Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE:	**Cambridge (0223) 323010**
INTERNATIONAL:	**+44 223 323010**
FAX:	**+44 223 359779**
E-MAIL:	**apm@ansa.co.uk**

ANSA Phase III

A Performance Framework

Francis Wai, Dave Otway, Nicola Howarth, Andrew Herbert

Abstract

This technical report introduces the ANSA workprogramme activities on real-time and performance management aspects of open, distributed systems. It is intended as an aid to reviewers of the workprogramme and to the ANSA core team as a statement of baseline and scope for work on real-time and performance management.

The document does not specify an architecture for real-time and performance management, nor does it review current real-time and performance management technology. The emphasis is on parameters, scope, and interaction and interference between timeliness, performance and other open, distributed system control functions.

The reader is assumed to have a working knowledge on the distinction between a computational viewpoint and an engineering viewpoint implied by the ISO Basic Reference Model for Open Distributed Processing.

APM.1051.00.05 **Draft** 11 October 1993

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

Distribution:
Supersedes:
Superseded by:

Copyright © 1993 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

A Performance Framework

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

A Performance Framework

Francis Wai, Dave Otway, Nicola Howarth, Andrew Herbert

APM.1051.00.05

11 October 1993

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

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

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

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1993 Architecture Projects Management Limited

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

Contents

5	**1**	**Overview**
5	1.1	Purpose and scope of document
5	1.2	Motivation and benefits
6	1.3	Scope
7	1.4	Timescales
7	1.5	How to read this document
8	1.6	Acknowledgement
9	**2**	**Introduction**
9	2.1	Real-time in ODP Systems
9	2.2	Architectural Principles
10	2.3	Quality of Service
11	2.4	Negotiation and Contract
12	2.4.1	Trading
12	2.4.2	Binding
12	2.4.3	Interaction
13	2.5	Resource Management
13	2.5.1	Functions
13	2.5.2	Allocation Strategies
13	2.5.3	Scheduling Strategies
13	2.5.4	Synchronisation Strategies
14	2.5.5	Federated, Cooperative scheduling
14	2.5.6	Resources Re-claimation
14	2.6	Automation, Tools and Abstractions
14	2.6.1	Objectives
17	**3**	**Concepts and Terminology**
17	3.1	Classification of Real-time Systems
17	3.1.1	Hard Real-Time Systems
17	3.1.2	Soft Real-Time Systems
17	3.1.3	Guaranteed Response
17	3.1.4	Best Effort
18	3.2	Scheduling algorithms
18	3.2.1	Rate Monotonic
18	3.2.2	Earliest Deadlines First
18	3.2.3	Least Laxity First
19	3.2.4	Integrated Time Driven
19	3.2.5	Heuristics
19	3.3	Real-time Synchronisation
19	3.3.1	Priority Inversion Problem
20	3.3.2	Priority Inheritance Protocol
20	3.3.3	Priority Ceiling Protocol
20	3.4	Reactive Systems and Synchronous Programming

21	3.4.1	Synchronous Model of Computation
22	3.5	Characteristics of Real-time Systems
22	3.5.1	Predictability
22	3.5.2	User Control
23	3.5.3	Timeliness
23	3.5.4	Mission Orientation
23	3.5.5	Performance
25	**4**	**Designing for Performance**
25	4.1	Enterprise issues
25	4.1.1	Policy
26	4.1.2	Scheduling
27	4.1.3	Monitoring
27	4.2	Information issues
27	4.3	Predictable performance
28	4.3.1	Timing uncertainties
28	4.3.2	Resource contention
29	4.3.3	Scheduling uncertainties
29	4.3.4	Interaction with dependability mechanisms
30	4.4	Responsive performance
30	4.4.1	Unpredictable events
30	4.4.2	Scheduling constraints
31	4.4.3	Scheduling guarantees
31	4.4.4	To pre-empt or not to pre-empt
32	4.4.5	Scheduling failures
32	4.4.6	Interaction with dependability mechanisms
32	4.5	Productive performance
32	4.5.1	Minimising latency
33	4.5.2	Liveness and fairness
33	4.5.3	Dependability mechanisms
33	4.6	Unspecified performance
34	4.7	Mixed performance
34	4.8	Design guidelines
34	4.8.1	Performance characteristics
34	4.8.2	Interaction with non real-time systems
34	4.8.3	System design approach
34	4.8.4	Protected scheduling guarantees
34	4.8.5	Timing failures
35	4.8.6	Predictable system resources
35	4.8.7	Synchronised clocks
35	4.8.8	Exclusion uncertainties
37	**5**	**Interdependencies**
37	5.1	Dependability
38	5.1.1	Replication
38	5.1.2	Fault-Tolerance
39	5.1.3	Persistence
39	5.1.4	Transactions
41	5.2	Federation
41	5.3	Monitoring and Management
42	5.3.1	Bottlenecks

42 5.3.2 Deadlocks
43 5.3.3 Statistics gathering
43 5.4 Transparency

45 6 Computational Abstractions
45 6.1 Deficiencies in the ANSA Computational Model
45 6.2 Objectives
46 6.3 Quality of Service
47 6.3.1 Streams
47 6.3.2 Conformance Checking
47 6.3.3 Contracts
48 6.4 Synchronous Constructs
48 6.5 Programming Languages
49 6.6 IDL and CORBA
49 6.6.1 CORBA IDL
49 6.6.2 CORBA C++ Language Binding

1 Overview

1.1 Purpose and scope of document

This technical report introduces the ANSA workprogramme activities on real-time and performance management aspects of open, distributed systems. It is intended as an aid to reviewers of the workprogramme and to the ANSA core team as statement of baseline and scope for work on real-time and performance management.

The objectives of the document are to

- outline the context and objectives for real-time and performance management in open distributed systems
- define the main concepts used in real-time and performance management
- provide a taxonomy for classification of real-time and performance management requirements
- identify the kinds of system components and programming abstractions required to meet these requirements
- state the design options available when building systems that have to satisfy performance guarantees.

The document does not specify an architecture for real-time and performance management, nor does it review current real-time and performance management technology. These are to be the subject of other reports.

The reader is assumed to have a working knowledge of the distinction between a computational viewpoint and an engineering viewpoint implied by the ISO Basic Reference Model for Open Distributed Processing.

1.2 Motivation and benefits

Open distributed processing is concerned with the use of commodity technology to build integrating applications that link together existing applications, databases, control systems and users. Integrating applications are built to respond to business goals by providing value added information services in a timely and accurate fashion or by automating some previously manual or undefined service function in the business.

Because these services are driven by business goals, which themselves often change rapidly in response to business forces, it is necessary to be able to design and deploy services quickly and cheaply. Since the focus is on integration, the platforms over which the service operates can be expected to be heterogeneous and to change during the lifetime of the service. These factors lead to pressure for integrating services to be implemented above standard interfaces which can be expected to persist over technological change, and for service creation and management to be automated to a significant degree.

Since the requirement for integration services comes from meeting business goals, the designer of the service must address the costs and benefits of providing the service. There are trade-offs between precision of results, timeliness of results, scale of access and so forth which need to be resolved against business requirements. These factors turn into performance goals for the service and for its supporting infrastructure; they will affect the structure of the service itself and the ability to deliver the service over alternative platforms.

Current reference models for open distributed processing, including ISO RM-ODP, OSI Management, OMG Object Management Architecture (OMA) and ANSA make no mention of performance or real-time issues.

Current standards for open distributed processing, such as the Open Software Foundation's Distributed Computing Environment (DCE) and the Object Management Groups Combined Object Request Broker Architecture (CORBA) do not address real-time or performance management requirements. As relatively new technologies, attention has focused entirely on functionality: efficiency, optimizations and meeting guarantees are not addressed.

Therefore the scope of the ANSA work on performance and real-time is to define the necessary framework and to investigate candidate technology for practical standardization.

The benefits of the work will be several:

- service providers will be able to specify and implement services that give performance and real-time guarantees, meeting their business goals
- except in the most demanding of situations, real time and performance management will cease to be special case, decreasing the costs and time to deployment for such systems open distributed processing technology will be able to support performance and real-time guarantees, increasing the range of applications and services it can support, increasing the market for the technology

1.3 Scope

The term performance is understood to cover the following:

- real-time: the completion of an activity can be specified to within a certain granularity of a clock from its start.
- timeliness: the delivery of results can be expected to be on time.
- deadlines: the beginning or the termination of an execution can be specified against a clock.

ODP systems are traditionally perceived to be slow because of latency incurred by the layers of management functions. Performance in ODP systems means the management of paths through the layers and which satisfy the application requirements.

The aim of the work is to enable the exploitation of the best of all technologies.

The statement of motivation and benefits generates the following issues for work on real time and performance management:

- statement of performance goals and structuring of a service to meet those goals (and links back to business goals)
- statement of performance requirements for each component of the service

- statement of the performance offered by the platform and network supporting the service (itself being a function of the components used to build the platform).

The ANSA work will focus on these issues on the context of the generic ANSA scenario [APM93], where the integrating service is seen as an application supporting a real-time dependable database of objects representing the state of an ongoing process, such as provision of a telecommunications service, passage of an assembly through a manufacturing system, or a document through an office process. The controlled process will put performance and real-time demands on the integrating application. To perform its function the integrating application may have to draw on external databases, take input and deliver output to user applications on personal workstations. It is therefore almost certain to be a mixed system i.e. one with boundaries between differing real-time and performance requirements.

1.4 Timescales

This space is deliberately left blank for the time being.

1.5 How to read this document

The rest of this document is organised as follows.

Chapter 2 identifies the role of the Quality of Service (QoS) in the performance control process. For integration and cooperation, negotiations of QoS, contracts and authorities are the essential concepts in a federated, heterogeneous environment. The support for QoS is part of the system resource management function. The separation of policy from mechanism is essential for an expressive and flexible management regime.

Chapter 3 gives an overview of the fundamental concepts associated with real-time and timeliness. Reactive systems and synchronous programming are discussed because of scheduling advantages they offer. A real-time system is characterised by system properties. Those properties are derived from objectives, some of which are elaborated here.

Chapter 4 looks at classifications of real-time systems. The classifications are discussed in isolation focusing on their design and programming. However, mixed performance systems are the overall goal. Some hints on design principles conclude this chapter.

Chapter 5 is a study of constraints between timeliness and system functions such as fault-tolerance, federation, monitoring and management, and transparency. It aims to identify areas where optimization, efficiency and meeting guarantees may or may not be extracted.

Chapter 6 is on computational abstractions for real-time programming. A proposal for extending the ANSA computational model is made. The roles of QoS contracts, streams and timed path expressions are put into context. Work on programming language is outlined in relation to CORBA IDL and C++.

1.6 Acknowledgement

Earlier versions of this report generated discussions, criticisms, and comments made by members of the ANSA team and colleagues at CNET. The authors wish to thank in particular, in alphabetical order, Nigel Edwards, Mike Eyre, David Iggulden, Guangxing Li, Chris Mayers, Ed Oskiewicz, Gomer Thomas and John Warne at ANSA and Laurent Hazard and Jean-Bernard Stefani at CNET.

2 Introduction

2.1 Real-time in ODP Systems

Real-time systems known hitherto are restrictive in that they are constructed for particular problem domains; mission oriented applications are one such area. This is largely due to cost, performance reasons.

The use of specialised technologies of one kind or another makes the solutions harder to adapt to new requirements. Can a flight control system for planes from one manufacturer be adopted for use on planes from another? For the same make of planes, can it be scaled up as well as down? Can it be linked up with ground radar tracking systems? And what are the costs?

New fundamental requirements arise few and far between. Most requirements either evolve with changes in business goals, e.g. banks merged, or get simplified or diverged as a result of better understanding of the nature of the problem. In other words, there are continuities and hence there are scope for adopting existing solutions to new circumstances. This is so provided that in the initial system design stage or at an upgrade stage, architectural factors such as scaling are taken into account.

The principles of Open Distributed Processing permit the construction of distributed system solutions with the capability to cope with heterogeneity, and to prepare to share, to communicate, to evolve, to federate etc. There are five projections in which an ODP system can be analysed: enterprise, information, computation, technology and engineering. The ability to scale up, for instance, may be desirable from an enterprise point of view but may require more engineering resources to be allocated. The projections permit a framework in which trade-offs are made.

Real-time systems are often closed systems. The benefits of ODP are well-recognised. The motivation here is to try to bring the ODP world and the real-time systems world together. The approach is to identify which ODP principles need modification and boundaries in the combined world where timeliness can be expected. A real-time ODP architecture will be beneficial to system builders, systems integrators in markets such as telecommunications, financial trading, office automation where openness is gradually becoming crucial to competitiveness.

2.2 Architectural Principles

The four principal characteristics of an ODP system are:

- *Heterogeneous*: Components of compatible functionalities always exist in a system for a variety of reasons such as cost, historical constraints, business policy etc. Their existence frees the system from being locked into a particular technology.

- *Federation*: Autonomy increases the need for de-centralisation and is intrinsic in a distributed system. Federation respects autonomy and is a viable basis on which common goals can be achieved.
- *Scaling*: Design decisions are made to enable a system to scale up or down. Telephone systems are good examples. However, once the scale is beyond a certain point, the system is no longer a single entity in a certain sense. For instance, the service it provides has to be continuous and cannot be restarted.
- *Evolution*: The first two capabilities imply that the system must have the ability to evolve: by accommodating technological changes, integrating new systems or amalgamating existing systems.

With respect to the generic scenario in 1.3, these characteristics mean,

- *Heterogeneous*: The integrating application uses new technologies to pull together legacy technologies.
- *Federation*: The integrating application will often span organizational boundaries e.g. in INA services.
- *Scaling*: The addition of users, database, sources/sinks increases the volume of traffic.
- *Evolution*: Change in business requirements, or in organizational boundaries implies changes in individual systems which must be handled smoothly and seamlessly by the integrating application.

A generic real-time ODP system is one where controlled performance can be tapped for use in a spectrum of integrating application scenarios. The delivery of controlled performance is the key feature, not outright performance.

An ODP system is dynamic: new technologies are introduced and must be integrated with the rest, components wear out and get replaced, components merge or split, new management functions get incorporated etc. In short, an ODP system cannot be perceived as a system with a specific performance characteristic. It is likely that it will have a range of performance characteristics. In this vein, it is appropriate to talk about boundaries against which guarantees can be made.

The timely availability of resources is vital to meeting real-time constraints. Resources belong to managers and their availability concerns non-computational issues such as ownership/access rights, security, accounting etc. Also, commitment to resource allocations is no guarantee to their availability for all sort of reasons, for instance network partition. Therefore, a resource management structure is necessary for the provision of guarantees.

The rest of this chapter is on scope and parameters in the areas of Quality of Service (QoS), QoS negotiations and contracts, resource management and automation, tools and abstractions.

2.3 Quality of Service

A system is composed of components of different performance and cost characteristics. System components are service users and service providers. Applications demand from the service providers certain functionalities and a

certain Quality of Service (QoS). The following are identified as attributes which can be negotiated as part of determining QoS.

- *Time*: e.g. what sort of delay or jitter bounds a service user can tolerate? What sort of deadline does a service user have?
- *Space*: e.g. what is the bandwidth requirement of an interaction with the service?
- *Cost*: e.g. what is the cost for not carrying out a service within time? What are the benefits of carrying out a service on time?
- *Criticality*: e.g. what is the consequence of a service failure to an external environment?
- *Dependability*: e.g. what is the probability of failure? What is the probability of survival against failures?
- *Precision*: e.g. what are the acceptable QoS trade-offs? For example, how many video frames need to be transmitted in a certain time frame in order to maintain a "coherent" picture at the receiving end?

It can be argued that for timeliness and performance, only the first item is relevant. This is an over simplistic view. The allocation of system resources to meet various QoSs simultaneously means that some requests may have to be deferred. For instance, the consequence of failing in life-critical system can be catastrophic with loss of lives. And this has to be avoided at all costs including the missing of deadlines. The use of shared resources therefore has inherently a real-time element and brings into play all of the other attributes.

2.4 Negotiation and Contract

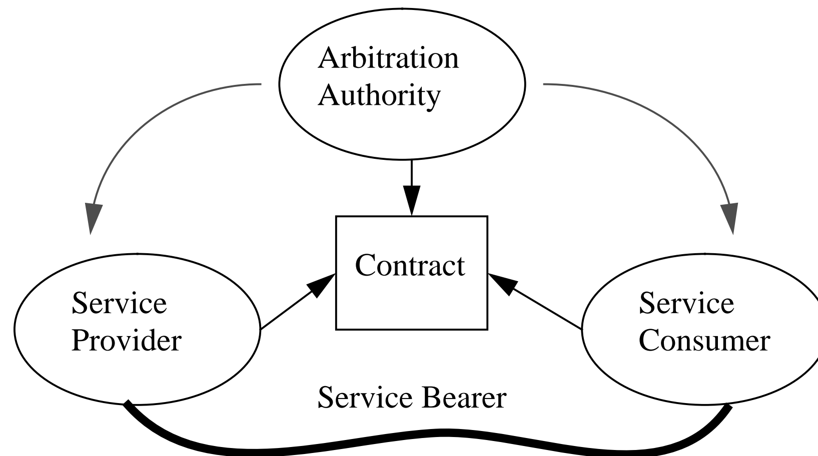
QoSs must be negotiated between service providers, service consumers and any supporting infrastructure (the service bearer). A contract is a binding set of agreements arising out of the negotiation. An arbitration (or billing or both) authority oversees the undertaking of the contract by all the parties concerned. Figure 2.1 illustrates the relationships among the service provider, service consumer, service bearer, the arbitration authority with respect to the contract.

In establishing a contract for QoS, the following kinds of guarantees can be negotiated:

- *Absolute*: The guarantee is delivered as demanded.
- *Pre-emptive, negotiable or voluntary*: The guarantee is delivered as requested but subject to withdrawal.
- *Time-variant, Average*: The guarantee is delivered at an average success rate over a certain time frame.
- *Conditional, Selective*: The guarantee is delivered upon the guarantee of some specific resources.
- *Atomic*: The guarantee is delivered all or nothing to every member of a group of service consumers or service providers.

ODP has already in place the concepts of trading, binding and interaction which facilitate service provision, service procurement and the detection of service failures.

Figure 2.1: Contractual relationships in QoS



2.4.1 Trading

Service providers advertise themselves through the trader. Service consumers look for available services that fit their requirements from the trader. A service provider provides a particular service and guarantees a certain quality of service. A service consumer requires a particular service and may require or be prepared to accept one or any provider within a range or on a list of quality of service.

There is no guarantee the required QoS can be met. The service consumer must accept all the termination conditions associated with the chosen provider.

2.4.2 Binding

The result of a trading process may be zero, one or more suitable service providers. The encapsulation of the consumer and that result is a binding. The consumer can expect a commitment to service from the providers in the binding.

A binding may be static or dynamic in the following sense. A static binding means the providers remain committed until the binding is explicitly destroyed. A dynamic binding means some provider may withdraw its commitment, or get substituted, or get replaced by a most-to-date version during the lifetime of the binding. To the service provider, a dynamic binding allows an exclusive service to be withdrawn if and when needed.

2.4.3 Interaction

Use of a service commences with an invocation from the consumer.

The provider expects to be invoked through a named operation in the advertised interface. It expects the types and number of parameters to conform to the advertised operations. If a particular sequence of invocations is expected, it is explicitly advertised and an internal mechanism in place to check the ordering and to raise a termination if necessary.

The service consumer expects the delivery of result to the quality of result specified. This may mean a certain precision and/or within a certain timing

constraint. It should be prepared to handle any of the advertised termination conditions.

2.5 Resource Management

2.5.1 Functions

QoS guarantees are achieved by the provisioning of adequate resources at appropriate times.

There are two management functions: resource allocation and resource scheduling. Resources are allocated to satisfy demands. Resource scheduling organises the availability of resources in keeping with the current scheduling policy.

In addition to explicit return of resources, resources can be temporarily re-assigned, whilst one activity is synchronised, awaiting another. Scheduling and synchronisation need to be carefully inter-related.

2.5.2 Allocation Strategies

There is a spectrum of strategies for resource allocation. At one end of the spectrum is so called “demand driven” where resource allocations are open to demand. An example is time-shared, multiple users systems. At the other end is “completely predictive” where resources are closed to demands and instead they are pre-allocated. Embedded mission-oriented systems are in this category.

Between the end points of this spectrum are degrees of “statistical predictability” where resource allocations are open to demands according to urgency or priority of the activities in hand.

2.5.3 Scheduling Strategies

The separation of policy and mechanisms means that any one of the following can be employed at any one time.

- *FIFO*: Resources are made available to activities that have wanted for them longest.
- *Pre-emptive, priority-based, deadline-based*: Resources are made available to the most critical or urgent activity or the one whose deadline is the nearest. An activity may have resources pre-empted to make them available to another.
- *Fair share:* Resources are quantized and allocated to all activities cyclically.
- *Round robin*: Resources are made available to activities which are ensured a chance to run until termination conditions arose.
- *Reservatio*n: Resources are partitioned and each partition is only available for scheduling particular activities.
- *Based on resource requirement*: Activities are scheduled according to the relative size of their resource needs.

2.5.4 Synchronisation Strategies

The following serves as a basis for synchronisation strategies for resources.

- *Shared*: No denial to access is possible.
- *Exclusive*: No access is granted, except the current owner.
- *Pre-emptive*: Access is open to all but can be closed at any instant.

With exclusive access mode, there is a phenomenon in priority-based systems known as “priority inversion” [SRL90]. This happens when a low priority activity impedes the progress of a high priority activity by acquiring exclusive access to a shared resource, and the low priority task is itself pre-empted by middle priority tasks.

2.5.5 Federated, Cooperative scheduling

Federation implies an agreement between parties on a common goal without compromising private, proprietary, independent knowledge. The scheduling of tasks in a federated system requires negotiation which might involve legal, contractual matters.

Cooperation implies the advancement towards a common goal based on shared information. The scheduling of tasks in a cooperative system requires exclusive mechanisms so that no task is scheduled more than once.

2.5.6 Resources Re-claimation

Resources may be re-claimed in one of the following ways:

- *Garbage collection*: Resources no longer referenced can be freed.
- *Negotiated*: Resources are re-allocated only when the owner agrees.
- *Non-re-claimable:* Once allocated, resources cannot be reclaimed until explicitly relinguished or reaching same termination event.
- *Time-out*: Resources are re-allocated when the owner fails to respond to polls conducted over a period of time.
- *Pre-emptive (with and without notification)*: Resources may be taken away and the owner may or may not be informed.

2.6 Automation, Tools and Abstractions

All of 2.5 is indicative requirements for a real-time “tool kit”.

Setting performance goals and managing resources is an intrinsic part of service and application design and programming.

The role of abstractions is important in this respect. They permit a user-oriented view: application programmers are given a language in which they can express QoS demands. They also permit automation: engineering details are hidden from programmers and tools are deployed to map the requirements onto the underlying engineering. Moreover, the mapping can take the same requirements onto different engineering substrates.

2.6.1 Objectives

The objectives of this automation and abstraction can be summarised as:

- to mask scheduling details from application programmers
- to minimise the opportunities for errors

- to minimise the amount of programming effort
- to enable scaling
- to support earliest possible QoS checking and hence scope for optimization

3 Concepts and Terminology

3.1 Classification of Real-time Systems

Following Kopetz and Verissimo (ch. 16, [Mul93]), a taxonomy of real-time systems is given below.

3.1.1 Hard Real-Time Systems

These are systems in which the timely delivery of results are critical and failure to deliver result on time can be catastrophic.

3.1.1.1 Fail safe

One or more safe states can be identified and accessed in case of a failure. The system can be halted from one of these states thus avoiding a catastrophe.

3.1.1.2 Fail operational

No safe state can be accessed in case of a failure. However, should a failure occur, a minimal level of service is guaranteed.

3.1.2 Soft Real-Time Systems

These are systems where the timely delivery of results are crucial but failure to meet deadlines can be tolerated.

3.1.2.1 Highly available

The total outage time is a small fraction of the operational lifetime of the system.

3.1.2.2 High integrity

The consistency of resources being managed is of utmost importance. Everything else being secondary.

3.1.3 Guaranteed Response

Systems are constructed based on the principle of resource adequacy. There is enough computing resources to cope with the specified peak load and the worst fault scenario.

3.1.4 Best Effort

Systems are constructed without consideration to resource adequacy. It is assumed that the provision of sufficient resources to cover all possible load and fault scenarios is not economically viable. Dynamic resource allocation based on resource sharing are acceptable.

3.2 Scheduling algorithms

A scheduling algorithm animates a set of rules. The rules determining which request must be processed at any particular moment.

Distributed scheduling is seen as a function over and above local scheduling. It is so because a certain degree of autonomy may have to be sacrificed and is beyond the scope of the current chapter.

3.2.1 Rate Monotonic

This algorithm is pre-emptive and priority driven. Once assigned, the priority associated with an activity is fixed and do not change over time. It is suitable for real-time systems such as process control and monitoring in which all significant activities exhibit periodicity. The characteristics of the activity set the algorithm can be found in [LL73]. The two principle assumptions are that all hard deadlines are periodic and there is no execution order among all periodic activities.

The highest priority is assigned to the activity with the shortest interval between requests to run. The next highest priority is assigned to one with the next shortest interval, and so on.

The algorithm is optimal in the sense that for a given activity set if a priority assignment exists, the rate monotonic priority assignment is feasible for that activity set.

Except in the trivial case, it can be shown that the processor utilisation cannot be 100%. On the other hand, the worst processor utilisation is known to be asymptotic to ln(2) for any arbitrarily large activity set.

3.2.2 Earliest Deadlines First

In deadline scheduling, the highest priority is assigned to the activity whose deadline is nearest to the current time. In contrast to the rate monotonic algorithm, this kind of scheduling is necessarily dynamic. At each scheduling decision point, the nearest deadline has to be re-established; the object is to arrange for the execution of an activity to be completed before its deadline expires. There is no processor idle time.

This algorithm is optimal in that if a schedule for a given activity set exists, it can be found under the deadline based algorithm.

The strength of this algorithm is in high processor utilisation. It makes no assumption on the periodicity of activities; periodic and aperiodic activities are treated the same. Under transient overloads, however, deadlines of periodic activities may be missed.

3.2.3 Least Laxity First

This algorithm is similar to the earliest-deadline-first algorithm in many respect. The difference is in the policy for priority assignment. In this algorithm, the highest priority is assigned to the activity with the shortest laxity. Laxity is defined to be the difference between the deadline and the execution time.

3.2.4 Integrated Time Driven

Hard, periodic activities and soft, aperiodic activities are catered for in this algorithm which was first reported in ARTS [TM89]. Rate monotonic scheduling is applied to the hard, periodic activity set. The processor utilisation is determined and the remaining processor time is then allocated to a so called Deferred Server. The latter server then selects to run an activity from the soft, aperiodic activity set using a value function which determines the semantics importance of the activities.

The use of value function which determines the criticality of the activity is to overcome the problem posed by transient overload phenomena. The use of the Deferred Server bounds the overall processor cycle time for the soft, aperiodic activities so that hard deadlines are not in danger of competing the same processor cycles.

In contrast to rate monotonic, the characteristics of the hard, periodic activity set are relaxed and more practical in that they are not required to be independent and their runability could be constrainted by shared resources e.g. semaphores.

3.2.5 Heuristics

In [GJ75], it was shown that if there is a partial order on the execution of activities in an activity set, even with only one shared resource, scheduling is known to be *NP*-complete. The only viable approach for scheduling is based on heuristic algorithms. In [ZRS87], a scheduling approach is proposed for meeting both timing and resource constraints. A schedule is considered to be made up of a sequence of slices. Each activity is assigned one or more slices and the object is so that each activity completes by its deadline. Assuming there are n independent system resources and that each activity requires at least one resource, there are at most n activities that can be allocated to one slice. Timing constraints take precedence over resource constraints in the scheduling process. For each slice, there is a primary activity and a number of secondary activities. The primary activity is scheduled by either earliest-deadline-first or least-laxity-first algorithm. The secondary activities are scheduled using a heuristic function. The motive here is to arrange all the suitable secondary activities to be executed in parallel with the primary activity.

A schedule is feasible if all the intermediate partial schedules are feasible. Backtracking is used where a partial schedule is not feasible.

3.3 Real-time Synchronisation

3.3.1 Priority Inversion Problem

The problem occurs when the progress of a high priority activity is blocked by a lower priority activity because they shared a critical region and the latter activity has acquired the semaphore first.

The blocking period is potentially unbounded. This is known as the *unbounded priority inversion* problem. This happens when intermediate priority activities which do not share the same critical region pre-empt the lower priority activity. Until all intermediate priority activities terminate, the

lower priority is not scheduled to run. The problem of unboundedness is that it causes deadlines to be missed.

In a client-server architecture, the same problem could occur. The scenario is that of a server which provides exclusive service to clients. What is the priority of the server? If it depends on the priority of the client requests, then high priority requests may be blocked indefinitely because a low priority request arrives first.

Two synchronisation protocols were reported in [SRL90] which tackle this kind of phenomenon.

3.3.2 Priority Inheritance Protocol

The priority of the low priority activity is raised by inheriting the highest priority of all blocked, high priority activities. The newly acquired priority is retained for the duration of the critical region. Upon exit from the region, the priority of the current activity is reverted back to normal.

3.3.3 Priority Ceiling Protocol

In this protocol, semaphores as well as activities are assigned priorities. A semaphore is assigned a priority which is the highest of all activities that may acquire it.

The object of this protocol is to avoid deadlocks and to eliminate chains of blocking periods.

Deadlocks arise because of lack of coordination among activities which acquire semaphores in arbitrary orders. Semaphores are in the same group if they are accessed in sequence by any one activity. Two activities can access the same semaphore group in any order. This protocol assigns the same priority (of the highest priority activity) to all semaphores in the same group. Deadlocks are avoided because once a semaphore is locked, no activities could possibly acquire a lock on any semaphores in the same group.

A high priority activity may suffer from a chain of blocking period if it is in a process of acquiring semaphores in some serial order and some semaphores had been acquired by other low priority activities. This protocol guarantees that an activity can be blocked for the duration of only one critical region.

3.4 Reactive Systems and Synchronous Programming

Reactive systems are ones that maintain a permanent interaction with their environment [BB91]. Examples of reactive systems include man-machine interfaces where typically the depression of a button immediately triggers an invocation.

Reactive systems are not necessarily real-time systems; they do not have a notion of time. But they can be made amenable to timing constraints defined by the environment they are interacting with.

Synchronous programming is one way of programming real-time reactive systems.

In contrast to synchronous programming, the traditional approach towards real-time programming as exemplified by Ada and Occam can be described as asynchronous. For instance, the flow of control may be interrupted between

adjacent program fragments. It is therefore impossible to guarantee temporal validity of any (real-) time dependent codes.

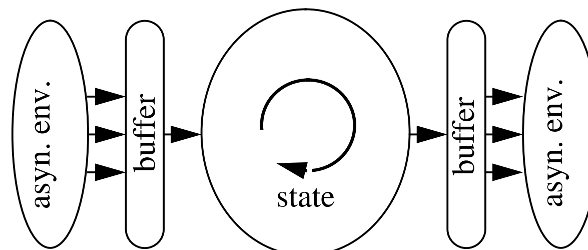
The foundation for synchronous programming is the hypothesis that it takes zero time for the system to response to external stimuli [BC84]. It assumes an infinitely fast execution machine. Such a machine does not exist in reality. But the model offers a conceptually simple programming framework. The following section describes the synchronous model of computation and elicits the scheduling advantages the model offers.

3.4.1 Synchronous Model of Computation

The synchronous model of computation is based on events. A synchronous program is seen as a finite state automaton. An automaton is defined by a set of transition rules and states. The rules specify the conditions for state transitions. In terms of events, an automaton waits for an input event at each state. Upon the reception of an input event, it undergoes a state transition and outputs an event. Since state transitions take zero time, the input event and the output event occur at the same instant!

An input event can be the reception of one or more signals from the environment. Similarly, an output event can be the transmission of one or more signals to the environment.

Figure 3.1: Interfacing with the asynchronous world



It is assumed that all input signals are available at the same instant and all output signals are consumed in a single instant. But the environment may be asynchronous. Fig. 3.1 illustrates one technique for interfacing with the asynchronous world. This technique was reported in [HHS93a].

Communication between automata is via signals.

Time is not given a primordial role. It is transmitted as a signal e.g. from a real clock. The use of signal in this way permits the so called multiform notion of time; “stop in 10 meters” and “stop in 10 seconds” are acceptable notion of time.

The transition rules indicate the causality relationship between input and output events. A causality graph of all input and output events can be constructed. The graph is an execution schedule.

For parallel automata, a global execution schedule can be obtained by serialising the causality graphs. The serialisation criteria are:

- dependency between signals,
- simultaneity of signals, and
- externally defined temporal constraints.

It should be noted that serialisation removes resource contention. All system resources are available for the execution of a state transition. And because synchronisation between communicating automata is completely deterministic, communication between co-located automata incurs zero cost.

It is axiomatic in synchronous programming that state transitions take zero time. This is unlikely to be true in practice. Given that all resources are available for the execution of state transition, uncertainties in resource allocation are eliminated from the execution machinery. It is therefore possible to calculate the execution time of each transition based on the timing characteristics of a model of execution machine.

Synchronous programming has, nevertheless, two weaknesses:

1. It is not a general purpose model of computation. It is useful only for a particular class of applications.
2. Synchronisation can only be local to a host. A serialisation cannot be obtained with dynamic activities -- an inherent feature in a distributed computing environment.

3.5 Characteristics of Real-time Systems

The following are excerpts from [Li93].

3.5.1 Predictability

Predictability is the tendency of the system to perform a set of operations in a well-defined, or determined fashion, and each operation's 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 non-deterministic system is one in which operation times have no guaranteed upper bound.

Predictability applies to every level of the components of a real-time distributed system environment. Such an environment must provide a certain degree of predictability, even though it is not always possible to be fully predictable, to support any useful performance guarantee.

3.5.2 User Control

This means a user has ultimate control of the behaviour of the system. This feature comes from the fact that many real-time applications are embedded systems (which are often static systems, and therefore it is possible to control the system's behaviour) and that real-time applications have immense behaviour diversity. Fixed system behaviour cannot cater for many real-time application requirements.

The simplest method of user control on system behaviour is probably the choice of priorities for real-time activities. By allowing a user to indicate the

relative priorities of activities, 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” approach. Users may also be allowed to select the scheduling policy, pre-allocation of system and application resources to critical services and so on.

3.5.3 Timeliness

Real-time 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 real-time system depends not only on the functional behaviour of the system, but also depends on the temporal behaviour as well. A real-time system environment must provide mechanisms which take these time related issues into account and must help application programs to meet these timing constraints. A simple example is to allow an application to associate deadlines with real-time activities, and the system employs a deadline based scheduling policy to ensure deadlines are met or to identify or cancel obsolete operations. Other required functions include the description and enforcement of temporal relations among related computational activities.

3.5.4 Mission Orientation

This 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 node. In the real-time sense, mission orientation also means **mission critical** -- the degree of mission success is strongly correlated with the extent to which the overall system can achieve the maximum dependability regarding real-time constraints. In its simplest form, mission orientation requires that a priority or deadline associated with a mission has global meaning when it spans 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.

3.5.5 Performance

Real-time applications often have stringent raw performance requirements. In this vein, the optimized integration of application software and its supporting environment is certainly desirable. This is in contrast with the popular layered design approach for non real-time applications. Also, real-time applications often require trading off modularity, flexibility and functionality to maximise performance.

4 Designing for Performance

This chapter reviews the issues that arise in designing and programming systems with defined performance characteristics.

The performance requirements of distributed systems will vary widely, but can be broadly separated into five categories which distinguish activities on the basis of whether they:

- have *predictable* performance under all circumstances
- are *responsive* to external events
- are as *productive* as possible
- have no specific performance requirements
- have a mixture of performance requirements

Most practical systems will contain activities with different performance requirements.

The first two sections 4.1 and 4.2 are a foray into performance categorisation and requirements from enterprise and information viewpoints. The following four sections (4.3 -- 4.6) then consider design and programming a particular performance category in isolation. Section 4.7 considers the complications caused by mixing activities from different performance categories. This mixture of requirements brings additional constraints and problems but is unavoidable in a federated system. Design guidelines which summarise this chapter are given in the final section 4.8.

4.1 Enterprise issues

The management principles embodied in the enterprise projection are expressed in terms of a direction-management-execution framework. In terms of policies, plans and resources, there are a number of interactions between entities in that framework. These are depicted in Fig. 4.1.

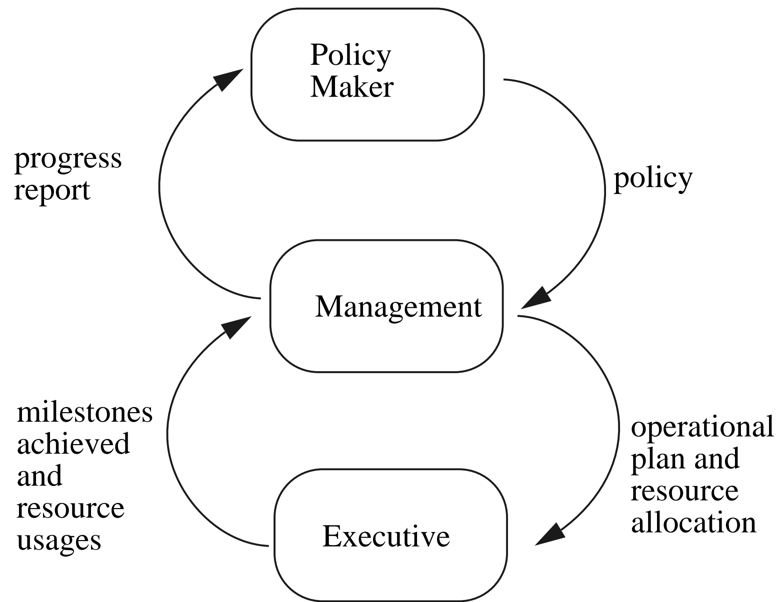
4.1.1 Policy

Real-time systems interact heavily with their environments. A typical environment is likely to be complex, dynamic, and evolving. In a system interacting with an environment of this type there is a great variety of activities, which need to be treated according to certain general classifications.

Elements of a system are classified according to their criticality to achieving the objective of the system. It is also important to identify the interference and interactions between them, what needs to be guaranteed in advance, and how to handle failures.

Timing requirements vary, and include hard deadlines, soft deadlines and periodic execution requirements, while some activities may have no specific timing requirements. Based on this there are three main types of activities:

Figure 4.1: Interactions in an enterprise model



- Critical activities are those that must meet their deadlines, otherwise a catastrophic result might occur. Such activities must be guaranteed beforehand to meet their deadlines subject to some number of failures.
- Essential activities are necessary to the operation of the system, have specific timing constraints, and will degrade the performance of the system if these constraints are not met. If they fail to complete on time, however, this will not cause a catastrophe.
- Unessential activities may or may not have timing constraints, and they execute when they do not affect critical or essential activities.

4.1.2 Scheduling

Once the category of each activity within the system has been determined, resources can be allocated according to requirements and the criticality of the relevant activity. For instance, a more important activity with a later deadline will be scheduled prior to a less important activity with an earlier deadline, if it is apparent that it is not possible for both activities to meet their deadlines.

Predictable systems require careful scheduling. A thorough assessment of the time-constrained, functional requirements of the process the system intended to control must be carried out.

Scheduling can only be carried out successfully when the boundaries of the system are known. If the boundaries are variable, then there is insufficient information to reliably predict use of resources. When an activity is added to a previous bounded, predictable system, then this activity will use up resources which may be required by the existing system. This will obviously affect the predictions made previously and affect the ability to meet deadlines. A federated system where load may be added uncontrollably places severe limitations on the ability to give guarantees as to the availability of resources. Similarly if elements of the federated system are not guaranteed to remain in operation, then predictability will suffer.

4.1.3 Monitoring

Objectives for monitoring are:

- Identification of bottlenecks, for avoidance or elimination with respect to the criticality of the activities in question.
- Prevention of deadlocks which might downgrade the performance of the system.
- Identification of availability of unused resources and free time, so that throughput can be maximised, within the timing constraints.
- Confirmation that activities are scheduled such that they meet their deadlines according to their criticality and importance.
- Notification of actual use of resources for accounting or statistical purposes.

4.2 Information issues

The primary requirement of a predictable system is that it must exhibit predictable behaviour. Predictable, reliable and timely operation rests on resource availability.

There are various problems involved in calculating resource availability. These include:

- How to express system state - both past, present and predicted?
- How to capture or enable the calculation of resource requirement in an application?
- How changes in system state affect an application’s ability to react (e.g. running out of CPU)?
- How to schedule changes in resource requirements: system reaction, and effects on other applications?
- What abstractions are required to enable applications to adjust to changes in resource availability/guarantees?

In distributed systems such behaviour prediction is exacerbated by the use of multiple nodes, and latency in the transmission of packets between nodes. Further complications are brought about by fault tolerance techniques, with various recovery mechanisms which are themselves subject to timing considerations.

4.3 Predictable performance

Time critical or hard real-time systems have predictable performance requirements. They are used for applications where timing failures are considered to be catastrophic.

Activities with predictable performance requirements are expensive in both design effort and execution resources. They also place severe constraints on the rest of the system.

In order to guarantee that the performance of a set of activities in a system is completely predictable, it is necessary to avoid all uncertainties in the execution timing, resource contention and scheduling of those activities.

4.3.1 Timing uncertainties

To avoid timing uncertainties in the execution of a predictable activity, the activity must have a bounded execution time.

This means that it cannot use programming constructs that are unbounded (such as while loops and recursion), or that it must use a fixed time algorithm which adjusts the amount of work done (and the precision of the result) to fit the time specified for its completion.

4.3.2 Resource contention

The maximum execution time of each activity can be calculated without reference to the needs of other activities; but in most situations it will need to share the resources it needs with other activities and its execution time bound must be protected from degradation by resource contention from other activities or be adjusted to compensate. This protection or adjustment of the execution bound must also be predictable.

The two basic mechanisms for dealing with resource contention are exclusion and segmentation.

4.3.2.1 Exclusion uncertainties

An activity may use an exclusion mechanism (such as a lock, semaphore or monitor) to prohibit other activities from concurrently accessing a resource or set of resources. An excluded activity must wait for access to a resource until it is released by the activity currently using it.

If exclusion mechanisms are to be prevented from introducing uncertainties into the execution time bound of predictable activities, they must also be bounded and the maximum exclusion time included in the maximum execution time of the activity.

In general, it is difficult to guarantee exclusion time bounds, especially if the resources are being shared with concurrent non-predictable activities.

4.3.2.2 Segmentation uncertainties

Resources may be segmented in the space and/or time domains and resource access for predictable activities scheduled in advance.

Occasionally segmentation may be completely in one domain (a processor dedicated to a single activity or each activity on a single processor system having access to all resources while it is being executed). Permanent spacial segmentation of resources is easy but not sufficient as it prevents sharing. Time segmentation is more difficult but is almost always necessary.

To segment resources in the time domain, the resource access requirements (in terms of time windows and exclusion bounds) of each predictable activity must be calculated and scheduled in advance to avoid contention with other activities. The exclusion of un-scheduled access to predictably scheduled resources is then the responsibility of the scheduler.

This pre-scheduling of resources used by predictable activities particularly applies to shared communication resources such as interrupts, buses and networks. These must be segmented on a time basis in the same way as processors.

Those predictable activities which have all their resource access pre-scheduled and which therefore don't dynamically contend for resources have no need of priorities relative to each other.

4.3.3 Scheduling uncertainties

To preserve the execution time bound of a predictable activity from degradation by the scheduler, it must either not be pre-emptable; or the number and duration of any pre-emptions must be bounded and the maximum pre-emption time included in the maximum execution time of the activity. Strictly, processor time must either be segmented and pre-scheduled or its exclusion time must be bounded; just like any other resource.

To remove scheduling uncertainties, all predictable activities must be pre-scheduled. As this can't be done for an infinite time line then all predictable activities must be periodic and they must be reserved a regular time segment even if they sometimes have no work to do when it arrives. Predictable activities can be scheduled in advance using whatever scheduling parameters are appropriate to the application.

The scheduling of predictable activities is therefore time-triggered rather than event-triggered and in a distributed system this requires synchronised time. Synchronising time around the whole planet is difficult to do (and manage) to the accuracy required to pre-schedule communications. Fortunately this is not necessary, all that is required is that time is synchronised on the set of nodes executing a set of pre-scheduled predictable activities.

This boundary around the scheduling and time synchronisation domains must be carefully defined and policed. Any change to, or federation across, this boundary is effectively a redesign of the application implemented by the set of predictable activities within the boundary.

It is not always possible to design predictable applications which can be exclusively time-triggered for all possible events. Certain events may so change the required future behaviour of the system that it is impossible to allow for this in a periodic schedule. Such traumatic events must be catered for by switching to a new (pre-planned) schedule at a pre-defined point in the existing schedule. This extends the notion of a pre-planned periodic schedule into a graph of such schedules which take account of all foreseeable traumatic events by switching schedules to cater for them.

Any resource segments not pre-scheduled for the servicing of predictable activities may be dynamically scheduled for non predictable activities provided that those activities can guarantee a bounded execution time (including bounded exclusion and pre-emption times) and that their scheduling guarantees can be temporary strengthened enough to protect it.

4.3.4 Interaction with dependability mechanisms

Predictable activities are invariably required to be dependable; although the reverse is not as common. This requires that the predictability mechanisms and the dependability mechanisms do not degrade each other when used in conjunction.

Predictability imposes the constraint on the dependability mechanisms that they must be bounded and pre-scheduled. This means that all agreement protocols (for ordering, exclusion, reformation, etc.) must be bounded and that recovery actions must be scheduled into the normal operational schedule; for

instance, a replica group could periodically multicast its state so that a new replica could acquire it just by listening. Dependability mechanisms not intended for use in predictable systems need not take account of these constraints.

As well as constraining the dependability mechanisms, predictability can help in fault detection. Because all communication is pre-scheduled then the non-appearance of a message at its prescribed time can be taken as evidence of failure; to this end, activities must always produce scheduled output even if it is null. Also if the schedule allows it, then some redundancy can be provided in the time domain as well as the space domain (performing calculations or transmitting messages more than once).

4.4 Responsive performance

Timely or soft real-time systems have responsive performance requirements. They are used for applications where timing failures must be avoided wherever possible. Timing failures in a responsive system are not catastrophic and alternative courses of action are available, even if undesirable.

4.4.1 Unpredictable events

Responsive activities are those that must react with their environment by responding to aperiodic events with sufficiently low latency. These activities cannot be pre-scheduled because the event arrival times are unpredictable. Therefore, the scheduling of responsive activities must be event-triggered.

The problem is to compute a schedule that guarantees the completion of all currently triggered activities within their respective scheduling constraints and to dynamically update this schedule as new unpredictable events cause extra activities to be scheduled.

Synchronised time is only needed for scheduling responsive activities if the scheduler is scheduling resources shared with other nodes.

4.4.2 Scheduling constraints

In order to compute and guarantee any schedule for responsive activities, each activity must have a bounded execution time. In addition, each activity will have a latest completion time (deadline) as this is what characterises its required responsiveness to the event.

Occasionally, a responsive activity will also have an earliest start time, reflecting a predictable time relationship between two events.

The scheduler must also take into account access to shared resources. As for predictable activities, this can be scheduled in the same way as processor time; or the exclusion time must be bounded and included in the maximum execution time when scheduling.

As for predictable activities, the responsive activities must be non pre-emptable (once started); or the maximum pre-emption time must be bounded and added to the maximum execution time when scheduling.

Finding a viable schedule that meets the deadlines of all currently triggered activities is computationally hard and heuristics may have to be used to give the best estimate in a reasonable scheduling time. If a viable schedule cannot

be found for all activities then the scheduler must drop some activities from its schedule.

It is not satisfactory for the scheduler to use any of its normal scheduling parameters (earliest start time, maximum execution time, deadline, arrival order) to decide which activity to drop from its schedule, because it has no way of knowing how this would effect the application.

Instead, the application must provide specific information, on which the scheduler can base its decision, by specifying a priority for each activity. The scheduler can then successively drop the least critical (lowest priority) activity until it can compute a viable schedule. Priorities are not used in computing a schedule; they only define what order activities will be dropped from the schedule.

4.4.3 Scheduling guarantees

Because the events that cause responsive activities are aperiodic, any guarantee that responsive activities can be scheduled is necessarily statistical.

However, the scheduler could provide a number of absolute guarantees to activities which have been scheduled or which have started executing:

1. scheduled activities could not be re-scheduled
2. scheduled activities could not be dropped
3. executing activities could not be pre-empted and re-scheduled
4. executing activities could not be pre-empted and dropped

Providing these guarantees is easy but reduces the responsiveness of the system. Not providing the guarantees to scheduled activities is not difficult, so it is essentially a matter of application requirements. But not providing the non pre-emption guarantees to executing activities considerably complicates the resource scheduling problems.

4.4.4 To pre-empt or not to pre-empt

This is the really difficult decision for designers of responsive systems. Both alternatives introduce undesirable complications.

If pre-emption is allowed then it becomes more difficult to bound execution times and calculate an accurate schedule; and these may have to be adjusted dynamically. But the responsiveness of the system can be improved if the problem of resource contention between the pre-empted and pre-empting activities can be avoided or solved. The contention problem can sometimes be avoided by segmentation in the space (but not the time) domain. If the contention problem cannot be avoided then a solution must be found to the problem of the lower priority pre-empted activity excluding the higher priority pre-empting activity from a shared resource.

If pre-emption is not allowed then the responsiveness of the system can only be improved by dividing the long running activities into smaller scheduling units. To protect the deadlines of activities at a particular priority p from lower priority activities then all activities with priorities less than p must have maximum execution times less than the priority p activity with the smallest scheduling slack (= relative deadline - maximum execution time).

4.4.5 Scheduling failures

An activity that has to be dropped from the schedule can be dealt with in a variety of ways depending on the application requirements; it can:

- be forcibly terminated immediately it is dropped from the schedule
- have its priority increased (and deadline extended) over time until it is scheduled
- have an alternative (shorter duration and/or higher priority) activity substituted for it
- be executed in any spare (otherwise unscheduled) time and forcibly terminated at its original deadline
- be executed in spare time and allowed to complete late

Executing dropped activities in spare time will also cause complications with pre-emption by or exclusion of higher priority activities which can subsequently be fitted into the schedule. Unless these problems can be avoided or bounded then activities which couldn't be reliably fitted into the current schedule cannot be allowed to execute in any spare time.

If a scheduled responsive activity finishes early then its unused time may be used to adjust the schedule to include previously dropped activities which weren't forcibly terminated on their initial failure to be scheduled.

4.4.6 Interaction with dependability mechanisms

Responsive activities may also be dependable. Responsive dependability mechanisms must be bounded but the recovery mechanisms do not have to be pre-scheduled.

4.5 Productive performance

Systems with productive performance requirements are those that strive to maximise their throughput. They do not have any critical timing problems, only those related to liveness and fairness. Their primary use of time is to help manage their workload and resources efficiently.

4.5.1 Minimising latency

Maximum throughput can be achieved when system load is at the minimum. A smaller overhead in resource management contributes towards a faster resource turn-around time. Further, the system must not be burdened with multitudes of application requirements. A tactically simple system is one which caters for fewer exceptions. And a tactically simple system is an efficient system. A number of measures can be taken towards that goal. These include:

- no reserved processor time
- no deadlines
- no synchronised time
- no priorities
- low activity switching cost

All except the last are policy issues. The last item is an engineering concern and it is important that the engineering technology does not itself incur high overheads to use.

In order to arrange for more activities to complete within the shortest time frame, the scheduling policy must be based on shortest-execution-time-first or least-resource-requirement-first. This requires known execution time bounds. They can be either be estimated or based on statistics gathered.

There are no notions of priority. Activities may not pre-empt others but they may be pre-empted by the scheduler. An activity may be pre-empted to make way for a newly arrived activity which has a shorter execution time. In addition, scheduled but unused time may be reclaimed for other productive activities.

4.5.2 Liveness and fairness

If the scheduling policy favours shortest execution time, then long activities may be neglected or get pre-empted more frequently than shorter activities. In the extreme case, there may not be a least upper bound on how long an activity is placed on hold. A productive system may not be a fair system.

An activity may require a resource which has been allocated. This activity is blocked and must be re-scheduled. In general, there are three problems in resource management which affect throughput; these are:

1. Deadlocks arise when two activities try to acquire resources which had been allocated to the other. In the extreme case, all activities are in deadly embraced where each is awaiting the others to relinguish their resources.
2. Livelocks arise when an activity tries to acquire a resource by testing repeatedly a release condition which never becomes true.
3. Starvation occurs when an activity requiring a large amount of resources is constantly being denied allocation of subsets of those resources because they are required by activities with shorter execution time.

Deadlocks and livelocks must be avoided, otherwise no progress can be made.

It is not un-reasonable to arrange shortest activities to complete first at the expense of starving long activities. However, the latter must be allowed to complete eventually.

4.5.3 Dependability mechanisms

Productive activities may be dependable. And there is no requirement that the execution time for dependability and recovery mechanisms to be bounded.

4.6 Unspecified performance

This category is really only here to allow it to be mixed with the three categories with specific performance requirements. Systems in this category have no effective control over performance. Some systems differentiate activities on the basis of priority, but do not use time as a scheduling parameter, offer scheduling guarantees or attempt to optimize performance.

4.7 Mixed performance

Systems with mixed performance requirements present the most severe problems but deliver the most functionality. The presence of activities with more severe performance requirements constrains the guarantees and optimizations that can be delivered to activities with less severe performance requirements. This category considers the problems of delivering performance in open federated systems.

4.8 Design guidelines

4.8.1 Performance characteristics

Open, distributed systems do not have specific performance characteristics.

Components of a distributed systems are heterogeneous and they get replaced or updated. Also, configurations and management functions get changed when the needs arise. The dynamism means that an open, distributed, federated system cannot be perceived to have specific performance characteristics. It is likely to have a range of performance characteristics.

4.8.2 Interaction with non real-time systems

Interactions between real-time and non real-time systems must be allowed.

Not all systems are designed or constructed to meet real-time constraints. In a real-time ODP world, interaction between real-time and non real-time systems must be allowed in order to avoid stratification which is a detriment to integration. However, the interaction must not be allowed to invalidate any guarantees.

4.8.3 System design approach

Predictable systems are by design.

Most existing systems are constructed so that resources are allocated on a fair basis. Predictable systems require guaranteed resource allocations and the protection of those guarantees.

In order to guarantee that the performance of a set of activities in a system is completely predictable, it is necessary to avoid all un-certainties in the execution timing, resource contention and scheduling of those activities.

4.8.4 Protected scheduling guarantees

Scheduling guarantees of predictable systems must be protected.

The execution time bound of an activity in a predictable system must be protected from degradation by resource contention from other activities or be adjusted to compensate. This protection of adjustment of the execution bound must also be protected.

4.8.5 Timing failures

Timing failures cannot be avoided.

Although scheduling guarantees for predictable systems are protected, there can be no guarantee against failures such as system or network failures. Such failures will cause a deadline to be missed. To avoid catastrophe, the system should be either fail safe or fail operational.

4.8.6 Predictable system resources

Resources and latent mechanisms for predictable systems must be predictable.

For predictable systems, resources are pre-allocated and pre-scheduled to ensure deadlines are not missed. Pre-scheduling requires predictability of all resources and mechanisms concerned. Predictability requirements apply to the whole range of reusable system resources, from CPUs, I/O devices, communication interfaces to system codes and reusable libraries as well as latent mechanisms (e.g. dependability mechanisms) that sit between the application and the hardware. Language constructs such as recursion and while-loops mean potential unbounded execution time.

4.8.7 Synchronised clocks

There is no global clock but there are synchronised clocks.

The scheduling of predictable activities is time-triggered rather than event-triggered and in a distributed system this requires synchronised time. The maintenance of a global clock incurs high overheads. But a global clock is not necessary. All that is required is that time is synchronised on the set of nodes executing a set of pre-scheduled predictable activities.

4.8.8 Exclusion uncertainties

Uncertainties in exclusion mechanisms must be scoped.

If exclusion mechanisms are to be prevented from introducing uncertainties into the execution time bound of predictable activities, they must be bounded and the maximum exclusion time included in the maximum execution time of the activity.

5 Interdependencies

Traditionally, the result of an execution can be qualified by the two attributes: completeness, and correctness. It is also possible to qualify by any discrete point within this spectrum, representing trade-offs between the two.

With a notion of time, a third qualifier, viz. timeliness, becomes apparent. The description of a result can be any point in the space delimited by the three axes: timeliness, completeness and correctness. It is possible to talk about a range of results including:

1. a correct result irrespective of timeliness and completeness, or
2. a complete result irrespective of timeliness and correctness, or
3. a timely result that is neither correct nor complete, or
4. a correct, complete result that is not delivered on time, or
5. a complete, timely but inaccurate result, or

...

6. results that are not exactly timely, nor correct nor complete.

This is a useful range since, for instance, a timely, inaccurate result can be more useful than an accurate but slow result. A timely system therefore can be more appropriate in meeting cost/criticality requirements. Applications such as mission-oriented, life-critical, or real-time transaction processing etc. all have a time-cost dimension in their problem domains.

Further, since timeliness is an attribute of dependability, there are obvious overlapping concerns. The failure model in [Edw93], for example, discussed expectations in terms of regions in a value-time space. If a value falls inside one of these regions, then that value is judged to be correct. If a value falls outside of any regions, then it is judged to be a failure. The proximity of a failure to the regions indicates more precisely the nature of that failure. The trichotomy described here complements that value-time space.

Timeliness and failure modelling are one area where real-time system control interplays with the dependability function. There are other system function areas. This chapter discusses structural relations between desirable real-time system control attributes viz. predictability, performance, timeliness, user control and mission-oriented (see 3.4) and functions such as fault-tolerance, federation, monitoring and management, and transparency.

5.1 Dependability

Techniques for dependability include:

- Replication -- provision of state
- Fault-tolerance -- provision of redundant resources
- Persistence -- preservation of state

- Transactions -- isolation of actions on state

5.1.1 Replication

Replicas form a group and may or may not be transparent outside of the group [OE93]. In order to behave in unison, their states must be in synchrony. On a distributed system where message delays and losses are inherent, consistency of state cannot always be guaranteed at any given point in time. The state of a subset of the replicas may diverge from the rest while another subset may converge.

The uncertainty does not necessarily represent a problem as the response of the group can be determined based on approaches such as leader-follower (cf. Delta-4 [Pow91]) or by reaching a majority verdict. With real-time constraints, there can be two types of responses. These are:

- responsive responses
- accurate responses

The choice of response depends on its criticality to the application at any given moment.

A notion such as temporal accuracy discussed in [KK91] is necessary in order to capture the trade-offs between the two. Applications would be allowed to expect or to offer the QoS required.

5.1.2 Fault-Tolerance

System resilience requires safeguard against failures. One such safeguard is redundancy[1] that is built-in at design time or dynamically engaged when the system is up and running. Redundancy are transparent and can be called upon should an error occur.

The mobilisation of redundancy relies on mechanisms for the detection of error and mechanisms for switching in the redundancy.

The detection of error over a distributed system requires communication and processing to be scheduled and, unless carefully engineered to overlap, may degrade the performance of the systems concerned.

Mechanisms for switching in redundancy may require alteration of existing schedules and initialisation of state which the redundancy can pick up where the failed component had left off.

Alteration to schedules can only take place at scheduling points. If the system is to be fail-safe or fail-operational, the scheduling of the redundancy must take precedence over all the rest.

In addition to the scheduling of redundancy, initialisation of state requires resources to be allocated. In a distributed environment, this may take place on a system which is different from where the decision was made. This will take time and may therefore have un-wanted effect on predictability and responsiveness.

In the design of a system with redundancy, the cost of detection and switching-in should be minimised. And in meeting a deadline, the scheduler should take into account the start-up costs in addition to the execution time bound of the activity in hand. In effect, the scheduler is scheduling redundancy for the

1. This can be hardware, software modules or even databases.

activity in question. This does not necessarily disadvantage other activities which have different system failure tolerance because the un-used scheduling slots can be re-deployed. It does, however, take away some freedom in the choice of size of scheduling slot.

5.1.3 Persistence

Persistence [ABC+83] [Ole92] is the concept whereby the lifetime of a data object is guaranteed beyond that of its creator. The availability of persistent data is only constrainted by synchronisation. A ramification of persistence is a one-level store. The where-about of data is not a concern to the programmer. The benefit is that the programmer is able to concentrate on the problem domain and not distracted by the properties of the local file system.

Another ramification arising out of generalising the concept over a networked environment is a network-wide integrated one-level store. The benefit reinforces the ODP principle of location transparency.

Intrinsic in persistency is the ability to hide representational difference on disk and in memory. This is usually achieved by automated flattening of structured data from memory to disk and re-constituting data from disk to memory format. The automation incurs a hidden translation cost. The cost is proportional to the structure of the data in question and is therefore not bounded.

Some persistent mechanisms support dynamic linking and loading. Only data that are needed are retrieved. Memory usage cannot be reliably predicted. Together with translation, they bring space and time uncertainties into the system.

The state of an object need not be made up of embedded data but may contain references to other objects. Some of those references may be for remote data. De-referencing triggers data to be brought into memory. However, due to migration, there is no guarantee that all data referenced can be located at the local site. The location service is required and a cost incurred.

The loading of remote data raises the question: where do the data go? There is no easy answer since there are security and federation implications. But from a pure technical point of view, there are two solutions.

1. The data could be loaded into the memory space of the site where it resides.
2. It could be loaded into the host where the de-referencing occurs.

Solution (1) introduces overheads into the local host which cannot be anticipated and may downgrade the performance of that host.

Solution (2) introduces overheads into the communication machinery which is shared by many.

5.1.4 Transactions

Transactions are atomic actions on object states [War93]. Atomicity requires resources to be secured before actions on states can be carried out. It then further requires a decision to commit or abort to be made before the states and resources can be released. In this respect, distributed, nested transactions have two implications.

1. In the absence of a coherent resource allocation or scheduling strategy, they can introduce deadlocks over the network.

2. The time it takes to reach a consensus to commit or abort is not guaranteed to be bounded.

These implications have further consequences on two classes of transactions: transactions with real-time constraints, and responsive transactions.

In a predictable system where resources are pre-allocated and pre-scheduled, real-time transactions are serialised with respect to their timing requirements. Actions on states can be carried out immediately without the need to securing resources[1].

With this arrangement, there can be no deadlocks possible on the local resources. However, invocation deadlocks may still occur in the absence of a coordinated scheduling regime[2].

This affects not only transactions, but in general a group of physically separated but interacting activities. A coordinated scheduling effort is therefore a pre-requisite for a predictable system that real-time cooperative activities. The construction of plausible schemes is not inconceivable given that scheduling algorithms for multiple machines exist [Gun84]. But the challenge lies, not in the construction of a new one from scratch, but on the control of a diversity of existing scheduling regimes, without compromising autonomy.

A decision to commit requires consensus on the readiness of all cohorts. Messages need to be exchanged before a decision can be taken. The number of messages required is proportional to the number of cohorts. The delivery of these messages takes time. And there is also the prospect of network and node failures.

The communication protocol charged with the delivery of messages must facilitate a decision to commit or abort to be made within the timing constraints. This, in turn, places a requirement on the schedulers concerned to schedule the protocol handlers. The efficiency of the communication protocol and the timely scheduling of protocol handlers are not sufficient to guarantee that timing constraints can be met. There is a commit protocol, often a two-phase commit protocol [GR92], which coordinates the commit on behalf of a transaction and its cohorts. In order that timing constraints are met, it is imperative for the individual schedulers to know, either statically or before runtime, bounded execution time of the commit protocol. Such requirement may place a restriction on the class of two-phase commit protocols for the purpose and one presumed-commit protocol [LL93] promises a logging and message cost lower that of presumed-abort.

In addition to determine whether to commit or abort, recovery actions are necessary if a decision cannot be made in the time scale required or an explicit abort is made. Techniques for recovery are diverse and include forward recovery, backward recovery, shadow paging [Cha78] etc. Each incurs a cost which the system has to take account of.

Among the three techniques, shadow paging may appear to be the cheapest as this involves the (atomic) toggling of few entries in the page table. The true

1. Higher level abstractions such as critical regions, semaphores etc. are not required.

2. Suppose two systems maintain two different schedules: A and B. Activity 1.A on schedule A may wish to transact with activity 2.B on schedule B and activity 1.B, which is ahead of 2.B in the schedule wishes to transact with activity 2.A on schedule A. None of these activities can proceed.

cost, however, is on memory management in that shadow pages have to be created and maintained. The number of shadow pages required may vary from one run of the transaction to the next depending on the execution path it takes through the code. Thus this introduces memory uncertainties into the system.

Recovery actions based on logging, whether for forward recovery or backward recovery, requires scheduling slots and perhaps resources extra to the transaction that is being recovered. Recovery actions may not be required to be carried out immediately. They can be scheduled later. However, they must be scheduled before the activities that depend on the original state are scheduled to run. Precedence relations between schedulable activities must be established. This requires computation to be carried out from time to time for a dynamic set of activities.

5.2 Federation

Federation concerns organisational agreement between two systems seeking cooperation in such a way that autonomy is not sacrified and homogeneity is not a pre-requisite.

Components and resources in a heterogeneous world have a wide range of performance characteristics. It is conceivable that there are federations that are functionally equivalent but with different performance characteristics. All or some of these federations can meet the real-time constraints in question. The choice of who-do-you-federated-with will depend on factors such as range of tolerance of those constraints, costs of the federation demands and benefits of the federation offers. Performance trading would be a useful device in resolving some of the issues.

Performance trading permits the formulation of a statement of obligation on the federation to meet a particular real-time constraint. The means to achieve this lies with resource allocation and their scheduling which are the dynamic aspect of the federation and are a local responsibility. In meeting the objectives,

- resource allocation may have to be integrated to prevent unexpected gaps in the chain of events that may span from node to node, and
- scheduling may have to be coordinated so that different priority regimes can be aligned, and
- the priorities of interacting activities may be allowed to adjust due to, for example, priority inheritance.

5.3 Monitoring and Management

A management structure is for interaction between a manager and the managed to take place in order to achieve a well-defined objective. Most objectives resolve into sub-objectives. A management structure should have the facilities to accommodate a range of objectives such as accounting, resource utilisation, performance analysis etc. A good management structure must be flexible as well as open since objectives get refined, formulated and introduced over the course of time.

Aspects of the managed need to be monitored to ensure, for example, activities are being initiated, coordination and cooperation are in order and problem areas spotted. The role of a monitor is for information gathering, information

to be disseminated and for events to be generated. A monitor may carry out specific functions to suit the need of the management.

Monitors do not change the way the managed behave but they provide the means whereby there is a cyclic flow of information among the managed, the manager and the decision maker.

A monitor and management structure is important for a large-scale networked system. However, it cannot be assumed that such a monitoring structure is separate from the system it is managing.

Consequently, it consumes resources and scheduling cycle, just like any other system functions. The costs of running and maintenance of such function should be kept small enough in relation to others the system normally runs to reap its benefits.

The management objectives for a high performance, timely system include:

- the identification of bottlenecks, and for them to be avoided or eliminated,
- full utilisation of resources at all time,
- the prevention of deadlocks from downgrading the performance of the system.

The full utilisation of resources at all time requires careful planning, budgeting and scheduling, and may benefit from user involvement. Statistics serve an important input to all these activities.

5.3.1 Bottlenecks

A bottleneck refers to the slow rate of execution, information flow of a particular component in relation to others in a system. A bottleneck can be tolerated unless the system is performing a critical activity, that it cannot afford to miss the deadline.

It is not always possible to tell when a more critical activity comes. Its accommodation with the least impact on the rest of the system may be achieved by eliminating bottlenecks in the hope that resources and unused scheduling slots can be reclaimed. Since monitors are the only means whereby the entire system can be observed, they play a part in locating those bottlenecks.

5.3.2 Deadlocks

The larger the memory and the more the available processors, the faster an activity can be executed. The availability of resources and their utilization aid the throughput of the system.

A federated system has, by definition, a higher resources capacity than individual ones in the federation. But this capacity does not necessarily translated into higher throughput. There are two reasons:

1. There is no centralised control over resource allocation and scheduling.
2. Resources can be withdrawn without notice.

The lack of centralised control means deadlocks can occur easily. They may be more difficult to spot because they may span over several nodes in circle. The strategic positioning of monitors is vital for the detection and elimination of such deadlocks.

5.3.3 Statistics gathering

The criticality of an activity is reflected in its scheduling priority. However, an activity may migrate (or diffuse) to other sites during the course of execution. A more global sense of criticality must be established across boundary.

If the migration is anticipated, measures must already be in place to ensure the preservation of criticality.

If the migration is un-expected, then the question is: are there sufficient system resources to cope with the demand? From a longer term of view, is the migration pattern regular in time and in shape?

Statistics gathering helps in a number of aspects:

1. It can be used to establish peak load and peak traffic into and out of individual systems and the communication medium as a whole.
2. It can be used to establish system performance profiles under peak load and/or peak traffic.
3. It can be used to feed back the relevant information into the programming development cycle for adjustments towards realistic requirements.

Monitors can be used to generate events, mock activities of various sizes and degrees of criticality to aid statistics gathering.

5.4 Transparency

The principle of selective transparency favours a toolset approach towards the realisation of computation requirements such as transactions, concurrency etc. This permits applications to evolve without references to the underlying technologies.

A tool transforms an application with a specific requirement into another whose function conforms to that of the original, and in particular the specific requirements. A transformation may be the result of a single application of a tool or the application of a chain of tools. This kind of transformer technology must not only preserve the functional meaning of the original program but also its temporal meaning. This means the timing constraint of the original program must be invariant to any sequence of transformation.

6 Computational Abstractions

6.1 Deficiencies in the ANSA Computational Model

The ANSA computation model [Ree93a] is comprised of two parts: a construction model and an interaction model.

The interaction model is concerned with a scheme for invocations through object interfaces and a scheme for typing interfaces.

The construction model is concerned with the provision to,

- construct objects that conform to the interaction model, and
- be computationally complete.

The interaction model is deterministic in the sense that a server cannot start a service until a client made a request.

This implies that the passage of time is in one direction.

In the context of real-time programming, the computation model does not cover the following:

- quantification of time,
- synchronisation,
- ordering,
- QoS and explicit binding, and
- streams.

A notion of quantifying time is needed because it is then possible to relate time with computations.

Synchronisation is needed because it allows conditions to be specified upon which two computations must not be performed in parallel.

Ordering is needed because it allows the expression of precedence among computations.

The notion of QoS is needed in order to objectively qualify computation. In this respect, the notions of binding and stream are required to be computationally explicit.

6.2 Objectives

It is useful to distinguish between what the programmer thinks in terms of a programming language semantics, the computation model semantics and the execution semantics. A programmer thinks in terms of a programming language. The computation model semantics gives a formal (e.g. functional) description of the kind of computation that can be carried out. An execution semantics gives a description of how computations are carried out. A tool chain maps a program directly (as an optimisation) onto its execution

semantics or indirectly through the computation model semantics resulting in a more portable program.

It would be useful if the three are identical resulting in a much simplified framework. An interpreter, for instance, unifies the computation model semantics and the execution semantics.

In addition to the distinction, it is important to distinguish between a programming language, its programming environment and the supporting platform or the nucleus which forms the execution environment. The computation model is not a programming environment.

In admitting time, synchronisation and ordering into the ANSA computation model, it is proposed to extend the construction model only. The intention is to keep the interaction model unchanged.

The proposed extension is to enrich the construction model semantics. The aim is so that new tools and new language constructs can be explained in terms of the construction model semantics and that of the existing interaction model.

Tools are in the programming environment and can be:

- deployed by the compiler in the construction of executable, or
- called upon during program execution.

Declarative programming is preferred over the imperative style. The details are left to the responsibility of automated transparency -- a mechanism for the deployment of tools. The architectural benefits are that the toolset is allowed to evolve over time and across platforms, and because the tools are faithful to the computation model semantics, the programmer is free to concentrate on the problem domain independent of the platforms.

Programmer control has been suggested to be necessary in real-time programming. Language constructs, such as timed path expressions, allow programmer control over synchronisation. In addition, computational interfaces to resource management allow the deployment of resources when and where they are needed.

6.3 Quality of Service

Objective statements to the computation can be made as QoS declarations. The following constructs will be allowed to be decorated with QoS declarations:

- type,
- interface,
- object,
- activity,
- invocation, and
- binding.

Bindings are implicit in the model at present. They will be made explicit to allow enhanced bindings to the quality specified. In addition, it will be possible to lump together several bindings as a single entity to which a certain QoS applies.

The notion of selective transparency will be made explicit in the language. This will allow the use of QoS declarations to dictate bindings to the quality required. Such declarations are meant to meet static requirements prior to program execution. As indicated earlier, in order to allow fine-grain programmer control and to meet dynamic requirements, computational interfaces to resource management will be made available.

6.3.1 Streams

Multimedia applications require the support for streams (e.g. for transport of video and audio data). The QoS required over a stream is programmable through declarations of QoS to interfaces and bindings that make up a stream management interface.

A stream has two endpoints. It is proposed that references to endpoints are to be allowed as well as stream management interfaces.

6.3.2 Conformance Checking

There are three aspects in QoS:

- *Expectation*: A service consumer expects a certain quality of service from a service provider.
- *Promise*: A service provider promises to deliver a certain quality of service.
- *Requirement*: A service provider requires the service consumer to provide the necessary information (e.g. in the form of parameters) and the service bearer to allocate resources. The resources and perhaps the information may be required to have a certain QoS.

The service bearer can be modelled as a collection of services. Therefore there is no need to distinguish between expectation and requirement.

Typing (or more precisely subtyping) is one way of modelling information content.

Conformance checking in the context of QoS is the process of matching service expectations and service promises. For instance, is the jitter level of a service provider within the tolerance bounds expected from a service consumer? The CNET work on QL [HHS93b] illustrated one way to tackling this problem.

A service provider promises to deliver a certain quality of service. Apart from the required information and resources, it seems natural to ask the question: what mechanism is there to ensure that the provider indeed delivers the quality of service promised? The issue here is similar to that of the type of a function body must “match” the result type advertised in the function heade, except that such a task is usually part of a mechanical typechecking process. Good faith, programming expertise or (re-)modelling of the problem domain have been used to side-step the issue. But how far can they go? More thoughts are required.

6.3.3 Contracts

Conformance checking is essential to negotiation of contract. The notion of a contract will be made an explicit computational entity. It is to be supported as a common mechanism for encapsulating agreements not only to service provisioning but also accounting, responsibility to manage, and federation matters.

6.4 Synchronous Constructs

The introduction of a notion of signalling into the construction model seems promising.

A signal is a named, record-like structure. A signal takes the following form,

```
<name> ({<type> <arg>}*)
```

There can be many instances of signal. Each signal instance carries distinct information, or temporal or ordering significance.

Signals are for emission and reception between activities within an object or between activities in an object and the execution environment.

An invocation can be explained in terms of signals and the associated synchronous semantics. Four signals (or two pairs of transmit-receive action) are required to achieve an invocation in the synchronous mode of computation.

Figure 6.1: Invocation as signals

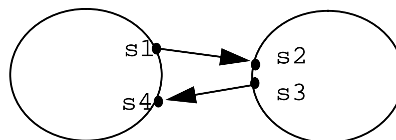


Fig. 6.1 shows two signals are needed for emission and reception at either end of an invocation.

It is more general to talk about four signals. This permits a common execution environment over a network.

It is important that input and output signals are paired up properly. There is a fairly straightforward solution using an IDL. This is described in a following section.

It is proposed that timed path expressions will be investigated at the language level for synchronisation purposes. It is intended that the runtime semantics of a timed path expression can be mapped in terms of signals. The reasons being that,

1. path expressions can be translated into state automata, and
2. signals can be used to model time inputs to a state automaton.

An application of the idea in (1) can be found in [Ree93b] where path expressions are used as concurrency guards.

6.5 Programming Languages

The syntactic manifestation of the extended ANSA computation model can be made in three areas:

- DPL [Otw93] will be made in line with the proposal. DPL will remain the vehicle for illustrating theoretical discussion of distributed programming. It is free from non-distribution concerns.

- A re-design of PrepC will be made to take account of the proposal. This effort is to check backward compatibility but no implementation effort is envisaged.
- PrepC++ is the major work area. It is meant to succeed PrepC. C++ is chosen because of its popularity and because object-orientation is more appropriate to distribution.

6.6 IDL and CORBA

It is the intention in Phase III that technologies will be developed on commercially popular platforms, and CORBA and C++ are within the range of choices.

6.6.1 CORBA IDL

In 6.4, it was mentioned that there is a need to pair up input and output signal declarations. This can be arranged fairly straightforward with an IDL as follows. Signal declarations are automatically generated from an IDL definition. A server and its clients are not required to share the same signal declarations. It is only required that each signal declaration matches the IDL it is compiled against. This allows the use of different IDLs. Binding to a server involves compatibility test between the IDLs and conformance test between the corresponding signal declarations.

There is a signal declaration for each operation and there is a signal declaration for each associated termination. There can be many signal declarations. However, for an invocation, there are only four signals involved.

Flow analysis at compile time is required to ensure signals are transmitted as a result of each request.

As indicated earlier, the major work areas envisaged are to allow the expression of behavioural constraints in IDL definitions. These constraints may be expressed in terms of:

- quantification of time,
- synchronisation,
- ordering.

6.6.2 CORBA C++ Language Binding

This work on PrepC++ is envisaged to be carried out in the context of CORBA C++ language binding.

Reference

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R Morrison. An Approach to Persistent Programming. *Computing Journal*, 26(4), 1983.

[APM93] ANSA. Application of the Architecture: a scenario. APM.1041. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[BB91] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. RR1445. INRIA, Rocquencourt B.P> 105, Le Chesnay Cedex, France. 1991.

[BC84] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In Seminar on Concurrency, S.D. Brookes, A.W. Roscoe and G. Winskel (editors) LNCS 197. Springer-Verlag. 1984.

[Cha78] M. Challis. *Databases: Improving Usability and Responsiveness*, chapter Database Consistency and Integrity in a Multi-user Environment. Academic Press, 1978.

[Doe86] T. Doeppner. Towards a Workstation Operating System. *Proc. of the 19th Annual Hawaii Int'l Conference on System Sciences*, pages 855–862, 1986.

[Edw93] N. Edwards. A Model for Failure. APM1027. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[GJ75] M.R. Garey and D.S. Johnson. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal of Computing*, 4:397–411, 1975.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufman, Redwood, CA., 1992.

[Gun84] D. Gunsfield. Bounds for Naive Multiple Machine Scheduling with Rlease Times and Deadlines. *Journal of Algorithms*, 5:1–6, 1984.

[HHS93a] L. Hazard, F. Horn, and J-B. Stefani. Some Computational and Engineering Aspects of Streams. RC.W03.LHFH.001. CNET, rue du General Leclerc, 92131 Issy les Moulineaux, France, 1993.

[HHS93b] L. Hazard, F. Horn, and J-B. Stefani. Toward the Integration of Real-Time and QoS in ANSA Architecture. CNET.RC.ARCADE.01. CNET, rue du General Leclerc, 92131 Issy les Moulineaux, France, 1993.

[KK91] H. Kopetz and K.H. Kim. Real-time Objects and Temporal Uncertainties in Distributed Computer Systems. PDCS Technical Report Series 42, Technische Universität Wien and University of California, 1991.

[LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming

in a Hard Real-time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[LL93] B. Lampson and D. Lomet. A New Presumed Commit Optimisation for Two Phase Commit. Technical Report 93/1, DEC CRL, 1993.

[Lam78] L. Lamport. Time, Clocks and the ordering of Events in a Distributed System. *CACM*, 21(7), 1978.

[Li93] Guangxing Li. *Supporting Distributed Realtime Computing*. PhD thesis, Computer Laboratory, University of Cambridge, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England, July 1993.

[Lun84] J. Lundelius. Synchronizing Clocks in a Distributed System. Master's thesis, MIT, 1984.

[Mul93] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley and ACM Press, 2nd edition, 1993.

[Nic91] C. Nicolaou. Support for Multi-media Operations. TR 28, APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1991.

[OE93] E. Oskiewicz and N. Edwards. A Model for Interface Group. AR.002. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England.

[Ole92] M. Olsen. A Persistent Object Infrastructure for Heterogeneous Distributed System. RC.343. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1992.

[Otw93] D.J. Otway. The DPL Programmers' Manual. TR.031. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[Pow91] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Research Reports ESPRIT Porject 818/2252, Springer Verlag, 1991.

[Ree93a] R.T.O. Rees. The ANSA Computational Model. AR.001. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[Ree93b] R.T.O. Rees. Using Path Expressions as Concurrency Guards. TR.022. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehockzy. Priority Inheritance Protocols: An Approach to Real-time Sychronisation. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.

[War93] J. Warne. ANSA Atomic Activity Model and Infrastructure. AR.004. APM, Poseidon House, Castle Park, Cambridge CB3 0RD, England, 1993.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John Stankovic. Preemptive Scheduling under Time and Resource Constraints. *IEEE Transaction on Computers*, 36(8):949–60, August 1987.