stringent timing constraints, but fast computing alone does not bring realtime properties.

2.4 Example Distributed System Environments

This section presents a brief overview of the Chorus COOL, Mach MachObjects, DCE and ODP distributed system environments.

2.4.1 COOL

The Chorus Object-Oriented Layer (COOL) is a system designed to provide generic support for distributed object-oriented programming. The system was developed by Chorus systems and INRIA. The architecture of COOL is shown in Figure 2.1.

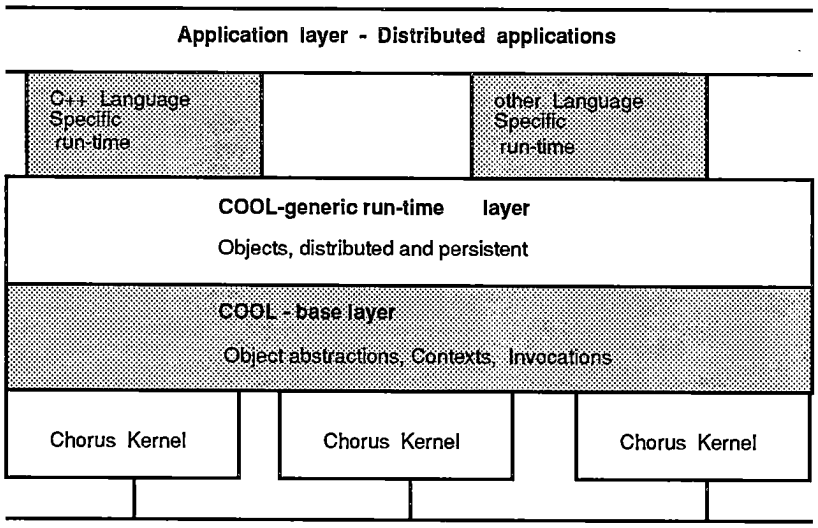

Figure 2.1: The COOL Architecture

The COOL base layer is designed as a generic support platform for object based systems. The layer provides a set of abstractions mapped directly on to the Chorus microkernel which extend the microkernel interface to include the notions of objects and object management. The layer supports the creation of typed objects and the subsequent invocation of operations on objects. The layer also manages the mapping of objects into Chorus virtual address spaces (referred to as contexts in COOL) and the saving of such contexts as a form of coarse grained persistence.

The COOL run-time layer provides mapping from a language or system level object model to the COOL base facilities. The current system has been focused on a run-time layer for C++. This layer transparently maps C++ objects on to COOL base facilities thus providing C++ programmers with the ability to develop dynamic distributed applications. The COOL/C++ objects may be persistent, and can be invoked both locally and remotely.

The ANSA Testbench has been used by the COOL environment as an RPC mechanism for remote object invocations.

2.4.2 MachObjects

MachObjects is an object-oriented run-time environment on top of the "pure" Mach microkernel. It provides dynamic typed objects, delegation, and a generic transparent RPC mechanism. Transparent access to remote objects is the key feature of MachObjects.

The Mach system servers are written using MachObjects. All the entities of the system, like files, directories, etc., are represented by objects. The interface presented by the system is defined as the set of operations exported by the operating system objects. Server-side objects are represented at the client by a proxy which completely hides distribution from the user code. Proxies may forward invocations to the server or may themselves perform some work.

2.4.3 DCE

The Distributed Computing Environment (DCE) from the Open Software Foundation addresses the problem of interoperability by providing a distributed platform, a run-time system, that may span multiple architectures, protocols, and operating systems. DCE is a layer between the operating system and network on the one hand, and the distributed application on the other. The architecture of DCE is shown in Figure 2.2.

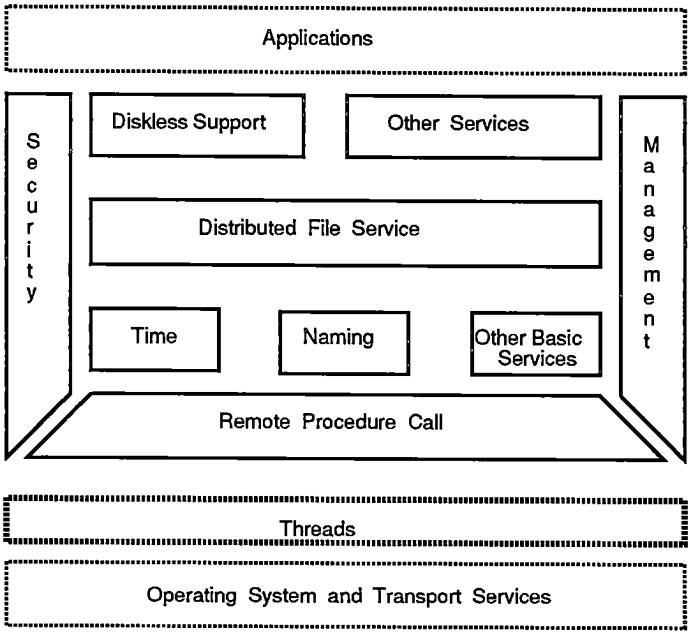

Figure 2.2: The Distributed Computing Environment Architecture

DCE's services are organized into two categories: Fundamental Distributed Services and Data-Sharing Services. The Fundamental Distributed Services are: RPC service, naming (directory) service, time service (distributed time reference), threads service, and the

security service. The DCE threads service is conceptually a part of the operating system layer. If the host operating system already supports threads, DCE can use this, otherwise DCE presents a user-level threads package instead. The DCE Data-Sharing Services are based on the Fundamental Distributed Services, and include distributed file system service, diskless support service and personal computer integration service.

2.4.4 Open Distributed Processing

The ISO standards community have established a work program to define a Basic Reference Model of Open Distributed Processing (RM-ODP) [61] The primary goal of ODP is interworking between applications and sharing of data between organizations. The RM-ODP is organized as a set of five languages, called the enterprise, information, computational, engineering and technology languages, for describing different *views* of distributed systems.

The enterprise and information languages are introduced as the means to make the link between technical solutions and user requirements. The computational language is a formalization of a programming language based on abstract data types suitable for specifying distribution in functional terms. The engineering language focusses on the structural aspects of the support for distribution. **Declarative, selective** and **modular** distribution transparency is the main characteristic of the engineering language. The technology language is used to map the logical architecture to product architectures.

APM's Advanced Networked Systems Architecture (ANSA) is a refined model for ODP. APM also developed a prototype DSE for ODP, called the ANSA Testbench.

The ODP community has not made any serious attempt at supporting realtime applications, but there is some ongoing work to incorporate stream interfaces with ODP for multimedia applications [31]. Realtime is currently an important issue in the development of ANSA Phase III at APM.

2.5 Realtime Scheduling

Resource scheduling plays a central role in any non-trivial realtime system. Substantial research has focused on the performance of realtime systems in terms of **schedulability analysis** — the feasibility of scheduling the required workload onto the available resources (processors and so on).

A scheduling approach that allows schedulability analysis is considered a basic requirement for designing hard realtime systems [45]. *Hard realtime systems* are those realtime systems in which the time constraints of a task set play a major role; not meeting such time constraints may lead to catastrophe results. *Soft realtime systems* are those realtime systems in which meeting the time constraints of a task set are desirable, but failing to do so does not cause a system failure. A system is said to be *schedulable* if it meets all deadlines of a task set.

Cyclic executive and rate monotonic scheduling are two most popular scheduling approaches in current realtime applications. There is also an increasing interest in using deadline based scheduling for practical realtime systems.

Cyclic Executive Scheduling. Cyclic executive scheduling [7] was a common technique used before concurrent programming became popular, and is still used where concurrent programming is not supported. This approach offers a framework for scheduling periodic tasks. It performs a sequence of actions during a fixed time period. The execution is divided into two parts. The major cycle schedules computations to be repeated indefinitely, and is composed of minor cycles. Each task is divided into subcomponents so that the execution of each subcomponent fits into a minor cycle in a way that satisfies the task timing constraints. In other words, this approach forces a programmer to *pre-schedule* programs based on their static knowledge to ensure predictable execution timing.

Rate Monotonic Scheduling. Rate monotonic scheduling [76] uses a preemptive fixed-priority scheduling algorithm that assigns higher priority to the tasks with shorter periods. This approach is *optimal* among fixed-priority scheduling schemes. The CPU utilization of a task, U(i) is calculated by U(i) = C(i)/T(i), where C(i) and T(i) are the execution time and period of task i, respectively. Assume a task's deadline is the same as its period, n independent periodic tasks can meet their deadlines if the following formula holds:

$$\sum_{i=1}^{n} U(i) \leq n\left(2^{1/n} - 1\right) \qquad (\approx 0.69)$$

This formula is very simple but pessimistic. In the case of a harmonic task set where all periodic tasks start at the same time and all periods are harmonic, the CPU utilization is schedulable up to 100 percent. More precise schedulability analysis of the rate monotonic algorithm is discussed in [70]. Rate monotonic scheduling does not require programmers to split tasks manually as the cyclic executive does, but the tasks must be preemptive and there is some overhead for context switching.

Tasks	*Period*	*Execution*	*Utilization*
T1	50	12	24%
T2	40	10	25%
T3	30	10	33%

Table 2.1: A Set of Tasks with 82% Utilization

Deadline Based Scheduling. Deadline based scheduling uses explicit information about application supplied deadlines. With the earliest (or shortest) deadline scheduling, the scheduler runs the task with the closest deadline. Deadline based scheduling is a *dynamic* scheme that allows priority to change with time. It was shown in [76] that earliest deadline scheduling is *optimal* among all scheduling schemes. This approach which is preemptive will feasibly schedule a periodic task set as long as:

$$\sum_{i=1}^{n} U(i) \leq 1$$

As a comparison, the task set shown in Table 2.1 with 82% CPU utilization cannot be scheduled by the rate monotonic scheme, but can be scheduled by the earliest deadline scheme. Another attractive feature of earliest deadline scheduling is that it deals well with systems that are mainly composed of aperiodic tasks. The disadvantage of earliest deadline

scheduling is that during a transient overload, deadlines are missed in an unpredictable fashion [92].

It must be emphasized that predictable scheduling in a hard realtime system with general resource (such as multiprocessors and networks) constraints and processing constraints (such as processing orders) is very complex and a subject of much research [106]. General solutions are not possible in polynominal time and therefore are not appropriate for practical applications. Current realtime scheduling still relies heavily on manually tuning programs in an application-based style.

2.5.1 Scheduling Task Synchronization

When tasks interact via a synchronization primitive, they may become blocked in the middle of their execution. In order to calculate their worst-case (or average) execution time it is necessary to know the potential length of this blocking time. A static priority scheme is no longer adequate because it gives no bound for the period of time a task can be blocked. This is known as the *unbounded priority inversion* problem. Similar problems exist for the earliest deadline scheme.

Priority Inversion Problem Illustration. Suppose a low-priority task L is preempted inside a critical region which is guarded by a binary semaphore. A high-priority task H then attempts to enter the critical region and blocks. H cannot continue until L leaves the critical region, and runnable tasks with priorities in the gap (see Figure 2.3) between H and L prevent L from running. H blocks for an unpredicatable and possibly lengthy period of time.

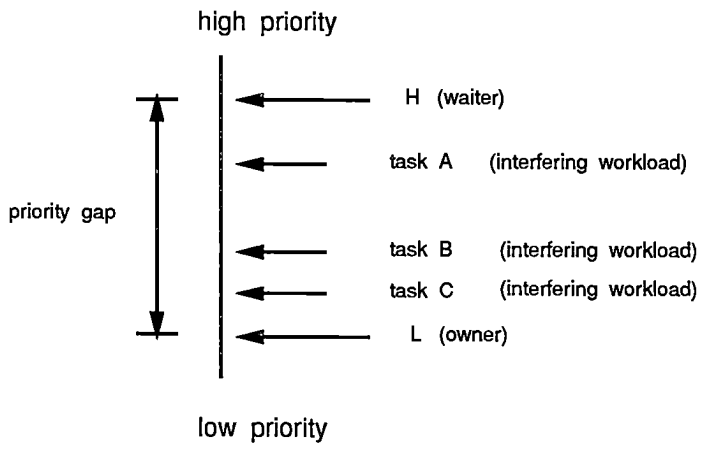

Figure 2.3: Priority Inversion

To address the problem, Sha et al [93] proposed two protocols based on binary semaphores (mutexes). They are called the basic **priority inheritance protocol** (PIP) and the **priority ceiling protocol** (PCP). The PCP provides better worst-case bound which is the duration of execution of a single critical section of a lower priority task. Both PIP and PCP have been adopted by the POSIX 1003.4a [59].

The idea of the PIP is that when a task blocks higher priority tasks, it executes its critical section at the highest priority level of all the blocked tasks. After exiting its critical

section, the task resumes its original priority level.

The **priority ceiling** of a mutex S is defined to be the highest priority of all the tasks that may acquire S. The priority ceiling value is assigned to a mutex at initialization time. Under the PCP a task t is allowed to acquire a mutex only if t's priority is greater than the highest priority ceiling of all the mutexes that are locked by other tasks in the system. Otherwise, t is blocked and the task which has locked the mutex with the highest priority ceiling inherits t's priority until it releases the mutex.

Like the case for resource constrained realtime scheduling, general-purpose solutions for predictable realtime synchronization are not possible in polynomial time. Mok [82] showed that the problem of deciding the schedulability of periodic precesses using semaphores for mutual exclusion is NP-Hard. Most practical approaches impose some constraints on interprocess synchronization. For example, the PCP allows only mutual exclusion and not condition synchronization between tasks.

2.6 Realtime Communication

Ensuring message-level timing correctness for interprocess communication is difficult using current technology and a substantial research challenge. There are at least *media, protocol* and *network scheduling* problems to be solved before a predictable realtime communication system become possible.

The problem of temporally (periodicity, jitter) and spatially (bandwidth) constrained communication has been studied in several directions. In the context of embedded realtime systems, these efforts have been directed mainly towards designing media access protocols for multi-access networks [57]. For example, Strosnider and Marchok [96] use a variation of the rate-monotonic scheduling algorithm to control access to a token-ring network. Priorities are assigned to message sources at design time based on the periodicity of message generation and the message schedulability can be checked then. This approach, however, works only for simple and static communication requirements and does not scale to internets.

In the context of multi-media communication systems, these efforts have been mainly conducted on the Asynchronous Transfer Mode (ATM) networks [78]. Media such as video and voice have both temporal and spatial communication requirements, and therefore their transportation needs to be done in realtime. ATM networks transfer data in small packets, known as cells, allowing fine granularity bandwidth sharing and reducing the delays due to contention and routing. ATM has been adopted as the basis for the Broadband Integrated Services Digital Network (B-ISDN) [58].

ATM communication is connection-oriented at the lowest level. All information is transferred in a *Virtual Circuit* assigned for the complete duration of the connection. Both network and operating system resources can be reserved for individual virtual circuits at connection setup time to guarantee their communication requirements which are commonly referred to as the desired *Quality of Service* (QoS). QoS specification, management and guarantee are still ongoing research topics in the network area, and there are no viable solutions currently. A survey by [64] shows various QoS definitions in terms of traffic characteristics. Dixon [23] discussed system mechanisms to support QoS in the Cambridge

ATM environment. Nicolaou [84] discussed the QoS management within the ANSA environment to extend QoS from operating systems to the applications. Sreenan [94] discussed QoS requirements for multimedia synchronization services.

QoS communication is not addressed in this dissertation. Rather, connection oriented IPC communication — virtual circuits that allow QoS associations — are assumed to exist on underlying operating systems. This allows the dissertation to concentrate on the integration of realtime communication with computation.

2.7 Summary of Background

Microkernel architecture structures an operating system as a modular set of system servers sitting on top of a minimal microkernel. From the viewpoint of realtime applications, the architecture offers three (potentially) important features — *performance, scale* and *preemptivity.* Realtime performance and distribution have been the two main drive forces in microkernel architecture research.

A DSE provides a set of *distribution transparency* mechanisms. These shield application designers and users from the technological complexities involved in distributed application programs. Despite the relative maturity of DSE research, realtime DSE remains a neglected topic.

Cyclic executive and rate monotonic scheduling are the two most popular scheduling approaches in current realtime applications. There is also an increasing interest in using deadline based scheduling for practical realtime systems.

ATM networks are considered to be of fundamental importance to realtime communications.

2.8 Addressed Issues

The RIDE system developed in this dissertation is concerned with the design and construction of a DSE for supporting realtime applications. The main issues addressed by RIDE are as follows:

- realtime programming model.
- timed remote procedure calls.
- description of temporal behaviours.
- empirical validation.

The following subsections provide a brief overview for each of the four major issues.

2.8.1 Realtime Programming Model

The essence of a realtime programming model is to provide the basic abstractions so that stringent timing constraints of realtime activities are respected (guaranteed at best). A serious difficulty is that the actual timing characteristics of software are determined not only by the raw processor speed, but also by the sharing policy for scarce resources. For example, the realtime response of a time-shared system depends heavily on the processor scheduling policy of its operating system. 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.

The RIDE realtime programming model is based on the ANSA computation and engineering models. As in the ANSA system, objects provide the basis for distribution, interfaces of objects provide service access points, and named operations of an interface provide the actual services. Abstractions, mechanisms and policies are developed to allow a programmer to access and control the resource allocation of the supporting environment. Tasks (representing processor resources, see Chapter 3) and communication channels (representing communication resources, see Chapter 4) 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**, **user control** and **mission criticality** are the main concerns of the RIDE realtime programming model.

2.8.2 Timed Remote Procedure Calls

ANSA is an RPC based system. A basic goal of many RPC systems is to make the semantics of a remote call as close as possible to that of a local call. As already mentioned, this is known as distribution transparency. However, distribution cannot be completely ignored: applications will have to deal with the possibilities of concurrent access to shared resources, variable latency in accessing resources and communication failures disturbing access to resources. The semantics of remote calls are implemented by RPC protocols. Perhaps the weakest semantics are to provide no guarantee when a failure occurs; an invocation might result in the actual program being called zero, one, or more times. Stronger semantics are more useful and are difficult to achieve. Two often referred to semantics are *exactly-once* and *at-most-once* executions. Realtime applications add another dimension to the problem: timeliness — arbitrary delays associated with synchronous RPC invocations cannot be tolerated.

The RIDE solution to the timed RPC is the design of a dependable RPC protocol through which reasonable timing constraints (representing different tradeoffs between consistency and strictness) of a remote invocation can be specified clearly and enforced. This relieves the additional burden of having to monitor and manage timing constraints by application programmers during remote calls. This is discussed in Chapter 4.

2.8.3 Description of Temporal Behaviours

Realtime systems and especially distributed realtime systems usually consist of many realtime activities with different but related time constraints. These activities must be temporally related to each other so all time constraints can be met. A realtime programming system therefore must provide a *temporal synchronization* facility. Some important requirements of this facility are as follows.

- the capacity to express different types of timing requirements.
- provision of a useful abstraction. This is better achieved by being based on a model that makes it easier to ensure the program's temporal correctness.
- preserve the separation of concerns so that the cooperative computations do not have to share assumptions about one another.
- provide mechanisms for run-time systems to enforce timing constraints.

The approach taken is to keep in-line with the ANSA Computational Model. The temporal synchronization facility is represented as a kind of special service accessible through well defined ANSA interfaces. Normal invocations on the interfaces are used to notify and enforce temporal synchronization conditions. The model used is **timed automata** as discussed in Chapter 5.

2.8.4 Empirical Validation

Given that there are few technology-independent lessons to be learned in systems research, it is important to evaluate some of the basic premises of systems design through actual design and implementation efforts. It seems clear that computer systems research must be validated by empirical studies — it is impossible to do credible systems research without actually building and using systems.

The systems concepts described in this thesis have been validated through the construction of a working system based on extensions and elaborations of a microkernel and a distributed system environment on bare hardware (see Chapter 6). This approach exposes many technology-driven, low-level engineering details that are critical to the validation of high-level concepts and are often not considered or neglected in other concept study, emulation or simulation approaches.

The Distributed Hartstone Benchmark [79] has been ported to the RIDE run-time system and used to evaluate the synthetic performance of RIDE. The multiprocessor effects on RIDE are also evaluated. These are discussed in Chapter 7.

2.9 Summary

This chapter has examined the background relevent to realtime distributed system environment and found it a neglected topic. The goal of this dissertation is to construct such a system environment which can better support distributed realtime applications.

Chapter 3

The RIDE Realtime Programming Model

RIDE inherits an object-based programming model from the ANSA architecture. This chapter discusses the realtime aspects of RIDE objects. The structure of RIDE objects is examined along with object invocation mechanisms, the handling of priorities and deadlines, resource allocations, scheduling mechanisms and policies, and the application's control over scheduling.

General distributed object execution is discussed first. This is followed by a discussion of the ANSA object execution model in terms of tasking and scheduling. The RIDE counterparts are then presented.

3.1 Distributed Object Execution

The use of an object-oriented data model and the client-server execution model makes the distribution of data and the processing implicit in nature. In non-realtime environments, object-oriented design has been successful in simplifying the design, implementation, and maintenance of software in many distributed systems such as Comandos [53], EMERALD [29] and ANSA.

Object interdependence can be classified into two categories: *static* interdependence — the structural relationships between objects, and *dynamic* interdependence — the interactions between objects. Many useful results are known about the static relationships between distributed objects [50] [13]. 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 [29], the *passive object* model [1], and the *actor object* model [6].

For realtime applications, this execution aspect is of vital importance — it has fundamental impact on the *predictability* of computational activities. Realtime 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 realtime activities). The latter issue is often neglected and considered irrelevant engineering detail in non-realtime computing. Distributed realtime systems must provide support for the specialized requirements of realtime 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 realtime world.

3.2 ANSA Object Execution

The ANSA Object Execution Model (AOEM) is defined by the ANSA Computation Model (ACM) and ANSA Engineering Model (AEM). ACM and AEM are reviewed first before AOEM is discussed.

3.2.1 ANSA Computation Model

A computation model is a framework for describing the structure, specification and execution of programs. The principle behind and the concepts underlying the ANSA architecture are articulated via the ACM [87]. This subsection briefly summarises the overall concepts of the ACM.

The key ACM concepts are:

(Computational) Object: a unit of program modularity having state and *operations* for initializing, accessing and updating that state. Object state may contain references to the interfaces of itself and other objects.

Interface: a view of an object as an abstract service. An interface is specified as a set of operations together with synchronization and ordering constraints on the use of these operations.

Operation: part of an interface. An operation has a *signature* and a body which defines the effect and outcome from an *invocation* of the operation.

Signature: a specification of the name of an operation, the number and interface types of the argument parameters and, optionally, a set of *terminations* which specify the possible outcomes from the operation.

Activity: the agency by which computations make progress. An activity may pass from one object to another by the first *invoking* an operation on an interface of the second. Activities may split into parallel sub-activities and later recombine. New activities can be initiated to proceed in parallel, these may be able to communicate with other activities but are not dependent upon their initiating activity.

Termination: the specification of a set of possible outcomes from *invocations* of an operation. A termination has a name and specifies the *interface types* of the result parameters from an outcome with that name.

Interface type: the signature of the operations in an interface of the type.

(Operation) Invocation: the execution of the body of an operation defined by a reference to an interface and an operation name in a context established by the referenced interfaces and a set of arguments.

Server: in the context of an invocation, the object which provides the interface containing the operation being invoked.

Client: in the context of an invocation, the object from which the invocation was initiated.

The ACM is in two parts:

- **an interaction model** defines permitted forms of interaction and a type scheme within which potential interactions are to be classified. The interaction model consists of an invocation scheme and a type scheme.
- **the construction model** defines elements from which the interacting objects may be constructed.

The invocation scheme defines how clients may use interfaces provided by servers. Two kinds of operation, *interrogation* (call) and *announcement* (cast), are permitted. Invocation of an interrogation is a synchronous request/response style. Invocation of an announcement is an asynchronous request only style, a new activity is created in the server and the invoking activity continues in the client.

The type scheme provides a set of types into which interfaces are classified and defines a relation over interface types that allows the detection of the possibility of interaction errors before the interaction commences.

The ANSA construction model provides the elements necessary to construct objects that conform to the ANSA interaction model.

3.2.2 ANSA Engineering Model

The AEM provides a framework for the specification of mechanisms to support distribution of application programs that conform to ACM. The details of AEM can be found in [48]. The AEM contains a number of sub-components and supports a number of application-level components as shown in Figure 3.1.

Transparency Mechanisms provide a uniform interface for distributed applications that address the problems and benefits of distribution. The transparency mechanisms communicate with one another via the nucleus and the network to achieve the desired transparency.

Nucleus is the part of the AEM which provides minimal and sufficient support for the implementation of distribution. It encapsulates all of the heterogeneity of processor and memory architectures. The Nucleus itself is not distributable.

The main concepts of the AEM may be summarised:

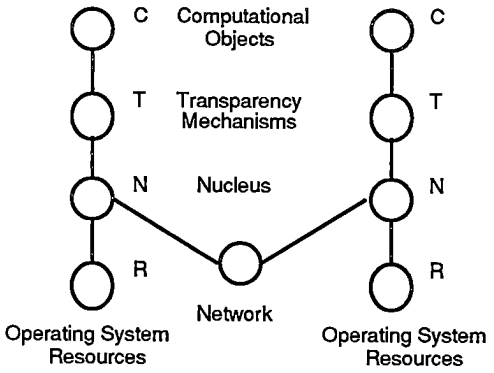

Figure 3.1: Engineering Model

Capsule: the collection of computational objects (in engineering form), transparency mechanisms and nucleus forming a virtual node of a network. It can be seen as the abstraction of an address space in a local operating system to provide the unit of protection and failure atomicity.

Thread: a sequence of instructions modelling a computational model activity within a capsule. It represents a unit of potentially concurrent activity that can be evaluated in parallel with other threads, subject to synchronization constraints.

Task: a virtual processor which provides a thread with the resources (e.g. a stack) it requires to progress. Tasks[1] provide the resources for real concurrency. An ANSA task is conceptually equivalent to an operating system *thread.*

Interface Reference: an interface reference is an identifier which contains sufficient information to allow the holder (the client) to establish communication with the interface denoted by the reference (the server). Interfaces have types (corresponding to their code component) which may be instantiated multiple times with different state (corresponding to their data component). Such instantiations are called **interface instances**, and interface references always refer to interface instances.

Channel: the abstraction for initiating operations to a specific remote interface and for receiving invocations on a specified interface. The initiating side (client) end-point of a channel is called a **plug**. The receiving side (server) end-point is called a **socket**. There is a one-to-one correspondence between channels and interface instances. Channels are asymmetric in that a channel may have many clients (plugs) bound to it, but only one server (socket).

Interpreter: a portion of the nucleus. It can be viewed as defining an instruction set for a distributed abstract machine. It interprets inter-object interactions (invocations), performs all argument and result processing, and links threads to sessions (a session is a cache of a plug or a socket) and transfers buffers between them. It also provides the necessary session, thread and task state changes to complete the execution of each instruction.

[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 avaliable to run it.

3.2.3 ANSA Object Execution Model

The AOEM can be summarised as follows.

- objects export services through interfaces.
- threads are created either explicitly for concurrent computational activities or implicitly by the invocations between objects. In the latter case, a thread embodies a distinct run-time agent for a client in its server side, representing the invocation on a computational interface.
- the infrastructure (capsule) is in charge of the management of resources (tasks, buffers etc.) in the system, and of their allocation to the different threads.

This means the system behaviour is completely dependent on the system's resource management policy. Also, the infrastructure offers no possibility of interacting with this management. Therefore, the resulting behaviour is totally non-deterministic, and nothing can be guaranteed; it depends entirely on the system workload.

3.2.4 ANSA Object Execution Model Deficiencies for Realtime Applications

To be more specific, ANSA Testbench 3.0 (the same applies to the Testbench 4.0) is used as an example to detail the AOEM. In the Testbench, its time-sharing characteristics of tasking and scheduling can be summarised as follows:

- multiplexing of one thread queue. The queue is used for all interfaces within a capsule; and all system tasks are homogeneous — they are allocated for serving any threads (requests on any interfaces).
- thread enqueue policy (and thus request service scheduling policy) is First Come First Service (FCFS).

Based on this single capsule-wide thread queue with a pool of tasks, the ANSA tasking system is very efficient at the task/thread resource sharing. However it imposes severe constraints on flexible and realtime scheduling. For example, it precludes the possibility of preallocating tasks for realtime interfaces (services). One aspect of the non-predictability caused by this design is if all system tasks have been assigned to some time-consuming non-realtime threads, newly arrived realtime requests (threads) have to wait until the completion of the non-realtime threads. Also this design precludes the possibility that an application performs its own resource management, synchronization and scheduling on the basis of services (interfaces) and tasks.

The simple FCFS thread enqueue policy precludes any realtime performance, when the object is executed in an open environment where time constrained and non-constrained operations are allowed to be requested dynamically.

3.3 RIDE Objects

The RIDE object model is inherited from the ANSA object model, with extensions to provide resource allocation and realtime scheduling support.

Like ANSA objects, a RIDE object is composed of data, one or more tasks of execution, and a set of exported interfaces. A new abstraction, **entry**, is introduced as the basic mechanism for realtime 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 with an entry id. The entry is an engineering concept which is confined within a capsule.

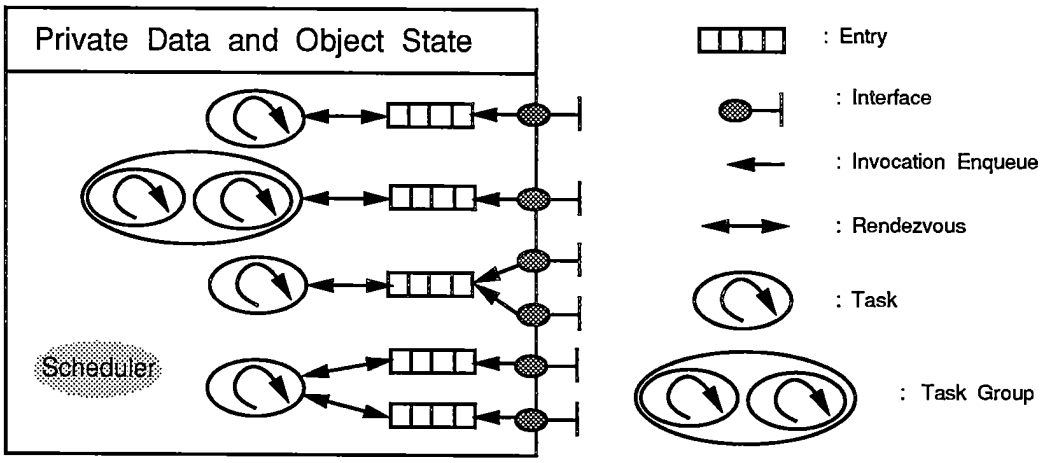

Figure 3.2: RIDE Object Illustration

In Figure 3.2, a graphical illustration of a RIDE 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 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. These policy issues are discussed in the further sections.

In an object model like RIDE 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 realtime scheduling properties, for example, *preemptivity* (as explained later in Section 3.6.2).

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 (an example is given in Section 3.10).

The user control over system tasking behaviour is further enhanced by the scheduling policy/mechanism separation used in the RIDE architecture (detailed in Section 3.5).

These resource management activities can all be done dynamically, increasing the flexibility and usefulness in an open dynamic environment. Some typical system configurations are illustrated below. Their combinations are straightforward.

The simplest form (Figure 3.3) is *Shared Single Entry configuration*, in which all interfaces share a single entry with all tasks serving all incoming requests on all interaces.

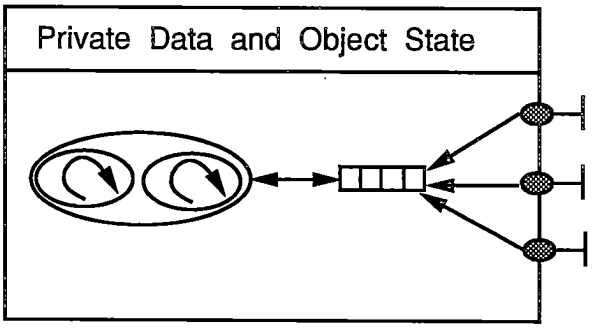

Figure 3.3: Shared Single Entry (ANSA) Configuration

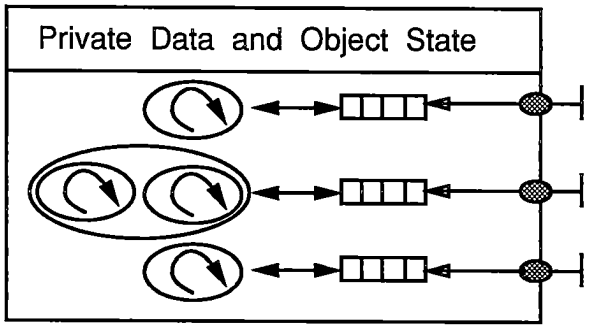

Figure 3.4: Multiple Single Entries Configuration

Another simple form (Figure 3.4) is *Multiple Single Entries*, in which each interface has its own entry.

Another interesting simple form (Figure 3.5) is *Single Task Multiple Single Entry*, in which the single task decides at its run-time which entry (interface) it would like to serve.

A combined configuration is illustrated in Figure 3.2. It contains the three simple configurations.

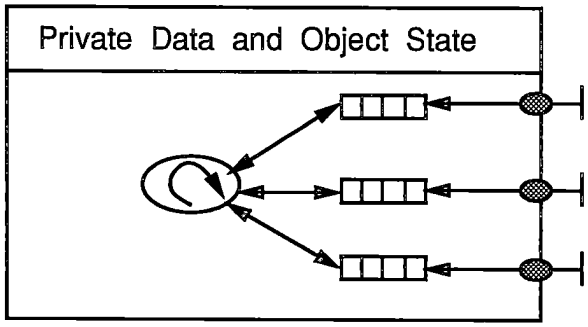

Figure 3.5: Single Task Multiple Single Entry

3.4 Object Invocation

The act of requesting that an operation of an interface be executed is termed an *invocation.* Two types of invocation are provided in ANSA, and are inherited by RIDE. These are *synchronous* calls and *asynchronous* casts. Each invocation is conveyed as a message to the invoked object, and is then transferred to a thread in the capsule where the invoked object resides.

To support the mission-critical requirements (see Section 1.1), there must be some means to enable the urgency of a computational activity to be spread among all the nodes it needs to access; and that urgency information should be used by the system resource scheduler to resolve resource contention so that important or more urgent computational activities have better access to system resources. This is done in RIDE by allowing the association of an optional priority 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 also passed and becomes a property of the thread, which may then be used as a scheduling parameter on the server site.

The priority and/or deadline of an invocation is independent of its contents (the invocation parameters) and context (the invocation thread). Allowing explicit invocation priority (and/or deadline) has several benefits: (1) it allows extra flexibility in conjunction with the server scheduler, in determining how the invocation is to be processed; (2) it allows a low-priority invocation to be sent from a high-priority task without having to enhance the server (thread) task's priority; (3) likewise, a low-priority thread may send a high-priority invocation to a server indicating the system has entered an urgent situation.

It should be pointed out that the priority and/or deadline is just a client's objective view of the criticality of an invocation; how that will affect the system resource management is also determined by the scheduling policy (the interpretation of the scheduling parameters) and the resources allocated for the service. This is further explained in the following sections.

3.5 Scheduling

The main goal of the RIDE 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 realtime programming. Realtime programming models (priority based, deadline based, imprecise computation [65], reactive kernel [46] etc.) have been devised for specific applications. No single existing model is likely to meet all realtime requirements. Therefore, an ideal general purpose realtime support environment should provide multiple models of realtime programming. This is supported in RIDE by the multiple application-selectable scheduling policy modules on top of a shared set of scheduling mechanisms.

The RIDE system scheduling behaviour is defined in layers as:

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

Task scheduling and thread scheduling are two separate, but related scheduling domains. Task scheduling is defined in the nucleus or the underlying operating system kernel. Thread scheduling is defined per entry. Task scheduling manages multiplexing of task executions over processor(s). Thread scheduling manages the multiplexing of requests (thread) over tasks. Figure 3.6 illustrates the structure of this multiplex.

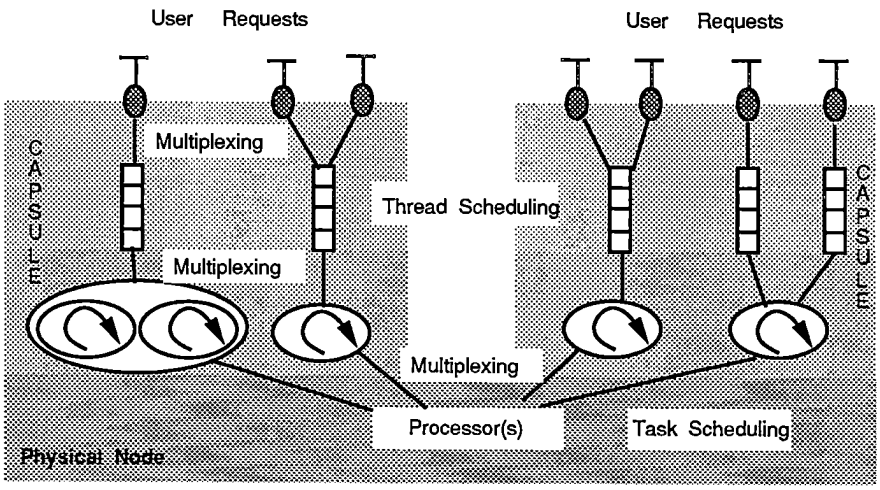

Figure 3.6: Threads, Tasks and Processor(s) Multiplexing

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 [83]. The use of separate entries to process requests on separate interfaces offers a number of (potential) advantages:

- allows the use of a specific scheduling policy (thread scheduling policy) suitable for each interface or each interface class.
- 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.

- allows the possibility of end-to-end scheduling and guarantees.
- preserves the modularity and separation of service interfaces.

Generally, in order to make good scheduling decisions, a scheduler needs to know a lot of information about the resources it is scheduling, the specific parameters of resource requirements, and it also needs to incorporate some scheduling policy. Placing this information inside an operating system kernel has a number of drawbacks: different kernels need to be built for different nodes within a distributed system and, once embedded in a kernel, any policies chosen are difficult to change, reducing the system's usefulness in a dynamic environment. The number of policies required by the kernel may change during the lifetime of the system. The scheduling processing also places a great deal of overhead in the kernel, particularly in multiprocessor systems.

The RIDE system scheduling behaviour is the integrated effect of thread scheduling and task scheduling. Task scheduling is managed by a nucleus or an operating system kernel. By attaching application-oriented policies to entries, RIDE is able to provide good scheduling decisions while needing only a small and simple kernel (e.g. with a priority-based preemptive scheduler). Complex scheduling decisions are made by the thread scheduling policies. This characteristic of minimum kernel support allows RIDE to be implemented on most practical realtime operating system platforms. Issues related to the RIDE scheduling are outlined in [73].

The nucleus scheduler defines how the real processor(s) is assigned to tasks, i.e. it manages the context switches between tasks. Preemption is used together with task scheduling parameters to order (either partially or completely) the otherwise non-deterministic behaviour of the task execution.

There are two issues in thread scheduling management. One is how a thread is enqueued in an entry (with the assumption that the first thread in the queue is executed first). Such a policy may be a RIDE defined one, like invocation priority based, invocation deadline based, or an application provided one. Another issue is how a serving task rendezvous with a thread in an entry, i.e. how the thread scheduling parameters (priority and/or deadline) are used/inherited by the task. This is defined by a task/thread rendezvous policy. Such a policy affects how the serving task competes for processor resources with other tasks. Priority inheritance is discussed further in Section 3.6.1.

It is worth noting that even though a minimum scheduling service is assumed on a kernel, alternative scheduling services, like deadline based preemptive scheduling or priority/deadline combination based preemptive scheduling are optional. If provided, they may be used to provide alternative mappings from threads to processors. The important fact is that RIDE itself is kernel scheduling policy neutral.

3.6 Priority Scheduling Models

This section discusses the mechanisms needed to provide the *static priority based* scheduling model in the RIDE framework. Static priority based scheduling is the most popular realtime scheduling method. There are well-known analytic methods [80] [70] to decide the schedulability of a set of periodic or aperiodic tasks.

Though obviously related, priority and scheduling are different issues. Associating a notion of priority with an invocation is an intuitive way of structuring realtime applications. The *priority queueing* of threads in the RIDE entry is incorporated to support such a view of realtime applications. While priority is a well defined and generally applicable notion, its role in RIDE task scheduling needs to be carefully examined. A clear definition of the *priority inheritance* (Section 3.6.1) and *priority ceiling* (Section 3.6.3) — used when the enforced synchronization during a task and a thread rendezvous — is needed to understand how priority works on tasking.

3.6.1 Priority Management and Priority Inheritance

A distinction is made between a task's *static* priority (that declared in its creation) and its *dynamic* priority (that is the static value potentially enhanced by a rendezvous or an explicit change of priority). It is the dynamic priority that is used by the nucleus (or operating system) schedule to determine the current system-wide "urgency" of a task.

The RIDE tasking model is designed to support a structured approach to priority management. Statically, the different task/entry/interface configurations allow important realtime services to be distinguished from non-realtime services. A dedicated entry may be allocated to realtime services, and high priority tasks may be allocated on the entry, so that request on the interface has better response time. Dynamically, a serving task may take into account the priority of an invocation, and use this priority as its dynamic priority. This is called **priority inheritance.**

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

3.6.2 Resource Allocation and Task Preemption

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

In RIDE, task preemption may be caused by task allocation and/or priority inheritance. By allocating tasks of different priority to different entries, an application programmer may anticipate where and when preemption is needed. Priority inheritance provides a complementary mechanism to allow a serving task to use dynamically an invocation priority — preemption happens if there is a serving task available and the invocation priority is higher than a current running task. This tasking model prompts a layered management

of priorities as illustrated by the following example.

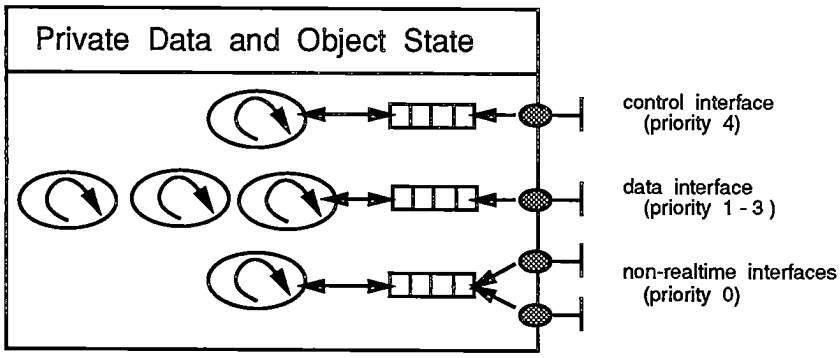

Figure 3.7: Layered Management of Priorities

One may allocate different levels of priorities to different realtime services, while priorities in one level may be used to identify the relative importance of an invocation among all the invocations on one interface. In Figure 3.7, three entries are allocated to serve non-realtime interfaces, a realtime data handling interface, and a realtime control handling interface separately. They are named as *n-entry, d-entry, and c-entry* respectively. In the n-entry, a task of priority 0 is allocated (assuming the smaller priority value means a lower priority), a FCFS thread enqueue policy is used, and therefore invocation priorities are masked, and have no effects on the scheduling activities. Priorities $1 \rightarrow 3$ are assigned to the d-entry, on which three tasks of initial priority 1 are allocated. Invocations on the d-entry may thus have a priority range $1 \rightarrow 3$. In a single processor system, the three serving tasks may provide two preemption possibilities among themselves with the priority inheritance mechanism: a 2 priority invocation preempts a 1 priority invocation, and later the 2 priority invocation is preempted by a 3 priority invocation. A task of priority 4 is assigned to the c-entry. It is guaranteed that any invocation on the d-entry will preempt any running thread on the n-entry, while any invocation on the c-entry will preempt any running thread on either the n-entry or the d-entry.

3.6.3 Dealing with Priority Inversion

Figure 3.8 shows an example of priority inversion in RIDE objects. Suppose there is a server object S with an interface I and client objects L and H. L is a low priority client — it runs a low priority task which sends low priority invocations to S. H is a high priority client — it runs a high priority task which sends high priority invocations to S. S has a task TS for serving invocations on I. Moreover, S has another middle priority task M running independently.

Priority inversion happens if the following sequence of actions appears: (1) L sends a low priority invocation to S; (2) TS begins processing L's request with the low priority; (3) M starts running, preempting TS; (4) H sends a high priority invocation to S, and has to wait until M finishes.

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

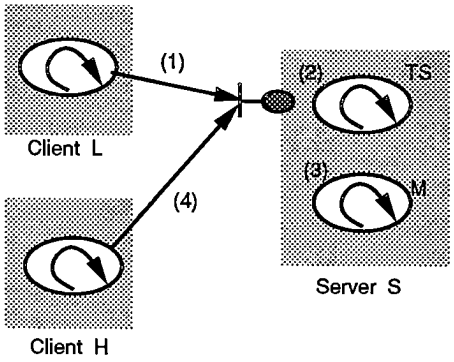

Figure 3.8: Priority Inversion in RIDE Objects

priority so that it can preempt M.

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

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

3.7 Deadline Scheduling Models

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

A simple deadline scheduling policy is to treat deadlines as priorities in thread queueing. An earlier deadline has higher priority than a late one. Let's call it *deadline based* thread scheduling. It is not assumed that the task scheduler understands deadlines. The resultant behaviour is a *non-preemptive earliest deadline first* execution of invocations.

Preemption is possible if the task scheduler provides an earliest deadline first preemptive

[2]To implement transitive priority inheritance, the infrastructure needs to maintain the dynamic task/thread relations and requires special kernel supports for transitive priority inheritance operations; see also Section 6.2.2

scheduling service and serving tasks are allowed to inherit thread deadlines. Under these conditions, deadlines can be handled exactly as priorities as defined in the last section. It should be pointed out that *deadline based* scheduling provides only a deterministic scheduling approach. It provides no guarantees for satisfying deadlines.

A system is said to be *hard* realtime if it has deadlines that cannot be missed for if they are, the system fails. A system is *soft* if the application is tolerant of missed deadlines. A hard realtime system has hard deadline types for its invocations. It is thus desirable to have some *deadline guarantee* scheduling policies.

As deadlines impose timing constraints directly to invocations, a late result produced by a server task has little or no meaning. This timeliness requirement suggests that the remote procedure call protocol — the Remote Execution protocol in the ANSA system — should take deadlines into account. Timed remote procedure calls are discussed in Chapter 4.

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

3.7.1 Guaranteeing Deadlines

A very simple scheduling model is chosen, as the aim of the design is to identify issues related to a *guaranteed* scheduling service, rather than a specific scheduling algorithm with various virtues. Deadline guarantee scheduling itself is an active research subject, and many scheduling models have been examined and many algorithms have been proposed [14]. This work focusses on scheduling mechanisms and how a guarantee algorithm may be possibly integrated with the RIDE scheduling infrastructure.

The chosen scheduling model is based on *single processor, non-preemptive, earliest deadline first* scheduling. The single processor scheduling model does not have serious restrictions on multiprocessor machines if the assignment of tasks to processors can be managed by applications as is allowed by the WANDA kernel.

Assumptions:

1. The RIDE infrastructure has full control over the scheduling activities on the scheduling processor. More specifically, only one task is allocated to the processor and the task is used to serve all incoming requests (for example, with a shared single entry configuration). In this case, The task functions as a processor, and the thread scheduler itself determines the scheduling behaviour of the processor.

2. The worst case execution time of each invocation is known.

Notations:

1. All accepted invocations are queued in the thread queue TQ and ordered by Early Deadline First rule.
2. Let T_e be the current executing thread, with last start execution time S_e, remaining execution time C_e. T_e is not in TQ.
3. Each thread T_i has a worst case execution time C_i and an adjusted deadline $D_i = d_i - Delay$, where d_i is T_i's deadline, $Delay$ is the worst case end-to-end message delay. T_1 is the first thread in the queue.
4. NOW is the current clock time.
5. The new incoming request is T_{in}, with D_{in} and C_{in} as its adjusted deadline and worst case execution time.

The algorithm:

IF (no current running thread)
  THEN $C_e = 0$;
ELSE $C_e = C_e$ - (S_e - NOW);
$S_e = NOW$;
enqueue T_{in} in TQ;
FOR $T_i = T_{in}$ and each thread after T_{in} in TQ
  IF (D_i - $NOW < C_e + \sum_{j=1}^{i} C_i$)
  THEN dequeue T_{in} and return *Unschedulable*;
return *Schedulable*;

Figure 3.9: A Deadline Guarantee Algorithm

3. The worst end-to-end message delay is known, and there is no message fragmentation.

4. Each invocation has a deadline.

A newly arrived request is called *schedulable* only if its execution does not jeopardize previously schedulable requests. The algorithm makes the schedulability test by first placing the request into the thread queue, then checking if there are sufficient processor resources to execute all the threads that may be affected by the request under the non-preemptive earliest deadline first execution rule.

It is not possible for all the assumptions of the algorithm to be true in most applications, but they are minimum requirements for providing a hard deadline guarantee service. Loosening any of the conditions will introduce non-predictable behaviour. Fortunately, hard realtime systems are normally static systems, in which interaction patterns and invocation execution times are normally fixed. The approach presented here shows how guaranteed hard deadline is possible in the RIDE architecture.

It is worth noting that the guarantee scheduling algorithm is not *optimal*. A scheduling algorithm is *optimal* if, for any set of threads, it always produces a schedule which satisfies

the constraints of the threads when ever any other algorithm can. Moore [82] showed that the earliest deadline first algorithm is optimal for non-preemptively scheduling a set of threads with the same ready (start) time in uniprocessor systems. Scheduling nonpreemptive threads with arbitrary ready times is *NP-hard* even in uniprocessor systems [71].

3.8 Other Scheduling Paradigms

Priority and deadline scheduling can be combined to provide alternative scheduling models. One combination is *priority first, and then deadline based*, in which deadlines are only used to break the tie when two thread have the same priority. This could apply in multi-media information systems, for example, priorities being used to identify information importance and deadlines being used to identify the relative order of frames in media streams (media interleaving).

Another combination is *deadline first and then priority based* [79], in which deadlines are used as first scheduling criteria, but in the case of unsatisfiable deadline, priorities are used instead for scheduling. This allows function priorities to be attached while at the same time, achieving the high throughput property of a deadline based scheduling algorithm.

3.8.1 Mixed Model Scheduling

Given the multiple thread scheduling policies and the flexible entry-interface allocations, it is possible in RIDE to have a mixed model scheduling arrangement. A specific thread scheduling policy attached with an entry may be used for a specific interface or interface groups. For example, in the example of Section 3.6.2, a deadline based policy may be used in the data entry, a priority based policy may be used in the control entry, and a FCFS policy may be used in the non-realtime-entry.

3.9 The Invocation and Entry Interface

The ANSA Testbench has an Interface Definition Language (IDL) to define ANSA interfaces, and a preprocessor (PREPC) which scans C programs for embedded statements (referred to as PREPC statements) which augment the original program to bind to interfaces and invoke remote operations. IDL has an associated compiler, stubc, which generates stub code from IDL definitions.

An example of the use of IDL and PREPC is given in Figure 3.10.

The RIDE extensions of PREPC allow realtime attributes to be attached to invocations. A realtime invocation is as following:

```
{results} <- IfRef$Operation(arguments) rtAttributes rt_attributes
```

The parameter rt_attrributes may have optional priority, deadline, deadline type, timeout, RPC protocol *id*, etc., values. Deadline type, timeout, RPC protocol *id* are explained

```
-- IDL Interface

Sample : INTERFACE
BEGIN
    Op1 : OPERATION [i : INTEGER] RETURNS [INTEGER];
END.

-- PREPC statements - server side

DECLARE ir : Sample SERVER
{ir} :: Sample$Create(arguments)
    -- create an interface instance of type Sample
{} :: Sample$Destroy(ir)

-- PREPC statements - client side

DECLARE ic : Sample CLIENT
    -- ic is an interface reference of type Sample
{result} <- ic$Op1(argument)
    -- an invocation on the interface bound by ic
```

Figure 3.10: IDL and PREPC Example

in Chapter 4. The rtAttributes item itself is optional, allowing non-realtime (ANSA) invocations to retain their original form.

An entry may be created by:

```
entry = Entry(enqueue_policy, rendezvous_policy, control);
```

The enqueue_policy argument selects which thread enqueue policy is used. The rendezvous_policy argument selects which priority inheritance protocol is used for task/thread rendezvous. The control argument defines the enqueue_policy related arguments, for example, the allowed priority range of all invocations on the entry, the priority ceiling values etc.

An interface may be created and bound to an entry:

```
{IfRef} :: IfName$Creat(arguments)
 EntryBind(entry, IfRef)
```

System tasks may be allocated to an entry:

```
TaskSpawn(entry, tasks, arguments);
```

Six thread enqueue policies are defined, which are *FCFS*, *PB* — static Priority Based, *DB* — earliest Deadline first, *PDB* — static Priority first, and then earliest Deadline

Based, *DPB* — earlier Deadline first, and then static Priority Based, and *USER* — an application supplied policy. For the USER policy, its behaviour is defined by the control argument in the entry create operation. The argument provides an upcall function and a thread enqueue policy *id* selecting one of the other five policies FCFS, PB, DB, PDB or DPB. The upcall function is called whenever an invocation has arrived on the entry. It returns a flag indicating if the invocation is *schedulable* — (to be queued and served), *ignored* — a bogus client is detected, or *unschedulable* — not enough resource to serve the invocation. In the case of schedulable, the function also returns the value of the invocation deadline and/or priority to be used by its selected thread enqueue policy. The RIDE system level scheduling itself is not going to provide any *guaranteed scheduling*, instead the architecture allows an application to provide its own schedulable test algorithm — with its own standard of guarantee and resource management policy, and integrate it with the RIDE infrastructure. For example, the guaranteed deadline scheduling of Section 3.7 is such an experimental USER policy.

Six task/thread rendezvous protocols (policies) are defined. They are rendezvous_N — the null protocol (the priority and deadline of a thread have no effect on its serving task), rendezvous_PI — the priority inheritance protocol, rendezvous_TPI — the transitive priority inheritance protocol, rendezvous_C — the priority ceiling protocol, rendezvous_DI — the deadline inheritance protocol, rendezvous_PDI — the priority and deadline inheritance protocol.

Initially, a RIDE capsule has a default system entry and all (ANSA) system services like *Capsule, Object* and *Notification* interfaces are bound to this entry. The default system entry has a FCFS enqueue_policy, and a rendezvous_N rendezvous policy. This is the native behaviour of an ANSA capsule. Any newly created interface reference is by default bound to the system entry. The EntryBind function may be used to force the interface bound to an specific entry. An interface may be reset, and bound back to the system entry by the operation *UnBind(IfRef)*. An entry may be closed by *EntryClose(entry)*.

3.10 Application Controlled Rendezvous

In addition to allocating system task(s) on an entry for serving requests, RIDE also allows tasks (when serving threads) to rendezvous with entries at run-time. The interface is as follow:

```
Accept(entry_set, timeout)
```

An entry_set may be just one entry, or the union of several entries like ($entry_1$ | $entry_2$ | ... | $entry_n$). The effect is that the task waits for at most timeout to serve one request on any entry of the entry_set.

Using an application task rendezvous to implement a bounded buffer is shown in Figure 3.11. From the example, it is worth pointing out that the application controlled rendezvous model has the following characteristics:

- Clients do not see any difference from the standard object invocation semantics.

```
-- IDL Interface
BufferIn : INTERFACE
BEGIN
    In : OPERATION [in : INTEGER] RETURNS [ ];
END.
BufferOut : INTERFACE
BEGIN
    Out : OPERATION [ ] RETURNS [INTEGER];
END.
-- server.dpl — server program
#define MaxBuf 10
static int count = 0;
DECLARE ir_in : BufferIn SERVER
DECLARE ir_out : BufferOut SERVER
static RIDE_Entry in_entry, out_entry;
int BufferIn_In(ansa_InterfaceAttr *_attr, int in)
{
    count++;
}
int BufferOut_Out(ansa_InterfaceAttr *_attr, int *out)
{
    count--;
}
static int svr_task()
{
    for(;;) {
        if (count == 0)
            Accept(in_entry, 0);
        else if (count == MaxBuf)
            Accept(out_entry, 0);
        else Accept(in_entry | out_entry, 0);
    }
}
Body()          -- server program starts here
{
    create ir_in and ir_out interface reference;
    create in_entry and out_entry;
    bind ir_in to in_entry, ir_out to out_entry;
    spawn the server task svr_task;
}
```

Figure 3.11: A Bounded Buffer

- The Accept statement ensures that only one request is executed in the accepting task (the server task). Other requests are queued, until the server task executes a subsequent Accept statement.
- The application task performs its own synchronisation. It accepts requests only when it is able to finish immediately. In contrast, if system tasks are used instead, the synchronisation has to be checked inside each operation. Synchronization is performed by suspending the task at a condition variable or semaphore. This form of synchronisation is expensive (it takes system resources like stacks etc.) and cannot be solved by simply limiting the number of concurrent requests allowed. Application task rendezvous may circumvent the problem by synchronizing before a request starts executing, and not after.
- The application task may initiate object invocations like other client tasks.
- The application task may perform its resource management when not responding to external requests. Therefore, it is possible to have interface specific tasks with pre-allocated resources and optimized synchronisation management.

3.11 End-to-End Scheduling

The client-server model of computing presents *many-to-one* interactions. While the model is natural for server design — to enable the sharing of server resources, its use in realtime systems needs revising because realtime applications often require client-based resource preservation and guarantees. This is often called *end-to-end* guarantee. Most realtime interactions have a periodic, or continuous nature. Their timing constraints have to be met for the entire duration of the client-server interaction lifetime. This periodicity also suggests that the server resource management should be solved, or solved efficiently, on an end-to-end basis.

Several levels of resource management support are needed to provide end-to-end service guarantees. At the communication level, an end-to-end communication QoS guarantee is needed. This has been partially addressed by Nicolaou [83] and Dixon [23]. At the service processing (the processor management) level, an *end-to-end scheduling* guarantee is needed, to preserve enough processor power at the server site for a client. Also, the end-to-end requirements should extend to the application level, to be able to preallocate application related resources on a per client basis.

The RIDE architecture does not provide end-to-end scheduling directly. However, applications that require the function can be designed on top of the RIDE mechanisms. ANSA allows the dynamic creation and passing of interface references. It is therefore possible to have a resource management interface which provides an operation of the following signature:

```
GetIfRef : OPERATION [ resource-requirements ]
           RETURNS   [ service-interface ]
```

Before a client starts using the server, it tells the server about its resource requirements with the GetIfRef operation. The management interface may then create a new interface

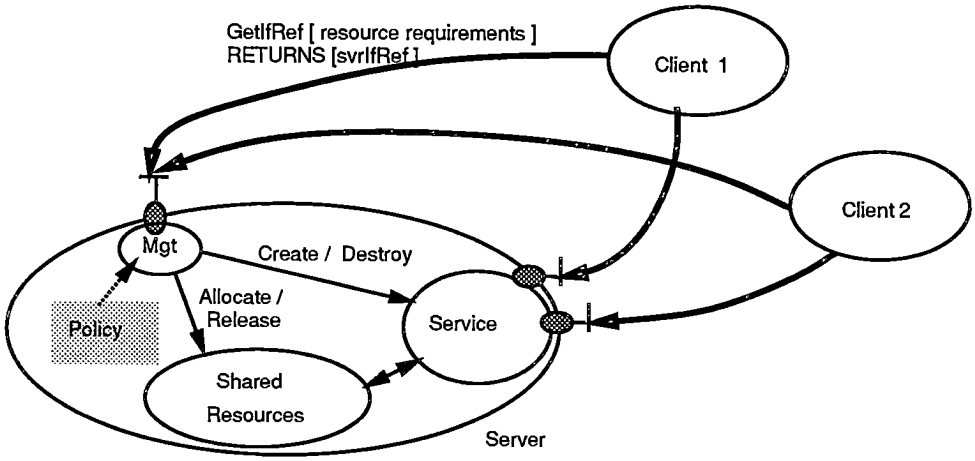

Figure 3.12: End-to-End Scheduling

instance of the service, allocate required resources (tasks, entry and application level resources) to the interface instance, and pass the interface reference back to the client. The client is then able to use the server (by the interface reference) with reassurance about the resources it reserved earlier. Figure 3.12 illustrates such an arrangement.

It seems natural for RIDE to adopt this interface-based scheme for resource reservation, because an interface is a service access point and is a programming level concept. The scheme is further enhanced by allowing the setup of a private communication channel between a client and its server interface as explained in Chapter 4.

3.12 Summary

This chapter has described the realtime programming model of the RIDE. Its scheduling flexibility has been demonstrated by its two-level scheduling multiplexing. Policy/mechanism separation is used to address the diversity of realtime programming. An integrated priority management scheme is introduced for preemption control. Guaranteed scheduling is shown to be possible as an application provided policy. The application controlled rendezvous is shown to be a powerful mechanism for resource management and synchronisation. End-to-end scheduling is supported indirectly.

Chapter 4

The RIDE Communication System

This chapter begins with a brief overview of the ANSA communication system, using the Testbench 3.0 as an example. Then three extensions aimed at making the communication system more suitable for realtime applications are presented. These extensions are:

- a parallel communication protocol stack to allow the preallocation of communication resources and the removal of layered multiplexing.
- a timed RPC protocol to allow the association of deadlines with invocations.
- a decomposable RPC protocol to allow the synthesis of the protocol to provide different levels of invocation semantics (such as exactly-once, at-most-once), so that an application programmer can customize the system to application-specific requirements of functionality and performance.

The three designs are integrated within a coherent architecture to provide a communication infrastructure for realtime applications.

4.1 The ANSA Communication System

The Testbench communication system implements three protocol layers:

- Message Passing Services (MPS): provide an interface to the transport protocols provided by the underlying operating system.
- Execution Protocols: implement the invocation of ANSA operations. Only the Remote Execution Protocol (REX) for point to point invocations is included in Testbench 3.0. Another protocol for group invocations, called Group Execution Protocol (GEX), has been included in Testbench 4.0.
- 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.

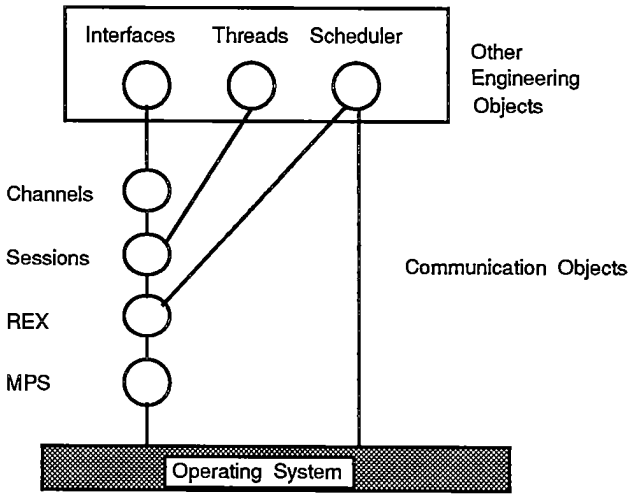

Figure 4.1: ANSA Communication System

The components concerned with communications are illustrated in Figure 4.1.

Efficient resource utilization is achieved by multiplexing the channels provided by each of these between those of the next layer.

Each MPS provides a single channel (called MPS channel in this dissertation) to each execution protocol. MPS's provide a stateless and an unreliable datagram service. It relies on the execution protocol to provide complete addressing information for each message transmitted, and on the underlying operating system for each message received. Connection-based MPS, such as MPS TCP and MPS MSNL implementations, hide their connection state from high level protocols with a connection cache.

Execution protocols provide *channels* for issuing operations to a specified remote interface and for receiving invocations on a specified interface. These are called *plug* and *socket* channels respectively. There is a one-to-one correspondence between channels and interfaces. Servers transmit invocation replies over sockets, and clients receive replies over plugs.

Sessions are created dynamically for each client/server interaction pair; therefore, a different session is required for each client invoking operations in a single interface. Channels are multiplexed between all of the sessions supported by the interface in question. Sessions are shared, when possible, across all of the operations in an interface, that is, all operations invoked by a particular client thread on the same interface will use the same session.

The Testbench has a *binding* module which maintains the bindings of interfaces with MPS addresses, channels, and sessions.

Figure 4.2 illustrates this multiplexing structure for a server/client interaction. The server supports three interfaces, X, Y and Z. All operations between the client and server share the same MPS communication channel.

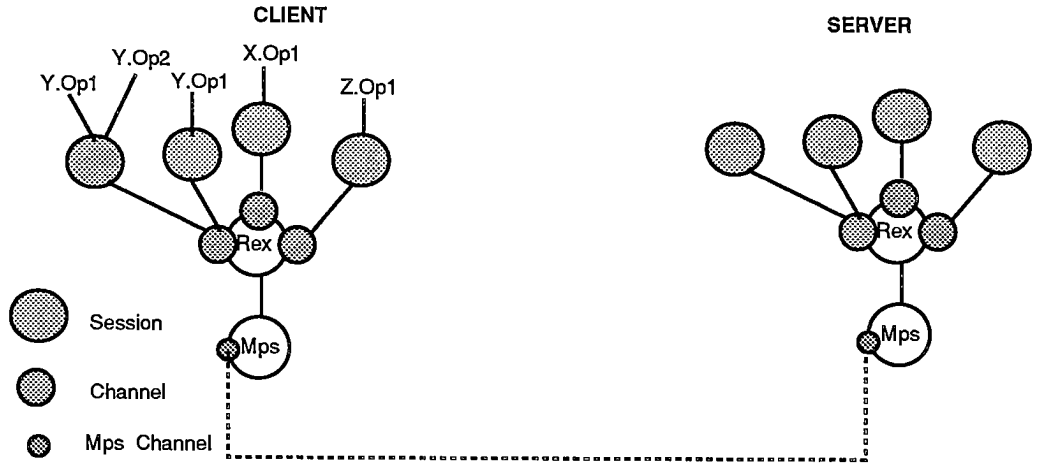

Figure 4.2: Multiplexing in the Testbench

4.1.1 The Remote Execution Protocol

REX provides a simple service for process-to-process interactions across a network. It is based on the remote procedure call protocol [12] with a number of extensions to support some of the generalizations of "procedure calls" permitted by the ANSA computational model.

These extensions are:

- asynchronous messages.
- rapid bulk delivery.
- rate based flow control.

The REX protocol provides transport and session functions for the types of remote operation interactions permitted by the computational model. Two types of interaction are supported:

- Calls. REX calls are synchronous. The execution reliability semantics of REX calls are exactly-once in the absence of total communication failure.
- Casts. REX casts are asynchronous invocations without waiting for a response. The reliability of asynchronous casts depends upon the underlying message passing service. The execution semantics of REX casts are at-most-once.

Probe, retransmission and acknowledgement are used for communication error detection and recovery in REX calls.

Server Dispatching

Incoming requests are assigned a buffer and a thread from the corresponding pool. This thread is activated by making an upcall when a task is scheduled to execute the thread.

The application language level stub code generated by the stub compiler will decode the incoming request message and perform the requested operation on an interface. On completion, any response generated is put in a response buffer which is passed back to REX.

How a task and a thread are scheduled to execute depends on the scheduling subsystem as discussed in Chapter 3.

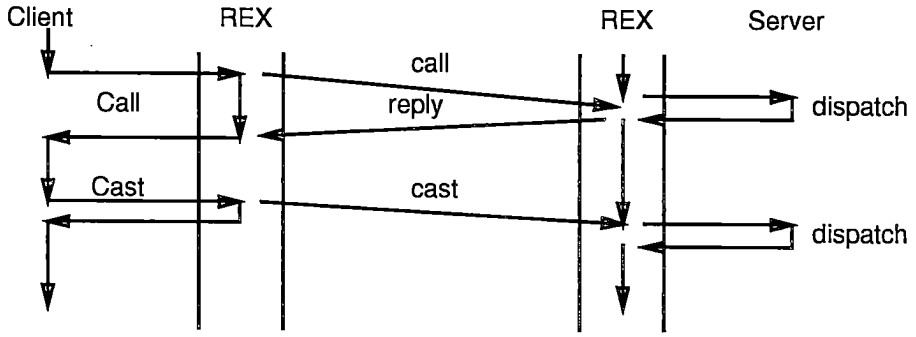

Figure 4.3: A Simple Call and Cast

A simple call and cast are illustrated in Figure 4.3.

4.2 Towards a Parallel Protocol Stack

The main advantage of the ANSA communication system design is its efficient resource utilization. The price, however, is the heavy use of multiplexing. This raises the following problems for realtime applications:

- there is no association between the (interface level) channels and MPS channels, and the two level modules have no interactions when channels are created and destroyed; the two are independent of one another. The end result is that even through it is possible to distinguish interfaces providing realtime services from those providing non-realtime services at a high level, communication to/from these interfaces may share the same MPS communication channel (such as a connection or virtual circuit), which inevitably introduces non-determinism.

- IDL and PREPC provide no way of selecting between multiple execution protocols. Even through the protocol suites adopt a generic architecture.

Detailed discussions of the adverse effect, known as *performance cross-talk*, of multiplexing several channels onto a single channel can be found in [97] and [83].

The RIDE communication system attempts to overcome the two problems as follows:

- MPS interface is redesigned as connection-based, it maintains simple states of its channels. The Execution Protocol is extended to use this connection-based interface. The result is a parallel protocol stack as explained further in this section.

- IDL stubc and PREPC compilers are upgraded to allow an application to select a dedicated execution protocol (for a specific semantics) on a per-call basis. This is discussed further in Section 4.4.

The RIDE MPS module offers a connection-based interface. The module provides a light-weight channel (LWC) abstraction. If the operating system can provide a connection-based service, a LWC is directly mapped on to an operating system IPC socket. The module also uses a default LWC to indicate a hidden MPS channel. A hidden channel can be used to represent a connection-less IPC system, or where the upper layer does not care which MPS channel is used for the communication. In such cases, the execution protocol still passes a full destination address to the MPS (along with the default LWC), so that it can use the address for communication (in the connection-less case), or choose the right connection from the connection cache (in the connection-based case).

Some important operations of the ANSA MPS module and RIDE MPS module are listed in Table 4.1. All the operations are synchronous. The socket and plug are ANSA channels, and are not to be confused with the operating system IPC socket.

ANSA MPS	*RIDE MPS*
mps_startup(address)	mps_startup(address)
mps_send(buffer, length, address)	mps_send(LWC, buffer, length, address)
mps_receive(buffer)	mps_receive(LWC, buffer)
	address = mps_offer(socket, execution_protocol)
	LWC = mps_connect(address, plug)
	mps_release(LWC)

Table 4.1: ANSA MPS vs RIDE MPS

REX is also extended to match with the RIDE MPS interface by adding some operations for managing the connections; this is shown in Table 4.2.

ANSA REX	*RIDE REX*
ex_startup()	ex_startup()
ex_send()	ex_send()
ex_receive()	ex_receive()
ex_reply()	ex_reply()
ex_cleanup()	ex_cleanup()
	address = ex_offer()
	ex_connect(address)
	ex_release()
	ex_control(LWC, status)

Table 4.2: ANSA REX vs RIDE REX

When a new interface is created, the binding procedure allocates a socket *id*, and calls the ex_offer operation of the execution protocol to get the appropriate MPS address that

a client can communicate. The ex_offer gets the address by a call of the MPS mps_offer operation. The mps_offer, if it is within a connection-based MPS module, may acquire an IPC address and spawn a thread to listen on the address. The socket *id* and the MPS address comprise an ANSA interface reference, through which a client may communicate with the server.

PREPC is extended to understand the connection-based nature of communications. A client may set up a private MPS communication channel to a server interface by the following PREPC statement:

```
IfRef$Connect(arguments)
```

The arguments are expected to include communication QoS in the future, when the underlying operating system can provide the required service.

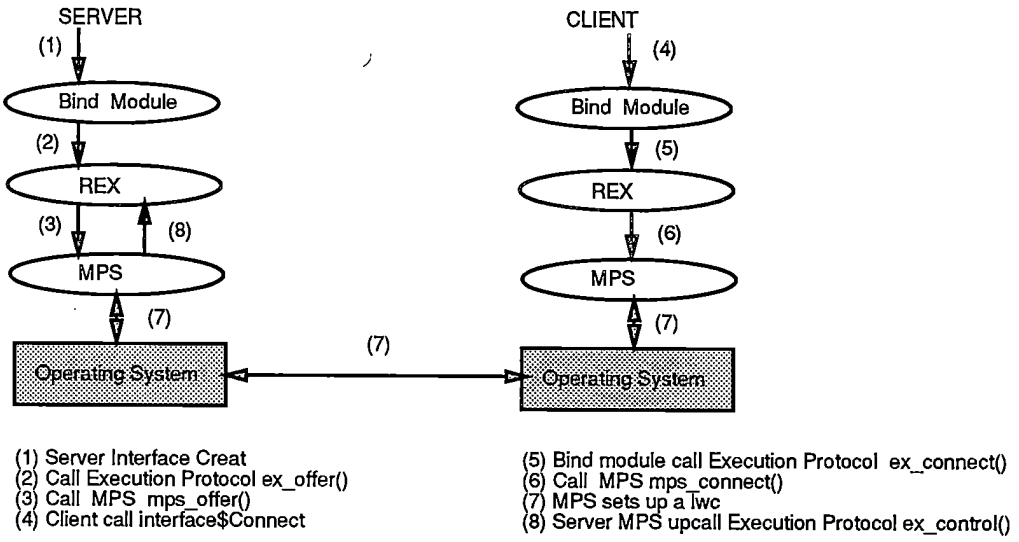

Figure 4.4: Private Lightweight Channel Connection

This PREPC operation calls the REX ex_connect operation, which then calls the mps_connect to make the real connection. On the server side, when the MPS module perceives a connection request, it upcalls the REX ex_control operation so that REX can check the validity of the connection and take some adminstration actions. The ex_control operation may also returns an action *id* to ask the MPS module to take an appropriate action on the connected LWC, two actions are defined in the current implementation: (1) disconnect the LWC; (2) set the priority of the LWC listener task/thread to a specific level. The private LWC set up procedure is illustrated by Figure 4.4.

Private MPS communication channels can be released by another PREPC operation:

```
IfRef$Disconnect()
```

The disconnect operation calls the REX ex_release, which then calls the MPS mps_release to do the real MPS channel disconnection operation. If this operation is initiated from a

client site, it releases the single LWC between the client and its server associated by the interface reference IfRef. If this operation is initiated from a server site, all LWC's to this server interface instance will be disconnected. This reflects the asymmetric aspect of the ANSA channels.

If either a connection is released or an IPC socket error (to reflect the fact that the LWC is a light-weight mechanism) occurs, the MPS module upcalls the ex_control operation, allowing it to make appropriate reactions, such as to initiate a new connection or invalidate the current one and so on.

It is not necessary for every client to call the IfRef$Connect operation to set up its own private MPS channel; a default channel may be used, in which case the MPS module manages cached connections hidden from upper layer protocols.

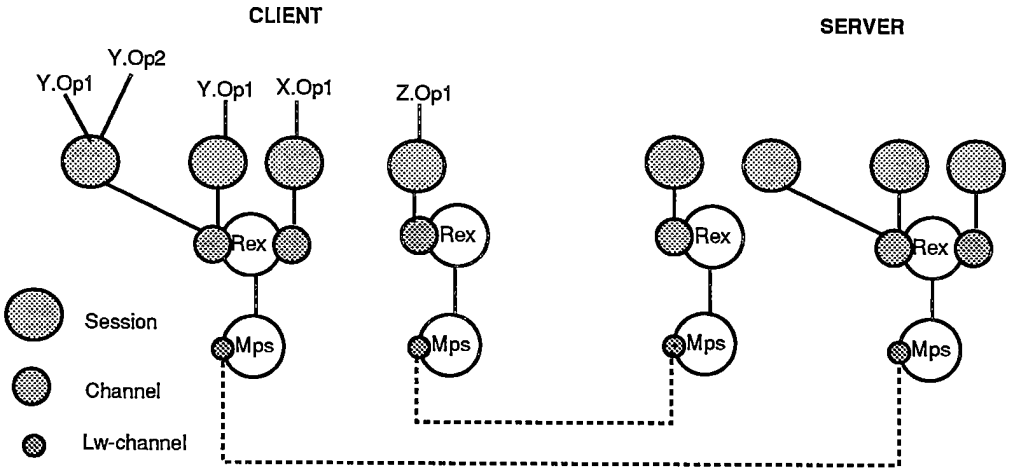

Figure 4.5: Parallel Protocol Stack

The result is a parallel communication protocol stack, as illustrated by Figure 4.5.

4.3 Towards a Timed RPC Protocol

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

Realtime invocations in RIDE 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 global sense of time — the deadline. It is thus necessary to assume a global sense of time is provided by the infrastructure.
- the interpretation of deadlines.
- a communication protocol to implement reasonable meanings of deadlines.

To the author's knowledge, there is no clear definition of TRPC yet when examined in the distributed setting. The interpretations applied significantly affect the implementation. The problem will be approached in the dissertation by first making a strictly unsatisfiable definition, and then relaxing the problem to lead to realistic solutions.

The TRPC call can be defined as follows. At time C_s, the client sends a request with a deadline D, which is the latest time the client is willing to wait for successful invocation. At some time S_s the server gets the request; the server checks if the deadline can be met, and if it is unsatisfiable a fail acknowledgement is sent back at time S_n. Otherwise, the request is accepted and the request is processed at time S_p, and a reply is generated at time S_f. This is illustrated in Figure 4.6.

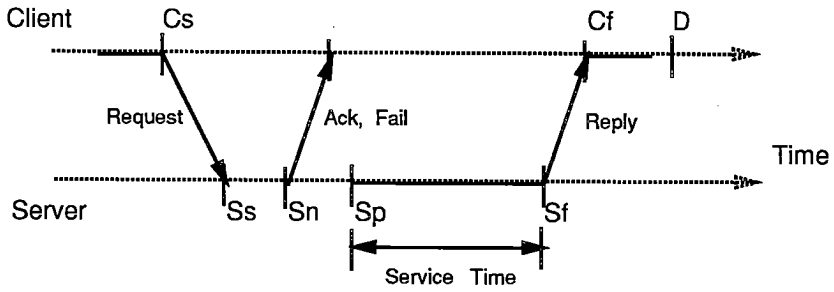

Figure 4.6: Timed RPC Communication Sequence

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

4.3.1 Discussion of Problem

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

- Goal 1: to establish a bound on the time at which the delay in awaiting a TRPC call expires.
- Goal 2: to establish a bound on the time at which a TRPC call is either scheduled to execute and finish or is unschedulable and cancelled.

The design of a TRPC is complicated by the fact that the end-to-end delay of messages can be arbitrary or even infinite (messages can get lost). It can be shown that the two goals are not mutually compatible. In its simplest form, in which each request takes zero service time, the TRPC problem is equivalent to the *timed synchronous communication* problem [68]. 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 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 like the one presented in Chapter 3 makes many of the impossible assumptions such as that the invocation service time is known, operations are independent etc.

The intention of this work is to develop a protocol for TRPC that works in reasonable environments. Therefore, an upper bound on message delivery and a guarantee scheduler cannot be assumed. Instead, various *relaxations* of the problem are investigated, this yields to a parameterised generic protocol, allowing different combinations of the parameters to represent different relaxed goals.

4.3.2 The Protocol

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

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

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

The second relaxation is using a deadline to specify 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.

The second relaxation can be further extended by relaxing the meaning of a deadline. Instead of bounding the finishing time of a request, a deadline can be used to bound the start time of a request in the server — to bound the start time by which the request is rendezvoused with a server task. If the rendezvous is issued before the deadline, then the request is successful and a success acknowledgement is sent back to the client, otherwise the request is cancelled and a fail acknowledgement is returned. At the client side, there are two possible actions to be taken when it receives a success acknowledgement. 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 4.7.

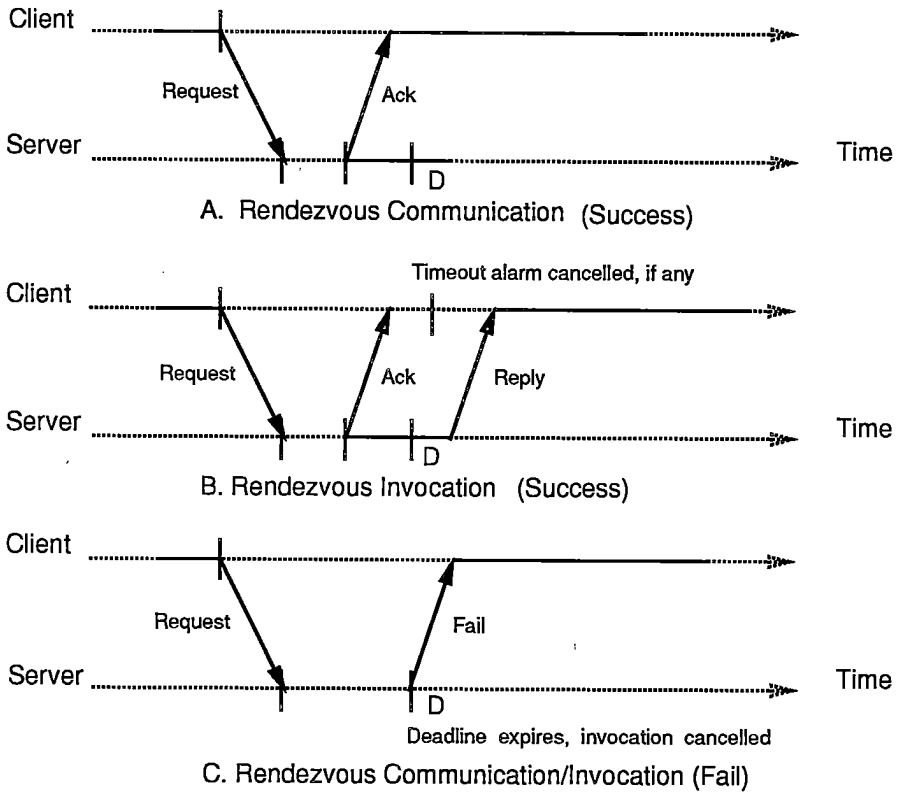

Figure 4.7: Rendezvous Communication/Invocation Interaction

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

Obviously, it is not necessary to choose the timeout and deadline the same value. A timeout may be smaller than a deadline, to specify that an acknowledge should be returned earlier; it may be greater than a deadline, to allow the request to have a better chance of success.

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

4.3.3 Server Deadline Expiry

There may be two types of deadline expiry at a server side. One type is defined by the TRPC protocol, as illustrated by the rendezvous communications and rendezvous invocations. The required semantics are enforced by the communication protocol (with support from the RIDE rendezvous mechanism).

Another type of deadline expiry may be caused by the tasking components. An active thread serving an invocation may be notified of a deadline expiry signal — if the operating system scheduler understands deadlines. If the service routine is designed to accept and

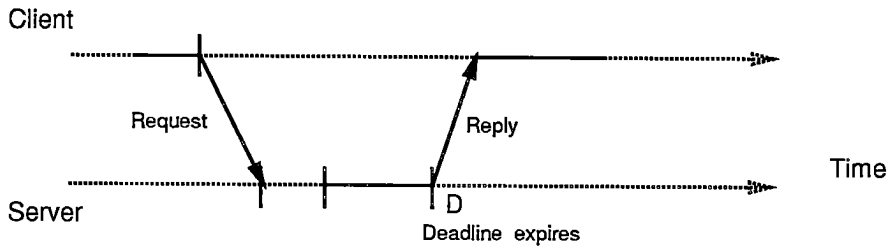

Figure 4.8: Server Thread Deadline Expire

handle the signal, a deadline exception may be raised. This deadline exception, however, is different from the one processed by the TRPC protocol. The active thread itself detects the deadline expiry, and may therefore cancel its execution and returns a special value *deadline-exception* to the client. This kind of interaction does not require special TRPC protocol support, as the deadline-exception is just a special value of reply. This is illustrated in Figure 4.8.

4.4 Towards a Decomposable RPC Protocol

An RPC protocol is normally required to provide *exactly-once* call semantics. The exactly-once protocol is used to ensure that calls are executed once and only once in the absence of crashes or prolonged communication failure, in order to preserve the local procedure call semantics for the client. Probes, acknowledgements and retransmissions are used for error-detection and error-recovery in such protocols. Error detection and error recovery both introduce significant performance overheads.

For realtime applications probes and retransmissions are not normally suitable techniques for error control, and exactly-once semantics are sometimes not a desired feature because retransmitted data or control information could be a *late* message, and have little meaning in realtime sense. Alternative *light-weight* protocols with *at-most-once* semantics are desirable instead.

RIDE is assumed to operate in a system which may consist of a mixture of realtime and non-realtime applications, therefore both the exactly-once and the at-most-once semantics are desirable. It is possible to implement the two protocols separately [45], but because the two protocols share many similarities, alternative integrated design is more interesting for the purposes of better structure, flexibility and efficient coding. This raises the desire to design a decomposable RPC protocol.

The ANSA REX service provides exactly-once semantics of RPC calls. REX can be decomposed into three layers as illustrated in Figure 4.9. It should be pointed out that, the three layered decomposition does not mean three layers of protocols like the approach used by the X-kernel [34]. The three layers are layered functions sharing the same protocol data structure — sessions, and to provide just one protocol service.

The *message* layer uses the underlying MPS service to provide a simple unreliable, unfragmented message passing service. This layer sends/receives messages not larger than a single MPS packet size. The *fragmentation* layer provides unreliable, but persistent

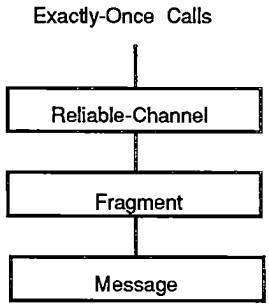

Figure 4.9: REX Functions Layers

(recovery from dropped fragment) transmission of large messages. The *reliable-channel* layer provides reliable transmission of large messages (recovery from lost and duplicated messages).

The three layers have been reassembled to provide a multiple service interface. In RIDE, the experimental transportation protocol now looks like Figure 4.10. The RIDE communication system is illustrated in Figure 4.11.

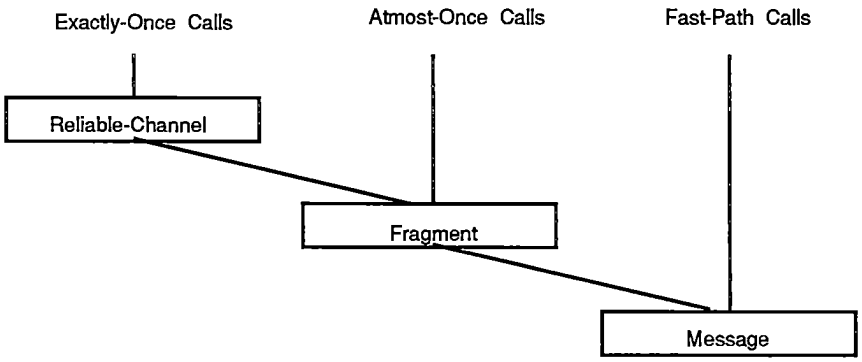

Figure 4.10: A Decomposable RPC Protocol

In addition to the exactly-once service, two other services, the at-most-once service and fast-path services are provided.

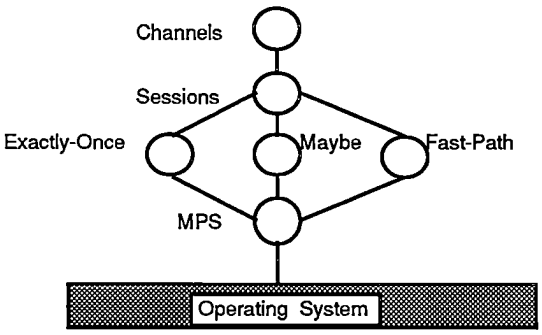

Figure 4.11: RIDE Communication System

The multi-transportation service protocol is still one execution protocol in the ANSA sense. But it provides additional call semantics. The call semantics can be chosen by the

rtAttributes parameter of an invocation (Section 3.9). The exactly-once service is used by default. The *MayBe* protocol *id* selects the at-most-once service. The *FastPath* protocol *id* selects the fast-path service.

The fast-path service is designed to execute operations within the *critical data path* of the RPC system. It is assumed that the request is independent of other invocations (no resource sharing with others and no nested invocations), and both the request and result fit in one single MPS packet. Under these conditions, the server can execute the request within a communication task (thread), allowing significant performance improvement by saving the cost of thread dispatches and task context switches.

The fast-path execution is not safe generally, it is not allowed on normal entries when initialised. A control function is provided to turn on the permission when required.

4.5 Summary

Realtime applications present more complicated functional requirements to the underlying communication systems. This chapter details some mechanisms for providing such functions within an RPC communication infrastructure. The facilities provided include:

- a parallel protocol stack for the preallocation of communication resources and the removal of layered multiplexing.
- a timed RPC protocol for the association of deadlines with invocations.
- a decomposable RPC protocol for the tradeoffs between functionality and performance.

Chapter 5

Temporal Synchronization

Realtime systems usually consist of many realtime computations with different but related time constraints. These computations must be temporally coordinated with one another if all time constraints are to be met. A realtime programming system must therefore provide a *temporal synchronization* facility. Some important requirements of this facility are as follows.

- the capacity to express different types of timing requirements.
- provision of a useful abstraction that makes it easier to ensure the temporal correctness of a system.
- the separation of concerns so that the cooperative computations do not have to share assumptions about one another.
- mechanisms for run-time enforcement of timing constraints.

Timing constraints can be classified into two — the *intra* and *inter* system timing constraints — according to whether they spawn across different address spaces or not. The intra system timing constraints impose temporal constraints in computational blocks within one address space. This type of constraint has received considerable research attention. For example, Real-Time C++ [59], FLEX [65] and Maruti [85] are designed for the purpose. The inter system timing constraints impose temporal constraints on computational blocks on different address spaces, and little work, if any, has been done in this domain. Because of the distributed nature of the RIDE programming system, this work concentrates on inter system timing constraints.

The approach taken is to keep in-line with the ANSA Computational Model. The temporal synchronization facility is a kind of special service accessible through well-defined ANSA interfaces. Normal invocations on the interfaces can be used to set and enforce temporal synchronization conditions. The model used is one of the *timed automata*, and is discussed in Section 5.1. Section 5.2 shows various modelling techniques for applying timed automata to the description of temporal behaviours. Section 5.3 shows how the timed automata can be used as a synchronization mechanism.

5.1 Timed Automata

Various extended automata, called timed automata (TA), haved been used to extend the automata approach to realtime system specifications [3]. The transitions of such automata also depend upon the time elapsed since the previous transitions. For this purpose, they are enriched with a set of variables, called timers. In this work, a variant of TA without accepting states is considered where timing constraints are associated with both states and edges.

Timing constraints of TA are defined first. Let X denotes a finite set of timers ranging over a time domain D. V: X $\rightarrow$ D denotes the set of timer valuations. The set Φ of timing constraints is generated by the following grammar:

$$\phi ::= x \leq m \mid y \geq m \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

where $x, y \in X$ and m is a natural number.

A timed automaton is a tuple $G = (S, X, s_0, Sig, R, con, E)$ where:

- S is a finite set of states.
- X is a finite set of timers.
- $s_0 \in S$ is the initial state.
- Sig is a finite set of input signals.
- R is a finite set of output responses.
- $con : S \rightarrow \Phi$ associates a timing constraint with each state.
- $E \subseteq S \times (\Phi \times Sig \times R \times 2^X) \times S$ is the set of edges. An edge $(s, \phi, a, r, T, s') \in E$ is written as $s \frac{\phi, a, T}{r} s'$, in which T is a subset of X and $a \in Sig$.

The function *con* associates to each state $s \in S$ a *safety* timing constraint, in the sense that the automaton can stay in state s only while the current timer valuation satisfies $con(s)$. This constraint forces the automaton to move before it becomes false, that is, it prevents the automaton from getting stuck at the state s. A null safety timing constraint means always true.

A timed automaton starts at state s_0 with all its timers initialized to zero. The states of the automaton represent the control state. Moving through an edge $s \frac{\phi, a, T}{r} s'$ is called a *transition*, and it takes *no time*. A transition sets to zero the values of all timers in T. The values of all the timers increase (but not necessarily uniformly) with time and, at any instant, the value of a timer is equal to the time elapsed since the last time it was reset. The time domain is further discussed in Section 5.4. The automaton can perform a transition only if the timing constraint ϕ associated with the edge is satisfied by the current values of the timers. The automaton may stay at a given state s but it cannot let time pass beyond the bound imposed by the associated safety time constraint $con(s)$.

At any instant, the whole state of a TA can be fully described by the current state of the automaton and the value of all its timers. A transition corresponds either to the move of the automaton through an edge at a particular instant of time by an input signal, or simply the fact that time progresses. A transition may respond to its environment with an output r.

5.2 Description of Temporal Constraints

Timing constraints impose temporal restrictions on a system or its user. For a realtime system, there are two categories of timing constraints [96]:

- *performance* constraints which set limits on the response time of a system.
- *behaviour* constraints which are concerned with the occurrences of events involving explicit time references.

It is the philosophy of this dissertation to address the two categories of timing constraints separately. Performance constraints are related to resource management, scheduling and allocation, which are tackled by the RIDE realtime programming model. Behaviour constraints are related to the association of events and responses with explicit time references, and are addressed by the timed automata synchronization mechanism introduced in this chapter. The idea is to assume that the performance constraints of a system have been satisfied, and then address the enforcement of behaviour constraints. Ideally, these two types of timing constraints should be addressed in an integrated way. But this is only possible under very restrictive assumptions, such as a system is static — all system loads and system resource usage are known at design time. The ESTEREL language [9] (also see Section 8.5) is perhaps the only approach trying to address the two types of constraints in an integrated manner. It needs to adopt the synchrony hypothesis — all computations "take a null time". In the approach adopted by this dissertation only transitions are assumed to "take a null time".

5.2.1 Instantaneous Timing Constraints

Dasrathy [20] was one of the first researchers to introduce timing constraint expressions in a systematic way. He constructed a language called the Real-Time Requirements Language based on the state machine model. He classified timing constraints into three types of temporal restrictions:

- *maximum*: no more than t amount of time may elapse between the occurrence of one event and the occurrence of another.
- *minimum*: no less than t amount of time may elapse between two events.
- *duration*: an event must occur for t amount of time (see Section 5.2.2).

The "event" means either an input signal (S) or an output response (R) of a system. These three types are not mutually exclusive, and may be combined.

A telephone switch example used by Dasrathy is adopted here to illustrate the various modelling/specification techniques.

There are four kinds of meaningful maximum timing constraints in realtime systems:

A1. S-S Combination: a maximum time is allowed between the occurrence of two signals. Example: after the first digit has been dialled, the second digit shall be dialled no more than 20 seconds later.

A2. S-R Combination: a maximum time is allowed between the occurrence of a signal and the system's response. Example: the caller shall receive a dial tone no later than 1 second after lifting the handset.

A3. R-S Combination: a maximum time is allowed between a system's response and the next signal from the environment. Example: after receiving a dial tone, the caller shall dial the first digit within 30 seconds.

A4. R-R Combination: a maximum time is allowed between two system responses. Example: after a connection has been made, the caller will receive a ring-back tone no more than 0.5 seconds after the callee has received a ring tone.

A1 and A3 are behaviour constraints imposed on the system's users. A2 and A4 are requirements on the system's behaviour.

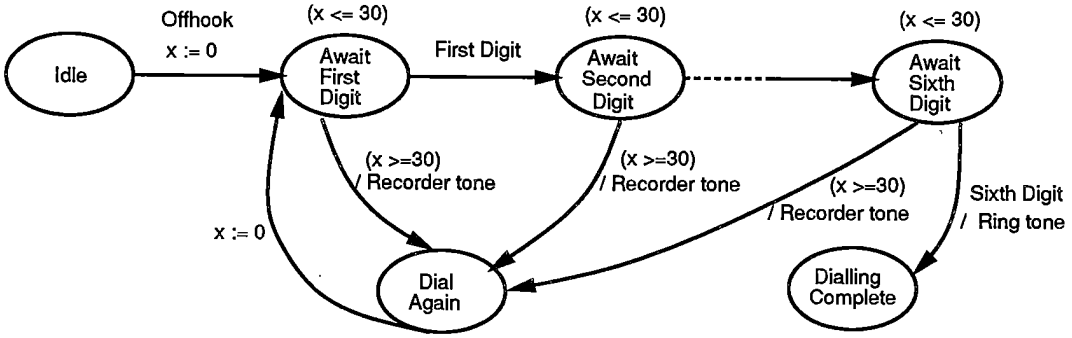

Figure 5.1: Description of a Maximum Timing Constraint

Figure 5.1 shows how a TA is used to specify a maximum timing constraint with one timer. It assumes that in a telephone system a caller should dial six digits in 30 seconds or less (total dialling time is 30 seconds or less), after lifting the handset. It also assumes that a user who meets this behaviour requirement correctly will be rewarded with a "ring-back" tone; failure to meet this constraint will trigger a "recorder" tone.

There are also four possible types of minimum timing constraints in realtime systems:

B1. S-S Combination: a minimum time is required between the signals. Example: a minimum of 0.5 seconds must elapse between the dialling of one digit and the dialling of the next.

B2. S-R Combination: a minimum time is required between the arrival of a signal and the system's response to that signal. Example: after the caller has dialled 0, the system shall wait 15 seconds before responding. (Presumably, this will allow the caller to complete dialling the operator-assisted call by himself.)

B3. R-S Combination: a minimum time must pass between a system response and the arrival of the next signal.

B4. R-R Combination: a minimum time must pass between the occurrence of one system response and the occurrence of the next.

B1 and B3 are behaviour constraints on the system's users. B2 and B4 are behaviour constraints of the system itself.

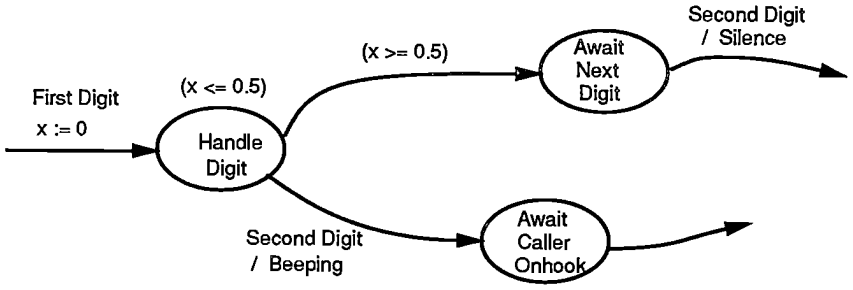

Figure 5.2: Description of a Minimum Timing Constraint

Figure 5.2 shows an example using one timer to specify a minimum timing constraint. In the figure, `First Digit` and `Second Digit` are the two signals between which the minimum time requirement is to be imposed. The requirement also states that the response `Beeping` should occur if `Second Digit` is dialled too soon.

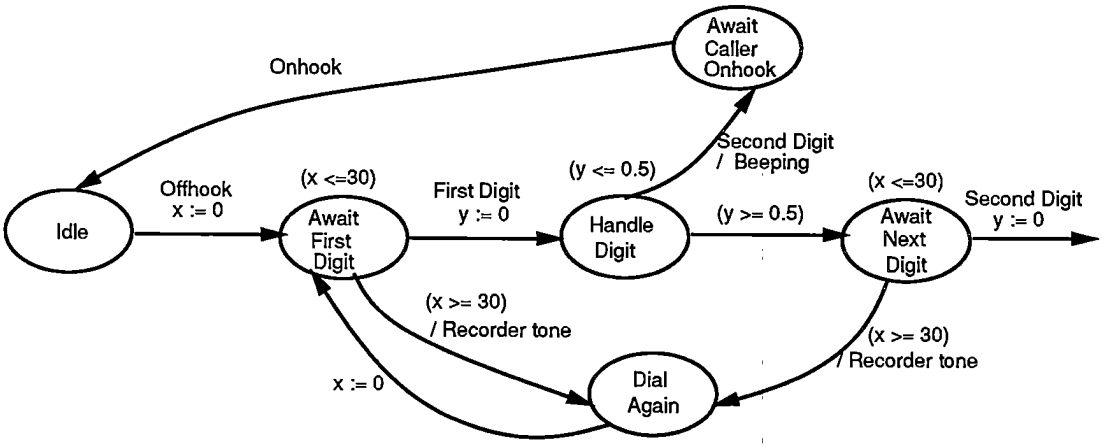

Figure 5.3: Combining Maximum and Minimum Timing Constraints

Figure 5.3 shows an example using two timers to specify the combination of maximum and minimum timing constraints. It combines the effect of Figure 5.1 and Figure 5.2.

5.2.2 Interval Timing Constraints

Duration time constraints associate *time intervals* with events. Interval time constraints are also useful in expressing temporal conditions. Allen [2] presents a temporal logic based on intervals and defines thirteen relations operating on pairs of intervals. Figure 5.4 shows seven of the relations. The others are just symmetric relations of the seven, for example `B start A` is a symmetric relation of `A start B`.

Dasrathy extends his finite-state machines to allow the association of a time interval value with a transition (event). This extension unfortunately violates the spirit of the finite-state machine model (the transition is assumed to be instantaneous). Dasrathy's Real-

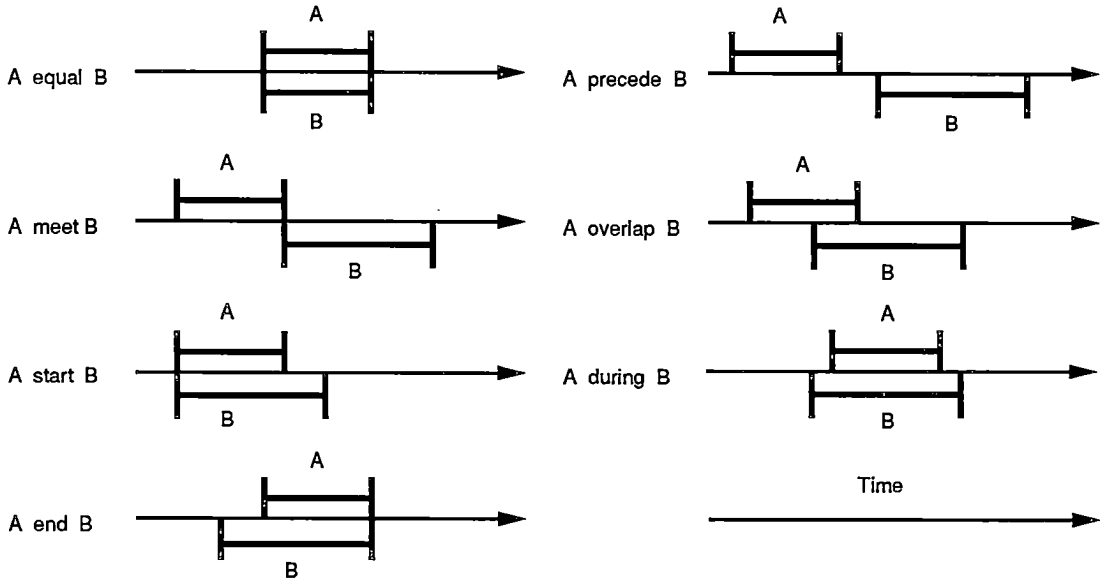

Figure 5.4: Interval Binary Relations

Time Requirement Language is a specification language and is not intended for automatic execution.

A equal B	A: (finish $\geq$ B'.finish) {
	A':(start $\geq$ B.start) { ... }
	}
	
	B: (finish $\geq$ A'.finish) {
	B':(start $\geq$ A.start) { ... }
	}

A meet B	A: (finish $\geq$ B.start) {
	A':(1) { ... }
	}
	
	B: (start $\geq$ A'.finish) { ... }

Figure 5.5: FLEX Constraint Blocks

FLEX [65] is another attempt at describing timing constraints in a manner which is consistent with the state machine model (the stimuli and response model). FLEX uses *constraint blocks* as the basis for the association of time and resource requirements. A constraint block identifies a constraint that must apply while a section of code is in execution. A FLEX programmer describes time and resource requirements by specifying constraint blocks and propagating information among them.

An interval of time representing the lifetime of the block is associated with each constraint block. In this sense, FLEX is an ideal language for defining interval timing constraints. Each FLEX block may be associated with a label. Another block might refer to the *start*

and *finish* times of a given block by using the block's label. With this timing mechanism, FLEX can enforce many temporal relations as shown in the Figure 5.5.

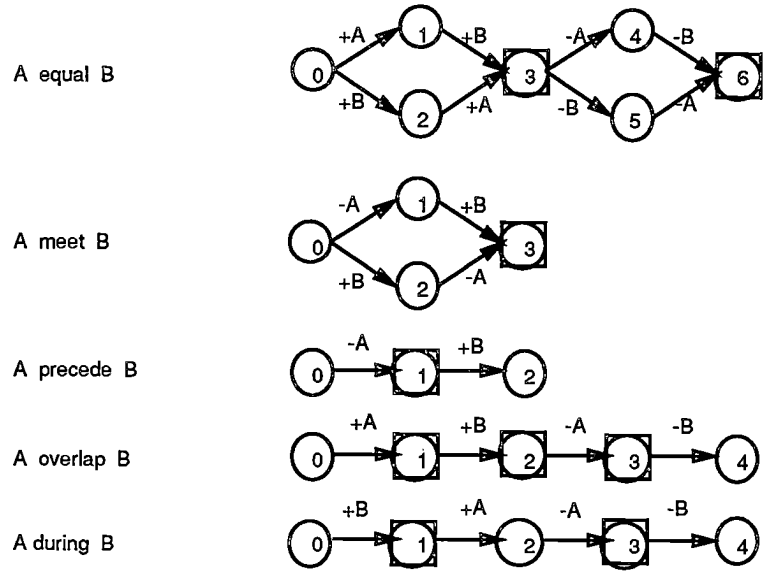

Figure 5.6: Transition Graphs for Interval Binary Relations

Timing constraints are implemented and enforced by using *wait* or *synchronisation point* semantics in FLEX. The timing constraints in FLEX come in two varieties: the *earliest* and *latest time* constraints. The *wait* semantics introduces delays to allow maximum chance for timing constraints to be true. In this sense, it is the *delays* that enforce the relative timing relations between constraint blocks.

The FLEX spirit of describing interval relations can be used by the TA for the same purpose. Let the start events of an interval be identified by a '+' before the name, the finish events by a '-', and the synchronization points by boxed states. Figure 5.6 gives some examples of describing interval relations with TA. The usage of these graphs is further explained in Section 5.3.1.

5.3 Timed Automata Synchronization Mechanism

Faulk and Parnes [36] were perhaps two of the first researchers that introduced finite state machine as a synchronization mechanism to practical realtime applications. They adopted finite state machines to model and implement high-level application-specific synchronization requirements. The design of the TA synchronization mechanism shares the same motivation as theirs, but with emphasis on temporal and distribution aspects.

The TA synchronization facility is designed to fit the ANSA Computational Model. A TA is intended to provide a temporal synchronization service, and to determine whether an activity requesting an invocation may proceed immediately, or some other action must be taken. Delaying a synchronization operation until a synchronization condition is satisfied is the main function of a TA.

There are three major functional requirements for the TA synchronization mechanisms:

1. provide a facility for defining and signalling events;

2. define the allowed transitions;

3. provide a facility for defining and waiting on synchronization conditions.

Either the ANSA announcements or interrogations can be used for (1). (2) is defined by a TA specification language as explained later in this section. The ANSA interrogations have the property required by (3).

5.3.1 Syntax and Semantics

The synchronization services are provided by the *timed automata interface*. TA interfaces are specified by the TA specification language. A TA is an instance of a TA specification which encapsulates its own states and timers.

A TA specification consists of a set of *states*, a set of *timers*, a set of *transitions* on these states and a set of *timing guards*. In addition, named subsets of the set of states may also be defined.

Figure 5.7 shows the syntax of the TA specification language. The BNF notation is extended by the use of "..." to denote a list of zero or more of the surrounding items. In the definition, the & denotes the logical operation *and*, and '|' denotes *or*.

An Example System

The virtual radar device module used in [36] is extended to illustrate the usage of the TA synchronization facility. Three timers are added to demonstrate how temporal constraints can be described.

The radar device has three designated modes (states) of operation: ranging, tracking, and standby. Under certain operating conditions, usually temporary, data from the radar becomes unreliable. This condition can be detected by the software and may occur in any of the three modes. If the radar stays in an unreliable mode for a sufficiently long time, the radar is assumed to have failed. In addition, the radar hardware may fail completely. Once a failure occurs, the radar never resumes operation. The states and state transitions of the radar are depicted in Figure 5.8 and the TA specification is shown in Figure 5.9.

An interface of the type Radar provides the required synchronization service. Other activities (clients) in the system wait for state changes, signal changes in it and otherwise use the radar state. In the original example, there are three processes which keep track of the radar state: (1) the mode monitor process polls the hardware register for changes to the radar mode and signals changes in its state; (2) the reliability monitor does the calculation necessary to determine if radar inputs are reliable and signals any changes in the reliability; (3) the fault monitor process detects and signals hardware failure of the radar.

A client may issue the following operations to a TA service:

TA_Interface ::= InterfaceName : TA_INTERFACE = Spec
Spec ::= BEGIN Body END.
Body ::= TimerSpec StateSpec SetSpec GuardSpec TransitionSpec Initialization
TimerSpec ::= | TIMER : TimerList;
StateSpec ::= STATE : StateList;
SetSpec ::= SetDecl ... SetDecl
SetDecl ::= SET SetName : StateList;
GuardSpec ::= GuardDecl ... GuardDecl
GuardDecl ::= GUARD StateName TimerExpr;
TransitionSpec ::= TransitionDecl | TransitionDecl ... TransitionDecl
TransitionDecl ::= TRANSITION TranName TranList;
TranList ::= TranItem | TranItem ... TranItem
TranItem ::= (StateName, TimerExpr, Action, StateName)
Action ::= | < TimerList >
TimerExpr ::= | TimerOp | (TimerExpr) | TimerExpr & TimerExpr |
TimerExpr '|' TimerExpr
TimerOp ::= TimerItem TimerOperator TimerItem
TimerItem ::= TimerName | Number
TimerOperator ::= $\geq$ | $\leq$
Initialization ::= STARTAT StateName Action;
TimerList ::= NameList
StateList ::= NameList
NameList ::= Identifier | Identifier, ... Identifier
SetName ::= Identifier
StateName ::= Identifier
TimerName ::= Identifier
TranName ::= | Identifier

Figure 5.7: Timed Automata Specification Language Syntax

Signal an Event. State transition operations on a TA are given by the TRANSITION attributes defined on the interface.

```
{} <- IfRef$Signal(Transition)
```

For instance, the `Transition` may be the *reliable* transition. Let the tuple $(s1, t, act, s2)$ be an element of the transition T. Then invoking the transition operation corresponding to T will change the state of the TA to $s2$, if TA's current state is $s1$ and the timing constraint t is satisfied. While making the transition, the timers in the *act* are reset to zero.

Wait on Synchronization Conditions. There are two kinds of synchronization conditions that can be applied to a TA. One of which corresponds to each TRANSITION attribute and the another corresponds to each SET attribute. There are also two

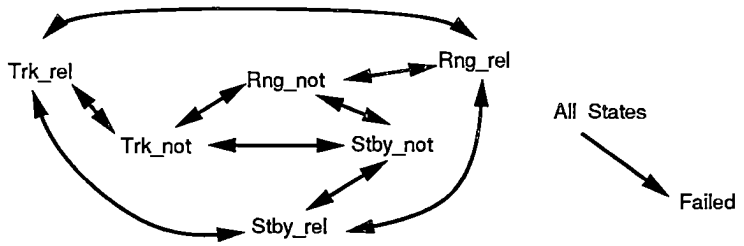

Figure 5.8: Radar States

combined synchronization operations that can be applied. The operations are shown below.

```
Wait on transition:
  {state} <- IfRef$Await(TRANSITION, Transition, timeout)
Wait on set:
  {state} <- IfRef$Await(SET, Set, timeout)
Combined operations:
  {state} <- IfRef$SigAwait(Transition, Relation, Name, timeout)
  {state} <- IfRef$AwaitSig(Transition, Relation, Name, timeout)
```

The `Await` operation can choose either TRANSITION or SET as the required synchronizaton condition. If TRANSITION is chosen, the operation will wait until a transition of the name `Transition` occurs in the TA. The `timeout` argument takes its intuitive meaning — a positive value will force the operation to return when the timeout expires; a zero value enables the operation to wait until the required synchronization condition is satisfied. The result parameter `state` contains the state of the TA when the operation returns. If the SET is used with the `Await` operation, it waits until the state of the TA is a member of the Set. A convention is adopted to allow a single state to denote the state set of itself. The `SigAwait` and `AwaitSig` operations combine the effect of a `Signal` and an `Await` operation. The `SigAwait` first signals a transition `Transition`, and then awaits the required synchronization condition. The `Relation` is either TRANSITION or SET, and the `Name` is a `Transition` or `Set` correspondingly. The `AwaitSig` first awaits a synchronization condition and then signals a transition.

All operations on a TA are atomic — there is no concurrent execution of operations on a TA.

Inquiry. The following operation can be used to get the current state of a TA.

```
{state} <- IfRef$State()
```

Some example operations to the Radar TA are shown below:

```
Signal a reliable transition:
  {} <- IfRef$Signal(reliable)
Wait on the reliable transition:
  {state} <- IfRef$Await(TRANSITION, reliable, 0)
```

```
Radar : TA_INTERFACE =
BEGIN
  TIMER : Trk_t, Rng_t, Stby_t;
  STATE : Trk_rel, Trk_not, Rng_rel, Rng_not, Stby_rel, Stby_not, Failed;
  SET down : Trk_not, Rng_not, Stby_not, Failed;
  GUARD Trk_not (Trk_t ≤ One_Second);
  GUARD Rng_not (Rng_t ≤ Two_Seconds);
  GUARD Stby_not (Stby_t ≤ One_Minute);
  TRANSITION reliable (Trk_not, , , Trk_rel)
                      (Rng_not, , , Rng_rel)
                      (Stby_not, , , Stby_rel);
  TRANSITION unreliable (Trk_rel, ,<Trk_t>, Trk_not)
                      (Rng_rel, ,<Rng_t>, Rng_not)
                      (Stby_rel, ,<Stby_t>, Stby_not);
  TRANSITION track (Rng_not, ,<Trk_t>, Trk_not)
                      (Rng_rel, , , Trk_rel)
                      (Stby_not, ,<Trk_t>, Trk_not)
                      (Stby_rel, , , Trk_rel);
  TRANSITION range (Trk_not, ,<Rng_t>, Rng_not)
                      (Trk_rel, , , Rng_rel)
                      (Stby_not, ,<Rng_t>, Rng_not)
                      (Stby_rel, , Rng_rel);
  TRANSITION standby (Trk_not, ,<Stby_t>, Stby_not)
                      (Trk_rel, , , Stby_rel)
                      (Rng_not, ,<Stby_t>, Stby_not)
                      (Rng_rel, , , Stby_rel);
  TRANSITION failure (Trk_rel, , , Failed)
                      (Trk_not, , , Failed)
                      (Rng_rel, , , Failed)
                      (Rng_not, , , Failed)
                      (Stby_rel, , , Failed)
                      (Stby_not, , , Failed);
  TRANSITION          (Trk_not, (Trk_t ≥ One_Second), , Failed)
                      (Rng_not, (Rng_t ≥ Two_Seconds), , Failed)
                      (Stby_not, (Stby_t ≥ One_Minute), , Failed);
  STARTAT Stby_rel;
END.
```

Figure 5.9: The Specification of a Timed Automaton

```
Wait on the down set:
  {state} <- IfRef$Await(SET, down, 0)
Signal a reliable transition and wait on the down set:
  {state} <- IfRef$SigAwait(reliable, SET, down, 0)
```

The combined operations are useful in enforcing interval timing constraints. For example, the "A equal B" relation in Figure 5.4 can be implemented as:

```
Block A :
  {} <- IfRef$SigAwait(+A, SET, 3, 0)
   code of block A
  {} <- IfRef$SigAwait(-A, SET, 6, 0)

Block B :
  {} <- IfRef$SigAwait(+B, SET, 3, 0)
   code of block B
  {} <- IfRef$SigAwait(-B, SET, 6, 0)
```

5.3.2 Embedded Code

The TA specification language has no component to specify the "output" responses. This is the main reason why the language must support embedded code in another language. Output responses can be handled by "callbacks". A callback is an operation supplied by the application which is executed when a synchronization point is reached. Another reason is that many useful programming functions already exist in a variety of languages, and it should not be necessary to be supported as the fundamental functions of the TA specification language.

Embedded code takes two forms: execution code and initialisation code. The target language initialisation code can be inserted before the BEGIN part of a TA specification, using a "*data*" and a "*code*" clause. The initialisation code contains the data and the prelude operations specific to the TA. Similarly, a "*code*" clause can be used to specify the body of a target language execution code associated after the *Action* part of a TA specification.

Figure 5.10 shows an extended Radar TA whose "failure" transition causes a callback to a failure handling interface.

The current ANSA Testbench supports only the C language, therefore only C is allowed as the embedded language in the TA specification.

5.4 Virtual Time

It is often the case that the time used within an application is not in terms of world time or realtime [4] [101]; rather a transformation of the realtime is used to define an application specific clock (rate). This is supported by the *virtual time* mechanism in the TA. The virtual time model is illustrated in Figure 5.11.

```
Radar : TA_INTERFACE =
data [ InterfaceRef fail; ]
code [ {fail} <- traderRef$Import(arguments);]
BEGIN
  ...
  TRANSITION failure (Trk_rel, , code [{}<-fail$trk_fail();], Failed)
                     (Trk_not, , code [{}<-fail$trk_not_fail();], Failed)
                     (Rng_rel, , code [{}<-fail$rng_fail();], Failed)
                     (Rng_not, , code [{}<-fail$rng_not_fail();], Failed)
                     (Stby_rel, , code [{}<-fail$stby_fail();], Failed)
                     (Stby_not, , code [{}<-fail$stby_not_fail();], Failed);
  ...
END.
```

Figure 5.10: Embedded Code Example

Associated with each TA, there are some (infrastructure provided) time management operations to enable an application to define and use its own virtual time. A virtual time y is a linear function of the real time x i.e. $y = ax + k$, in which $a = m/n$. By default, the virtual time of a TA is directly mapped to the real time i.e. $y = x$. The following operations can be applied to control the clock tick of the virtual time of a TA:

```
Set the virtual time function:
  {} <- IfRef$Time_Virtual(m, n, k)
Freeze the virtual time by delay:
  {} <- IfRef$Time_Freeze(delay)
Resume the virtual time:
  {} <- IfRef$Time_Resume()
Catch up the delayed virtual time:
  {} <- IfRef$Time_Catchup(delay)
```

The virtual time function can also be set at the creation time such as:

```
{IfRef} :: Radar$Create(arguments)
IfRef$Time_Virtual(m, n, k)
```

It is worth to note that if the value `m` is set to zero, then the virtual time would be a constant, which turns off all time effects of a TA — the TA would be equivalent to a normal finite state machine without timers.

The `Time_Freeze` operation freezes the virtual time for `delay`.[1] The `Time_Catchup` operation adds the `delay` to the virtual time;[2] the effect of this time advance on the TA transitions is equivalent to a smooth increment of the virtual time clock.

[1] A zero value may be used to denote an infinite delay.

[2] A zero value in this case denotes the default *now* time.

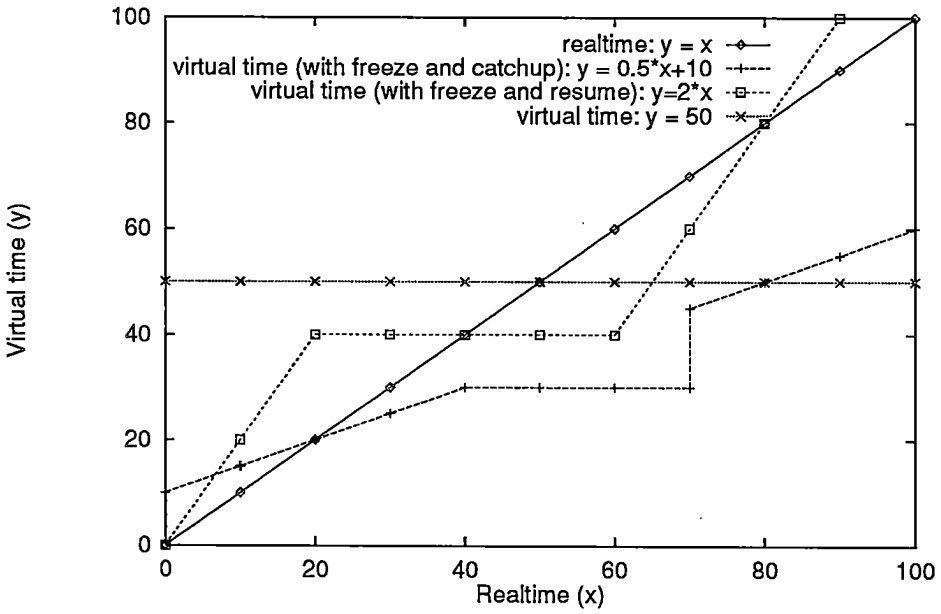

Figure 5.11: Virtual Time vs. Realtime

The virtual time mechanism is very useful for many realtime applications. For example in a multi-media presentation system, a "hold" event may cause a presentation server to hold the current screen and the time in the server would therefore be frozen; while a "resume" event would cause the server to continue displaying media objects with the pre-defined time order.

5.5 Implementation

A compiler *tagen* generates ANSA IDL interface specifications and server PREPC programs from TA specifications. The Testbench is also enhanced to process application-allocated timer alarms. The detailed description is given in section 6.5.

5.6 Summary

The TA temporal synchronization facility has the following characteristics:

- the TA provides a high-level abstraction of the temporal synchronization activities.
- both the instantaneous and the interval temporal constraints can be expressed.
- by keeping in-line with the ANSA Computation Model, the facility allows distribution.
- the facility preserves separation of concerns in the sense that (1) event-detecting computation activities are able to signal events without having to wait for activities

that synchronize with the signals; (2) computational activities that signal events do not embody assumptions about how these signals are used; and (3) activities that respond to events or use state information do not need to embody assumptions about the activities that signal the events or about one another.

- the facility is implemented by automatic transformation. It relieves the application programmers from the tedious and error-prone task of managing timing and a state machine.

Chapter 6

A Prototype Implementation

This chapter details the implementation of the proposed RIDE system design. The prototype system is built on the Cambridge Systems Environment. The environment is described in Section 6.1 and some kernel extensions are explained in Section 6.2. The implementation of the RIDE tasking system is outlined in Section 6.3. The implementation of the RIDE communication system is described in Section 6.4. Finally, the implementation of the timed automata facility is given in Section 6.5.

6.1 Systems Environment

The systems environment used for development and evaluation consists of a collection of heterogeneous host and target machines, all connected by the Computer Laboratory Ethernet. The host machines are workstations running UNIX operating systems, providing the tools and basic services for development. The target machines are the Fireflys [98] (prototype multiprocessors developed at DEC SRC) running the WANDA microkernel, on which experimentation and evaluation were carried out.

WANDA provides a high performance, lightweight platform upon which system servers and application programs could be run. A WANDA *domain*, or process, consists of a set of independently schedulable threads[1] which all share a common address space. Scheduling decisions are made by the kernel based upon the statically assigned priorities of runnable threads.

The WANDA IPC facility supports communication between threads in different domains on the same machine and on different machines connected by a network. The IPC interface is derived from the Berkeley UNIX socket abstraction. The main communication protocol is the Multi-Service Network Level (MSNL) protocol, defined within the Multi-Service Network Architecture [77]. MSNL is implemented within both WANDA and most local versions of UNIX, supporting unreliable communication based upon the use of lightweight virtual circuits in an inter-network environment.

The ANSA Testbench 3.0 has been ported to run over both WANDA and the Laboratory UNIX systems, providing a distributed system environment. The Testbench enables easy

[1] A WANDA thread is conceptually equivalent to a ANSA task.

access to many traditional operating system components. It has facilitated the building of distributed software in the current host-target setting. Many system services, such as the WANDA Process Server (a WANDA program loader) and its UNIX side server the Request Server[2], are ANSA programs.

The Firefly is a close-coupled multiprocessor with a central main memory shared by some number of processor-cache subsystems. One of the processors is connected to a QBus I/O bus. Network access is via a DEQNA device controller connecting the QBus to a 10 Megabits/Second Ethernet. The machines used in this experiment are a later version of Fireflys. Each such system has a Micro-VAX II CPU as the I/O processor (which provides about 1 MIPS of processor power) and 4 CVAX CPUs as the non-I/O processors. Compared to the Micro-VAX CPU, a CVAX CPU supplies a processing power approximately twice as fast and caches four time as big.

6.2 WANDA Extensions

The realtime capabilities provided by WANDA are limited in terms of scheduling, synchronisation and kernel monitoring. This section explains some of extensions made to WANDA for realtime applications.

6.2.1 Scheduling

The WANDA kernel has a static priority based preemptive scheduler. This is not sufficient when deadlines associated with realtime activities have to be respected. Also, to explore the *policy-free* characteristics of the RIDE architecture, a flexible kernel scheduling service is desirable. These requirements plus the experimental nature of the implementation have led to a design and implementation supporting multiple kernel scheduling policies. A policy/mechanism separation in the scheduler is adopted.

A thread is created and assigned to a specific processor.[3] Each processor has its own thread run queue and a policy *id* which selects a scheduling policy used for the processor. A scheduling policy is a compare function which decides the position of a runnable thread in the run queue, given that the first thread in the queue gets the processor first. Preemption happens if a thread with higher criticality (depending on the scheduling policy) than the currently running thread becomes available.

Four scheduling policies are placed in the new kernel. They are priority scheduling, deadline scheduling, first come first served, and priority and deadline combined scheduling. The latter combines the priority and deadline scheduling, using a priority as the main criterion and an optional deadline as the secondary criterion. By default, each processor uses the priority scheduling policy, and can be changed to use other policies by a kernel call at run-time. In figure 6.1, a graphical illustration of the new kernel scheduler is given.

[2]The Request Server is a file server on a UNIX machine, it reads a WANDA program image from a UNIX file system and transfers the image to the Process Server.

[3]In the current mixed setting of the Firefly processor boards, thread migration from one processor to another is not always possible [18], therefore the WANDA kernel adopts fixed thread/processor assignment.

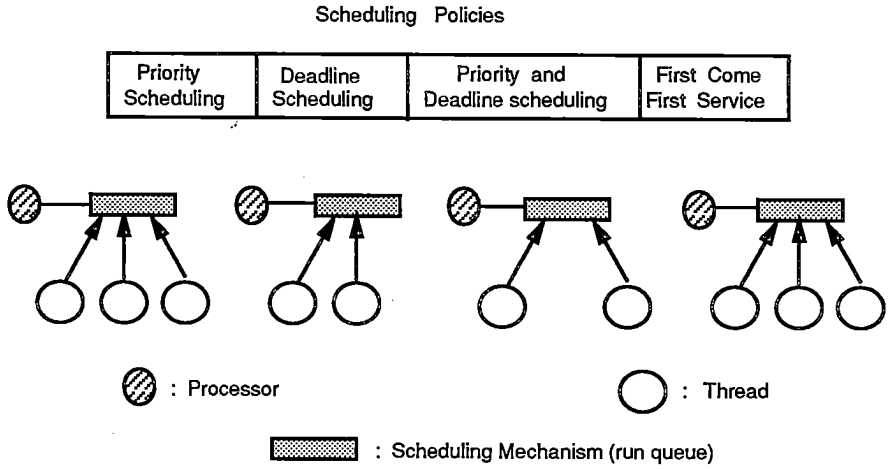

Figure 6.1: The New Kernel Scheduler

The WANDA kernel provides an *event mechanism* to pass kernel-level events, including hardware exceptions, to the relevant user level thread. When each address space is started, an event-waiter thread is created, which handles all the events raised in the address space.

In the new kernel, a thread may be associated with a priority, a deadline, a deadline type and a period value. A deadline type can be either *soft* or *hard*. A hard deadline forces the kernel to monitor whether the deadline has been missed, as well as being used as a scheduling parameter by the kernel scheduler. A soft deadline is purely used for scheduling purposes. The period defines the periodic nature of a thread, and is implemented by the user level part of the thread package. If the period is not met, the thread package launchs a *Period Miss* event, and a technique known as *load-shedding* [103] is used to synchronise the period.

The WANDA kernel maintains a timer queue for system clock related activities. Upon each system clock interrupt, the interrupt handler examples the timer queue and processes the expired timers. If a thread has a hard deadline, a deadline timer for the thread is allocated on the timer queue. If the deadline timer expires before the thread finishes, a *Deadline Miss* event is launched. The system clock rate determines the deadline granularity, which is currently 10 milliseconds. A faster clock rate may be used, although this also introduces extra interrupt overheads.

6.2.2 Thread Synchronisation and Priority Inheritance

The basic synchronisation mechanism for WANDA threads is the counting semaphore. The traditional 'P' and 'V' operations and a facility for waiting on a semaphore subject to a timeout are provided.

To solve the *unbounded priority inversion* problem (see Section 2.5.1), the implementation of binary semaphores with both PIP and PCP has been investigated.

There are several reported implementations of realtime thread synchronisation with PIP or PCP. Heuser [51] developed a collection of low-level synchronisation tools and a client-server coordination mechanism to implement the PIP in a shared memory multiprocessor

UNIX operating system. Goodenough and Sha [40] proposed a client-server method to implement PCP in the Ada task model. Tokuda et al. [100] and Khanna et al. [66] give implementations of PIP for realtime Mach and SunOS 5.0 respectively. No report of the PCP implementation on a multiprocessor has been found.

The work reported here extends Heuser's work to implement both PIP and PCP in the WANDA kernel on a shared memory multiprocessor (Firefly) machine. The approach is different from others in that it tries to shift as much functionality from kernel to user level as possible. The detailed design and algorithm is given in [74]. Only a skeleton is discussed here.

The following implementation requirements have been taken into consideration.

- *Maintaining the Protocol Graph.* The operation of PIP and PCP depends on the knowledge of the dynamic mutex/thread owner and block relations, which can be represented as a graph. For PIP, the relationships between threads and mutex can be represented by a directed bipartite graph [76]. For PCP, an extra relation is needed to define the priority ceiling order among the mutexes.

- *Minimum Kernel Interference.* Critical sections are usually short, and the operations to enter and exit from them should be fast. Ideally, the acquiring of a mutex that satisfies synchronisation conditions is better done in user mode without a trap to the kernel. Similarly, the releasing of a mutex on which no threads are blocked should be done in user mode. This scheme requires *fine grain synchronisation* in user space. This is usually implemented through a low-level *busy-wait spin lock.* While holding the lock, a thread can check whether the protocol-related synchronisation conditions are satisfied, and can decide if it can acquire a mutex or should be blocked on it.

- *Maintaining the Running Thread Invariant* — the running threads are the runnable threads with the highest effective priorities. A special kernel interface is needed to coordinate the scheduling of processors and the management of threads which are blocked on or holding mutexes. When a thread is holding a mutex, it may inherit a new priority and later revert to its old priority. Priority inheritance and de-inheritance cannot be done without kernel assistance.

- *Restriction of Mutex Contentions.* In a shared memory multiprocessor environment, a nontrivial policy is needed to handle the release of a mutex when there are blocked threads awaiting. For example, if a simple policy that wakes up all the blocked threads is used, then because the unblocked threads may run on different processors, it is not possible to guarantee that the highest priority thread will get the mutex first. Also, it is a waste of processor time since only one thread will be successful in getting the mutex.

In the current implementation, the functions of PIP and PCP are divided between the operating system kernel and a user library. The implementation consists of the low-level fine grain synchronisation mechanisms, kernel calls and user interface library routines.

The mechanism of a *preemption variable* is introduced to provide low-overhead control of processor rescheduling. Busy-wait spin locks (providing two operations: *spin_lock* and *spin_release*) are used to provide mutually exclusive execution of the protocol codes. Kernel

calls are provided to support dynamic priority and priority inheritance. The user interface library maintains the thread-mutex relation graphs defined by the protocols and their dynamic transformations. This approach contrasts other implementations which store the protocol graphs inside an operating system kernel.

6.2.3 Monitoring Scheduling

The WANDA event mechanism was designed primarily for address space management and thread management. The kernel launches *thread exit* events and *domain stop* events to the related event waiters. This simple event mechanism has been extended to monitor kernel activities, especially to monitor the kernel scheduling activities.

The new kernel can be conditionally compiled to have code to generate scheduling related events to a well known event port. Each such event consists of an event type, a thread *id*, a address space *id*, a processor number, and a time stamp. The following types of events can be generated: *thread create, thread exit, thread switch, thread preemption, thread block, domain start, domain stop, domain context switch,* and *deadline miss.*

An application program can spawn a thread to listen at the event port. The thread may register events of interest to it by an event mask. Only one such thread is allowed per kernel in the current implementation. This is not a restriction because only privileged users are interested in scrutiny of how the kernel scheduler behaves.

A number of applications can be built based on the new event mechanism. For example, a user level schedule server [72] can be built upon it. The event mechanism allows the server to maintain up-to-date information of the current thread execution state, the current processor usage etc, and carry out dynamic scheduling adaptation and dynamic scheduling enforcement. Another example is execution replay. An application was implemented to cache the kernel scheduling activities and pass them to a UNIX-side server, which then graphically displays the kernel execution trace on an X-terminal.

6.3 Implementation of RIDE Tasking

This section has three parts. The first part reviews the tasking system in the ANSA Testbench. The second part gives the details of the RIDE preemptive tasking implementation. The third part outlines the RIDE thread scheduling mechanisms.

6.3.1 Tasks and Threads in the ANSA Testbench

ANSA tasks and threads have been outlined in Section 3.2.2. ANSA threads represent points of execution and provide the notion of logical concurrency. ANSA tasks represents the resources required (stacks) to execute an ANSA thread and provide the actual concurrency.

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

execute the user program code. Application programs may create additional threads for concurrency. For example, a client may communicate with two different servers using two different threads. On the server side, additional threads are created implicitly. The receiver thread creates one additional thread for each request from a client.

In the original Testbench, ANSA 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. The ANSA 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.

The Testbench nucleus is driven by the following shared data structures holding the capsule state.

- the channel table controlling the inter-capsule bindings.
- the session table controlling the communication sessions, which themselves control the sending and receiving of independent sequences of messages.
- the thread table controlling the application threads.
- the task table controlling the tasks.
- the timer module maintaining a list of all currently outstanding timers in temporal order.

The coroutine nature of the ANSA task package has a major effect on the maintenance of the shared data structures. Since scheduling is non-preemptive, the shared data structures can be maintained in a global area without any synchronisation mechanisms. The Testbench took another advantage of non-preemptive scheduling. 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 design decision made almost all the procedures non re-entrant.

The WANDA Port of the ANSA Testbench

The ANSA Testbench over WANDA was ported by Dixon [23]. The port is almost the same as the Testbench over a UNIX system. But the implementation exploits the concurrency facility (lightweight threads) provided by WANDA. Each ANSA task is implemented as a WANDA thread. The ANSA scheduler is redundant here: task scheduling is done by the WANDA kernel. The initial threads do not share tasks: each has its own task. Due to the connection-based nature of the WANDA IPC interface, for each end point of a communication connection there is also a communication task.

Because of the real concurrency in the implementation, 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 ANSA operation, a task must first acquire the lock, then operate on the shared data, and finally release the lock when finished. Preemptivity is not exploited: all tasks are assigned the same priority. In this

way, most of the ANSA software remains unchanged. However, as the ANSA operation path is serialized, throughput is severely limited. An alternative implementation approach is pursued in this dissertation.

6.3.2 Preemptive Tasking Implementation

A more modular approach to locking is to protect each item of shared data separately with its own lock. Each operation on a data structure can then be surrounded by a lock acquisition and release. For the ANSA Testbench, this involves separate locking for operations on the channel table, the session table, each session item, the thread table, the task table, and the timer queue. The RIDE extension introduces another data structure — the entry table — on which a separate lock is also used.

There is a basic tradeoff between latency and throughput in the choice between using a single lock or multiple locks in protecting shared data structures. As less of the total ANSA activity is in a critical section, and since it is split among several locks, the maximum rate of ANSA operations, and task preemptivity, is higher with multiple locks than with a single lock. There is a cost to this benefit, however; more lock accesses are needed, which increases latency.

Tasks are allowed to have different priorities in the new implementation, therefore preemption is possible. This means that ANSA procedures have to be re-implemented to be re-entrant. The variables forming the context of a task are no longer global variables, but are grouped into a private data structure and the pointer to this structure can be accessed by an indirection operation on the task *id* which can be acquired through a kernel call.

Timer Task	Communication Task_1	Communication Task_n
P(*timer_lock*);	P($session\_lock_{i1}$);	P($session\_lock_{in}$);
...	...	...
P($session\_lock_i$);	P(*timer_lock*);	P(*timer_lock*);
...	...	...
V($session\_lock_i$);	V(*timer_lock*);	V(*timer_lock*);
...	...	...
V(*timer_lock*);	V($session\_lock_{i1}$);	V($session\_lock_{in}$);

Figure 6.2: Fine Grain Synchronization and Deadlock

Another design decision that is affected by the introduction of preemptive scheduling and making the context of each task private, is the policy of allocating memory contiguously in a dynamic manner. The Testbench 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. This scheme is not valid in the case where each task holds its own private context (which is basically a set of pointers to the shared memory). The solution in this work is to allocate memory statically. This is a simple solution, but also loses the scalable property of the Testbench. A more rational solution is perhaps to use hash table based dynamic memory allocation.

Timer Task	Communication Task$_1$	Communication Task$_n$
WriterP(*rw_lock*);	ReaderP(*rw_lock*);	ReaderP(*rw_lock*);
P(*timer_lock*);	P(*session_lock*$_{i1}$);	P(*session_lock*$_{in}$);
...	...	...
P(*session_lock*$_i$);	P(*timer_lock*);	P(*timer_lock*);
...	...	...
V(*session_lock*$_i$);	V(*timer_lock*);	V(*timer_lock*);
...	...	...
V(*timer_lock*);	V(*session_lock*$_{i1}$);	V(*session_lock*$_{in}$);
WriterV(*rw_lock*);	ReaderV(*rw_lock*);	ReaderV(*rw_lock*);

Figure 6.3: Deadlock Resolution

The fine-grain nature of synchronisation is often accompanied by two common problems: *deadlock* and *contention*. Deadlock occurs when two or more tasks each hold a lock the other requires, and thus prevent their executions. In most cases, deadlock can be avoided by examining each execution path and enforcing a proper order on lock-acquire and lock-release operations. Contention occurs when two or more tasks can simultaneously access unguarded shared data, and is avoided by reordering some of the Testbench code. This is tedious work, but is vital for the correct execution of the Testbench in a real parallel and preemptive environment.

There is also a deadlock case that must be handled by a special technique. Deadlock may occur between the timer task and the communication tasks (note, for each WANDA IPC connection, there is a communication task). The timer task needs to acquire the timer lock first, and then a session lock to process the timing work related to the execution protocol. A communication task needs to acquire a session lock first, and then acquire the timer lock (also for the purpose of timing management related to the execution protocol). The situation is shown in Figure 6.2. The deadlock could be solved by using another lock (guarding the whole code in the tasks) to enforce mutually exclusive execution at a coarse granularity. But this also introduces unnecessary blocking among the communication tasks. The solution adopted here is to use a reader-writer lock[4] as shown in Figure 6.3. The reader-writer lock is implemented by several WANDA semaphores.

6.3.3 Thread Scheduling

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

In RIDE, a new thread is queued on an entry (see Section 3.3), instead of the capsule FCFS queue. In case (1), the application gives an additional entry *id* when a new thread

[4] A reader-writer lock guards the exclusive access of a critical section between a set of writer tasks and a set of reader tasks. In addition, it allows as many as readers to enter the critical section simultaneously, but only one writer is allowed to enter the critical section each time.

is created. In case (2), the binding between an interface and an entry determines on which entry the new thread should be queued.

Each entry is associated with a thread scheduling policy, which is executed each time a thread is queued on the entry. Thread scheduling policies are detailed in Section 3.9.

One or more system tasks may be allocated for each entry. Such a task runs an infinite loop to dequeue and execute one thread each time on the entry. Upon getting a thread, a task also adjust its kernel scheduling parameters according to the thread scheduling parameters before executing the thread. This is defined by the task/thread rendezvous policy (also see Section 3.9) of an entry. Some policies require support from the kernel. For example, if the kernel does not support deadline based scheduling, it is meaningless for the task to inherit the deadline of a thread. Upon finishing a thread, a task always reverts to its original scheduling parameters. The transitive priority inheritance policy has not been implemented yet.

Another extension is to allow a task to rendezvous with an entry dynamically (when executing a thread), each such rendezvous executes one thread on the entry. The rendezvous of one task with a set of entries has not been implemented yet.

6.4 Implementation of the RIDE Communication System

An outline of the ANSA communication system was shown in Section 4.1. This section explains some of the implementation details for the RIDE extensions.

6.4.1 Parallel Protocol Stacks

The fine-grain synchronisation of the RIDE tasking system provides the basis for the parallel execution of the RIDE parallel protocol stacks (see Section 4.2). Each RIDE MPS channel (LWC) is mapped to a WANDA IPC socket. The connection management operations of LWC are mapped onto the WANDA IPC connection management operations.

The RIDE parallel protocol stack also raises two management problems: connection management and multi-threaded network reception. It seems they are better addressed in an application-based manner rather than a default or system-based manner as explained in the following.

Connection Management. An interface may or may not want to have private LWCs from its clients. Also, in the case that a connection is allowed, there is a problem of connection error management, because a LWC is not assumed to be reliable, and may be torn down by the underlying operating system.

Multi-threaded network reception. A dedicated task is spawned to listen on each end point of a LWC. Such tasks execute the whole ANSA communication protocol stack functions and may consume considerable processor time for each network reception. These tasks' priorities must be consistent with the importance of their corresponding interfaces. This priority difference enables the preemption among the communication tasks, so that important services have better response time.

On creating an interface, a server may pass its management requirements through the control arguments in the PREPC statement:

```
{IfRef} :: IfName$Create(arguments)
```

The `arguments` specify (1) whether connections are allowed; (2) the priority of the server side network reception task.

When making a private LWC with a server interface, a client may also specify its management requirements through the control arguments in the PREPC statement:

```
IfRef$Connect(arguments)
```

The `arguments` specify (1) a connection management policy — either to initiate a new connection or to invalidate the current connection when a connection error occurs; (2) the priority of the client side network reception task.

If a client is making an invocation on a broken (invalid) LWC, an exception is raised to allow it to take further actions.

6.4.2 Timed RPC Protocol

Based on the ANSA Remote Execution protocol, a timed remote execution protocol (TREX) is implemented to support the timeliness of TRPC calls.

Like REX, TREX constructs its communication protocol based on the send/receive asynchronous message passing service provided by MPS.

TREX extends REX in the following aspects:

- the header for the REX packets is expanded to include the information about the priority, deadline and deadline type.
- the session function is extended to process timeouts at client sides and deadlines at server sides.
- extra packet types are used to handle deadline exception and confirmation.
- a *session timeout recovery* scheme is introduced to cope with the session inconsistency between a client session and its server session when a timeout expires.

Timeouts 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 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.

The RIDE approach for session timeout recovery is illustrated in Figure 6.4. With this approach, the client continues immediately after the timeout, and the client session is

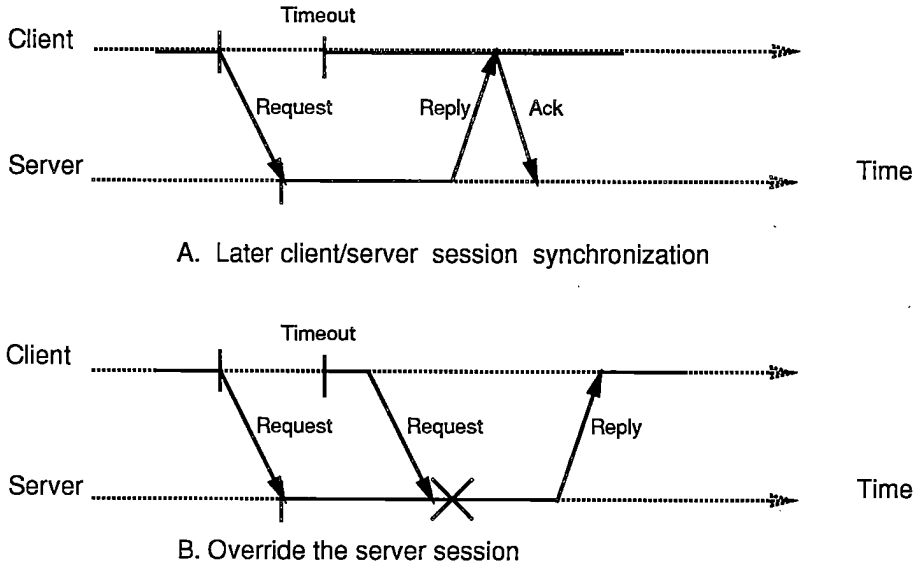

Figure 6.4: Session Timeout Recovery Illustration

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 ANSA REX 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.

The primitive packets and their semantics used in TREX are shown in table 6.1.

Primitive Packet	*Semantics*	
call	client call message	*
cast	client cast message	*
reply	server reply message	*
call_ack	server ack of client call	*
reply_ack	client ack of server reply	*
frag_nack	message fragmentation nack	*
deadline_nack	notification of unsatisfiable deadline from server	
deadline_ack	notification of satisfied deadline from server	

* means an REX defined packet.

Table 6.1: Timed Remote Execution Protocol Packets

6.4.3 Decomposable RPC Protocol

The implementation of the decomposable RPC protocol (see Section 4.4) introduces another two primitive packets: *at-most-once call* and *fast-path call.* For these two kinds of calls, a default timeout value is automatically associated if an application does not provide one. This is necessary to prevent a client getting stuck at an invocation when a communication failure occurs.

6.5 Implementation of the Timed Automata

A compiler *tagen* generates ANSA IDL interface specifications and server PREPC programs from the timed automata specifications (written in the TA specification language described in Section 5.3.1). For example, given a TA specification radar.ta, "tagen radar.ta" generates an ANSA interface specification radar.idl and a server PREPC program radar_svr.dpl. The sequence of commands to generate the radar temporal synchronisation server is shown as follows:

```
tagen radar.st          (generate radar.idl and radar_svr.dpl)
stubc radar.idl         (generate stub routine cradar.c sradar.c etc.)
prepc radar_svr.dpl     (generate radar_svr.c)
compile radar_svr.c sradar.c  (generate radar_svr.o sradar.o)
link server radar_svr.o sradar.o Testbench_Capsule_Library
```

The compiler is written in YACC. From a TA specification, the compiler constructs an IDL file of "TYPE =" statements which provide IDL data types to define the transitions, sets and timers used for the execution of the machine. The compiler also constructs the signatures of the `Signal, Await, SigAwait, AwaitSig`, and other time management operations in the IDL file.

The generated server PREPC program provides a state machine plus the operations to support the ANSA interface of the machine. The unusual aspect of the state machine is the handling of timing constraints. The timing constraint on a state is a set of time expressions (timer <= time_value) or (timer >= time_value) combined with logical 'and' or 'or'. The time expression may change its logical value due to the time progression. The effect of this change may cause the TA to conduct an internal state transition. The detection of this change is handled by a time module which becomes active when a timer expires at a timeout value. For each state with a timing constraint, the compiler generates a timer handler routine and a set-alarm routine. The set-alarm routine is called when a transition caused the state machine to enter the state. The routine adds the required timeout alarms on the infrastructure-provided time alarm service. When a timeout expires, the appropriate timer handler routine is called to check the time constraint and take appropriate action if required, such as to conduct a transition.

The Testbench nucleus has an internal time module to provide a timeout alarm service for the communication protocols. This module has been extended to provide a general interface to afford other (application level) timeout alarm processing, such as that required to handle the timing constraints on a timed automaton.

6.6 Summary

The prototype described in this chapter represents an almost complete implementation of the RIDE system. Extensions have been made to the WANDA kernel in order that a broad range of scheduling policies, under management control, may be supported. Realtime synchronization is addressed by the provision of binary semaphores with PIP and PCP.

Preemptive tasking is achieved via re-implementing the Testbench procedures based on fine-grained synchronizations. The parallel protocol stack is achieved by mapping each MPS light-weight channel to a separate WANDA socket. The TRPC protocol is implemented by a timed REX. The timed automata synchronization facility is achieved via the automatically generated implementation.

Chapter 7

Performance Measurement and Evaluation

This chapter describes the performance of the RIDE prototype system. The basic performance of the WANDA kernel is presented first. Then the Hartstone Benchmark [103] is used to give the synthetic performance of the kernel thread management. Next, the Distributed Hartstone Benchmark [78] is used to measure the synthetic RIDE realtime performances in a network environment. Finally, the multiprocessor speedup of the parallel execution of the RIDE protocol stack is evaluated.

7.1 WANDA Basic Performance

Type	Time (μs)
Null Procedure Call	7
Null System Call	31
Thread Switching	228
Preemption	248
Semaphore Ping-Pong	563

Note: the performance was measured using a single CVAX CPU on a Firefly WANDA machine.

Table 7.1: The Basic WANDA Performance

The basic WANDA performance shown in Table 7.1 is obtained from three categories of basic activity that are most crucial to the performance of realtime systems, irrespective of the actual application.

- *thread switching* time is the average time the system takes to switch between two independent and active threads of equal priority.
- *thread preemption* time is the average time it takes a higher priority thread to preempt the control of the processor from a running thread of lower priority. Though conceptually similar to thread switching, preemption usually takes longer. This is

because the kernel must first recognize the wake-up action and access the relative priorities of the running and requested threads, and only then switch threads if appropriate.

- *semaphore ping-pong* time is the delay between a thread's release of a semaphore and the activation of another thread blocked on the semaphore. No other threads are scheduled in between.

7.2 Controlled Priority Inversion

This section explains the experiments and performance of controlled priority inversion by using the PIP and PCP mutex.

In the absence of contention in the Firefly multiprocessor, the busy-wait functions (see Section 6.2.2) *spin_lock* and *spin_release* together cost 21 μs, the *semaphore_acquire* and *semaphore_release* of PCP together cost 100 μs, and the *semaphore_acquire* and *semaphore_release* of PIP together cost 79 μs. The performance is good in the sense that even a Firefly null system call costs 31 μs.

To test controlled priority inversion in the presence of contention, an experiment was designed as shown in Figure 7.1. A set of three non-harmonic periodic threads is used in the test. Their priorities are assigned according to the *rate monotonic scheduling rule* (see Section 2.5). H is the high priority thread with the smallest period, L is the low priority thread with the longest period and M is the medium one. L and H's job are to access a critical section guarded by mutex S. The experiment is to see the effect of a PIP mutex (a PIP or a PCP mutex behave the same here because only one mutex is used) in comparison with a common binary semaphore mutex. First, a suitable initial period value and workload for each thread are selected, so that all the threads can finish their workload within their deadlines (periods) for both kinds of mutexes. Then, M's workload is increased gradually in units of a millisecond to repeat the test. The experiment stops when any deadline of the threads is missed.

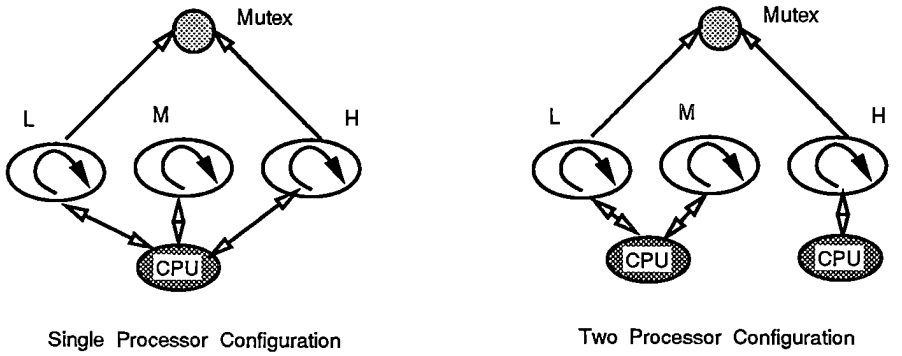

Figure 7.1: Controlled Priority Inversion

The initial test parameters are set as follows:

Thread	Period (ms)	Load (ms)
L	600	40
M	500	100
H	123	8

In the single processor test, when a WANDA semaphore is used for synchronization, H's deadline is missed when M's workload is increased to 137 ms. While if a PIP mutex is used, L's deadline is missed when M's workload is increased to 396 ms.

In the two processor test, H's deadline is missed when M's workload becomes 163 ms under the WANDA semaphore. L's deadline is missed when M's workload is 432 ms under the PIP mutex.

This experiment clearly shows the effectiveness of the PIP mutex in controlling priority inversion.

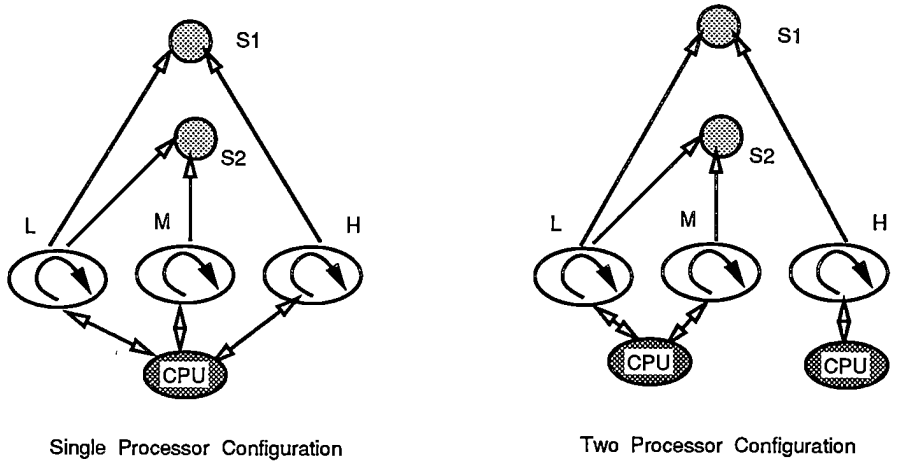

Figure 7.2: Strictly Controlled Priority Inversion

The second experiment, shown in Figure 7.2, is to test strictly controlled priority inversion with the PCP. The experiment uses three non-harmonic periodic threads, following the pattern of the last experiment. H accesses a critical section with a workload guarded by a mutex `S1`. M accesses a critical section with a workload guarded by a mutex `S2`. L accesses a nested critical section guarded by `S1` and `S2` with a sequence of processing steps { *acquire*(`S1`), workload, *acquire*(`S2`), workload, *release*(`S2`), *release*(`S1`) }. We use two PCP mutexes and two PIP mutexes for comparison. A set of initial values of periods and workloads is chosen to enable the threads to finish their workloads within their deadlines for both kinds of mutexes. Then, M's workload is increased gradually in units of a millisecond to repeat the test. The experiment stops when any deadline of the threads is missed.

The initial test parameters are set as follows:

Thread	Period (ms)	Load (ms)
L	600	24+24
M	500	80
H	123	8

In the single processor test, when two PIP mutexes are used for synchronization, H's deadline is missed when M's workload is increased to 194 ms. If two PCP mutex are used, L's deadline is missed when M's workload is increased to 403 ms.

In the two processor test, H's deadline is missed when M's workload becomes 240 ms under the PIP mutex. L's deadline is missed when M's workload is 449 ms under the PCP mutex.

This experiment clearly shows the effectiveness of the PCP mutex in strictly controlling priority inversion.

7.3 Hartstone Benchmark

Benchmark programs have been used for many years to compare the relative speeds of computers and language implementations. Among these, synthetic benchmark programs are used widely for the evaluation of different computer architectures and compilers. For example, the Whetstone [19] synthetic benchmark is a program which is meant to be representive of applications in scientific computing. The Dhrystone [102] synthetic benchmark is defined to address the need for measuring the performance of system programs. The objective of a synthetic benchmark suite is to evaluate the performance of a system as a whole, rather than to evaluate individual performance of separate components of a system as defined by some analytic benchmark programs.

Recently, there has been a growing interest in defining standard synthetic benchmarks for realtime computing systems. The Hartstone Benchmark (HB) [103], Distributed Hartstone Benchmark (DHB) [78] and Hartstone Distributed Benchmark (HDB) [64] are three examples of this effort. The HB is a set of timing requirements for testing a system's ability to handle hard realtime 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 realtime 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 realtime distributed systems' behaviour, while the DHB has a concrete definition of the series of tests.

The HB and DHB are used for the measurement of the RIDE system prototype. The HB performance is now discussed, followed by the DHB performance in Section 7.4.

The HB was developed by the Software Engineering Institute at Carnegie Mellon University to gauge the performance of realtime operating systems (specifically the Ada run-time executives). Five categories of tests are defined, each stressing one aspect of realtime operating systems. The five categories are (1) PH series: Periodic Tasks, Harmonic Frequencies; (2) PN series: Periodic Tasks, Non-Harmonic Frequencies; (3) AH series: PH series with Aperiodic Processing Added; (4) SH series: PH series with Synchronization; (5) SA series: PH series with Aperiodic Processing and Synchronization. The synthetic workload for HB is based on the Whetstone benchmark. It is designed such that the architecture level performance (raw performance) is automatically taken into account.

The current HB implementation published by the Software Engineering Institute gives

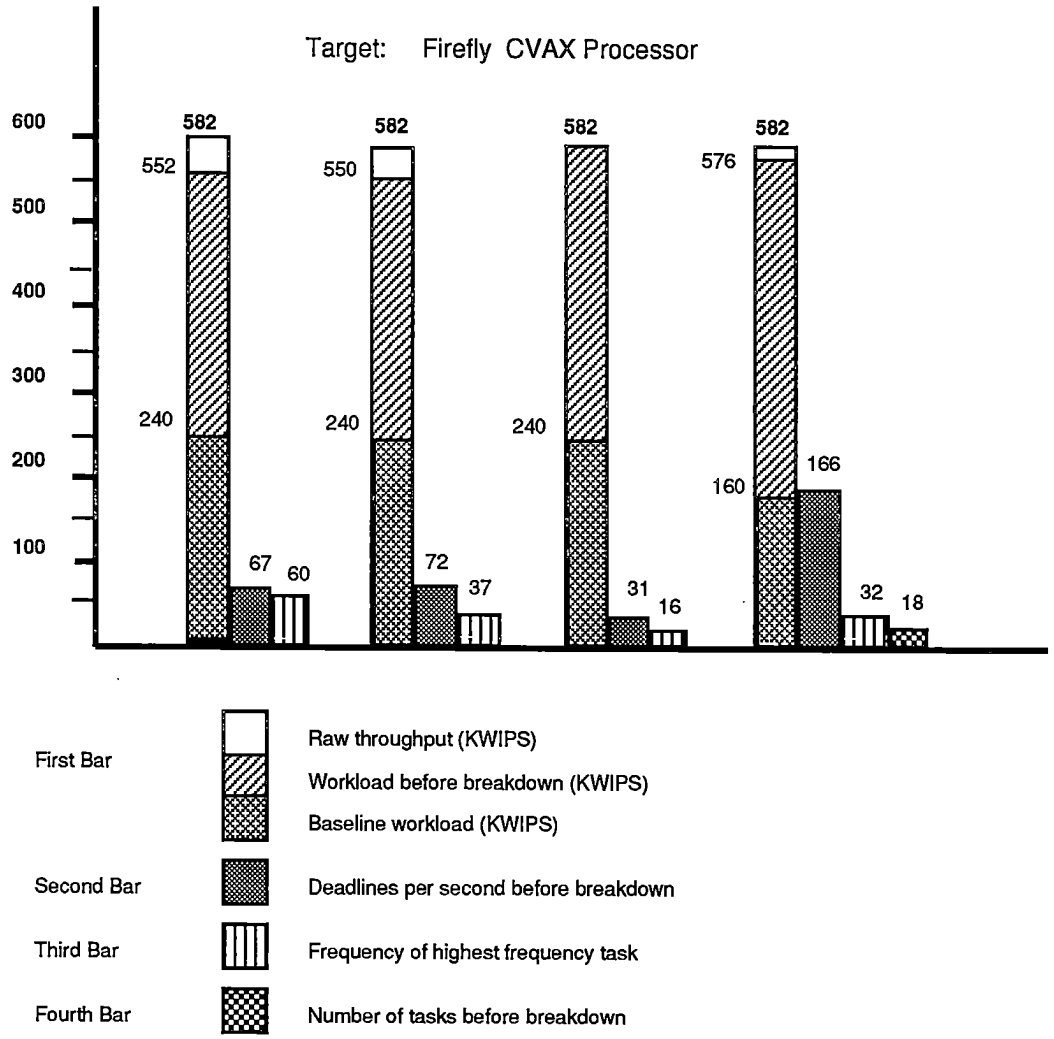

Figure 7.3: Summary Results: Firefly WANDA

only the PH series tests, and the source code is in the Ada programming language. A conversion of the Ada program to the WANDA C program is done first, resulting in about 2000 lines of C code.

The objective of the PH series is to provide simple test requirements with a load of tasks that are purely periodic and harmonic. This series might represent a program that monitors several banks of sensors at different rates and displays the results with no user intervention or interrupt requirements. The baseline system consists of five periodic tasks, each has a frequency, a workload, and a priority which is a function of the frequency — the higher the frequency, the higher the priority. This is consistent with the rate-monotonic scheduling discipline. Each task frequency is a multiple of every higher task frequency. The task workload is expressed in Kilo-Whetstone. The Whetstone calculation is the self-verifying version specified by [19]. A Hartstone task is required to execute a specific amount of Kilo-Whetstone within its period. The rate at which it does this amount of work is measured in Kilo-Whetstone Instructions Per Second (KWIPS). The deadline for completion of the workload is the beginning of the task's next period.

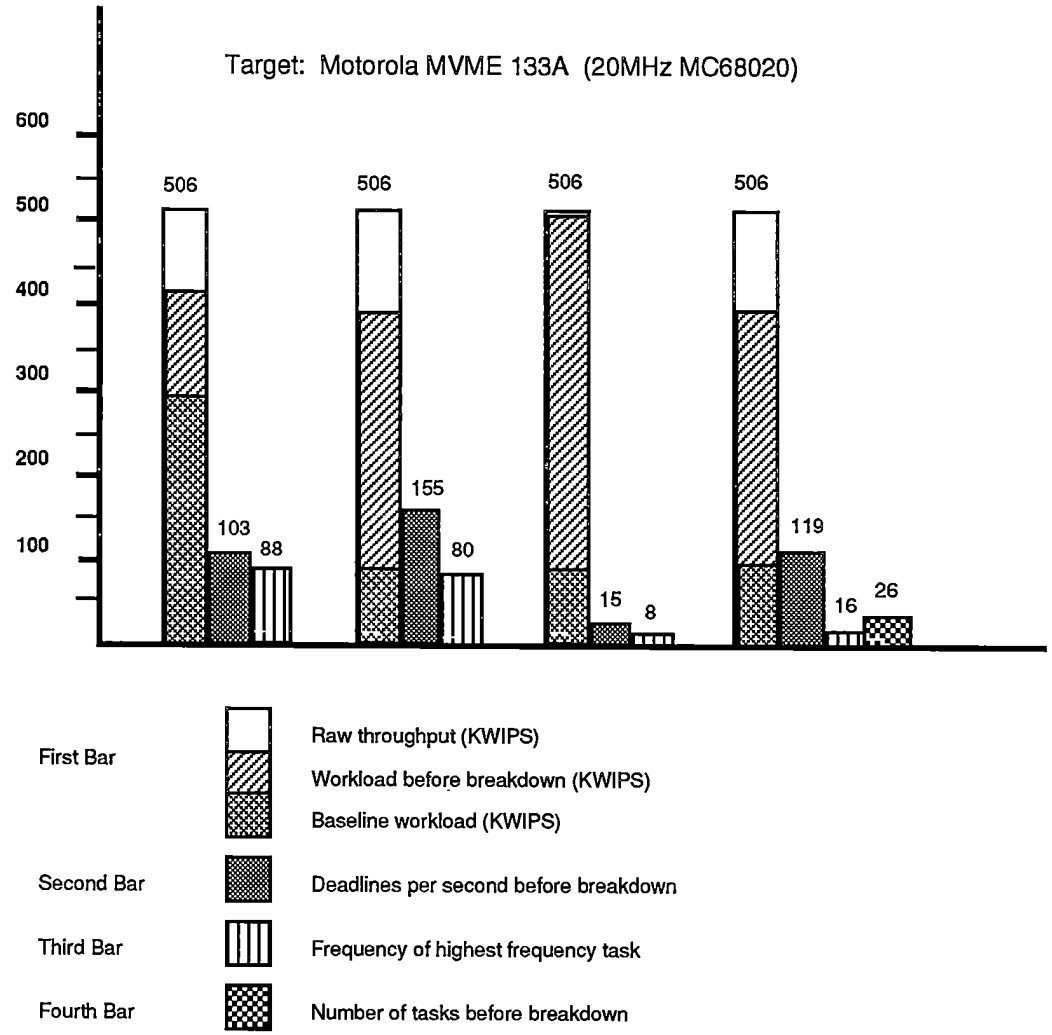

Figure 7.4: Summary Results: Verdix VADS

The raw speed of the benchmark is the number of KWIPS achieved by the test system. This calibration test is performed by the experiment package when an experiment is initialized. The performance requested of Hartstone tasks is expressed as a percentage workload utilization, which is computed as the ratio of the requested task speed to the raw benchmark speed. The utilization required of the entire task set is the sum of the individual task utilizations. Successive tests in an experiment increase the requested utilization to the point where deadlines are not met.

The Hartstone PH series defines four experiments as follows:

- *Experiment 1* starts with a baseline task set, and increases the frequency of the highest frequency task (Task 5) for each new test until a task misses a deadline.
- *Experiment 2* starts with the baseline task set after which all the frequencies are increased by a factor of 1.1, then 1.2, then 1.3, and so on for each new test until a deadline is missed.
- *Experiment 3.* The workload of each task is increased by 1 Kilo-Whetstone per

Experiment 1 (step size: 4.12 %)

Task No.	Frequency (Hertz)	Kilo-Whets Per Period	Kilo-Whets Per Second	Requested Workload Utilization
1	0.50	96	48.00	8.24 %
2	1.00	48	48.00	8.24 %
3	2.00	24	48.00	8.24 %
4	4.00	12	48.00	8.24 %
5	8.00	6	48.00	8.24 %
			240.00	41.40 %

Experiment 2 (step size: 4.12 %)

Task No.	Frequency (Hertz)	Kilo-Whets Per Period	Kilo-Whets Per Second	Requested Workload Utilization
1	1.00	48	48.00	8.24 %
2	2.00	24	48.00	8.24 %
3	4.00	12	48.00	8.24 %
4	8.00	6	48.00	8.24 %
5	16.00	3	48.00	8.24 %
			240.00	41.40 %

Experiment 3 (step size: 5.32 %)

Task No.	Frequency (Hertz)	Kilo-Whets Per Period	Kilo-Whets Per Second	Requested Workload Utilization
1	1.00	48	48.00	8.24 %
2	2.00	24	48.00	8.24 %
3	4.00	12	48.00	8.24 %
4	8.00	6	48.00	8.24 %
5	16.00	3	48.00	8.24 %
			240.00	41.40 %

Experiment 4 (step size: 5.49 %)

Task No.	Frequency (Hertz)	Kilo-Whets Per Period	Kilo-Whets Per Second	Requested Workload Utilization
1	2.00	16	32.00	5.49 %
2	4.00	8	32.00	5.49 %
3	8.00	4	32.00	5.49 %
4	16.00	2	32.00	5.49 %
5	32.00	1	32.00	5.49 %
			160.00	27.45 %

Figure 7.5: Hartstone Benchmark

period for each new test, continuing until a deadline is missed.

- *Experiment 4* starts with the baseline task set. New tasks with the same frequency, workload, and priority as the *middle* task (Task 3) of the baseline set are added until a deadline is missed.

The Hartstone PH series is executed on a single Firefly CVAX processor. The raw speed is 582 KWIPS. The baseline tests for each experiments are given in Figures 7.5.

The summary results of the four experiments on a Firefly CVAX processor over WANDA is given in Figure 7.3.

As a simple comparison, the Verdix VADS Ada run-time system [26] performance is given in Figure 7.4. It can be seen that VADS performs slightly better than WANDA in the first two experiments; while WANDA performs better in the last two experiments.

7.4 Distributed Hartstone Benchmark

The Distributed Hartstone Benchmark is an extension of the Hartstone work for the distributed realtime computing environment. In DHB, realtime task sets are defined to stress specific aspects of the environment such as communication latency, communication bandwidth, prioritised message handling, and preemptability of the communication processing. The intention of the DHB is to measure the realtime 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.

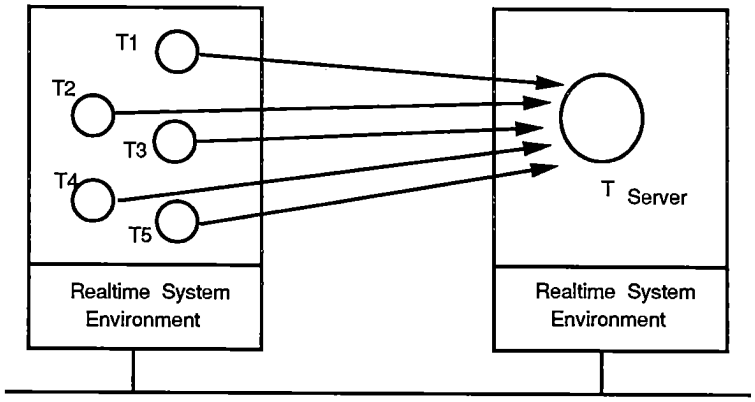

Figure 7.6: Five Clients with Single Server

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 7.6 shows the structure of the DSHcl series task set. Table 7.2 gives the timing requirements of the DSHcl series baseline test task set.

Task	Workload (KWIPS)	Period (ms)
T1	1	80
T2	1	160
T3	2	320
T4	2	640
T5	8	1280
Tserver (remote)	variable	N/A

Table 7.2: DSHcl Series 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 DSHpq series task set is a Distributed, Synchronized, and Harmonic task set designed to test for priority queueing 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 KWIPS to measure server workload. Table 7.3 gives the timing requirements of the DSHpq series baseline test task set.

Task	Workload (KWIPS)	Period (ms)
T1	1	160
T2	1	320
T3	2	640
T4	2	1280
T5	8	2560
Tserver (remote)	variable	N/A

Table 7.3: DSHpq Series Task Set

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.

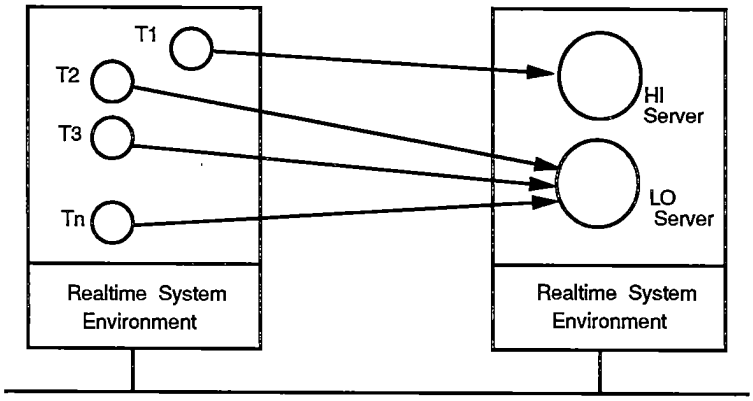

Figure 7.7: N Clients with Multiple Servers

Figure 7.6 shows the structure of the DSNpp series task set. Table 7.4 gives the timing requirements of the baseline test task set.

Task	Workload (KWIPS)	Period (ms)	Priority
T1	1	50	high
T2	1	5120	low
...			
Tn	1	5120	low
HI SERVER (remote)	1	N/A	high
LO SERVER (remote)	0	N/A	low

Table 7.4: DSNpp Series Task Set

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.

Figure 7.8 shows the structure of the DSHcb series task set. Table 7.5 gives the timing requirements of the baseline test task set.

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.

The benchmark results of the RIDE system over the WANDA kernel are presented in Table 7.6. The RIDE performance was measured using two Firefly machines connected by the 10 Mbit/Second Ethernet. Only one CVAX processor for each Firefly machine was used in the experiment.

Task	Workload (KWIPS)	Period (ms)	Priority
T1	0	80	high
T2	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 (remote)	0	N/A	N/A

Table 7.5: DSHcb Series Task Set

Series	RIDE	ARTS
DSHcl	26 ms	35 ms
DSHpq	16 KWIPS	18 KWIPS
DSNpp	18 tasks	(13) 20 tasks[1]
DSHcb	15 tasks	14 tasks

Note[1]: ARTS has two protocol engines, one achieves 13 tasks in this test, and the other (with prioritised workers, also see Section 8.3) achieves 20 tasks.

Table 7.6: RIDE vs ARTS Performance

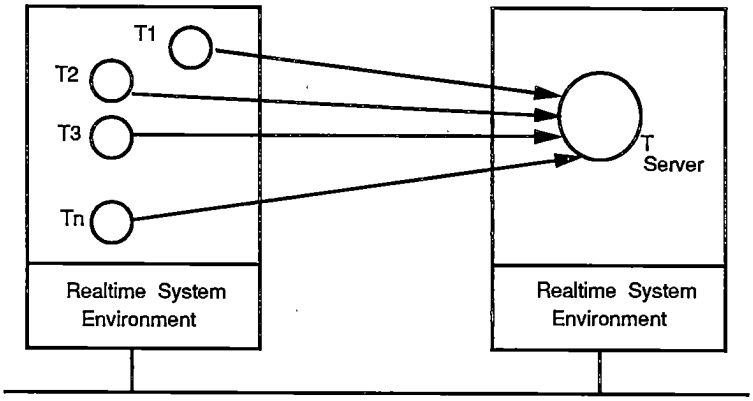

Figure 7.8: N Clients with Single Server

To make a comparison, the relevant performance of the ARTS distributed realtime operating system is also given in Table 7.6. The ARTS performance is copied from [78] which was measured by using SUN3/140s and a private 10 Mbit/Second Ethernet. 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 unmarshalling 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 queueing of communication packets, in the preemptability of the protocol engine, and in the provision of communication bandwidth.

7.5 Parallel Protocol Stack and Multiprocessor Speedup

Technical improvements in integration technology and optical media have led to very high bit rates at the physical network level. But the performance of communication protocol processing is not keeping pace with the present rate of improvement in network transmission bandwidth [77, 39]. Several options are available to increase protocol throughput. These include low-overhead protocols such as MSNL, specialized hardware architecture [47], parallelizing the execution of existing protocols, or off-loading protocol processing to auxiliary subsystems. These options overlap and can be employed together.

The RIDE parallel protocol stack architecture plus the real physical parallelism provided by the Firefly shared memory multiprocessors makes it possible to explore the parallel execution of the ANSA communication system. The parallel implementation includes the corresponding higher layers of the OSI protocol stack [21], i.e. the transport, session, presentation, and application layers. The overhead of these higher layers dominates all other processing in a communication system [16], therefore, the parallel execution is expected to be able to largely improve the protocol throughput.

The parallel execution of the communication stacks on a multiprocessor is a stringent test of the concurrency correctness of the protocol program. Although semantically the same as uniprocessor execution, subtle differences exist. The difficulty in parallelizing a set of protocols is not introducing parallelism, but is rather introducing synchronization and mutual exclusion where parallelism is not desirable (see Section 6.3). On a multiprocessor the execution stream of two threads may be interleaved at the sub-instructional level, adding much stronger race conditions than on a single processor.

7.5.1 Basic Performance

In order to determine the cost of implementing RIDE and to illustrate the benefit of the parallel protocol stack, the performance of the RIDE prototype implementation is compared against that of the original version of the Testbench on which it is based.

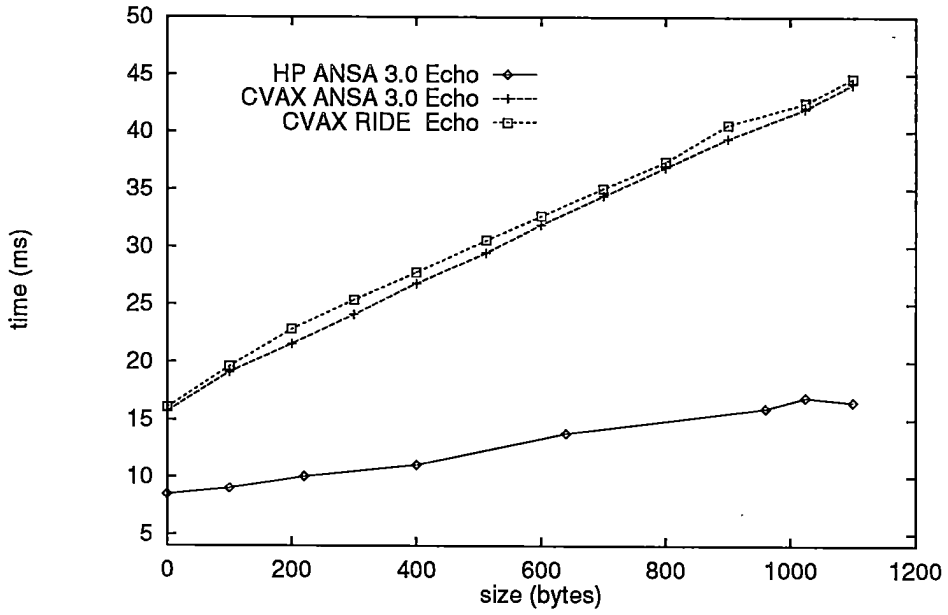

Figure 7.9: Basic RPC performance

All experiments were run between lightly loaded Firefly WANDA machines connected by the Computer Laboratory Ethernet. The results presented were obtained by averaging the results of multiple experiments, where each experiment consisted of 1000 or more invocations.

Figure 7.9 shows the performance of Testbench 3.0 on the Firefly WANDA, the performance of RIDE on the Firefly WANDA and the performance of Testbench 3.0 on the Hewlett Packard Series 9000/375 workstations running HP/UX (source [83]). All measurements shown are for an *echo* operation which sends and receives n bytes of data. The measurements on the Firefly WANDA were done by using a single task on a CVAX processor as a client and a single task on a CVAX processor as a server.

The performance of Testbench 3.0 on the Firefly WANDA compares poorly with the

same Testbench on the HP workstations. This can be explained by the fact that the HP workstation has a much more powerful processor which rates at approximately 11 VAX MIPS whereas a CVAX processor rates less than 2 VAX MIPS. The performance achieved on the Firefly machines is restricted by the relatively poor processor throughput.

The RIDE performance appears to be worse, by an average of 0.51 ms, in comparison with the Testbench 3.0 counterpart. This is the cost of the fine-grain synchronization and the use of task-based data structures.

7.5.2 Multiprocessor Speedup

The experiment shown in Figure 7.10 is designed to test the multiprocessor speedup of the RIDE RPC performance. Parallelism is achieved by associating each client/server task pair (by using the RIDE entry mechanism) with a separate network connection (by using the RIDE parallel protocol stack mechanism). On each side of a client/server pair[1] (such as T1 client and T1 server), a dedicated processor is used as the client/server processor. The separate connections are therefore processed by separate processors. Parallelism occurs when multiple clients are active concurrently.

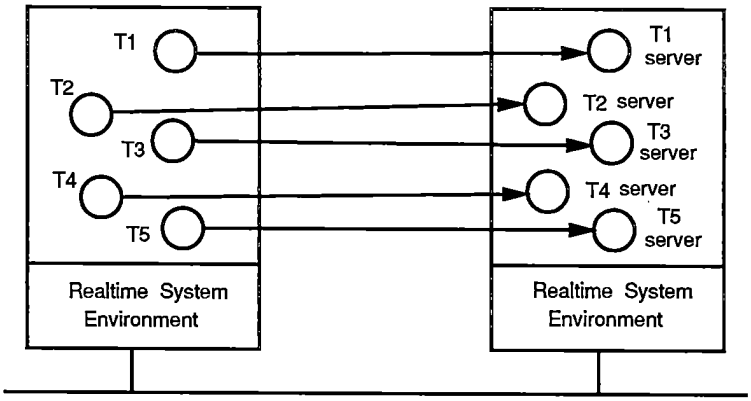

Figure 7.10: Experiment Setup of Multiprocessor Speedup

Figure 7.11 shows the results obtained using the connection-based RPC calls on one to four processors. For comparison, the result obtained using normal RPC calls (without explicit connection) on a single processor, is also given.

Under the single processor circumstance, an RPC call using a connection is improved by an average of 0.72 ms compared with that of not using a connection. This saving is achieved because, in the connection-based case the MPS module does not need to search into a table of cached network connections, as is the case with normal RPC calls when packets are being passed.

The exact multiprocessor speedup is shown in Table 7.7. This experiment shows that parallelisation of communication protocol processing can provide a significant performance improvement. The number of RPC calls processed is increased by about 80 percent when the second processor is used. This increase follows the same pattern when the third

[1]There are two tasks at a client side: the client task and its network reception task. There are also two tasks at a server side: the server task and its network reception task.

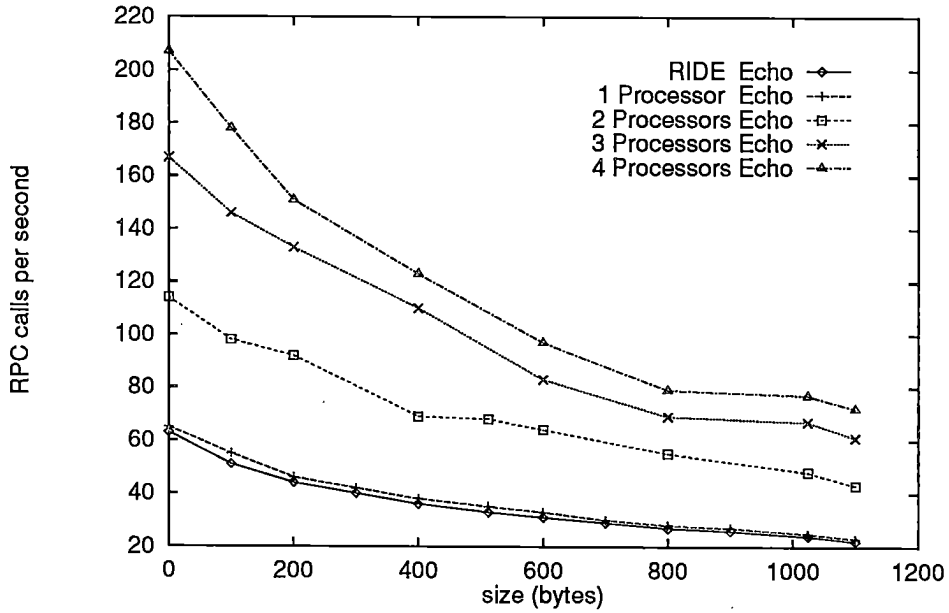

Figure 7.11: Multiprocessor Speedup

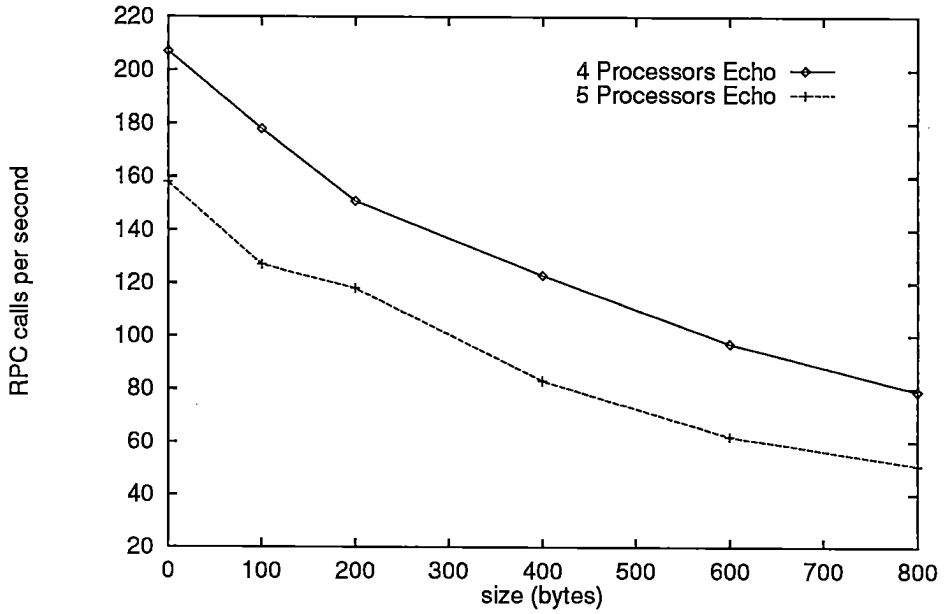

Figure 7.12: Adverse Effect of the Fifth Processor

Configuration	*Null RPC*		*Echo*			
			100 bytes		1024 bytes	
	Seconds for 1000 Calls	Speedup (%)	Seconds for 1000 Calls	Speedup (%)	Seconds for 1000 Calls	Speedup (%)
ANSA 3.0	15.7	N/A	19.10	N/A	42.00	N/A
RIDE	16.02	N/A	19.60	N/A	42.50	N/A
1 Processor	15.35	N/A	18.45	N/A	40.55	N/A
2 Processors	8.80	74.43	10.24	80.17	21.16	91.63
3 Processors	6.02	154.98	6.86	168.95	15.03	169.79
4 Processors	4.83	217.80	5.61	228.87	13.08	210.01

Table 7.7: Multiprocessor Speedup of RPC

processor is used. When the fourth processor is used, another 65 percent increase is achieved. It is not surprising that using four processors does not increase the performance by four times, but only by 3.2 times. This smaller relative improvement is likely to be the result of the contention for hardware resources such as bus bandwidth, memory cycles, and especially on the network interface; the contention on software resources is also a source of performance loss. This would become more of a problem as more processors are added to the shared bus, shared memory and shared network interface architectures.

A subtle adverse effect comes up when the fifth processor — the I/O processor with a Micro VAX II CPU — is used to test five processors speedup. Figure 7.12 shows the experiment results when using five processors in comparison with those of using four processors. Unexpectedly, the fifth processor does not bring any performance improvement, rather it degrades the performance by about 20 percent. The reason is likely to be the asymmetrical nature of the Firefly and the fine-grain synchronisation used by RIDE. Note that the raw speed of the fifth processor is one time less than that of the other processors (see Section 6.1 and all the interrupts (including those from the network interface) are processed by that processor. Therefore tasks on the processor are executed much more slowly than tasks on other processors. Because of the fine-grain synchronisation used by the parallel protocol stack, these slower tasks on the I/O processor unfortunately also slow down tasks on other processors (recall that all the communication tasks share some exclusive resources such as channel table etc.). This brings an overall performance degradation. The experiment also suggests that the fifth processor has became the bottleneck of the whole system. Performance improvement on the processor is vital to improve the overall system performance.

7.6 Summary

In this chapter, several aspects of the RIDE implementation are examined. The validity of the implementation is justified by comparing its performance with that of some typical systems, using the Hartstone and Distributed Hartstone performance benchmarks. In addition, the PIP and PCP semaphores are shown empirically to be a useful facility for controlling priority inversion. The parallel execution of the ANSA communication stack is shown to be able to largely improve the protocol throughput.

Chapter 8

Related Research

This chapter describes recent research on the design and implementation of realtime system environments. A brief examination of eight projects and systems and comparisons with RIDE are given. This places the work of this dissertation into context.

8.1 Ada 9X

Ada is a general purpose language with special applicability to realtime and embedded systems. Ada was originally developed by an international design team in response to requirements issued by the United States Department of Defense.

Ada 9X [8] is a revised version of Ada updating the 1983 ANSI Ada standard and its 1987 ISO standard equivalent in accordance with ANSI and ISO procedures. The major language enhancements provided by Ada 9X are in the areas of *programming in the large*, *object-orientation*, *realtime*, and *parallel programming*. Distribution is addressed under the requirements of programming in the large. A brief description of some of the Ada 9X proposals that are intended to address realtime [25] and distribution [24] requirements is given below.

Ada (and Ada 9X) is one of the few mainstream programming languages with support for multiple threads of control (tasks) incorporated into the language definition. Ada also incorporates support for high-level synchronization *between tasks*, based on its *task rendezvous* and *task entry* mechanisms.

8.1.1 Realtime in Ada 9X

Ada 9X addresses the need for asynchronous inter-task communication, and particularly the need for user control of scheduling, by providing a combination of primitive features that enable a user to program solutions to fit specific application requirements. Central among these new primitives is a data and task synchronization construct called a *Protected Record*. A protected record is a passive object of a protected type that exports a set of protected operations. A protected type is specified by a program unit that defines its private components and its protected operations. These are either non-queueing read-only

protected functions, read/write protected procedures, or potentially suspending entries (analogous to Ada task entries). Each protected record entry is associated with a Boolean expression called an *entry barrier*, which controls whether a task may suspend itself to wait for a specified condition.

Ada 9X provides a default priority scheduling model that refines and extends the Ada 83 tasking model. This model attempts to unify the treatment of priority across all the operations that affect processor scheduling. The most notable additions are: (1) an operation to change a task's priority; (2) a mechanism for a user to choose an entry queueing policy (such as priority queueing); and (3) the priority inheritance through protected records.

Each task is assumed to have both a base priority and an active priority. The active priority starts out equal to the base priority but may be raised when the task is executing a protected record. The priority scheduling model specifies that: (1) higher active priority tasks will be dispatched before lower active priority ones; (2) a rendezvous is executed with an active priority which is the maximum of the active priorities of the caller and callee; (3) task activation will execute at an active priority which is the maximum of the active priorities of the creator and created tasks.

Each protected record has a fixed ceiling priority associated with it. The ceiling priority specifies an upper bound on the active priority of any task that may call a protected operation of that protected record. While a task is executing a protected operation its active priority is raised to the ceiling priority of the protected record.

Ada 9X also addresses user needs for *time measurement* and periodic computation via a new standard time service package, called `Monotonic`, and extensions to the delay statement. The Monotonic package provides a time type, a clock function, and operations for comparison and arithmetic with monotonic time and a standard Duration type.

With the Ada 83 delay statement the only way to specify the expiration time is in *relative* terms:

```
delay 0.1;
```

The Ada 9X includes a new form of delay statement which specifies an absolute expiration time on a particular clock:

```
delay until Monotonic.Time'(T);
```

8.1.2 Distribution in Ada 9X

The Ada synchronous task-to-task communication ability inherent in the rendezvous has been adequate for building multitasking embedded systems. Unfortunately, this style of communication is by nature static, and is less useful in a dynamic distributed environment.

Distribution is addressed in Ada 9X by adding flexibility to library unit naming and to the definition of a *program*. The term *partition* (similar to an ANSA capsule in function) is introduced to represent the unit of a system that is loaded and elaborated at one time. Partitions are allowed to communicate with each other using shared packages and remote subprogram calls (with at-most-once semantics). Programs that may be used by remote access must be declared as shared passive packages or as remote call interface packages. RPC is accepted as the underlying communication facility and is considered as the only mechanism that could be standardized on. There are also suggestions [22] that Ada 9X

should explain how an implementation could set the priority of an RPC.

8.1.3 Ada 9X and RIDE

Ada 9X addresses realtime and distribution as two separate issues whereas RIDE addresses the two in an intergrated manner. It is not clear in Ada 9X, for example, how the priority scheduling model works in a distributed environment. Although protected records are designed to enhance realtime capabilities, they are not allowed in the visible part of a remote call interface specification.

8.2 Alpha

Alpha [17] is an experimental operating system kernel which extends the realtime domain to encompass distributed applications. Alpha's application targets are those realtime distributed systems that are asynchronous, dynamic, non-deterministic, and mission-critical.

Distribution is provided by a passive object model implemented by kernel mechanisms. Objects are passive abstract data types in which there may be any number of concurrently executing activities called *distributed threads.* A distributed thread is the locus of control moving among objects via operation invocations. It is a distributed computation entity which transparently spans physical nodes; this is contrary to conventional threads which are confined to a single address space.

Alpha's principal realtime strategy is to schedule all resources — both physical and logical — according to realtime constraints associated with distributed threads and using a *Benefit Accrual Model.* With this model, each distributed thread is associated with a time-value function which identifies the criticality of the thread as time varies. Because distributed threads may span across nodes, the system exhibits a uniform approach to resource scheduling.

The Alpha approach of systems research is contrary to the popular microkernel approach. It provides all functionality inside its kernel. For example, even transaction management, which is often considered as an application-level service, is provided as a kernel mechanism.

Alpha and RIDE share the same motivation to support mission-critical applications, but their design approaches and system architectures differ substantially.

8.3 ART

The Advanced Realtime Technology (ART) project [99] at Carnegie Mellon University has developed a testbed for distributed realtime systems. The testbed has a distributed realtime operating system called the ARTS kernel, a realtime toolset called the ARTS toolset, and an object-oriented distributed realtime programming language RTC++ [59] based on C++.

The ARTS kernel supports the explicit association of timing attributes with its threads (called *realtime threads*). Typical timing attributes are priority, deadline, period, and

worst case computation time. Realtime threads are supported by an *Integrated Time-Driven Scheduler*. Rate-monotic scheduling is used by the scheduler to provide *capacity preservations* for realtime threads. The scheduler adopts a policy/mechanism separation scheme to allow multiple scheduling policies to co-exist in the kernel. The priority inheritance protocol and priority ceiling protocol are implemented within its lock operations.

ARTS supports kernel-provided distributed objects. Object invocations can be both synchronous and asynchronous. An unusual aspect of the object invocation mechanism is that it allows each operation to be associated with a worst case execution time, called a *time fence* value, which is used by the called object to decide if it is possible to finish the operation before a required deadline. The protocol used for the checking is called *time fence protocol.*

The ARTS kernel has an unusual communication manager. It uses priority queues for incoming and outgoing messages, allowing each message to have a priority. Multiple prioritised workers (threads for communication protocol processing) are spawned; each handles messages at a priority level. Therefore, the communication protocol processing is preemptive. This gives high priority messages a better chance to access network interfaces and to be processed more promptly at their destination nodes.

The RTC++ objects are extensions to C++ objects. RTC++ extends C++ class specifications to have an *activity* part. This specifies how to associate a thread (or a group of threads) to an operation (or a group of operations) in the class. Priority inheritance is defined on object invocations.

8.3.1 ART and RIDE

RIDE has been influenced by ARTS in the design of its priority scheduling models. RIDE also shares a few common features with RTC++ in the handling of object invocations. However, the object models of the two systems differ substantially. RIDE 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. In contrast, RTC++ tasking resources have to be declared at specification time, limiting its usefulness in dynamic environments. On the priority management aspect, RIDE also allows priority ceiling with object invocations. On the communication aspect, RIDE has a timed RPC protocol which provides richer timing semantics than the simple time fence protocol of ARTS; RIDE also allows the preallocation of communication resources (channels) to interfaces whereas ARTS can only associate priority to messages.

8.4 CHAOS

CHAOS [42] [41] is a realtime system environment for concurrent, hierarchical, and adaptable object-based systems. It is designed and implemented on a multiprocessor system.

CHAOS implements a kernel-embedded object model. It proposes the notion of *decomposable* and *synthesizable* invocation primitives. Invocation functions are broken down into independent primitives, and different combinations of them provide different invocation

semantics. This provides an application programmer with the mechanisms by which he can customize the system to application-specific requirements of functionality and performance.

An unusual feature of the CHAOS system is its ability to accomplish dynamic behaviour adaptations automatically. A declarative language, named COLD, is used to describe the structure of CHAOS objects, the interaction patterns of a CHAOS application, and the initial values for various object and interaction attributes. Such pre-defined system behaviour is monitored by the CHAOS run-time system, and the behaviour history is stored in a database as feedback to an *adaptation controller*, which is the repository for all the adaptation algorithms. On the basis of the monitored behaviour of the application, the adaptation controller will select a particular adaptation to perform. The actions of the adaptation controller are designed to modify the behaviour of the executing application software so that it behaves predictably even when the conditions of operation differ from the conditions for which the application was designed.

Distribution is not addressed in CHAOS.

8.5 ESTEREL

The ESTEREL language [9] is a parallel imperative language which models *reactive systems.* A reactive system is one that reacts to inputs coming repeatedly from a controlled environment and produces outputs to that environment. The ESTEREL language is also a *synchronous* language which adopts the *synchrony* hypothesis — the outputs are absolutely synchronous with the inputs, so their computations *take a null time*, or the controlling processors are infinitely fast. ESTEREL has been found useful in modelling realtime controlling process because of its intrinsic determinism. Recent work [54] shows the use of ESTEREL as a script language in programming multimedia object synchronization.

ESTEREL offers the ability to specify:

- delays relative to some events;
- some calculus on delays;
- the occurrence of input events;
- the occurrence of output events relative to some conditions;
- parallel treatment inside an object;
- repetitive or cyclic behaviour;
- exceptions and their associated handles.

The execution of ESTEREL programs is supported by a special compiler which transforms ESTEREL modules into deterministic sequential automata. Another project [32] considers supporting ESTEREL in the ANSA architecture for distributed multimedia applications, but has to rely on the assumption that the ANSA Computational Model exhibits *synchrony.* ESTEREL does not yet easily support large distributed realtime systems where limited resources must be scheduled.

8.6 MARUTI

The MARUTI [35] is an experimental hard realtime, distributed and fault-tolerant operating system. The kernel supports objects as primitive entities, and provides a communication mechanism that allows transparent distribution. Fault tolerance is provided through replication and consistent-control mechanisms.

An unusual aspect of the MARUTI system is that it supports *guaranteed-service scheduling*, which means that given a job with a set of service requirements and time constraints, the system automatically verifies the schedulability of each component of the job with respect to the job's constraints and those of other jobs in the system.

The important contributions of the MARUTI system are that it has developed a set of notations, algorithms, and a framework so that complicated time-constrained resource requirements can be expressed, analysed and reserved within an object-based structure. The initial MARUTI implementation was carried out on top of a modified UNIX operating system. It seems that the system concepts still need to be verified by practical applications.

8.7 MARS

MARS [28] is a distributed realtime system architecture for process control applications. It is intended for use in industrial, hard realtime systems.

To achieve the determinism required by hard realtime systems, MARS adopts a strictly time-driven and periodic architecture, where all events are predetermined in a time-triggered manner. This is in contrast to the more popular event-triggered systems, where the system activities are initiated as a consequence of the occurrence of external or internal events.

The sequence of processing and communication steps between an observation of the environment and a response to the environment is called a *realtime transaction* in MARS. During the *design stage* the realtime transactions are refined into a sequence of task executions and message exchanges. In this stage, the task dependencies are analyzed and an execution time limit for each task is established, so that all transactions can be scheduled on the given hardware resources. The schedule is generated off-line for verification of the established execution time bounds.

Communication among tasks and components is realized by the exchange of periodic state-messages over the Ethernet. A *time division multiple access* protocol is used to provide a collision-free access to the Ethernet.

8.8 IMAC

The Integrated Multimedia Application Communication (IMAC) architecture [83] provides a framework which facilitates the construction of multimedia applications.

IMAC is based on the ANSA architecture and has been implemented as an extension to the ANSA Testbench. IMAC provides a mechanism for the specification of communication

oriented QoS on a per-operation basis. Interface operations may specify a set of QoS options with which they are prepared to be invoked. The QoS options are expressed as constraints on the underlying communication system. A method of mapping from application level QoS to communication level QoS is provided.

IMAC is complementary to the work detailed in this dissertation.

8.9 Summary

A great deal of effort has been applied to support realtime applications as displayed by the eight related projects and systems reviewed in this chapter. A common feature of all the systems presented is the provision of a purposely built environment. This is in contrast with RIDE which is intended to provide an open and standard-based architecture.

Chapter 9

Conclusion

This chapter firstly summarises the contributions of this dissertation, and then gives suggestions for future work.

9.1 Contributions

The goal of this work was to design and construct a distributed system environment for supporting realtime applications. The contributions of this dissertation towards that goal consist of the RIDE realtime programming model, the timed RPC protocol, the timed automata synchronisation facility and the empirical validation.

The realtime programming model provides a framework to facilitate the enforcement of stringent timing constraints found in distributed realtime applications. The model incorporates tasks and communication channels (the two most important resources in realtime 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 development of the timed RPC protocol contributes a capability to associate timing constraints with object invocations. Using the timed RPC protocol allows a programmer to express and enforce reasonable timing requirements (representing different tradeoffs between consistency and strictness) on a per-call basis.

The definition and infrastructure support of the timed automata provide a temporal synchronisation facility. The facility contributes to the understanding of temporal synchronisations in a distributed world. The temporal synchronisation problem can be found in many forms in many applications and the timed automata facility should be a useful tool for the development of general solutions to it.

The implementation and empirical evaluation of the RIDE system environment is both time and resource consuming. The approach taken in this work has been to construct as complete a prototype implementation as time and resources allowed and to use it to evaluate the feasibility of the design. An implementation helps to identify many of the low-level engineering details that are critical to the validation of high-level concepts. The

performance of the 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.

9.2 Future Work

The design and implementation of a realtime distributed system environment is an ambitious task both in its complexity and scope. Consequently, there are some weaknesses and conscious omissions in this work that need further investigation. Aspects of the design of the model, computational constructs, engineering constructs, and the implementation can be improved. The flexibility of the architectural concepts and run-time system should be demonstrated by trying them in different operating environments and applications. These limitations of the current work provide motivation for future work.

Model

A major weakness in the model is that it lacks support for intra system timing constraints (see Chapter 5). Allowing intra system timing constraints to be expressed in the computational constructs, as in RTC++ and RTC [104], would be a promising future addition to the RIDE realtime programming model.

Performance

The current run-time system implementation is still somewhat crude without a systematic investigation of performance. Performance is concerned with improving system speed without necessarily providing any guarantees. The ANSA performance issue is currently being addressed by [84], and partially by [89]. They should provide useful insights to improve the RIDE run-time system performance in the future.

Timed Path Expression

The timed automata temporal synchronisation facility does not scale well to large systems. This is an intrinsic drawback of the state machine model: state machines are low-level abstractions and applications have to deal with states. It is difficult, for example, to understand a state machine consisting of hundreds of states.

A natural extension is to develop a high-level script language for describing temporal synchronization, and using the timed automata as an implementation mechanism for the high-level language. The ANSA architecture has chosen *path expressions* [88] as its concurrency control mechanism. [88] has also developed a method for mapping path expressions to state machines. It should be promising work to extend path expressions to have constructs for the description of temporal synchronizations, and extend [88]'s work to map the resulting *timed path expressions* to timed state machines. The timed path expressions would be able to enforce both temporal synchronization and concurrency control requirements, and should be useful for many concurrent realtime applications.

Bibliography

[1] J E Allchin and M S Mc Kendry. Synchronization and Recovery of Actons. In *Proc. of Second Symp. on Priniciples of Distributed Computing*, August 1983. (p 19)

[2] J F Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983. (p 57)

[3] R Alur and T Henzinger. Logics and Models of Real-Time: A Survey. In *Proc. REX Workshop Real-Time: Theory and practice, LNCS 600.* Springer-Verlag, June 1991. (p 54)

[4] D P Anderson and R Kuivila. A System for Computer Music Performance. *ACM Computing Surveys*, 8(1):56–82, February 1990. (p 64)

[5] ANSA. *The ANSA Reference Manual.* Poseidon House, Castle Park, Cambridge, CB3 0RD, 1989. (p 3)

[6] A Attoui and M Schneider. An Object Oriented Model for Parallel and Reactive Systems. In *IEEE Real-Time Systems Symposium*, December 1991. (p 19)

[7] T P Baker and A Shaw. The Cyclic Executive Model and Ada. In *Proc. of Ninth IEEE Real-Time Systems Symp.* IEEE CS Press, 1988. (p 13)

[8] J Barnes. *Ada 9X Project Report, Introducing Ada 9X.* Office of the Under Secretary of Defense for Acquisition, US Department of Defense, February 1993. (p 97)

[9] A Benveniste and G Berry. The Synchronous Approach to Reactive and Real-Time Systems. Technical Report 1445, INRIA/IRISA Rennes, June 1991. (pp 55, 101)

[10] E Biagioni, E Copper, and R Sansom. Designing a Practical ATM LAN. *IEEE Network*, March 1993. (p 10)

[11] K P Birman and T A Joseph. Exploting Replication in Distributed Systems. In S Mullender, editor, *Distributed Systems*, pages 319–365. ACM Press, 1989. (p 9)

[12] A Birrell and B Nelson. Implementing Remote Procedure Calls. *ACM Computing Surveys*, 2(2):39–59, February 1984. (pp 9, 42)

[13] G S Blair and R Lea. The Impact of Distribution on the Object-Oriented Approach to Software Development. *IEE/BCS Software Engineering Journal*, 7(2), March 1992. (p 19)

[14] S Cheng and J A Stankovic. Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey. In J A Stankovic and K Ramamritham, editors, *Hard Real-Time Systems: Tutorial.* The Computer Society Press, 1988. (p 32)

[15] D Cheriton. The V Distributed System. *Communications of the ACM*, 31(3), March 1988. (p 5)

[16] D D Clark and D L Tennenhouse. Arhitectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM 90, Communication Architectures and Protocols*, pages 200–209, 1990. (p 92)

[17] R K Clark, E D Jensen, and F D Reynolds. An Architectural Overview of the Alpha Real-Time Distributed Kernel. In *Proc. of the USENIX Workshop on Microkernels and Other Kernel Architectures*, Seattle, USA, April 1992. (p 99)

[18] Digital Equipment Corporation. *DECSRC Firefly Documentation, Part 1*, November 1988. (p 69)

[19] H J Curnow and B A Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1), January 1976. (pp 84, 85)

[20] B Dasarathy. Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them. *IEEE Transactions on Software Engineering*, 11(1), January 1985. (p 55)

[21] J Day and H Zimmerman. The OSI Reference Model. *Proc. of IEEE*, September 1991. (p 92)

[22] P de Bondeli et al. Summary of the 6th International Workshop on Real-Time Ada Issues. In *Ada Letters*, March 1993. (p 98)

[23] M J Dixon. *System Support for Multi-Service Traffic.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 245, September 1991. (pp 3, 5, 15, 38, 73)

[24] Ada 9X Documents. *Ada 9X Project Report, Distributed Systems Annex.* Office of the Under Secretary of Defense for Acquisition, US Department of Defense, February 1993. (p 97)

[25] Ada 9X Documents. *Ada 9X Project Report, Real-Time Systems Annex.* Office of the Under Secretary of Defense for Acquisition, US Department of Defense, February 1993. (p 97)

[26] P Donohoe, R Shapiro, and N Weiderman. *Hartstone Benchmark Results and Analysis.* Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-90-TR-208, June 1990. (p 88)

[27] S J Mullender ed. *The Amoeba Distributed Operating System: Selected Papers 1984 - 1987.* CWI Tract No. 41, Amsterdam, Netherlands, 1987. (p 5)

[28] A Damm et al. The Real-Time Operating System of MARS. *Operating Systems Review*, 23(3), July 1989. (p 102)

[29] A P Black et al. Distributed and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 12(12), December 1986. (p 19)

[30] D L Black et al. Microkernel Operating System Architecture and Mach. *Journal of Information Processing*, 14(4), 1991. (p 5)

[31] G Coulson et al. Extensions to ANSA for Multimedia Computing. *Computer Networks and ISDN Systems*, 25:305–323, 1992. (p 12)

[32] K Moody et al. OPERA: Storage, Programming and Display of Multimedia Objects. Technical Report 294, University of Cambridge Computer Laboratory, April 1993. (p 9)

[33] L Hazard et al. *Notes on Architectural Support for Distributed Multimedia Applications.* CNET, France, Technical Report ESPRIT ISA Project Number 226, March 1991. (p 101)

[34] M Rozier et al. CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4):305–370, December 1988. (p 5)

[35] N C Hutchinson et al. RPC in the *x*-Kernel: Evaluating New Design Techniques. In *Symposium on Operating Systems Principles*, pages 91–101, 1989. (p 50)

[36] S T Levi et al. The MARUTI Hard Real-Time Operating System. *Operating Systems Review*, 23(3), July 1989. (p 102)

[37] S R Faulk and D L Parnas. On Synchronization in Hard-Real-Time Systems. *Communications of the ACM*, 31(3):274–287, March 1988. (pp 59, 60)

[38] Open Software Foundation. *Introduction to OSF DCE*, December 1991. (p 9)

[39] M Gien. Micro-kernel Architecture Key to Modern Operating systems Design. *UNIX REVIEW*, 8(11), November 1990. (p 5)

[40] M W Goldberge, G W Neufeld, and M R Ito. A Parallel Approach to OSI Connection-Oriented Protocols. In *Proc. of 3rd International IFIP WG6.1/6.4 Workshop on Protocols for High-Speed Networks*, Stockholm, Sweden, May 1992. (p 92)

[41] J B Goodenough and L Sha. The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High Priority Ada Tasks. In *Second ACM International Workshop on Real-Time Ada Issues*, Devon, UK, June 1988. (p 71)

[42] P Gopinath. *Programming and Execution of Object-Based, Parallel, Hard, Real-Time applications.* PhD thesis, Ohio State University, Department of Computer and Information science, 1988. (p 100)

[43] P Gopinath and K Schwan. CHAOS: Why One Cannot Have Only An Operating System for Real-Time Applications. *Operating Systems Review*, 23(3), July 1989. (p 100)

[44] P Guedes. *Use of Object-Oriented Technology in the Implementation of a Distributed Operating System.* Position Paper, Presented at the Workshop on Object Orientation in Operating Systems, OOPSLA/ECOOP'90 Conference, Ottawa, October 1990. (p 9)

[45] W A Halang and A D Stoyenko. *Constructing Predictable Real Time Systems.* Pluwer Academic Publishers, 1991. (p 12)

[46] K G Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 70, December 1984. (p 50)

[47] D Harel and A Pnueli. On the Development of Reactive Systems. In K R Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, 1985. (p 27)

[48] M Hayter and D McAuley. The Desk Area Network. *Operating Systems Review*, 25(4), October 1991. (pp 1, 92)

[49] A Herbert. Engineering Model: Conceptual Framework. Rc.282, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1991. (p 21)

[50] A Herbert. The Challenge of ODP. Technical Report 33, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993. Also Appeared as an Invited Paper for the Berlin ODP Conference, October 1991. (pp 5, 9)

[51] A Herbert. Distributing Objects. Technical Report 18, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993. (p 19)

[52] M Heuser. An Implementation of Real-Time Thread Synchronization. In *1990 Summer USENIX Technical Conference*, 1990. (p 70)

[53] D Hildebrand. An Architectural Overview of QNX. In *The Proc. of the Usenix Workshop on Micro-kernels and Other kernel Architectures*, Seattle, April 1992. (p 5)

[54] C Horn and S Krakowiak. Object Oriented Architecture for Distributed Office Systems. In *Proc. of ESPRIT Conference.* Amsterdam:North-Holland, 1987. (p 19)

[55] F Horn and J B Stefani. On Programming and Supporting Multimedia Object Synchronization. *The Computer Journal*, 36(1):4–18, 1993. (p 101)

[56] M Hubley. Distributed Open Environments. *BYTE*, November 1991. (p 9)

[57] J F Hurose, M Schwartz, and Y Yemini. Multi-Access Protocols and Time-Constrained Communication. *Computing Surveys*, 16(1), March 1984. (p 15)

[58] CCITT Study Group XVIII Draft Recommendation I.121. *Broadband Aspects of ISDN*, February 1988. (p 15)

[59] IEEE. *POSIX Std 1003.4 (Draft 13)*, September 1992. (pp 8, 14)

[60] Y Ishikawa, H Tokuda, and C W Mercer. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. In *OOPSLA/ECOOP'90*, Ottawa, October 1990. (pp 53, 99)

[61] ISO. *Basic Reference Model of Open Distributed Processing — Part2: Descriptive Model*, December 1992. (p 12)

[62] P W Jardetzky. *Network File Server Design for Continuous Media.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 268, August 1992. (p 3)

[63] D B Johnson and W Zwaenepoel. The Peregrine High-performance RPC System. *Software—Practice and Experience*, 23(2), February 1993. (p 10)

[64] J Jungok and T Suda. A Survey of Traffic Control Schems and Protocols in ATM Networks. *Proc. of the IEEE*, February 1991. (p 15)

[65] N I Kamenoff and N H Weiderman. Hartstone Distributed Benchmark: Requirements and Definitions. In *Proc. of Twelfth IEEE Real-Time Systems Symp.* IEEE CS Press, 1991. (p 84)

[66] K B Kenny and K J Lin. Building Flexible Real-Time Systems Using the Flex Language. *IEEE Computer*, May 1991. (pp 27, 53, 58)

[67] S Khanna, M Sebree, and J Zolnowsky. Realtime Scheduling in SunOS 5.0. In *1992 USENIX-Winter*, 1992. (p 71)

[68] R Lea and J Weightman. Supporting Object Oriented Languages in a Distributed Environment: The COOL Approach. In *Proc. of TOOLS USA'91*, Santa Barbara, July 1991. (p 9)

[69] I Lee and S B Davidson. A Performance Analysis of Timed Synchronous Communication Primitives. *IEEE Transactions on Computers*, 39(9):1117–1131, September 1990. (p 47)

[70] J P Lehockzy, L Sha, and Y Ding. The Rate Monotonic Scheduling Algorithm — Exact Charicterization and average-case Behavior. In *Proc. of Tenth IEEE Real-Time Systems Symp.* IEEE CS Press, 1989. (p 13)

[71] J P Lehockzy, L Sha, and J K Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *IEEE Real-Time Systems Symposium*, December 1987. (p 28)

[72] J K Lenstra, H G Kan, and P Bruchker. Complexity of Machine Scheduling Problems. In *Annals of Discrete Mathematics*, January 1977. (p 34)

[73] G Li. *A Real-Time Scheduler Server.* Internal Document, University of Cambridge Computer Laboratory, April 1992. (p 72)

[74] G Li. *Supporting Real-Time Distributed Computing.* Position Paper, Presented at the Fifth European Workshop on Dependable Computing — Dependability, Decentralization and Distribution, Lisbon, Portugal, February 1993. (p 28)

[75] G Li and J Bacon. Realtime Thread Synchronisation in a Microkernel. In *Proc. of IEE International Workshop on Systems Engineering for Real Time Applications*, Cirencester, England, September 1993. (p 71)

[76] C L Liu and J W Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, January 1973. (p 13)

[77] M L M Luttmer, H Ribbers, and P G Jansen. TUMULT-X: A Real-Time Executive. Technical Report INF-89-26, Department of Computer Science, University of Twente, 1989. (p 71)

[78] D R McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 186, September 1989. (pp 15, 68, 92)

[79] C W Mercer, Y Ishikawa, and H Tokuda. Distributed Hartstone: A Distributed Real-Time Benchmark Suite. In *International Conference on Distributed Computing Systems*, 1990. (pp 18, 81, 84, 92)

[80] F W Miller. Predictive Deadline Multi-Processing. *Operating Systems Review*, 24(4):52–62, October 1990. (p 34)

[81] A Moitra. Scheduling of Hard Real-Time Systems. In *Foundations of Software Technology and Theoretical Computer Science, LNCS 241*, pages 352–381. Springer-Verlag, 1986. (p 28)

[82] A K Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology, May 1983. (p 15)

[83] J M Moore. An n Job, One Machine Sequencing Algorithm for Minimize the Number of Late Jobs. *Management Science*, 15(1), 1968. (p 34)

[84] C Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 220, May 1991. (pp 16, 27, 38, 43, 93, 102)

[85] C Nicolaou. ANSA Phase III: Nucleus Redesign and Reimplementation Overview. Rc.357, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1992. (p 105)

[86] V Nirkhe, S Tripathi, and A Agrawala. Language Support for the Maruti Real-Time Systems. In *IEEE Real-Time Systems Symposium*, December 1990. (p 53)

[87] J D Northcutt. *Mechanisms for Reliable Distributed Real-Time operating Systems: The Alpha Kernel.* Orlando FL: Academic Press, 1987. (p 3)

[88] O Rees. The ANSA Computational Model. Technical Report 01, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, 1993. (p 20)

[89] O Rees. Using Path Expressions as Concurrency Guards. Technical Report 22, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, February 1993. (p 105)

[90] T Roscoe and S Crosby. MSRPC2 User Manual. Draft document, University of Cambridge Computer Laboratory, June 1993. (p 105)

[91] M Schiebe. The Origins of Real-Time Processing. In M Schiebe and S Pferer, editors, *Real-Time Systems Engineering and Applications*, pages 1–10. Pluwer Academic Publishers, 1992. (p 1)

[92] L Sha, J P Lehockzy, and R Rajkumar. Solutions for Some Practical Problems in Prioritizing Preemptive Scheduling. In *Proc. of Seventh IEEE Real-Time Systems Symp.* IEEE CS Press, 1986. (p 14)

[93] L Sha, R Rajkumar, and J P Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990. (p 14)

[94] C J Sreenan. *Synchronisation Services for Digital Continuous Media.* PhD thesis, University of Cambridge Computer Laboratory, Technical Report 292, October 1992. (p 16)

[95] J A Stancovic. Misconceptions about Real-Time Computing: A Serious Problem for the Next Generation. *IEEE Compputer*, 21(10), October 1988. (p 1)

[96] J K Strosnider and T E Marchok. Responsive, Deterministic IEEE 802.5 Token Ring Scheduling. *The Journal of Real-Time Systems*, 1:133–158, 1989. (p 15)

[97] B Taylor. Introducing Real Time Constraints into Requirements and High Level Design of Operating Systems. In *Proc. 1980 National Tele. Communication Conf.*, volume 1. Houston, TX, 1980. (p 55)

[98] D L Tennenhouse. Layered Multiplexing Considered Harmful. In *Protocols for High Speed Networks*, IFIP WG.1/6.4 Workshop, May 1989. (p 43)

[99] C Thacker, L Stewart, and E Satterthwaite. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988. (p 68)

[100] H Tokuda and C W Mercer. ARTS: A Distributed Real-Time Kernel. *Operating Systems Review*, 23(3), July 1989. (pp 7, 99)

[101] H Tokuda, T Nakajima, and P Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *USENIX 1990 Mach Workshop*, 1990. (pp 7, 71)

[102] M Vazirgiannis and C Mourlas. An Object-Oriented Model for Interactive Multimedia Presentations. *The Computer Journal*, 36(1):78–86, 1993. (p 64)

[103] R P Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984. (p 84)

[104] 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. (pp 70, 81, 84)

[105] V Wolfe, S Davidson, and I Lee. RTC: Language Support for Real-Time Concurrency. In *IEEE Real-Time Systems Symposium*, December 1991. (p 105)

[106] J Xu and D L Parnes. On Satisfying Timing Constraints in Hard-Real-Time Systems. *IEEE Transactions on Software Engineering*, 19(1), January 1993. (p 14)