Chapter 9

Extra Performance Architecture (XPA)

The OSA architecture offers a wide range of solutions for distributed fault tolerance. However, the conjunction of the attributes offered, especially the complexity inherent in its generality and openness, precludes its successful use in certain niche applications.

Indeed, for a number of critical applications, where high-performance, real-time and fault-tolerance are simultaneous goals, those major attributes of OSA become a constraint. The difficulty arises in the time domain, i.e., on the ability of offering assurances of timeliness, and on the user-perceived responsiveness and throughput of the system.

The Delta-4 *Extra Performance Architecture* (XPA) introduces mechanisms that support explicitly the requirements of real-time systems, with the introduction of priorities and deadlines, and the requirements of high-performance, with respect to both throughput and response. However, XPA will inherit as much of the OSA architecture as is possible within the constraints imposed by these requirements, and XPA and OSA will retain compatible interfaces as far as possible, see annexe H.

To meet performance and timeliness targets, XPA will inevitably lose in openness and generality. The functionality of the communication system is limited to that required to support XPA applications running under XPA-Deltase. Standards that cannot address real-time needs will not be used. There will be layer and service elimination in the communication protocols, and fail-silent hardware will be used to avoid the need for validation of computation results. Only homogeneous hosts will be accommodated, to avoid the performance overhead of conversion of data representation.

The XPA workpackage began much more recently than other work in Delta-4. This chapter therefore contains a synthesis of concepts and ideas that are less mature than those presented elsewhere. Since the present project is nearly completed at the time of writing; it is likely that a complete implementation of the concepts described will require further work.

9.1. Objectives and Definitions

The strategic objective of XPA is to give a Delta-4 solution to specialized application scenarios, where the OSA architecture proves to be inadequate. In Delta-4, these scenarios are found in the fields of real-time and high-performance systems. Since Delta-4 is aimed at providing distributed fault tolerance, technically, the XPA architecture has the difficult goal of combining:

- distribution;
- real-time;
- dependability;
- high performance.

Thus, XPA is required to provide timeliness assurances, by performing actions within bounded and known time intervals. The start and end of such an interval are called *liveline* and *deadline* in this chapter and chapter 5. All system support mechanisms, including communications, must have controlled latency and synchronism.

XPA is decoupled as far as possible from any particular approach to the issue of real-time scheduling mechanisms. The architecture is therefore open to whatever scheduling strategy a particular application requires, and can make use of whatever scheduling primitives are offered by the underlying LEX.

Each service of an XPA system is therefore assigned a *precedence* that provides a generic measure of both its importance and urgency. The precedence is an abstract data type that is interpreted by a system-dependent "Precedence Manager" into a set of scheduling parameters suitable for presentation to the underlying LEX scheduling mechanism.

The same precedence manager is used to interpret the precedence throughout a single XPA system, and schedulers of different resources, such as processor time and network bandwidth, use compatible strategies, in the following sense:

- XPA permits deterministic preemption of computation at predefined points a bounded execution distance apart (see section 9.4.3.2). Similarly, message transmission can be preempted in bounded time between individual frames. Whereas the LEX machinery for deciding between the instantaneous precedences of two computations might be complex and vary with time (e.g., [Jensen et al. 1985]), in the communication system precedence comparison can be simple, since the time granularity of communication is normally much shorter than that of computation. Indeed, a message carrying a typical RPC is normally contained in a single communication frame.

Thus the precedence manager exports an operation to tell an XPA resource allocator which of two precedences should be preferred. Precedences are inherited by all components on which a distributed computation depends, and the "inherit" operation is also exported by the precedence manager, since it is closely associated with the scheduling strategy.

In the XPA prototype, the LEX is the Real-Time UNIX developed in phase 1 of the Delta-4 project [Bond 1987, SVC200], and the precedence manager interprets the precedence as a "priority", or measure of importance, and a "targetline", or measure of urgency. In the context of this LEX, a service A is said to have higher precedence than a service B, if:

- either *A* has higher priority than *B*;
- or *A* has the same priority as *B*, and *A* has an earlier targetline.

In another implementation, the XPA precedence may be interpreted in a radically different way. For example, section 9.6.2.3.1 describes how a fixed cyclic schedule calculated at design-time can be encoded into an XPA precedence and reproduced at run-time by a suitable precedence manager and LEX.

The differing importance and urgency of services are recognised by assigning precedences to actions, and allowing high precedence actions to preempt low precedence ones. In a distributed system, the precedence concept should of course be propagated throughout the information path. In consequence, high-precedence messages in the communication system should also be allowed to overtake low-precedence ones.

Distributed fault tolerance in Delta-4 relies on message passing. The order and agreement attributes needed to preserve consistency of replicas should be guaranteed by the distribution support (group management and communications).

A distributed real-time system requires the maintenance of a distributed time service, which provides an abstraction of global time. This is required to ensure *participant and replica agreement on time*, for example: to trigger actions at pre-determined instants; to recognise when events occurred; or to provide replicas of a component with a common knowledge about time, a general condition for replica group determinism in real-time. Both an internal and an external time reference may need to be provided. The internal time is provided through a global clock approximation, maintained by a distributed time server. The external time consists of one of the standards of time, whichever is necessary for the particular use of a given XPA system.

A Delta-4 XPA sub-system is able to coexist and inter-work with Delta-4 OSA sub-systems. Application objects are portable between XPA and OSA systems; in both cases the application objects are supported in Deltase envelopes or *capsules*. XPA Deltase supplies the same services as OSA Deltase, with some variations to support real-time and high-performance applications (see section §9.4).

Disclaimers: XPA is aimed at providing support for application in which the cost of failure — including the failure to meet real-time constraints — is measured in financial terms rather than in terms of potential loss of human lives, i.e., XPA is *not* aimed at safety-critical applications although, like OSA, it may be usefully applied to safety-related applications (see annexe A).

Fault-tolerant sensing and actuating are not directly addressed within the present phase of the Delta-4 project. Input/output implications and requirements are however discussed in a Delta-4 context in chapter 12.

9.2. Overview

This section surveys the issues treated in the present chapter.

9.2.1. Real-Time and Performance

One major objective of XPA is to support real-time applications, which are defined as applications that are able to offer an assurance concerning the timeliness of service provision.

XPA in fact aims to support both hard real-time applications, which are deemed to have failed if they miss their deadlines, and soft real-time applications, which may miss their deadlines in some circumstances. An XPA system may also support non-real-time applications, which have no timing constraints.

XPA is also addressing applications where high performance is required, not only locally to hosts, but in the distributed environment. Special attention is thus given to the responsiveness of the system to distributed actions, optimization of the distributed user-to-user data paths, and development of efficient group communication protocols.

9.2.2. Dependability and Computational Models

The fact that distributed fault-tolerance is associated with real-time, also has implications in the dependability techniques to be used in XPA.

In XPA, fail-silent hosts provide the basis for error detection and replication provides the basis for error recovery.

Consequently, backup replicas can either be active or passive. The possible models differ in the role and degree of activity they assign to each replica. The requirements of XPA were in the origin of a new dependability model in Delta-4: the *leader-follower*, or *semi-active replication* model. The rationale and advantages of this model for real-time fault tolerance are discussed in chapter 6.

Computation in XPA relies on the XPA version of the Delta-4 applications support environment, XPA Deltase, which differs from OSA Deltase in two main ways:

- Performance enhancements: XPA Deltase and the communication manager are combined in one library, which copies data directly between the application and the LAN message buffers, bypassing the LEX.
- Support for real-time: XPA Deltase includes a real-time local execution environment, and interprets the precedence of each computation in terms of suitable LEX scheduling parameters.

9.2.3. Support for Distribution

The original design of XPA envisaged the simplification of the interface between Deltase and the communication support, for performance. Deltase was collapsed onto the group communication layer, implying the introduction of new communication services that led to the extended AMp (xAMp), and of an intermediate harmonising layer — the *group management layer (GM)*. The architecture is shown in figure 1.

The *Group Manager* (GM) is a distributed object, represented locally on every node of an XPA system. In fact, the group management entities are pseudo-objects, since they present an object interface to the Deltase world, but interact with the group communication layer as another layer above. The group manager is concerned with the management of groups of objects, and with the support of the distribution of such groups. The group manager incorporates knowledge of the different modes of replication (active, passive, semi-active, etc.), and based on that view, it is able to provide, transparently to the objects themselves, the appropriate level of support, using the xAMp.

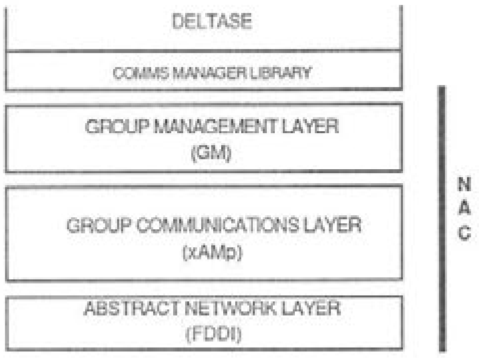
DELTASE
COMMS MANAGER LIBRARY
GROUP MANAGEMENT LAYER (GM)
GROUP COMMUNICATIONS LAYER (xAMp)
ABSTRACT NETWORK LAYER (FDDI)
NAC

Fig. 1 - XPA System Architecture

The communication sub-system comprises the *abstract network layer*[1], and the *group communication layer*. As detailed in chapter 10, the implementation of the abstract network over different LANs allows the communication sub-system to be LAN-independent. The local area network used in the current XPA prototype is an ISO 8802/5 token-ring, featuring 4 Mbit/s. Taking into account the needs of XPA, both from the reliability and performance viewpoints, FDDI — a 100 Mbit/s high-speed, high-throughput fibre-optic LAN — is an envisaged alternative, whose advantages are discussed later in this chapter.

The group communication services of XPA are based on an extended version of AMp, xAMp (see chapter 10), which provides a set of primitives, offering extended group

[1] We recall that "network" is not taken in the sense of layer 3 of OSI; it refers generically to the several communication infrastructures for xAMp.

management facilities and different tradeoffs between quality of service and efficiency. The services range from an *atomic* service to simpler and more efficient alternatives, down to multicast datagrams. The communication protocols will generally be executed on a special-purpose *Network Attachment Controller* (NAC) board. However, given that both host and NAC are fail-silent in XPA, it is also possible that the NAC be combined with the host itself.

The time service is provided at the xAMP layer, in order to improve precision.

9.2.4. XPA as an Integrated Machine

The previous sections have described the options available to the system designer in XPA. This section shows how the bricks just described interact, in one of the possible combinations to fulfil the real-time, dependability and performance objectives stated in the beginning, in the XPA prototype built.

9.2.5. System Administration

System Administration (SA) is concerned with monitoring and managing changes in the operational status and degree of replication of objects, and changes in the system configuration.

This does not have to be implemented by a central "system administrator", or distinct *system administration subsystem*: system administration functions can sometimes be delegated to ordinary application objects and the support environments of the objects to be managed.

In Delta-4 XPA, because of its emphasis on high performance, there is an overall bias in favour of the local management of objects, and on delegated rather than centralized system administration. Objects are encouraged wherever possible to perform administrative operations themselves, when invoked by the appropriate manager, exploiting their self-knowledge to optimize performance.

9.3. Real-time and Performance

9.3.1. Introduction

One of the major objectives of Delta-4 is to support real-time applications. The techniques used to support real-time were developed first for XPA and in some cases depend on its restrictions, such as the assumption that hosts are fail-silent and have real-time local schedulers.

Real-time systems require predictably bounded performance. This is distinct from the other main objective of XPA, which is to support the highest achievable performance for all aspects of distributed computation.

Chapter 5 defines many of the concepts used in this chapter and explains the Delta-4 approach to real-time. In this chapter, section §9.3.3 discusses the requirements and computational philosophy of XPA real-time. The main techniques used to support real-time and high performance are described in sections §9.4, §9.5 and §9.6.

9.3.2. Markets Addressed

The market areas addressed by XPA were described in chapter 1, but a brief reminder at this point will assist understanding for readers who are more familiar with other real-time or high-performance market areas.

XPA is designed to address large dynamic systems in the process control and command and control areas. These systems are notable for the large, complex sets of assumptions that have to be made before it is possible to predict their behaviour during the design process.

For example, state-of-the-art process control systems typically contain hundreds of analogue loop controllers, each with analogue input sampling, control calculation and analogue outputs, and possibly thousands of digital inputs and outputs. Scan and response times may be of the order of milliseconds, and plant integrity may depend critically on reliable and regular execution of the scanning and control logic.

Such systems must support concurrent interaction with many operators. The operator interface has its own response time requirements in various conditions of use, and its quality can exert disproportionate influence on the commercial acceptability of an architecture.

Most process control events are periodic, but some aperiodic, and some "sporadic", the term used in [Burns 1990a] for events that are aperiodic, but have a minimum specifiable duration between their arrival times. We reserve the term "aperiodic" for cases where no such minimum duration can be specified.

Command and control systems contain large numbers of real-time events, periodic, sporadic and aperiodic. There are typically several modes of operation and several different priority levels. However generously the system is sized, peak activity levels depend on an uncontrollable environment, and may still cause a system overload.

9.3.3. XPA Real-Time

Chapter 5 contains a generic discussion of real-time concepts and requirements and defines many of the terms used in this chapter. This section is concerned with the more specific real-time requirements of the XPA prototype, nevertheless, for the sake of self-containment there may be some repetition of the material in chapter 5.

The *priority* of an activity is a measure of its criticality, i.e., the cost of not meeting its timeliness constraints. The highest priority is given to hard real-time components.

The *targetline* is the time at which the system designer aims to deliver the service; it lies between the liveline and deadline, which define a window within which service must be delivered. See chapter 5 for a generic discussion of these terms.

9.3.3.1. Hard Real-Time. A *hard real-time deadline* is one that must be met in order to avoid a costly timing failure. Such deadlines, the operational envelope in which they must be met, and any defensive behaviour must be captured in an adequate requirements specification, which must be unambiguous. This involves:

- Determining that there is sufficient system power. Worst-case component execution requirements and other resource usage must be determined. In particular, the system must have sufficient power, or "execution resource", and a means of allocating this power must be provided which has properties useable during the design process.
- Knowledge of execution times. Whatever events can occur, their worst-case timeliness consequences must be accounted for during system design. It follows that primitives that do not themselves give assurances of time bounds are unuseable as a basis from which hard real-time components can be built.
- Control over system component location. For example, [Burns 1990a] argues that migration of hard-real-time components cannot be permitted.
- Avoiding all deadlocks and bounding delays due to temporary resource conflicts (see section 9.3.3.4).

For example, the consequences of internal hardware faults must be bounded in time. Component failures will occur — they cannot be ignored in the design process. The consequence must be accounted for in specification, as a (specified worst-case) rate of occurrence, and perhaps in design, through the use of fault tolerance or recovery mechanisms.

If the time overhead for some such mechanism is not compatible with some computation's deadline requirements, then that computation cannot use the mechanism.

As another example, human operators are sometimes placed "in the loop" as a final resort to cover a possible emergency. Whereas it is obviously not possible to specify bounds to operator reaction times in such an open-ended situation, the information displayed and any commands issued must be processed by the system within specified time bounds so that it is not open to indictment as the cause of late response to the emergency.

9.3.3.2. Soft Real-Time. A soft real-time service has an identified timeliness requirement but no high cost is attached to a failure to meet it. An example is the updating of a display monitor that is intended to show the progress or present state of some production process. The majority of such information does not come into the category examined in the previous section, in that an operator will only be inconvenienced by an occasional delayed update. The quality of the interface is then measured in terms of its normal rather than its worst-case behaviour.

This is a very different world to that of hard real-time. Here, deadlines are regarded as desirable targets that it is not essential to meet on all occasions. The requirements specification must capture such possibilities. A consequence is that the system power can be less than implied by worst-case analysis. A system whose peak loads are many times normal loads, or only statistically predictable, might therefore be rendered economically viable.

Although this is distinct from hard real-time, it must be stressed that it is also quite distinct from the principles of "fair time sharing" employed elsewhere in the industry — notably in traditional mainframes and in communication systems, or the FIFO ordering used in batch control systems. In a "fair" system, when a new requirement arises, the execution resource is redivided; the new requirement gains a "fair" allocation of resources and every other requirement loses to a small extent. The effect is that if requirements arise arbitrarily, no assertions can be made about timeliness for any of them.

It is therefore necessary to be unfair, to give some requirements more importance than others. There may exist a spectrum of urgency to the different activities that arise. If not all outstanding activity can be fulfilled, that which is least important is postponed, but timeliness assertions can still be made about what is most important. This is one basis for controlled, or "graceful", degradation of behaviour.

9.3.3.3. Best-Effort. For soft real-time components such as the operator screen server, there are costs attached to each timing failure. The objective of a best-effort system is to minimize the probable cost function by achieving the lowest probability for the most costly failures.

The means of calculating the probable cost function for each proposed strategy must exist. This involves, as before:

- Determining that there is sufficient system power, this time as a factor directly determining the probability of meeting timeliness criteria.
- Knowledge of execution times, this time in terms of probability distributions rather than worst-cases. As discussed in chapter 5, we are usually obliged to build systems from microprocessors for which execution times are probabilistic due to internal pipelining and cacheing and the use of contention busses. Even were this not so, the complexity of operating systems, language compilers and realistic applications effectively prevents the construction of any rigorous timeliness proof, much less one short enough to be subjected to a meaningful peer review. Banning the use of unbounded constructs ([Puscher and Koza 1989] gives the example of unbounded loops) does not overcome these obstacles.

- Control over component location. If the pattern of external demands becomes such that certain nodes perform more than their share of high-urgency computation, some of this might not be completed whilst less urgent computation is completed elsewhere. This situation, if it persists, represents an unbalanced system. Components may be migrated to balance the urgency of processing achieved across nodes, providing care is taken to avoid migration-"thrashing".
- Avoiding all deadlocks (e.g., by always acquiring resources in the same order), and determining both the probability and the duration of any temporary resource conflicts (see section 9.3.3.4).

The method of devising schedules is important:

- Off-line schedulers (see chapter 5) do not provide direct support for the idea of preferring one computation to another.
- For on-line schedulers, there is a shift in emphasis. Some methods, like earliest deadline scheduling, can find a feasible schedule, in which all deadlines are met, if such a schedule exists. In a best-effort system, there may be no feasible schedule: such methods are then generally badly behaved.
- When not all deadlines can be met, the method must explicitly take account of which activities are preferred. For this, scheduling computations according to their priority is appropriate. It might be that under different modes of system operation, different priorities apply.

Note here the essential difference between use of priority and use of deadline. For feasible schedules, highest priority scheduling is not as well behaved as earliest deadline scheduling, which on one processor will find a feasible schedule if one exists. On the other hand, for unfeasible schedules, deadline scheduling is not as well-behaved as priority scheduling, which establishes a design preference over what is allowed to execute when not everything can be executed.

It follows that a best-effort system should rely primarily on highest priority scheduling, so that only lowest priority deadlines will be missed in a marginal overload. Only components of the same priority should be scheduled in the order of their deadlines or targetlines. For most priorities this will normally find a feasible schedule, but in a marginal overload some components of lowest priority start missing their deadlines. Since they are of lowest priority, the cost of missing their deadlines is minimal, and may be less than the cost of the extra system size needed to avoid such overloads.

9.3.3.4. Resource Conflicts. Resource conflicts do not cause unpredictable delays in simple static schedules, since the times when resources are required are known during design; the points of reservation and release are embedded in the static schedule.

However, in complex dynamic systems, where resources other than those for execution are being contended for and mutual exclusion must be practiced, there is no known generic mechanism that permits absolute (as opposed to probabilistic) assurances of timeliness to be made during the design process. In commercial systems, partial solutions have made use of application-specific knowledge, as discussed below.

If a highest precedence computation requires a resource, the current holder of the resource should start running with maximum precedence. The maximum delay is then the dedicated execution time before the holder releases the resource. This is a generic mechanism in that it is applicable to all types of resource. However, it works well only for the highest precedence computation. Furthermore, it does not resolve deadlocks; these must be avoided by application-level methods such as:

(a) declaring at the start of each transaction which resources may be acquired during the transaction [Eich 1988, Habermann 1969];

(b) restricting the order in which resources may be acquired [Kopetz and Schwabl 1989].

An alternative policy is to use a transactional model of computation and abort the holder of a resource required by a higher-precedence computation in much the same way as is proposed for deadlock recovery in [Moss 1981], through a sequence of messages from resource claimers to resource holders. The operation exported by the precedence manager to compare precedences is used to compare the precedence of a remote resource claimer with the local resource holder. This enables a shorter bound to be set for the highest precedence computation but leads to expensive re-execution of the aborted transaction; the timing properties are difficult to analyse, even probabilistically.

However, this mechanism will resolve cycles of deadlock, provided that precedence is always unique. To ensure this, one of the precedences must always be preferred, in a deterministic manner across replicas. For example, when importance and urgency are identical, precedence order may be resolved deterministically using unique computation identifiers.

The timing properties of these strategies are only calculable if applications hold resources for bounded periods, and only acceptable if these are short. This seems to be general; there is a similar requirement in static scheduling schemes[2], where components hold resources during pre-arranged parts of a static scheduling cycle (see section 5.2.4).

9.3.4. Performance

Homogeneity plays an important role in XPA performance: the communication system requires no presentation layer, and the use of nodes with similar power simplifies the synchronization of replicas. From an engineering viewpoint, the existence of only one hardware configuration optimizes performance enhancements.

The XPA communication sub-system is LAN-independent. LAN choice may then be conditioned, when possible, by performance improvement. Taking into account the needs of XPA, both from the reliability and performance viewpoints, FDDI — a 100 Mbit/s high-speed, high-throughput fibre-optic LAN — is an interesting alternative, in comparison with slower LANs, such as 4 or 16 Mbit/s token-rings or 5 or 10 Mbit/s token-busses.

The group communication service of XPA, xAMp, provides a set of primitives from an *atomic* service down to multicast datagrams. With this range of successively less costly services, the user incurs the minimum cost needed to provide a given functionality.

The communication protocols will generally be executed on a special-purpose *Network Attachment Controller* (NAC) board. However, given that both host and NAC are fail-silent in XPA, it is also possible that the NAC be combined with the host itself, yielding a simpler hardware configuration where data is moved around faster.

In fact, minimizing user-to-user transfer latencies also implies efficiency of the local executive and support services like buffer management. Mapping of user buffers into (and from) communication space (without copy) and scatter-gather DMA are desirable features.

The *Remote Procedure Call* (RPC) is one of several support mechanisms for distributed computation. This paradigm is one we wish to encourage designers to use. It underpins the

[2] A resource required by more than one component can only be held by each component for part of the static scheduling cycle. There is a tradeoff between the time for which resources need to be held and the need for such cycles to be short enough to meet the latency bounds of response to rare but important events. Unless such events have the overhead of several adequate processing "slots" reserved (but normally unused) in every cycle, such response must be achieved through installing a new schedule at a predefined point in the current cycle. At this point the current cycle must not hold resources required by the new cycle.

support environment, and its performance in support of replicated interactions continues to be the subject of intensive optimization effort within the project. The end point of this effort is not yet in sight[3].

9.3.5. Timeliness

Timeliness, i.e., the correctness of the system in the time domain, is the distinctive issue in a real-time system. XPA is concerned with assuring the timeliness properties of the support environment, by bounding all system latencies and queueing times. To construct a system with timeliness attributes, other factors are also essential: system sizing, knowledge of component execution times, and control of component location (see section 9.4).

However, as explained in chapter 5, timeliness can only be guaranteed where circumstances and events fall within a predefined "operational envelope". If they ever fall outside that envelope, timing constraints may sometimes be violated, but the designer still has a responsibility to detect such violations and limit their consequences. A time service is thus mandatory. The timing requirements of XPA are discussed in section 9.5.

In XPA, a loss of timeliness is normally detected as a missed targetline, so it is the consequences of not meeting a targetline that need management. If the targetline is set before the deadline, then action taken at the targetline may yet avert a missed deadline. If the targetline is set equal to the deadline, then a missed targetline is a timing failure, but there may still be ways of minimising the resultant cost.

There is considerable variation in the types of response that it is possible to design into different computations:

- For some computations, a missed targetline may have no consequence; execution simply continues after the targetline has passed and the service is provided when possible. Even here, the missed targetline should be reported to a system administration module, which might log the event or increment a count of missed targetlines that could be used for assessing system performance or to support a load-balancing algorithm.
- For other computations, some local action by the component concerned might be appropriate. This might take the form of a change in the nature of the computation performed, or a change in the circumstances of execution:
- In some cases, such as that of figure 1 in chapter 5, the benefit/delivery-time graph shows that it is less costly never to deliver the service than to deliver it after the deadline. In this case the component should abandon its computation, but report the exceptional condition to its invoking client (which may be able to fulfil the required service in some other way).
- If it is a periodic process, it might cause the start of its next period to be omitted or delayed. This, whilst harmless in terms of its effect on other components competing for the same resources, must be assured not to lead to other sorts of instability, for instance if the component is in the control loop of an external process. Any replica of the component must act in a consistent way, so inter-replica negotiation may be needed.
- It might limit its own functionality in some application-specific way, such as rejecting rather than deferring outstanding requests. The global ability to tolerate

[3] On the prototype, limitations are imposed by certain generations of VLSI medium access control chips, which appear to have been optimized for throughput rather than (as RPC demands) delay. Thus, figures of 3-4 ms for each network access have been measured on a particular Token Ring network interface chip. Such figures impose artificial limits that may not apply to the next generation chip or another medium.

such cavalier behaviour must exist. For example, a command and control system may reject older input data, but process newer input data that gives the recent position of an enemy aircraft.

- It might reduce the precision of some computation; indeed, it is quite reasonable to construct a certain class of components to refine a calculation until the targetline elapses (interpreted as "the deadline is about to elapse"), and then to deliver immediately the current "best estimate". An example is a chess-playing program. Such components exhibit hard behaviour.

There is such wide variation in possible responses that it is inappropriate to confine applications to a set of standard responses. When required, therefore, the support environment can inform the component of an elapsed targetline and allow it to interpret this as suits the application.

A system should normally be sized so that all hard real-time components can have worst-case execution time, and run right up to their targetlines, on every execution. So if a hard real-time component happens to terminate *before* its targetline, it may suspend until the targetline before starting its next task, without jeopardising the timeliness of its next task. By so doing, it releases resources to lower precedence components whose timeliness may be at risk in the current operating mode.

9.4. Dependability and Computational Models

9.4.1. Homogeneity

Chapter 7 discusses the use of heterogeneous hosts, languages and implementation conventions on Delta-4 OSA. The restriction to homogeneous hosts on XPA has the following advantages, from the computational and dependability viewpoints:

- *Performance:* the communication system requires no presentation layer, and performance enhancements need be developed for only one hardware configuration.
- *Synchronization:* it is important to bound the desynchronization between replicas, e.g., to maintain the accuracy of replica time (see section 9.6.6). This is easier if all nodes have the same power (see section 9.6.5.5).
- *Fail-silence:* XPA also requires fail-silent hosts, and it is easier to develop an acceptable degree of fail-silence for only one hardware configuration.

9.4.2. Replication Requirements

XPA requires a model of replication that supports real-time requirements such as bounded preemption and synchronization of replicas, and also the high-performance objectives of XPA, without compromising replica group determinism.

Several different models of replication are supported on Delta-4 (see chapter 6). They have different properties, and are not equally suitable for XPA. The differences are summarised below.

9.4.2.1. Active Replication. Active replication is only used with deterministic software components, e.g., those structured according to the state machine model [Schneider 1990], so that the replicas pass through identical states without any external support apart from identically ordered inputs.

Active replication carries the minimum overhead in terms of inter-replica messages, but an active replica's input messages need to use the *atomic* service of xAMp, which is not as fast as its *reliable* service. Computation can proceed at the speed of the fastest replica, but if the fastest replica fails, there is a delay depending on the desynchronization of the replicas, (see section §9.6.5.5).

Active replication does not at present allow high precedence input messages to preempt low precedence messages or low precedence computation within bounded times. Because of these unbounded delays, it is unsuitable for supporting real-time computations of more than one precedence level. However, the *slotted* service of the extended AMp (xAMp, see chapter 10) could support negotiating active replicas, so that bounded preemption could then be achieved, if not as efficiently as with semi-active replicas.

9.4.2.2. Semi-Active Replication. Semi-active replication supports non-deterministic software components, or components whose environments make opportunistic decisions about the preemption of messages or processes, in order to achieve bounded latencies for high precedence computation.

Messages to semi-active replicas can use the faster *reliable* service rather than the *atomic* service of xAMp (see chapter 10). The inter-replica messages cost less than those of passive replication, and can also be used to detect replica desynchronization and leader failure.

Semi-active replication can support preemption with bounded latencies. This is because a maximum precedence input message will raise the scheduling precedence of the destination leader to the maximum as soon as it enters the leader's environment. The leader must then run until the next preemption point, where the maximum precedence message preempts. The preemption latency is therefore bounded by the maximum execution time between consecutive preemption points, unless the leader fails.

If the leader fails, there is a longer but still boundable latency before a follower takes over. This depends on two other latencies:

- A follower can detect leader failure in a bounded time (see §9.6.5.6).
- The desynchronization between leader and follower can be bounded (see §9.6.5.5).

Because of this ability to bound latencies, semi-active replication can support hard real-time computation, as well as soft real-time and non-real-time components.

9.4.2.3. Passive Replication. Passive replication, like semi-active replication, can support non-deterministic capsules, and opportunistic mechanisms such as re-ordering of input messages or preemption of processing. Messages to passive replicas are also able to use the faster *reliable* xAMp protocol.

Because only one of the replicas is running, passive replication may require less processing power than active or semi-active replication. However, this has to be set against the extra processing power used in the generation and storage of checkpoints, which also consume extra network bandwidth. The generation of checkpoints also tends to reduce the response time of the primary replica.

When failure of the primary is detected, a backup has to repeat processing from the last checkpoint. The delays associated with primary failure, namely replica takeover, are therefore larger than with the other models. However, they can still be bounded, provided the frequency of "I'm alive" messages and maximum reprocessing times are known.

Since preemption and replica takeover latencies can be bounded, passive replication can support hard real-time computation, provided the delays associated with checkpoint generation and backup reprocessing are acceptable.

9.4.2.4. Replica Determinism. Active replication unsupported by inter-replica negotiation is unable to support preemption, needed to bound the execution times of hard real-time components. The first XPA prototype will therefore make no use of active replication.

The choice between semi-active and passive replication depends on whether the extra processing power and network bandwidth required by checkpointing, and the extra delays when a primary passive replica fails, outweigh the costs of extra active replicas. This is most likely to happen with components whose checkpoints are large, frequent, or time-consuming to generate, e.g., some real-time databases. Since the XPA prototype may be required to support such components, it concentrates on semi-active replication.

On OSA, applications may be implemented on fail-silent or fail-uncontrolled hosts. If fail-uncontrolled hosts are used, the applications can only be made dependable by active models of replication, with validation of network messages. These models require replicas to behave deterministically, which means they must be structured as state machines.

On XPA, applications are always implemented on fail-silent hosts, and can therefore use semi-active and passive models of replication. Some of these models support non-deterministic applications by enabling the non-deterministic decisions of one fail-silent replica to be imposed on the other replicas. Consequently, XPA applications need not be structured as state machines.

9.4.3. Scheduling Principles

To summarise the main requirements of XPA, we wish to allow both hard and soft types of behaviour to coexist. That is:

- We wish to provide support mechanisms that allow a system designer to assert that a defined subset of required activity is assured to meet deadlines.
- We wish to offer the designer the possibility of another subset exhibiting "best-effort" behaviour, but this must not compromise the deadline assurance of the first subset.
- Finally, we wish to offer the opportunity for any spare resource to be occupied with non-real-time activity, as long as this does not compromise the other behaviour.

In consequence, a notion of *precedence* was introduced, with the possibility for higher precedence computations to *preempt* lower precedence ones. The next two sections discuss these issues.

9.4.3.1. Precedence. Since the XPA prototype is intended to support best-effort systems, it uses the Real-Time UNIX LEX, which takes the on-line approach to local scheduling, using the instantaneous "precedence" of computation. In this LEX, XPA precedence is interpreted as a combination of the priority and targetline. The coarse precedence of groups of components is established through their levels of priority, and within a priority level, the precedence of individual components is established by the component targetlines.

Precedence does not just apply to individual components, but is made to apply to the whole of a distributed computation. When a requirement for some real-time action arises, *all* components supporting the action must be conducted with an appropriate precedence. The precedence of the action is therefore propagated with the request for service from component to component. Some such mechanism is needed since it would be impossible to predict timing behaviour if a high-precedence client raised a request for service on a server that ran at low precedence. The arguments for precedence inheritance are discussed in [Bond et al. 1987, Sha et al. 1987].

A capsule may therefore receive different requests for service with different precedences. The XPA environment must support this in such a manner that the system designer can assert timeliness properties of the components concerned.

Thus, in an input queue of requests to a component, higher precedence requests must overtake lower precedence requests, or preempt their processing, otherwise the hard characteristics required for the highest precedence activity will be compromised. The ability to ignore the effect of queued low precedence computation in calculating the requirements for higher precedence computation is essential.

To support Delta-4 dependability models, the overtaking and preemption must be performed deterministically, since replicas must be certain to receive requests in the same order. To achieve this, some form of *negotiation* technique must be used. In XPA, the replication models supported are limited to those where such negotiation is inexpensive.

This question of precedence ordering of inputs has an interesting analogue in the form of a current debate in the Ada world. The language creates problems for designers of exactly this sort in the ordering of entry queues. When several rendezvous are possible, the Ada programmer may make no assumption about which will be selected first. This destroys the possibility of bounding the time it takes for the server to rendezvous with the highest-precedence waiting client. The Ada 9X committee are examining such shortcomings in the present language definition.

On Delta-4, we have experimented with a distributed Deltase/Ada system in which standard Ada programs are mapped onto Deltase capsules (see chapter 7). The communication between these programs is validated syntactically by standard Ada compilers, and semantically it is similar to Ada; however, at run-time these communications are managed in precedence-ordered queues.

9.4.3.2. Preemption. Preemption of independent processes running in separate address spaces can take place as soon as the higher precedence process is runnable and the lower precedence process has finished the current machine instruction. If the processes sometimes share address space, e.g., sometimes run in the Real-Time UNIX kernel, preemption cannot occur in the middle of logically atomic operations, so that the maximum preemption latency is longer than a machine instruction.

Preemption between threads in the same capsule, which share the same address space and global data, is also supported on XPA, because some concurrent activities do need to share global data. However, the thread preemption must take place at the same point in all replicas so that the their states do not diverge.

Thread preemption mechanisms for semi-active replicas are briefly described below. Leader replicas may be preempted at the next point in processing that can be identified in a notification (or mini-checkpoint) to their followers, which will then be preempted at the same point. Such points are either *receive points*, at which the capsule requests the next input, or *preemption points*, at which the thread declares its willingness either to be preempted or to continue processing.

As soon as a high precedence input is waiting to cause preemption, the destination process can be executed at the same high precedence. An input of highest precedence therefore waits no longer than the maximum dedicated processing interval between two preemption points.

The code executed at each preemption point is essentially as follows:

```
Preempt_point_no := Preempt_point_no + 1;
IF Preempt_soon_flag = TRUE
THEN (Determine whether preemption to occur here, etc.)
     (A follower may be instructed to suspend here)
ELSE (Continue execution of current thread)
```

This code could be invoked as a procedure in any procedural language. With some high-level languages, notably "C", a more efficient solution is available: the preemption point can take the form of in-line assembler code (see annexe I). The details of the preemption protocol are described in section 9.6.5.3.

9.4.4. Timeliness Enforcement

XPA requires a group of generation-time tools to support the following functions:

- Measurement of execution times and preemption latencies.
- Insertion of preemption points in applications.
- Design-time proving of timeliness properties.

Maximum task execution times can be calculated by source code analysis [Puscher and Koza 1989], provided the application obeys certain restrictions such as bounded loops. The PDCS project is developing a tool to do this [PDCS 1990]. Real-time applications that do not obey the required restrictions must be modified, either by the application programmer or perhaps automatically by a sophisticated tool.

Maximum execution times can also be measured by test execution on a dedicated processor with "worst case" input parameters. To determine the execution time of a section of code, calls to read a real-time clock have to be inserted at the beginning and end of it. This could be the first stage in the insertion of preemption points that are to be separated by a specified maximum execution time, so as to achieve a known and bounded preemption latency.

The insertion of preemption points should ideally be transparent to the applications programmer. To see why this may not always be possible, consider an application containing the following code:

```
A: X:= Global_Variable;
B:
C: Global_Variable := X + 1;
```

where X is a variable local to the current thread, but Global_Variable is accessible to all threads in the software component. If the tool inserts a preemption point at B, one thread can preempt another at this point, causing a double updating of Global_Variable, which may have undesirable effects. If the tool cannot place a preemption point at B, the required maximum preemption latency may be exceeded.

All accesses to global variables could be protected by mutual exclusion mechanisms such as Deltase semaphores. (Here the programmer is advised to use a standard mechanism that will be recognised by the tool.) At first, the tool could ask the programmer to insert the semaphore operations; in the future, they could be inserted transparently by a more sophisticated tool.

However, even if the programmer acquires a semaphore before line *A* and releases it after line *C*, a preemption point at *B* may still cause embarrassment, because the thread that preempts at *B* cannot acquire the semaphore if it needs to; so the effective maximum preemption latency may still be too great. The tool should thus plant preemption points before *A* and/or after *C*. If the processing between *A* and *C* exceeds the required maximum preemption latency, the programmer should be asked to recode it.

Another problem is that a single statement of high-level language code, e.g., a for-loop, may take longer to execute than the required maximum execution time between preemption points. Such a statement must be recoded, either by the programmer, or transparently by a more sophisticated version of the tool. For example, a long for-loop could be made into two for-loops, one "nested" in the other, with one preemption point in each iteration of the outer for-loop.

To prove at design time that a whole system, or hard real-time subsystem, will meet its timing constraints, it is necessary to measure the worst-case execution time of each component on each computing element and the maximum latency of each system operation. Also required is the following information:

- livelines and deadlines, or the means of calculating them, for each distributed computation;
- a definition of the operational envelope in which timeliness must be achieved.

A static allocation of the components to the available computing elements can then be determined, with redundancy sufficient to support the fault tolerance specified in the operational envelope. (Dynamic allocation of components to computing elements makes it more difficult to prove timeliness and is therefore confined to lower priority components whose timeliness need not be assured.)

Targetlines that will meet the timing constraints are then set for each component on each computing element and the (on-line or off-line) scheduling algorithm to be used is selected. To prove that an off-line scheduler can achieve the targetlines, the actual fixed schedules to be used for every possible task set are constructed. To prove that an on-line scheduler will find a feasible schedule, it may be sufficient to prove that a feasible schedule exists, or that the processor load will never exceed a certain level [Sha et al. 1988].

If the required timeliness cannot be proved, it is normally necessary to re-size the system and repeat the above calculations. Only in marginal cases will a change of scheduler render timeliness provable. On-line schedulers are more able to take advantage of the fact that worst cases do not normally coincide, but cannot assure timeliness more easily in the extreme situation when all worst cases do coincide.

9.5. Support for Distributed Real-Time Computing

A distributed system is formed by processors with no shared memory, no centralized control or clock, hosting participants in distributed activities, communicating only by exchanging messages.

In designing the distributed real-time computing support of an architecture like XPA, several fundamental issues deserve discussion. One of them is order: what is the role of order? The relative ordering of events and/or actions (which may be distributed) must be preserved, for the system to progress correctly. A range of design alternatives are available, from a freely interacting concurrent system, to a highly sequential and blocking one. Another issue is time: what is time needed for in a distributed real-time system? It is required to ensure *participants and replicas agree on the time the environment changed, or on the time to act upon it.* The last issue discussed in this section are the requirements put on the communication service, to support distributed real-time computing.

9.5.1. Order Requirements

Given that participants communicate by exchanging messages whose transmission delays are not negligible, the cause-effect relations between events lead to partial orderings [Lamport 1978].

One type of ordering defined in section 10.1.2, is the one established when one message departs from a site, and arrives at another site, before another message is sent; the two messages are then said to be in a *logical* order. Preservation of such an order is normally adequate for non-real-time systems [Birman and Joseph 1987].

If a component is actively replicated — e.g., for fault tolerance — another requirement, established in chapter 6, is that the group of replicas receive the same information in the same order. This should happen, regardless of the fact that the group of replicas, as a whole, is still subjected to the cause-effect relationships just described. We have seen that some replication techniques may relax these order and agreement requirements, but let us focus on the general case.

In real-time systems, there are circumstances where the logical order does not correctly represent causality. In those cases, implementations based on it produce *anomalous behaviour*, i.e., participants may observe an ordering that is not consistent with their perception of system state. These situations occur when there is information flowing between participants, which is not controlled by the ordering discipline. Examples:

- when participants exchange messages without using the ordering protocol;
- when the system is expected to interact with the real world, in terms of real time.

This latter case, typical of real-time systems, can be reduced to the first, the interactions between participants and the environment becoming the outside "messages".

Clearly, the logical order is not causal, for these cases. For example, when a participant sends a broadcast *m1*, then issues an RPC to another participant, which in turn sends a broadcast *m2*. If the RPC is made outside the ordering discipline, there may be no way of knowing that *m1* Æ *m2*, and the resulting order may conflict with the view of the participants. The same may happen without using explicit messages. Physical feedback in a control and automation setting between two participants which otherwise communicate using a reliable broadcast protocol in a LAN, may produce similar effects. For example, suppose that a participant A performs an action, which generates a message *m1* and an output that physically affects participant B. As a result, B generates a message *m2* that, if only logica*ll*y ordered, could be delivered at destinations before *m1* — thus violating the real causal relationship established between them by the physical process. This may disturb distributed control algorithms programmed concurrently.

The solution for these situations consists in implementing *temporal order* of messages. In section §5.3, we saw it is possible to enforce temporal order with any synchronous communication system. Loose synchrony is the minimal condition for a reliable broadcast, so that correct behaviour be definable, in terms of real-time and ordering of events. This condition is relevant, since the basic communication protocols used in XPA are loosely-synchronous.

We have just discussed the ordering requirements of a general system. It may however be possible to reduce the generality of the distribution support, because certain assumptions about system ordering can be made. This is, of course, of particular interest in a highly optimized system such as XPA. Such assumptions, aimed at relaxing the default ordering and synchronism requirements of the system, may arise from:

- a particular knowledge of the applications, e.g., that concurrent senders are independent. A very immediate consequence is to reduce causality to individual senders, which yields the very well known FIFO order — in fact, a *temporal FIFO* order. Total order (see section 10.1.2) should still be provided when needed, to support replicas.
- a particular computational model semantics, e.g., that participant interactions are synchronous (RPC-like), and computations are single-threaded. This significantly reduces concurrency of the system; however, rendering it sequential simplifies the ordering requirements; as a matter of fact, causality follows quite simply the thread of control, as it moves from site to site.
- a particular replication model, e.g., the leader-follower. In this model, interactions are directed to a privileged replica (the leader). This replica represents the participant,

in the system, and is in charge of ensuring the consistency of its replica group. Total order at system level can therefore be relaxed in this example, since it is ensured by a private protocol inside the replica group. Causal order, incidentally, restricted to FIFO in the computational model used, must still be respected.

The xAMp communication service ensures a variety of ordering properties, seeing to it that the system designer can use a variety of computational and replication models.

To ensure replica group determinism, a message multicast to a group of replicas must be received in a total order by all recipients. This is ensured by the *atomic* service of xAMp, or by the *reliable* service in conjunction with the semi-active or passive replication protocols.

Causal ordering (see section 10.1.2) of logically related interactions may be assumed by the application and is therefore supported by some xAMp services, e.g., the *atomic* service. It can also be ensured by the use of blocking RPCs, since the events in each logical activity are forced into a known sequence. Blocking RPCs are compatible with parallelism if the blocking is suffered by only one thread in a multi-threaded capsule.

Casts, i.e., messages without reply, are often used to represent external events within the system. Such casts may be generated in the order of event occurrence and the destination may be written to expect them in this order. If so, they should be given the same precedence (or the same priority plus targetlines in the order of generation), so that their relative order will not be changed by message preemption.

The leader-follower semantics is such that some interactions can do without any order: notifications contain enough information to ensure that they need not be delivered in FIFO order. For example, if the follower is waiting at preemption point N+1, the notification "Continue from N+K to N+L" cannot be obeyed until receipt of the one or more logically-earlier notifications that bridge the execution from N+1 to N+K (see section 9.6.5.3). One or more of these may contain decisions, e.g., "Continue from N to N+K, and then process input HIGH_PREC". Such earlier notifications must eventually be delivered, but not necessarily in FIFO order.

Section 9.6 describes the interaction model of then current XPA prototype, and gives examples of how ordering is relaxed, thus requiring more efficient communication primitives.

9.5.2. Timing Requirements

If we look at the temporal order issue under the general perspective of *participant agreement on time*, we will find two facets:

1) agreement on the time to trigger actions: synchronization of the concurrent progress of a system;
2) agreement on the time at which events occurred: distributed recording (logging) of events.

These requirements will depend on the anticipated range of applications. For example, in the computational part of a real-time system, the synchronism requirements of (1) — i.e., steadiness (bounded variation in the transmission times of different messages), and tightness (bounded variation in the times of delivery of one message at several recipients) — apply only indirectly to computation (see chapter 12), whereas they apply directly to interactions with the environment. Whereas we might say that the synchronism requirements of (1) depend on *application granularity*[4], those of (2) depend on the "environment granularity"[5], more accurately, the granularity with which we wish to perceive its evolution.

[4] Granularity of an application is the minimum interval between any input event, and the subsequent response event.

Finally, replication for fault tolerance introduces the issue of *replica agreement on time*. This issue is a delicate one in real-time systems. The computation part is less demanding: whether for (1) or (2), solutions where agreement is slightly deferred, e.g., achieved by consensus, are acceptable. For example, a leader may notify its follower(s) that "the time of event E was T" to ensure replica agreement. Simplifying, there is not much difference between "the time is *T*" or "the time was *T*", provided that all replicas take the same "*T*". Fault-tolerant (i.e., replicated) I/O, however, requires a higher degree of synchronism. Non-replicated I/O already requires a certain accuracy towards real time; replicated I/O additionally requires precision among replicas. De-synchronism and loss of precision may lead to replica group non-determinism.

In conclusion, opportunities arise for relaxing the ordering requirements discussed above, and in consequence, the synchronism of communication and precision of the distributed time base. Those opportunities will be conditioned by the granularity and the real-action requirements of the envisaged applications.

An analysis of the various application domains for XPA, to discover relevant timing requirements, gives the following results:

Typical existing systems have required the time of the external world to be known only to the nearest second. This has conveniently been met through the use of radio broadcast time, received using compact specialized equipment at a single node and disseminated without any critical synchronization. If this service is temporarily lost, it has proved satisfactory to use internal dead-reckoning until the service is restored. As for internal time, most systems have relied on a centralized global time server (eventually derived from the source of external time). In normal conditions, such a system can yield low accuracy of the time read, due to delays. It is open to the unpleasant effects of arbitrary failure or unbounded dissemination time, unless a continuous check on the believability of the various time values is provided.

In XPA, it is expected to improve this situation. Within the system, there is a finer granularity called for, to distinguish the order of arrival of external and internal events such as alarm conditions. A typical current requirement is for about 10-50 ms granularity, for general events (local or remote). We hope in XPA to be able to determine event ordering to a finer granularity, even for distributed events. Internal clock synchronization is provided by the communication system, in order to achieve a time service of high precision (see section §9.6.6). This means bounding mutual deviations of the local clocks forming the global time approximation, to a close interval.

The clock synchronization protocol can in addition trigger actions at different sites within a defined time granularity, i.e., requirement (1) at the start of this section. Regarding requirement (2), the code that generates an XPA event when something happens in the external world (e.g., a LEX kernel interrupt routine) can be constructed to assign it a timestamp of the approximate global time of arrival.

Availability of good quality external time references (international time standards) will be considered (e.g., radio broadcast), when necessary for the relationship of the system with the external world. Given that this is not a general requirement, the time service will be independent of the approach to get external time. The latter may be injected in the system by one or more servers (fault-tolerant if necessary), having the required quality. Then, the external source may be used to *improve* accuracy of the approximate global time, i.e., keep the internal time as close as required to real time (typically 1 second).

---

5 Granularity of an environment is the minimum interval between any two physical events that must be perceived as being ordered (i.e., non-concurrent).

9.5.3. Communication Service Requirements

Basic concepts of real-time reliable multicasting have been discussed in chapter 5. Communication affects the timing of all distributed real-time activity, so the communication subsystem must respect the real-time constraints imposed by applications.

Previous material in this chapter has raised a number of requirements on communication, discussed below: bounded delivery *latency*; precedence (i.e., *urgency*) in the delivery of messages; *synchronism*.

A detailed description of the XPA communication system can be found in section 9.6.4.

9.5.3.1. Bounded Latency. Bounded delivery latency implies guaranteeing timeliness:

- of the underlying network, in the presence of disturbing factors such as *overload* or *faults*;
- of the communication protocols, whose executions must have a bounded and known termination time under specified fault scenarios; message delivery instants at several multicast recipients may differ, but only by a bounded and known amount.

To control frame delivery latency, the lower layers of the communication system should be engineered so that it is possible to parameterize the network size and load to the desired performance goals, if any guarantee is to be given on their achievement. Additionally, the operating conditions should be monitored so that error conditions can be detected and corrected. This includes both unsuitable loading situations and fault conditions, such as partitions, token loss, etc.

9.5.3.2. Urgency and Precedence of Messages. The requirement of precedence propagation implies that messages with different degrees of precedence receive privileged treatment, i.e., be delivered *ahead* of others. Despite such re-ordering, the following essential ordering properties are still assured by the replication and computational models used:

- total order of inputs to replicas (see chapter 6);
- sending order of messages with the same precedence and from the same source, e.g., checkpoints or notifications, and of causally-related blocking RPCs, whenever required.

Different urgencies imply distinguishing latency classes. In practice, there is a many-to-few mapping from (system-level) message precedence values to (communication-level) latency classes.

The urgency of a message should allow it to overtake other, less urgent messages, in the input queue of each destination access point. The overtaking time can be bounded for highest precedence messages at any particular destination. There could be similar overtaking in the output queue of a source access point, although this is not yet implemented.

In practice, high precedence messages often cannot overtake less urgent ones immediately, but they can do so after a bounded delay. If a low-precedence message is currently being transmitted when the high-precedence transmission is requested, the bounded delay will depend on the characteristics of the networks in use. For example, unless there is bandwidth sharing between latency classes, it will include the bounded time to complete the low-precedence transmission.

It is recommended that the lowest latency class on the network be reserved for hard real-time messages. These will then suffer only a bounded delay because of less critical messages. The delay which hard real-time messages cause each other must be bounded by system sizing and analysis at design time.

9.5.3.3. Synchronism. Real-time protocols must display synchronism, i.e., the duration of successive executions, as well as the skew of a protocol action at different sites, must be bounded (see chapter 5).

Tight synchrony, as implemented, for example, by protocols using approximately synchronized local clocks [Cristian et al. 1985], has the systematic performance overhead of having to take worst case communication delays into account. This would not be an extra overhead when proving the timing properties of a hard real-time subsystem, but is undesirable for best-effort or high-performance subsystems.

In a LAN, execution and inconsistency times may deviate considerably from their normal values, although the probability of these deviations occurring is normally very low. We plan to take advantage of this fact, for probabilistically tightening a loosely-synchronous protocol, since it behaves tightly most of the time. This would maintain recipients in synchrony in a best-effort manner, but of course, ordering could not be reliably ensured. In those cases where total order is needed (see section 10.1.2), another possibility is to build a higher-level tightly-synchronous clock-driven protocol on top of the clock-less one, to serve that part of the information flow requiring both total order and high synchronism. We believe both approaches cover the range of applications requiring performance and a high degree of synchronism.

9.6. XPA as an Integrated Machine

We now discuss how the computation and communication aspects of XPA are integrated to support real-time, high performance and dependability.

9.6.1. Prototype Architecture

In a bottom-up description (see figure 2), the XPA architecture is composed of the underlying LAN, on top of which the xAMp service is built. The group manager uses xAMp to supply the necessary distribution and replication management support to Deltase.

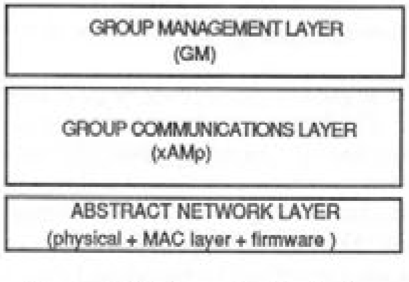
GROUP MANAGEMENT LAYER
(GM)
GROUP COMMUNICATIONS LAYER
(xAMp)
ABSTRACT NETWORK LAYER
(physical + MAC layer + firmware)

Fig. 2 - XPA Communication Stack

9.6.1.1. Choice of LANs for XPA. The LAN used in the current XPA prototype is a 4 Mbit/s token-ring, whose adapters were the ones available the earliest in the project. Note that thanks to the abstract network, XPA is virtually LAN independent. Still, its real-time guarantee and high performance aims, may condition the choice of LAN.

As an example, FDDI, which provides around 10 times the data rates of the other standard LANs used on the project to date (token-ring and token-bus), is under consideration, for future prototypes. It provides support for the implementation of urgent traffic classes (like token-bus), and has built-in medium fault tolerance, so that expected availability is very high (see section §15.4).

Although the sustained user-network data rate in FDDI is much higher, the objective in XPA, from a performance viewpoint, is achieving responsiveness not only to periodic but also bursty sends or receives. However, it has been shown, in section 10.7.2, that FDDI can enable increased responsiveness, reducing the end-to-end delay, and thence, the AMp primitives' execution times.

Compared with 8802/5 token-ring, FDDI has some speed enhancement measures like early token release and frame length limitation. A comparison with 8802/4 token-bus is more difficult, but these two LANs do have one thing in common: both access methods have a message priority mechanism based on bandwidth sharing by timed token priority classes. Although cable propagation velocity is basically the same, the transmission time of a frame in FDDI is about 30 times less that of slower LANs. For the same global load, medium length and number of stations, FDDI is capable of passing a much greater number of frames of the same octet length, per unit of time, than for example token-ring with which it has the most direct comparison. Added to the much larger number of nodes and distance allowed, FDDI also displays a better scaling capability. In the situations where frame dimension is not negligible, the throughput of FDDI becomes an important factor in the end-to-end delay.

9.6.1.2. Group Communication: xAMp. In the current OSA Delta-4 implementation the xAMp is part of an OSI-type protocol stack, with an inter-replica protocol implemented at the session layer, which also provides multipoint communication. In XPA, the application support environment (Deltase) is collapsed onto a group communication layer. A consequence of this is that the remaining layers must retain parts of the functionality of the removed layers. This is the reason for the introduction of new services in xAMp, and of the group manager as a harmonising layer.

As figure 2 shows, the XPA communication stack is based in a collapsed layering design where, to improve the efficiency of the whole stack, some of the layers present in the OSA architecture were removed. This means that the remaining layers must retain some of the functionality of the removed layers, to keep the system powerful and versatile. However, the increment of functionality cannot compromise the efficiency of the system, otherwise the result would be reversed.

The new services have emerged as variants of the original AMp primitive in order to profit from the previous design, development and validation work. Since the original AMp implementation was a starting point for the development of a multi-fold primitive, software is extensively re-used. The group communication layer is architecturally more elegant as a set of variants of the same protocol than as a bag of completely different protocols. For details about the extended service, dubbed *xAMp*, see section 10.4.

The group management services offered by the original AMp primitive were designed for the integration with an OSI-like stack, in the OSA architecture, where a single logical entity was the user of a gate group. With the original approach, where properties are only assured within a group, several high-level groups of entities were mapped on to a single AMp gate group. The mapping and multiplexing are done by some high-level entity, the session layer, with the original AMp. Evolution both in XPA and OSA has come to show that addressing support performed at low level can render those multiplexing and mapping functions more efficient. So xAMp should offer efficient primitives to ease the addressing of subgroups within one gate group and become the common communication support for both XPA and OSA. *Selective multicasting* is a key issue for this objective. With it, one can manage to address subsets of participants, e.g., separate between sender and receiver participants, in a dynamic, but userfriendly way. High performance address resolution is also obtained. These new features are extensively used in the XPA prototype.

Communication quality of service in XPA assumes a range from *atomic* to *datagram*, through variations of the same protocol, the xAMp, interpreted by a set of primitives, which we briefly recall: *bestEffortN*, *bestEffortTo*, *reliable*, *atomic* and *tight*. For details, the reader is referred to chapter 10.

9.6.1.3. Group Management. Turning the xAMp into a super primitive would be a negation of the end-to-end argument [Saltzer et al. 1984], as much as just providing datagram quality would be a bad use of it. In fact, the argument is against providing more functionality than needed at a given level of a system, because this goes against optimizing efficiency. Nevertheless, if a class of applications requires a certain functionality (or quality of service), however complex it may be, the lower level should provide it, since it frees the user from programming it, and will probably have been optimized and widely tested. Going back to xAMp, since it has to support a diversity of applications, both in OSA and XPA, it provides a range of qualities of service, rather than a single one. The upper layer user will select the one that best suits its request.

The xAMp provides a comprehensive set of communication primitives and low-level group management tools, to support interactions between groups of components in a distributed environment. However, these interactions may assume a certain complexity, as is the case if components are replicated for fault-tolerance, making it worthwhile for them to be managed by a dedicated entity between XPA Deltase and the xAMp layer. To take advantage of the facilities provided by xAMp, in such operations as message reordering, overtaking, or reuse, it should share structures with xAMp. The *group management* layer is thus introduced as a harmonising layer between the xAMp layer and the communication manager of XPA Deltase, sharing structures with both of them.

9.6.1.4. Deltase for XPA. XPA and OSA versions of Deltase (see chapter 7) present the same interface to application objects, except for the following extensions and restrictions for XPA:

- XPA applications can read and reset their precedence parameters, i.e., their priority and targetline (called "deadline" in the *Real-Time UNIX* documentation [Bond 1987, SVC200]), and the period of a periodic real-time task. However, only top-level client applications are expected to make use of these facilities; servers will normally inherit these parameters transparently.
- As explained in chapter 12, the management of *real actions* varies according to the model and degree of replication in use; yet it is an established ODP principle that replication should be transparent to the application. One way to achieve this is to perform the real actions in a transformer separate from the application.

  However, it may be that the application programmer himself wishes to supply a procedure that would perform the real action correctly in the simple non-replicated case. It may therefore be necessary to extend XPA Deltase so that it can call a real action procedure supplied by the application in a manner consistent with the actual replication details. For example, if the real action has already been performed by a leader replica, the Deltase environment of the follower would refrain from repeating it.
- XPA Deltase does not permit the recursive import of interfaces; i.e., a client may not import services from a server that also invokes the client. In the absence of this restriction, it would be more difficult to give timeliness assurances. Recursive

import is also disallowed by some languages (e.g., Ada) but is permitted by OSA Deltase for reasons of openness.

However, the *implementation* of XPA Deltase differs quite radically from that of OSA Deltase in the following ways:

- Support for high performance: XPA Deltase copies network messages directly between the application address space and the message buffers in the network attachment controller, doing a minimum number of data copies, system calls and context switches. Details of these mechanisms are included in the *Delta-4 Implementation Guide* [Delta-4 1991]
- Support for real-time: XPA Deltase interfaces to a real-time local execution environment, which in the prototype is the *Real-Time UNIX* developed in the first phase of Delta-4, and to the *Collapsed-Layered Communication System* (CLCS, see section §9.6.4). XPA Deltase also propagates the XPA precedence from client to server, ensuring that both are scheduled accordingly. It supports the semi-active replication model, including its thread preemption mechanism (see section §9.4.3.2).

Good system generation tools are required to ensure that compatible versions of both application and system level objects are included in the same system build. This problem will arise more frequently during the development of the XPA prototype because the continuous effort to improve performance is likely to lead to several successive versions of important system interfaces, e.g., the interface between a capsule and the communication system.

Existing Deltase trading mechanisms and system generation practices will therefore be reviewed. If further mechanisms are required for XPA, consideration will be given to run-time checks in which the client presents the version number of the interface specification in its first invocation of the server. This is a marginal overhead that will detect any errors during system commissioning.

9.6.2. Scheduling

9.6.2.1. Scheduling Algorithms. As explained in section 9.1, XPA makes the minimum of assumptions about the scheduling strategy used by the underlying LEX. The consequences of using each of several common strategies in an XPA context are nevertheless of interest and are now examined.

Highest priority scheduling preemptively schedules components in the order of priority. Components that interact with the environment derive their priority from the cost of their potential timing failures, and other components inherit their priority from dependencies.

All XPA on-line schedulers use highest priority scheduling, because it is *stable*, i.e., during overload conditions, where the underlying LEX makes use of highest priority scheduling, the most critical components are still likely to meet their timing constraints although the least critical components may stop doing so. Refinements such as the ceiling protocol [Sha et al. 1987] prevent deadlocks and reduce delays to high-priority components, at the cost of blocking other components more often.

At each priority level, a scheduling algorithm can be used which discriminates between components of the same priority on the basis of their arrival time, targetline, deadline, slack time, period or some other principle. Different algorithms may be used at different priority levels, as recommended by the draft IEEE standard on real-time operating systems [IEEE P1003]. Only the most important algorithms are described below.

Earliest deadline (Targetline) scheduling preemptively schedules components in the order of their deadlines (targetlines).

Targetline inheritance is the policy of assigning a component's targetline to any component on which it depends, e.g., assigning to a server the targetline of its client. *Inherited targetline scheduling* is earliest targetline scheduling with such targetlines.

Targetline calculation is the policy of calculating a targetline for each execution on each computing element, by subtracting (estimated) subsequent execution times from the targetline of the whole distributed computation. For example, a server's targetline is equated to the targetline of its client minus the expected execution time in the client after the server replies. *Calculated targetline scheduling* is earliest targetline scheduling with such targetlines.

Inherited targetline scheduling meets all targetlines on a mono-processor system, if any schedule can achieve this [Halang 1986]. On a multi-processor system, this can only be achieved by NP-hard calculations (e.g., [Zhao et al. 1987]) and inherited targetline scheduling becomes non-optimal. Nevertheless, it performs well in multi-processor experiments [Sha et al. 1988] and is intuitively sensible, favouring those components that are part of the most urgent computations. Also it is simple and does not require accurate foreknowledge of execution times. It is therefore being prototyped in XPA.

Intuitively, calculated targetline scheduling derives component targetlines in a more realistic way and should therefore meet more targetlines in a distributed system. It also performs well in experiments [Sha et al. 1988]. It requires a knowledge of execution times, but this is needed anyway in a provable real-time system.

Several other scheduling algorithms could be used on XPA, but are not included in the prototype because of limited resources. Two examples are:

Rate-monotonic scheduling preemptively schedules independent periodic components in the order of increasing period [Liu and Layland 1973]. On a single processor, rate-monotonic scheduling completes every periodic activation within the period if the worst-case load is kept within certain bounds. It is stable if the more critical components are given the shorter periods.

Least slack time scheduling preemptively schedules components in the order of increasing slack time, (i.e., time till inherited targetline - remaining execution time). Intuitively, it is likely to have similar properties to calculated targetline scheduling.

9.6.2.2. The Prototype LEX

9.6.2.2.1. The Precedence Parameter. A precedence is assigned to every distributed computation and communicated to every component of the computation. The precedence encodes the following information:

- The priority of the computation, i.e., whether it is hard or soft real-time or non-real-time computation, and how costly are the consequences of a timing fault.
- The targetline of the computation, which is normally derived from the deadline (and liveline, if any), or period for a periodic computation.
- A parameter whose interpretation depends on the scheduling algorithm used at this priority level, e.g., an estimate of the remaining execution time in the computation.

Every component of the computation (i.e., every message or process that can delay the computation for any reason) inherits the precedence and may not change the priority field, e.g., a component of a non-real-time computation may not claim that a hard real-time computation depends on it.

Each local scheduler preemptively schedules components in the order of their priorities and components of the same priority in an order that may vary from one priority level to another (see §9.6.2). The default is inherited targetline scheduling, i.e., components of the same priority are scheduled in the order of their targetlines. If two targetlines are equal or differ by less than the context switching time, the scheduler refrains from switching between the two components, which would increase the risk of missing targetlines.

9.6.2.2.2. Interpretation of the Targetline. If a deadline has been specified for a component, the targetline should normally be set at or before the deadline. If no deadline is specified, but the frequency of component activation is specified, targetlines should be set in a way that reflects this frequency, i.e.:

- Each periodic activation of a periodic process can be assigned a targetline equal to the start of the next period. If the periodic processing finishes before this targetline, the process suspends until the targetline — the targetline event unsuspends it and starts the next period. If the periodic activation finishes after the targetline, this is a warning of potential overload.
- A sporadic process, e.g., a process with a known minimum inter-arrival time, can be assigned a targetline equal to the earliest possible arrival of the next activation of the process. Again a missed targetline is a warning of potential overload.

9.6.2.2.3. Prototype Implementation. In practice the precedence parameters have to be converted into scheduling parameter(s) that are meaningful in each local execution environment. In the XPA prototype, this is done as follows:

- In the Real-Time UNIX developed in phase 1 of Delta-4, processes are scheduled in the order of their priority and processes of the same priority in the order of their "deadlines", which are set equal to their inherited targetlines.
- In VRTX32, which will be used in the prototype NACs, processes are scheduled in the order of a VRTX priority in the range 0 to 255, which encodes the priority and the most significant bits of the targetline.
- On a bus or LAN, messages typically belong to one of a few priority levels or latency classes. For example, only two latency classes may be used on the LAN, with only the hard real-time messages being assigned to the higher class.

The communication protocol software in the NAC will be organized on the basis of one thread per message to be transmitted. In the first prototype these threads are not prioritized, but it is intended that each thread should later derive its precedence from the message it manages, so that a high precedence output message overtakes concurrent low precedence output messages. (The group manager threads allow preemption only at certain points in processing to ensure protocol correctness.)

In the destination NAC, high precedence input messages may again overtake other messages; they may also preempt preemptible software components, provided the semi-active and passive replication models are used (see section 9.6.4).

9.6.2.3. Alternative LEX Mappings

9.6.2.3.1. Mapping of Off-Line onto On-Line Schedulers. If there is a static set of processes at one or more priority levels, an off-line scheduler might calculate a fixed cyclic schedule for them. It may be possible to communicate this schedule to an on-line scheduler by setting the precedence parameters to specify the times at which components are to run and the order in which components are to be preferred if they ever conflict for resources.

For example, if the static cyclic schedule is in fact the same as that which would be produced by a rate-monotonic scheduler (see above), the targetline can be set equal to the period, the third parameter of the precedence vector can encode the start time of the first periodic activation and a rate-monotonic scheduler can then reproduce the cyclic schedule planned at design-time by the off-line scheduler.

To reproduce a more complex static schedule, in which long-period processes sometimes preempt short-period ones, it would also be necessary to assign different priorities to the components. However, most static cyclic schedules can be encoded into a priority, period, start time vector for each component. (An exception would be any schedule that contains mutual preemption: *A* preempts *B*, which then preempts *A*.)

We can therefore envisage an XPA system in which the hard real-time components at the highest priority levels are executing according to a static schedule calculated at design time. The remaining system resources could be allocated to lower precedence components by a "real" on-line scheduler, calculating its schedules at run-time in the normal way.

9.6.2.3.2. Other Scheduling Strategies. The third parameter in the precedence vector is included so that XPA can remain open to other on-line scheduling strategies such as rate-monotonic and least slack time scheduling (see above). The detailed format and interpretation of the precedence are therefore hidden in a "precedence manager" object, which exports a fixed interface, e.g., will tell the caller which of two precedences is the higher, but which can be implemented in a number of different ways.

Various other scheduling strategies are described in the literature (e.g., [Johnson and Madison 1974, Minet and Sedillot 1987, Xu and Parnas 1990, Zhao et al. 1987]). Where possible, it is desirable to permit their use in XPA, although they will not be prototyped. The necessary restrictions are:

- All computations at the same priority level must compete using the same scheduling algorithm at each local scheduler.
- All local schedulers of the same resource, e.g., CPU time, should be identical on XPA. This makes it easier to predict timing properties, e.g., bound the desynchronization of replicas.
- Schedulers that perform extensive or NP-hard calculations are discouraged, as they tend to add to the problem they are trying to solve.

One reason for allowing the later introduction of other scheduling strategies is the wide variability of real-time system requirements. For example, a voice management system implemented by Ferranti has the following requirements:

- the start of service provision can be delayed if necessary, i.e., this is a "soft" requirement;
- but as soon as service commences, a fixed proportion of the processing power is required between the start and completion of each service, i.e., this is a "hard" requirement.

In this case there is no definable targetline, but the third parameter in the precedence vector could be used to hold the required proportion of processing power.

Non-real-time processes running at the lowest priority level may also have no "natural" targetline and may be given "fair shares" of any remaining resource. Alternatively, they may be assigned a targetline a fixed period after their activation times, leading to FIFO scheduling.

Figure 3 shows an example local scheduler that is using some of the above options. The arrows indicate the order in which the scheduler examines the tasks; it executes the first which is found to be immediately executable (i.e., not suspended waiting for any resources) and reexamines the list whenever a new task appears or becomes executable. There are four priority levels: one for the hard real-time tasks, two for soft real-time tasks of different criticality and one for non-real-time tasks. The hard real-time tasks are a static set, so they can be scheduled according to targetlines worked out by an off-line scheduler. The targetlines of the soft real-time tasks are equal to their deadlines; and the targetlines of the non-real-time tasks are derived from their activation times, so that they are scheduled like batch jobs.

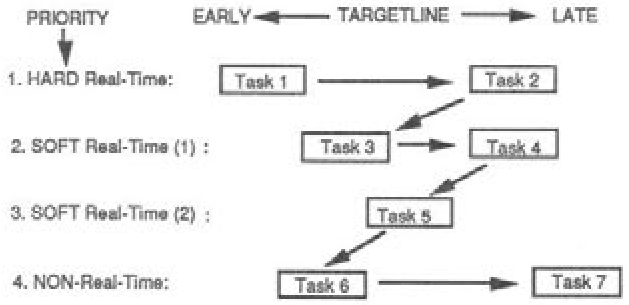

Fig. 3 - Scheduling Tasks of 4 Different Priorities

9.6.3. Relationship to System Administration

XPA system administration is described in section 9.5 and summarised in figure 7. System administration "events", such as missed targetlines, exhaustion of buffer space and changes in the replication status of the object are reported by the group manager to the Deltase communication manager. They are then handled by standard Deltase routines, unless the application code has substituted its own application-dependent handling routine.

Other "events" may take the form of UNIX signals, e.g., from asynchronous device drivers, but these too are handled transparently by Deltase unless the application explicitly substitutes some special handling routine.

All event handlers should preserve replica determinism (see section 9.7.6). A handler need not perform any local processing, but may simply report the event to the invoking client, as in the Ada exception model.

Application requests to read the time, or to be signalled at a particular time, are passed to the group manager and thence to the synchronized local clock, maintaining approximate global time, i.e., the local approximation of the common system time base (see [Kopetz and Ochsenreiter 1987]). For increased precision, clock synchronization is implemented within the communication system. Here the group manager is responsible for replica group determinism; e.g., having given a particular clock reading to a leader replica, it passes the same reading to the follower replicas.

9.6.4. Communication Issues

9.6.4.1. Communication System Design Aspects. XPA uses a communication service (xAMp, see chapter 10) which provides *group communication*, i.e., which permits and encourages communication using addresses that can refer to more than one end-point or entity. Such communication capabilities are inherently useful in distributed systems and especially so where the use of component replication means that some kind of replicated communication must be provided either in the communication system itself or at the application level.

We chose to hide the complexities of using a particular group communication service (with all its various options and subtly differing QOSs) in a *Group Manager* (GM). This is a different entity from the layer in the communication service provider that implements group communication (xAMp). The group manager provides all the services that a user requires to use the xAMp.

This has turned out to be good decision, because the GM can encompass much of the implementation-dependent detail that is necessary for efficient communication within a node and between host and NAC. It also implements most of the leader-follower protocol, the rest being taken care of by a *host-NAC interface library* that straddles the gap between Deltase and the group manager and is responsible for hiding the complexities of the interface from everything else.

The xAMp was specified to be versatile, offering an easy way to address subgroups of participants and a multi-fold communication primitive offering different qualities of service.

The xAMp service is based on the gate group concept. We briefly review the concepts relevant for this discussion (detailed in chapter 10). Each gate group, or *gate*, possesses a logical identifier that is implicitly an address. A message sent to a given gate group will be delivered to all group members. Only a group can be addressed at send time. A message to be sent to two different groups must be sent twice. However, *selective addressing* allows the selection of a *subgroup* of a given group as the recipients of a message. In the xAMp, these subgroups are identified by lists of station names, but logical subgroup identifiers are constructed for higher-level references.

Within a gate group, a receive queue is associated with the gate at every group member. Messages sent using *atomic* or *tight* qualities of service are ordered in the receive queue such that total delivery order is provided to all members. Messages of the same precedence are inserted with FIFO order on the receive queue. However, the position of high precedence messages within the queue can be negotiated when the *tight* quality of service (QOS) is used.

Potential causal order is preserved for messages exchanged in the system, within the same domain of causality, which is identified by a label (*clabel*) independently of group membership. This service is provided by *atomic* or *tight* QOS. Messages of different precedence must use different *clabels*.

9.6.4.1.1. Mapping between Components and Gate Groups. We now discuss how high level components can be mapped on to gate groups. The discussion is needed since a designer using xAMp has several options that can be expressed by the following questions:

(1) May a component belong to several gate groups, or should the membership be restricted to a single group?

(2) May several components belong to the same gate group or should the group membership be confined to several replicas of the same component?

The way each question is solved by the designer has consequences in terms of the degree of parallelism, concurrency and order relations offered within the communication layer.

Question (2) concerns the *visibility* of multicast, also discussed in section 8.1. In the following discussion we assume that a component possesses a logical identification. A component can be replicated, having several replicas in different stations, but it is still the same logical component, since all replicas have the same identity. Communication with a group of these replicas is such that their existence is hidden from the higher MCS layers, thus it is termed *invisible multicast*. On the other hand, different components can be seen as forming a group of different cooperating entities. The leader-follower model is an example of such a component since each participant can be addressed individually (the leader, the first-follower, etc.). In this case, we have *visible multicast*. Participant visibility can go all the way up to the top system layers, if so wished.

To emphasise that components do not directly call xAMp, we will give different names to the primitives called by the high level components: when a component desires to receive the messages addressed to a logical name *lname* it will call an *OpenRec (lname)* primitive. To send messages addressed to a logical name *lname*, it will call an *OpenSend (lname)*.

9.6.4.1.2. Managing Concurrency. With xAMp communication primitives there are two different ways of achieving concurrency in the communication:

- The first is to associate different receive queues to the same component, letting the component open more than one gate. In this procedure, no causality relations can be assured between messages received through different gates.
- The second is to use *clabels* to establish causality paths while communicating through the same gate. This approach obliges the use of a single group, but it allows the users to incur the expense of achieving causal delivery only when required, without compromising efficiency.

The first option is appropriate for the model of causality based on RPC and leader-follower replication that has been described. The second option is made use of when other interaction constructs must be used; for instance where Casts must be received in potential causal order. These observations clarify the designer options to solve question (1).

9.6.4.1.3. Interactions between Single Components. Let us consider the case where components are only addressed individually and no identification is given to groups of components. If a message needs to be sent to several components, one copy is sent to each component. Components can still be replicated and replicas are still addressed transparently.

In such a case, a component can receive messages through a single gate and there is no need for several components to be mapped onto the same gate. There is a one-to-one relation between components and gates. Logical order of all messages exchanged (through the appropriate QOS) is preserved in the system. Since there is a direct mapping between components and gate groups, there is no need for any extra translation protocol.

With this configuration there is a direct mapping between an *lname* and *gate address*. The translation between *OpenRec* and *GateOpen* primitives and between *OpenSend* and *AttachGate* is also trivial.

9.6.4.1.4. Interactions between Groups of Components. Let us now consider the case where cooperating components need to be addressed both individually and as a group. Components will then be grouped and an identification will be given to each group.

If total order relations need to be verified among the messages exchanged by a set of components — and this includes both the ones directly addressed to a component and those addressed to one of the groups to which it belongs — all the messages must be received by each component through the same receive queue. This answers question (2) put earlier: group membership cannot be confined to several replicas of the same component since, when total order relations need to be preserved, cooperating components must be mapped onto the same group.

It will then be impossible to map directly component and group identifiers into gate group addresses since several logical names will be sharing the same gate address. So, some protocol must be added on top of the xAMp layer, executing the necessary mapping.

With this configuration, there are still two different strategies to disseminate the messages to the appropriate components. The first is similar to the one used in OSA before xAMp was available, where frames were addressed to all gate group members. At the upper layers the local existence of the logical name addressed was checked and the frame discarded or delivered to the addressed component. The major advantage of this method was that there was no need to run agreement protocols to open or close a logical name at a given station: as long as the gate group already existed the operation was purely local. The disadvantage was obvious: at the AMp layer the agreement was run among all the group members even when only a small subset needed to receive the frame. This was a severe performance penalty, so a second alternative was devised, as explained below.

Since xAMp provides a selective multicast service, which allows the frame to be addressed only to the relevant stations, we explored it to achieve better performance. If the sender is able to know, at transmission time, the identification of those stations where a given logical name exists, it can use the appropriate selective multicast field in any xAMp primitive to send the frame just to the relevant stations. There an extra lateral cost, since now an agreement protocol must be run to disseminate the information about open and close actions. However, since the environment changes are supposed to be relatively few in relation to the total number of messages exchanged, this can represent a substantial performance improvement.

9.6.4.1.5. The Use of Selective Multicast. We now summarise a simple protocol to support the use of selective multicast, when several components share the same group. An elaborated form of this mechanism was developed for the OSA architecture after the adoption of xAMp as a common component of the architecture.

On top of the xAMp layer the protocol keeps a table with two fields: *lname*, *listOfStations*. The table is just a conversion table, mapping logical names into the lists of stations where that logical name exists.

When a new station enters a group, and also to support the cloning protocol described in section 9.7.9, consistency must be maintained. All group members must have the same view of the table and a new station entering the group must obtain this view. The agreement protocol for consistent subgroup management depends on the *atomic* service of xAMp. Each time a component issues an *OpenRec* request, the protocol will broadcast to all the group members an *enter* message carrying the station identifier and the logical name to which the component becomes associated. When this message is received, a new entry in the table is created or, if the entry already exists, the station is added to the list. Each time a component closes the receive queue a *leave* message is propagated in a similar way, throughout all the group members.

When a message must be sent to a given component, the protocol just fetches the appropriate xAMp selective multicast list and requests the transmission to the xAMp layer.

9.6.4.2. Flow Control. Flow control is the problem of coping with limited buffer space. For hard real-time components, this can be achieved by sizing the system so that buffer space will never be exhausted. For other components, flow control can be achieved by blocking producers of messages when it threatens to be exhausted. Three types of component can be distinguished: hard real-time, soft real-time and non-real-time. In the XPA prototype, they are mapped onto different priority levels.

Note that the proposed mechanisms for flow control management have not yet been implemented in the first (1990) XPA prototype.

9.6.4.2.1. The Hard Real-Time Priority Group. Hard real-time components are assigned to the highest priority level. If we are to prove they meet their timing constraints, they must not be blocked for lack of buffer space.

The maximum buffer requirements of these components must be determined at design time and these buffers must be *reserved* for components in the hard real-time priority level. Components with a lower priority must *never* be allowed to use the reserved buffer space.

So a hard real-time component cannot be blocked for lack of buffer space, provided conditions remain within the operational envelope (see chapter 5). Outside this envelope, two things may happen:

- A hard real-time component may seize free space normally used by lower priority components.
- If all else fails, the producer component receives an event indication that offers it a chance to limit the damage in some application-dependent way. This is analogous to the missed targetline event and might be serviced by a similar event handler (see section 9.3.5).

Since hard real-time components are supposed always to have buffer space, inaccessibility for those components is not defined. In other words, it equals failure. In consequence, an xAMp message is never rejected due to inaccessibility.

9.6.4.2.2. The Soft Real-Time Priority Group. Soft real-time components that are not part of the hard real-time subsystem may be denied buffer space even when conditions are within the normal operational envelope. However, it is important to make this an unusual event and therefore buffer space is reserved for the soft real-time priority levels, in the sense that non-real-time components may never use the reserved space.

When a soft real-time component cannot be given a buffer in its reserved space, two things may happen:

- A soft real-time component may seize free space normally used by non-real-time components.
- If all else fails, the producer component receives an event indication that offers it a chance to limit the damage in some application-dependent way. This is analogous to the missed targetline event and might be serviced by a similar event handler (see section 9.3.5).

When an xAMp message for a soft real-time component cannot be received in the destination NAC because of a shortage of soft real-time buffer space, the destination component is said to be inaccessible. The transmission may be retried later, but if this condition persists beyond pre-defined limits (duration, rate), it must be considered failed, either by itself, or through an action of system administration. When using the *reliable* QOS, that happens upon the first occurrence of inaccessibility, by definition. (In normal cases, the system administration module in the group manager is informed of the buffer shortage before it leads to such failure and may be able to avert failure by denying buffers to non-real-time components.)

9.6.4.2.3. The Non-Real-Time Priority Group. Non-real-time components may not use the reserved buffer space of real-time components and the buffer space that they normally use may be seized by these higher priority components in exceptional conditions. It follows that all non-real-time components on an XPA system must expect to be blocked for lack of buffer space, or lack of other resources, and it may be difficult to predict their timing properties.

Although non-real-time producers may be blocked, it is not proposed that non-real-time messages should ever be discarded to release buffer space. This would lead to incorrect operation or abandoning of non-real-time components, which may in fact be as costly as a loss of timeliness in real-time components.

If there is no non-real-time buffer space for a non-real-time message to be received at a destination node, the destination component is said to be inaccessible. If the message was transmitted by *atomic* multicast, the multicast fails and may be retried after an interval. If the message was a *reliable* multicast, the destination component is deemed to have failed.

Non-real-time applications may, if required, be alerted by the missed-targetline or no-buffer-space events. An application might handle these events by explaining to a terminal user why it is delayed.

9.6.5. Computation Issues

9.6.5.1. Causality in Component Interactions. Deltase supports the Remote Procedure Call abstraction, in which object interactions are synchronous RPCs involving the passage of a global thread of control. Global threads are represented within the single address-space of a component replica by lightweight threads, which are scheduled deterministically, at well-defined points in the code. In figure 4, objects A, B, C, E and F are shown supporting multiple computations in this way.

Figure 4 also illustrates the facility that is provided to objects for a "family" of threads to come into being on behalf of the single global computational requirement (labelled iii in the diagram). Within object C, a "parent" thread has created "child" threads of control that subsequently die within that object, but in the interim are despatched through RPC to objects E and F. If the components supporting these are located on distinct hosts, simultaneous computational progress may be achieved; local "pseudo" parallelism can therefore give rise to distributed "true" parallelism.

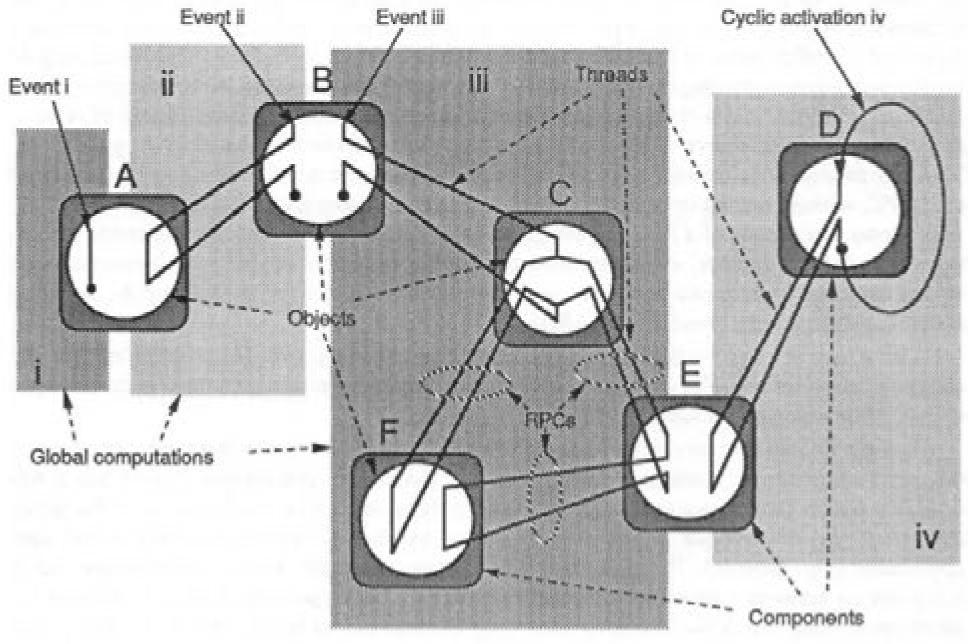

Fig. 4 - Distributed Computations

Apart from the purely within-component issue of birth and death, global threads that are "related" in this manner are causally-independent of each other during their life. In figure 4, this applies to those arriving at object F, which are just as independent of each other as those arriving at objects A, B and E. Causality follows each individual thread of control, as it animates different objects. Therefore, upon arrival at any one component, the messages that represent these threads, even from the same sending component, are necessarily independent if they are concurrent. They are viewed as "competing" and may be freely re-ordered by the component according to precedence. The machinery is described below.

9.6.5.2. Causality and Replica Determinism. XPA inherits the Delta-4 Open System Architecture lightweight thread scheduling. This is inherently replica-deterministic; it always takes place at well-defined points in code and the same data is used to take the same scheduling decision.

However, the group of replicas must reorder messages identically, and must permit these to sometimes cause preemption identically. The State-Machine approach to replica-determinism [Schneider 1990] used by the Delta-4 Open System Architecture cannot be used, since in effect it bans preemption. To overcome this, XPA has introduced the leader-follower, or semi-active replication model [Barrett et al. 1990].

Since stations are fail-silent, it is possible to permit all decisions on potentially non-deterministic behaviour to be taken by a privileged replica, the leader. Until it fails, this replica is in charge of ensuring the consistency of its replica group, both with respect to the order of receipt of messages that represent RPCs and with respect to any other potential causes of non-determinism.

9.6.5.3. Message Handling. The group manager takes messages from the network that are intended for recipients on its node, interprets their addresses and passes them on to the addressees. It ensures that messages with the same precedence and from the same source are delivered in FIFO order to the queue of messages passed up to Deltase. The processing of higher precedence messages is allowed to proceed at the expense of lower precedence messages, but FIFO order is maintained within a "precedence class", so that a series of reliable casts from the same source at the same precedence are delivered in the order of generation. Thus, for example, the various messages of the XPA cloning protocol can be sent by cast rather than RPC, without having to ensure that their order can be deduced from their internal content. This allows precedence of a thread of computation to migrate across the communication system from one object to another, within the limitations of the number of precedence classes provided by the underlying network hardware and the order-preserving properties of the *reliable* communication service provided by xAMp.

The group manager takes messages from local senders and sends them on via the communication service. The communication service is used even to send purely local messages so as to obtain the precedence properties that it provides.

The group manager consults a *precedence management* function to translate and compare the precedence classes associated with a message; the various precedence classes that it has available for its own computations (from the underlying execution environment of the group manager); and the various precedence classes of execution that are provided to the user application (i.e., Deltase). The precedence management function allows, for example, fuzzy comparisons between a message precedence that consists of a (priority, deadline) pair and the precedence with which the currently executing Deltase thread is running on the host. This comparison is then used to decide whether or not to interrupt the current Deltase processing in order to pass the (possibly more important) message to another thread or object. The comparison is not straightforward because it may, for example, use a cost-function which takes account of the time required to interrupt and schedule another thread. This may "cost" more to do, perhaps increasing the likelihood that one or both threads of computation may miss their deadlines, than simply letting the first thread run to completion, even though the new message has a slightly higher precedence when measured in the absolute sense.

As well as handling application messages, the group manager must also produce and process messages that are generated as part of the leader-follower protocol and administration messages controlling the creation, destruction and management of communication resources.

Concerning precedence management, it is mandatory for an RPC of highest precedence to cause bounded-latency preemption of lower precedence computation. The message must not only "jump the queue" on arrival; processing must begin at the same point in all replicas so that they do not diverge in state. Figure 5 illustrates the following solution to this problem:

Preemption points are small units of code, pre-installed in the object code a bounded execution distance apart (shown in figure 5 as small lines crossing the execution-path of the threads). Very few instructions are executed in the normal (non-preempting) path through a preemption point, which permits their distance apart to be small without significant overhead. A count is maintained to identify arrival at a preemption point; because of both deterministic lightweight thread scheduling and the machinery for handling asynchronous events about to be described, the same count (N-1, N, N+1, ... in figure 5) is assured to identify the same point in execution across replicas.

When the leader arrives at a preemption point, code examines a preemption flag and normally continues computation if it is not set; periodically, it sends "continue from N-K to N" instructions at such points (K is a constant and N-K is where the last instruction was issued). Such instructions consume very little network bandwidth and assist the follower to keep up or detect desynchronization. The set of such instructions provides a continuous description of execution requirements. When a follower arrives at such a point, the code compares its own preemption point count with the value supplied by the leader in the instruction currently being followed, and proceeds if it is less. When it arrives at an identified point, it obeys the leader's instruction to proceed or divert to accept a message (figure 5). Note that the leader must stay one instruction ahead; if this has not been sent, the follower blocks for a bounded time at the next preemption point beyond those for which it has received instructions.

When a message containing a request for service arrives at a station, its precedence is compared with that currently occupied by the destination replica (leader or follower), which is running with at least the precedence of its current computation. If the message precedence is higher, the replica is immediately given that precedence, and the preemption flag is set. If that precedence exceeds that of any other computation on that station, the replica gains exclusive access to the execution resource, and the execution distance to the next preemption point is then bounded in time.

The leader is diverted by this flag at its next preemption point, and will then select the highest precedence waiting message; this selection, together with the preemption point count when diverted, is sent to the followers. The service required is then invoked by running a local lightweight thread to temporarily represent the global thread in that replica, and will be identically invoked at the followers.

The group manager at a follower blocks the replica when it requires to read an input message until the corresponding notification has been received. Similarly when a replica requires to produce an output it is blocked until the notification confirming that the leader has sent it has been received. It would be possible to buffer output messages in followers, thus allowing computation to proceed, but for the moment we believe that the complications in message buffer management that this would entail do not justify the benefits to be gained. The argument is that the follower must be only a little way behind the leader, or the application would fail to meet its deadlines if the leader fails. There is thus very little to be gained by buffering output messages except programming complexity and probably reduced efficiency.

Follower replicas are also blocked by their group managers when they reach a preemption point for which a preemption notification has not been received from the leader. All such interactions between group manager and replicas (notification of reaching a preemption point; suspension/resumption of a replica, etc.) are handled by the *host-NAC interface library*. In the present implementation, this uses shared memory to reduce multi-processing context switches and a UNIX device driver for preemption of Deltase threads.

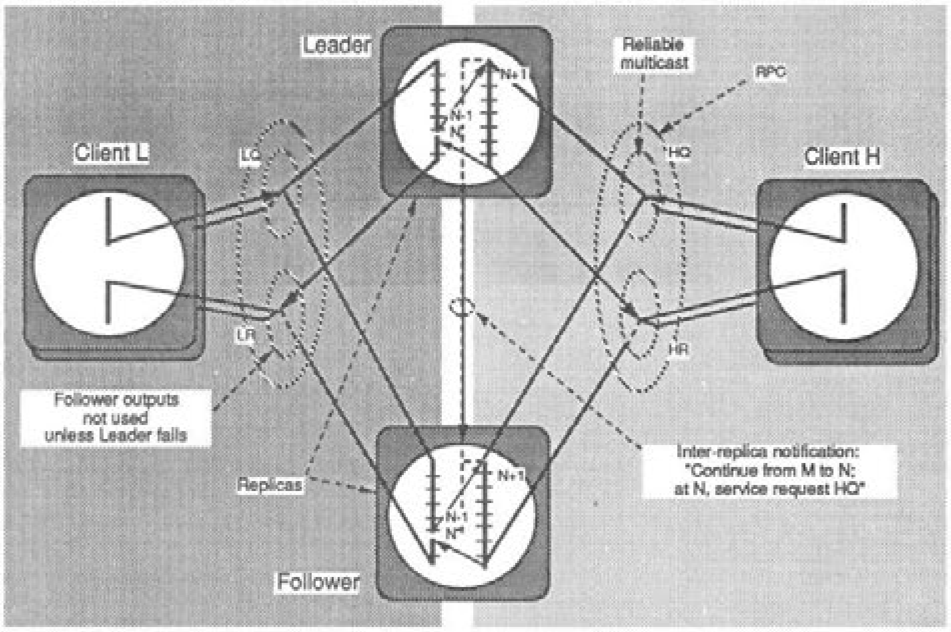
Leader
N+1
N-1
N
Reliable multicast
RPC
Client L
LQ
HQ
Client H
LR
HR
Follower outputs not used unless Leader fails
Inter-replica notification: "Continue from M to N; at N, service request HQ"
Replicas
N+1
N-1
N*
Follower

Fig. 5 - Replica Group Determinism, Leader-Follower Model

9.6.5.4. Supporting the Leader-Follower Protocol. The group manager relies on xAMp to inform it when a node containing a leader replica has failed, so that a follower must therefore become a leader. It is sufficient for the group manager of the follower that is about to become a leader to pass the information that the follower must become a leader in the normal flow of notifications to the follower. The follower can take appropriate action when it processes the notification (this is done within the host interface library). This assumes that correct ordering of messages and notifications is maintained within the group manager and that xAMp co-operates with the group manager by providing the indication that the leader node has failed in a suitable manner.

To support the leader-follower protocol, the group manager must support the following assumptions:

- total order of inputs to replicas;
- messages with the same precedence class do not overtake messages with the same or a higher class; i.e., FIFO — and therefore causal — ordering of messages with the same precedence and source is not violated;
- a similar consideration applies to notifications as messages.

The group membership management service for the *reliable* xAMp QOS must provide the following:

- notification of node failure (i.e., a group change indication) must be delivered to a node *after* any delivered messages from the failed node.

9.6.5.5. Replica Synchronization. With all forms of replication on XPA, the fastest replica will normally output messages to the network and to any non-replicated device. The role of the slower replica(s) is to provide a correct and timely service if the fastest replica fails. To provide this timely service, they must be synchronized with the fastest replica to within a bounded time.

Badly desynchronized replicas also present other problems: the slower replica may appear to have failed silently, or its input queue may overflow. When a follower replica reads the present time, it is given the value that was given earlier to the leader, to preserve replica group determinism; however this value is inaccurate unless replicas are well synchronized.

9.6.5.5.1. Detection of Desynchronization. One of the problems is to detect excessive desynchronization. Here a natural mechanism occurs as part of semi-active replication. A stream of notifications passes from leader to follower, indicating the results of non-deterministic decisions and successful transmission of the leader's output messages. Preemptible capsules send notifications of the form "Continue from N-K to N", with bounded inter-notification delays, which indicate that the leader recently passed preemption point *N*.

A follower's group manager can thus detect excess replica desynchronization. If the "Continue from N-K to N" notification is time-stamped *T* (in the approximate global time of the leader's node) and the follower has still not reached preemption point *N* at time *T+X*, (in the approximate global time of the follower's node), where *X* is an appropriate desynchronization event "trigger", the follower's GM raises the desynchronization event to both leader and follower.

This is a local event (see section 9.7.4.1) since it is handled differently at the two locations. The follower event handler reports the event to the local system administration object responsible for station management, which may merely log the event, or may initiate some long-term solution such as load-balancing. The leader event handler is discussed in the next section.

However, the follower may not only be desynchronized, but may have failed silently, e.g., because of a Heisenbug (software fault that manifests itself independently in different hosts) that causes it to fail at a point where the leader proceeded correctly. It is therefore necessary to define a time-out *F* larger than *X* and to raise the follower-failed event if the follower has still not reached preemption point *N* at time *T+F*. Such a follower is incapable of timely back-up of its leader, so it is correct to deem it failed, whatever the reason for its tardiness.

9.6.5.5.2. Synchronization Techniques. Two types of synchronization technique have been identified: lightweight probabilistic mechanisms that usually prevent the desynchronization exceeding *X* and, when the desynchronization is greater than *X* but less than *F*, a reliable protocol to keep the desynchronization below *F* unless the follower has failed.

Lightweight probabilistic mechanisms include:

1) Replicas should be allocated to hosts so that each follower (or slower replica) has as much processing power and other resources as its leader. For example, one host could support a group of leaders while an identical host supports the corresponding followers.
2) However, even with such a distribution of components, variations because of factors such as different disc interrupt times may cause some followers to finish early and others late, compared with the corresponding leaders. Those which finish earlier than the targetline, or which catch up with their leaders, should then suspend until the targetline or arrival of the leader's notification. This releases the spare resources for re-allocation to the late followers on the same node.

3) Synchronization of semi-active replicas can be optimized by conferring the role of leader on the slowest replica, i.e., on the replica that has fewest local resources such as processing power and buffer space. The followers will then complete most operations more quickly than the leader and cumulative desynchronization is unlikely.
4) A follower in fact requires slightly fewer resources than its leader, since it does not normally output network messages or perform real actions (see chapter 12). These factors help a follower keep up with its leader, although they are difficult to quantify.
5) When a high-precedence input message enters the input queue of either replica, it depends on the replica's current processing, which is therefore allowed to proceed at the high precedence of the pending input. If the follower is desynchronized, it will experience this "speed-up" effect for longer than the leader.

The protocol that ensures bounded desynchronization is as follows. The desynchronization event causes the leader and follower to switch roles, and the old leader then waits for the new leader (i.e., old follower) to catch up with it and send it an instruction. The new leader will do this unless it fails, which is detected when it reaches the failure desynchronization F (see above), if not sooner.

This strategy ensures resynchronization and tends to reduce the frequency of desynchronization events. Since the former leader is running on what was recently, and may still be, a less heavily-loaded host, it is likely to keep up with the new leader.

This strategy does not damage the timeliness properties of components where a design time proof of timeliness exists, since any such proof must apply to both replicas. However, in circumstances for which there is no such proof, the follower may miss a deadline that the leader achieves, so that role-switching can increase the probability of untimeliness. Some users may therefore prefer some other handling of the desynchronization event, and it may be necessary to offer applications a selection of desynchronization event handlers, plus a description of their properties.

Other synchronization mechanisms have been considered but rejected, e.g., it is undesirable to raise the precedence of a follower to speed it up, because this will adversely affect higher precedence components competing for the same resources. There could follow a useless "inflation" of many component precedences in an overloaded node. Before precedences are changed, there has to be a careful analysis to prove that the side-effects will be acceptable.

9.6.5.6. Latencies. As explained in section 9.5.2, a real-time system must bound all system latencies. This affects all parts of an XPA system, as can be illustrated by considering the various latencies that occur during a remote procedure call. Figure 6 shows the request phase of an RPC from a top-precedence client. Control paths are shown in solid lines and data paths in dashed lines.

The RPC starts in the client application. XPA Deltase copies its parameters directly into message buffers in the network attachment controller. For a top precedence application, which will not be preempted, this takes a bounded time, depending on the size of the parameters.

The client application then interrupts the NAC to draw attention to the output. There are then a bounded interrupt latency and context switching time and a bounded queuing time for a top precedence message, before the group manager processes the message.

On a token ring, there is a bounded latency before the top precedence message can be transmitted, since other stations may acquire the token first and transmit a bounded number of other messages, each of bounded length. Some LANs may entail unbounded latencies; a standard Ethernet, for example, may be unusable in a hard real-time context.

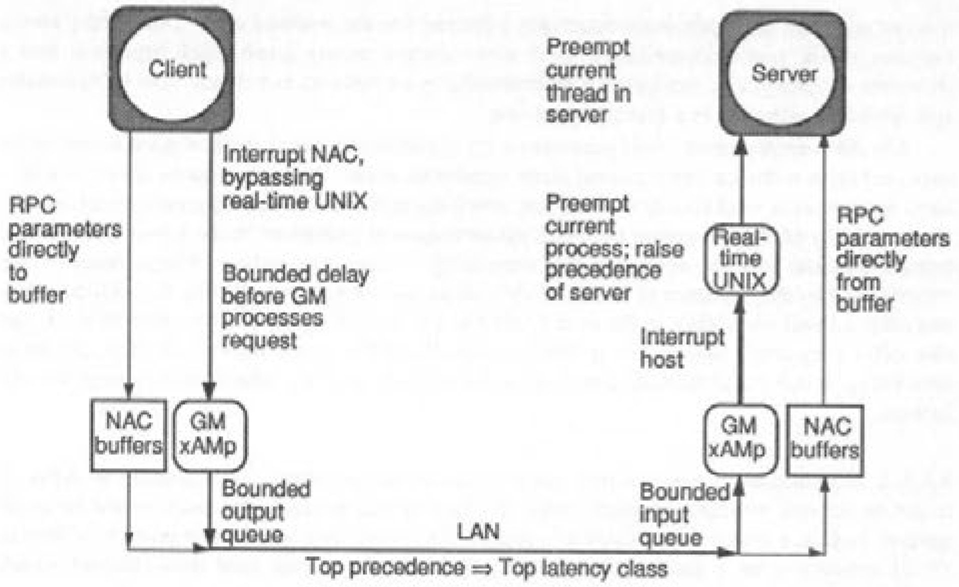
Client
Server
Preempt current thread in server
Interrupt NAC, bypassing real-time UNIX
RPC parameters directly to buffer
Bounded delay before GM processes request
Preempt current process; raise precedence of server
Real-time UNIX
RPC parameters directly from buffer
Interrupt host
NAC buffers
GM xAMp
GM xAMp
NAC buffers
Bounded output queue
LAN
Bounded input queue
Top precedence ⇒ Top latency class

Fig. 6 - Request Phase of a Remote Procedure Call

Communication latency must be bounded, at least for messages that support hard real-time services. The basic XPA communication system provides a choice of communication services, with different tradeoffs between latency and reliability (see section 9.6.4). RPCs use the Reliable QOS, which involves no overheads to ensure total or causal order (see section 10.4).

When a top-precedence RPC request arrives at the destination NAC, the xAMp and group manager should process it in a bounded time. There are then more delays whilst the host is interrupted and its current process preempted. These are bounded under the Real-Time UNIX developed in Delta-4 Phase 1 [Bond 1987, SVC200]. The latencies involved in preempting a particular server and diverting it to the highest precedence thread, are bounded for the semi-active and passive models of replication.

However, in the worst case the fastest replica of the server fails. The latency that then occurs depends on failure detection (except for active replicas) and on the synchronization of replicas. Methods for bounding desynchronization are discussed in section 9.6.5.5. A follower can detect leader failure in a bounded time if the leader's environment generates an "I'm alive" notification to the follower whenever there has been no other leader-follower notification for a preset period. A similar technique can be used with passive replicas.

If this is a top precedence RPC, the server inherits its precedence and runs without being preempted until it replies. The reply message suffers similar delays to the request message.

9.6.6. The Time Service

In real-time computing applications such as those XPA will support, a facility to measure durations and the position of an event relative to the environment, in the metric of the external real time, is a major demand. Furthermore, in distributed real-time control systems, it is often required to measure the duration between two events that have been observed by two different nodes, or to specify actions to occur at several places in the system at a given absolute time (these operations may require a high degree of precision). It is also often required to establish comparisons with the real world time, with the assistance of some external time metric, not always with the high precision standards required for the internal time. Due to the above reasons, clock synchronization in XPA must always ensure good clock precision and a moderate accuracy, i.e., clocks must be internally synchronized, but should also be externally synchronized, although in a less stringent way.

The XPA environment itself possesses a set of characteristics that allow good results to be expected from software implemented clock synchronization. The main reasons are the use of a local area network with broadcast facilities, which eases the task of message dissemination, and the possibility of implementing the clock synchronization procedure in the lower levels of the communication system, reducing the variability of message delays. These features are materialized by the existence of cheap reliable broadcast services provided by the xAMp, which can offer a small variability in the time needed to execute the protocol (execution time). It can also offer very small inconsistency time, which allows the use of the broadcast primitive to simulate a quasi-simultaneous event or to disseminate a clock adjustment throughout the system.

9.6.6.1. Discussion. Since external clock synchronization needs consideration in XPA, it might be argued whether a specific internal clock synchronization protocol would be at all needed. In fact, a protocol like Cristian's probabilistic clock synchronization protocol [Cristian 1989] appears to be a good solution to the problem of external (and thus internal) clock synchronization.

However, we may cite two drawbacks of Cristian's protocol, as far as XPA is concerned:

First, unless the system designer is able to afford a great expense in traffic to achieve synchronization, the reading delay must be chosen with a value such that there is a good probability p of achieving the desired precision. So, although Cristian's protocol does not depend on the variability of the message delay, it depends, for practical implementations, on the distribution of these delays. In particular, it depends on how far the expected delay is from the minimum delay. If the sources of external real time are not assumed to be fail-silent, some kind of agreement must be reached before the value is read. This can take the average delay far from the minimum delay thus reducing the accuracy of the synchronization protocol.

Second, a direct use Cristian's probabilistic clock reading method, as suggested in [Cristian 1989], is extremely dependent on the availability of a source of external real time. Internal clock synchronization algorithms, in presence of any number of crash faults, will keep the surviving clocks synchronized (if faulty processors remain in the right proportions to correct ones), which may be enough for many applications. Given that XPA is supposed to support a number of applications, of varying needs, the cost of having an available and reliable external source [Cristian 1989] would be undesirable for some of them.

To avoid the total dependency on the presence of an external real time source, we propose a configuration where virtual clocks are both internally and externally synchronized.

To synchronize clocks internally, an approach similar to Babaoglu's clock synchronization procedure might then be envisaged [Babaoglu and Drummond 1987]. However, in XPA, group communication is achieved through the use of gates, reducing the number of *Full Message Exchange* rounds to those on behalf of applications using the *Multicast Group of Stations* (MGS). In Delta-4, MGS is mainly used by system administration. These applications are not expected to produce traffic with the frequency required to satisfy clock synchronization needs. Furthermore, to achieve *approximate timed simultaneity*, a need expected to be frequent in a real-time distributed system, an appropriate protocol must be run. The protocol proposed by Babaoglu (DemandAts) has many similarities with a clock synchronization protocol proposed by Srikanth and Toueg [Srikanth and Toueg 1987]. We analyse next this protocol and assess its adequacy for our purposes.

Srikanth's clock synchronization protocol possesses several advantages, namely: it achieves optimal accuracy; it is easily tuned to tolerate different kinds of faults; it is easy to implement. However, being a convergence non-averaging algorithm, the precision obtained depends on the worst-case message transit delays. This is a severe limitation that needed to be overcome to gain the maximum profit from the XPA architecture.

Kopetz's *Clock Synchronization Unit* (CSU) [Kopetz and Ochsenreiter 1987] could improve protocol performance and would reduce the computational cost of the clock synchronization protocol. However, the performance obtained by the CSU is not necessary in the envisaged applications. Furthermore, the CSU has the commercial drawback of being a single source component. The use of the CSU will thus be avoided.

The XPA clock synchronization protocol relies on the NAC fail-silent characteristics. However, the underlying physical clocks do not need to be fail-silent. When the physical clocks are assumed to exhibit an arbitrary behaviour, a majority of correct clocks will be required to run the protocol, i.e., a total of $2f+1$ clocks will be needed to tolerate f faults.

For internal clock synchronization in XPA we developed a clock synchronization algorithm that exploits the intrinsic characteristics of broadcast networks. The algorithm is based on a new variant of the well-known convergence non-averaging technique, dubbed *a posteriori agreement*. The precision achieved by the algorithm is not limited, as opposite to most of the published works, by the variability or worst case values of network access delays. Furthermore, our solution does not require the use of dedicated hardware.

The algorithm, that is described in detail in [Rodriques and Verissimo 1991], achieves clock synchronization using broadcast messages to generate simultaneous events in the system. In most existing broadcast local area networks, frame transmissions arrive almost simultaneously at the successfully receiving sites. This property is exploited by the a posteriori agreement clock synchronization to achieve precision. Note that occurrence of faults may prevent a given broadcast to generate a simultaneous event. We use acknowledgments to detect the effects of faults and several simultaneous events can be generated on a single resynchronization. The agreement for the appropriate simultaneous event — to be used as the base of a new virtual clock — is thus executed *after* the generation of the event, thus the name "a posteriori".

Virtual clocks provide a continuous function being obtained from the value of the local physical clock and the adjustments from internal and external synchronization. The adjustment term for external synchronization may remain null when not implemented. This allows the use of the clock synchronization service in XPA systems where no sources of "external" real time are available. Several solutions will be studied for external synchronization with different fault assumptions, since the use of a non-fail-silent source of real time may be interesting for economic reasons.

9.6.6.2. Primitives Provided. The aim of a clock synchronization service is to provide a synchronized virtual clock. The more important primitive is then *virtualTime*, which returns the value exhibited by the virtual clock. Since the virtual clock will be based on a local physical clock a primitive is provided to set its value during initialization (*setLocalClock*) and to read its value (*physicalTime*). The virtual clock is "started" with the *startVirtualClock* primitive, which initiates a clock locally and includes the node in the synchronization protocol.

9.7. System Administration

This section is concerned with areas of difference between system administration in OSA (OSA-SA), see section §8.2 and system administration in XPA (XPA-SA). XPA-SA provides an augmented subset of the services provided by OSA-SA. Although it is exclusively concerned with the leader-follower replication model for capsules and can take advantage of simplifications resulting from use of this single model, the administration consequences of the high performance and real-time features of XPA must be explicitly addressed. Thus, using the terminology of [Le Lann 1989], many of the additional XPA-SA services and mechanisms are concerned with the management of the time domain (timeliness properties) as well as the logical domain (correctness properties)

XPA-SA provides similar network management services to OSA-SA. These are fully described in section 8.2 and may be classified as the management of events that directly affect the logical domain, such as:

- the initial Delta-4 system configuration;
- system startup operation;
- station failures;
- operational changes to the system configuration.

XPA-SA additionally provides services to manage and attempt to minimize, the effects of faults and overloads in the time domain. Lack of resources (e.g., exhaustion of a finite number of computational threads, or message buffer space, or processing power) is manifested as failure to meet deadlines. The treatment within XPA-SA consequent upon the detection of such an event depends upon what sort of *event handler* has been provided for the object concerned; because of the significance of event handling in the context of the discussion of chapter 5, this forms a large part of the body of this section.

Some OSA-SA services are not relevant for XPA-SA or have a changed purpose. For instance, there is no need to collect statistics; nevertheless functions that permit pre-delivery tuning, verification of design-time deadline assertions and so on, will prove essential.

XPA-SA is biased towards logically distributed rather than logically centralized administration activity; separate concerns are where possible separately encapsulated. Such object-orientation is of particular importance in the target market areas, where individual projects are sometimes required to implement specialized administration paradigms. There is a twofold basis for this: substitution of individual system administration components and local-to-object thread-related event management.

System administration components are deliberately structured as a distributed federation to minimize the granularity of substitution. Individual system administration components may be substituted provided the subset of SA interfaces which are "presumed always to be present", together with the object design-level assumptions about their use, are preserved. Examples of such components are discussed in several sections below. The semantics associated with these interfaces may thus be extended, and in some cases reinterpreted, according to the needs arising in particular projects.

Designers and applications programmers are encouraged to take advantage of XPA mechanisms that permit events to be intercepted and managed within objects wherever this is appropriate. Application-specific knowledge may thus be brought to bear in order to achieve a necessary performance or timeliness requirement. Particularly in the case of a missed-deadline event, such knowledge and such local-to-object management may be essential to the design. Section 9.7.4 below discusses the event-handling mechanism chosen.

The processing required for the complete implementation of all SA functions may not be fully automated. The involvement of a human administrator will often be necessary in order to initiate and implement the system administration function.

As standard, a very small kernel of general-purpose functions is provided, largely derived from the equivalent work under OSA-SA. These constitute an object-library, implementing alternative application-level SA objects that provide interfaces "presumed always to be present" and a support-library of local-to-object SA event management mechanisms. Together, these provide administration paradigms of general value, which will be maintained and added to as additional paradigms are recognised and implemented. The library will always maintain upwards compatibility.

The original system configuration establishes the replication domains of each capsule as an ordered list of stations available for cloning. Replicas are instantiated on these stations in this order, up to the defined replication degree. The order of instantiation establishes an initial *"takeover order"* for replicas; which replica is the leader, which is the second (follower) and so on. When a station fails, promotion of followers is in this takeover order. The station chosen for instantiation of a new clone is the first free station on the list of the replication domain. The new clone is given the last position in the takeover order, now free as a result of promotion. Similarly, when a station is repaired and restored, it appears last on the ordered list.

The takeover order therefore evolves with time; after a number of node failures and repairs, the takeover order may bear no resemblance to that originally established.

In XPA, variants on this model may well be desirable in implementing a particular system; this can be achieved by replacement of selected SA components. For instance, it may be desirable to restore an initial configuration when this becomes possible through repair, perhaps to restore the set of assumptions taken during design with respect to timeliness.

9.7.1. The Structure of XPA-SA

Section §9.6.4 describes the collapsed-layered communication stack and group manager that replaces the OSA 7-layer communication stack and SMAP. This design is a consequence of the XPA performance and real-time requirements. The initial stages of fault tolerance are handled within the XPA group manager and the XPA-Deltase support environment by means of (error) event-handling code. All such event-handling code is considered to be part of XPA-SA and, indeed, code resident within the capsule takes part in the immediate handling of an event (e.g., follower promotion to leader, deadline-elapsed event handling). In OSA, care is taken to separate functionality in such a way that event handling is totally transparent to capsules that may execute on fail-uncontrolled hosts. In XPA, the capsules — executing on fail-silent hosts — can with advantage be given event-handling responsibility.

Fault-tolerance is achieved in two phases:

1) The *first phase* of fault-tolerance is error processing; failures of (fail-silent) servers/stations are detected (as time domain errors) and raised as events to the group manager and Deltase support environment in order to trigger immediate recovery. After server/station failure (leader or follower), an alternative follower replica is selected to take over the role of the failed replica.

   This immediate error processing is achieved on-the-fly within the group manager communication and Deltase libraries, transparently to the applications programmer. The objective is to complete the promotion of followers without compromising any deadlines currently in force.

2) The *second phase* of fault-tolerance is fault treatment, i.e., the restoration of the specified degree of redundancy through cloning, or load balancing by migration, of the capsules concerned onto another station within their replication domains.

   These actions are implemented as either local or global services embodied in system administration server objects with support mechanisms built into the Deltase envelope and the XPA group manager. In common with OSA-SA, these entities include a domain manager plus database, local factory, catalogue server, etc. One difference in XPA-SA is that those functions of the OSA *Management Information Base* (MIB) which are required in XPA have been absorbed into the domain manager database; there might otherwise have been a need for a protocol to maintain consistency between the two databases.

A general model of XPA-SA is shown in the data-flow diagram of figure 7.

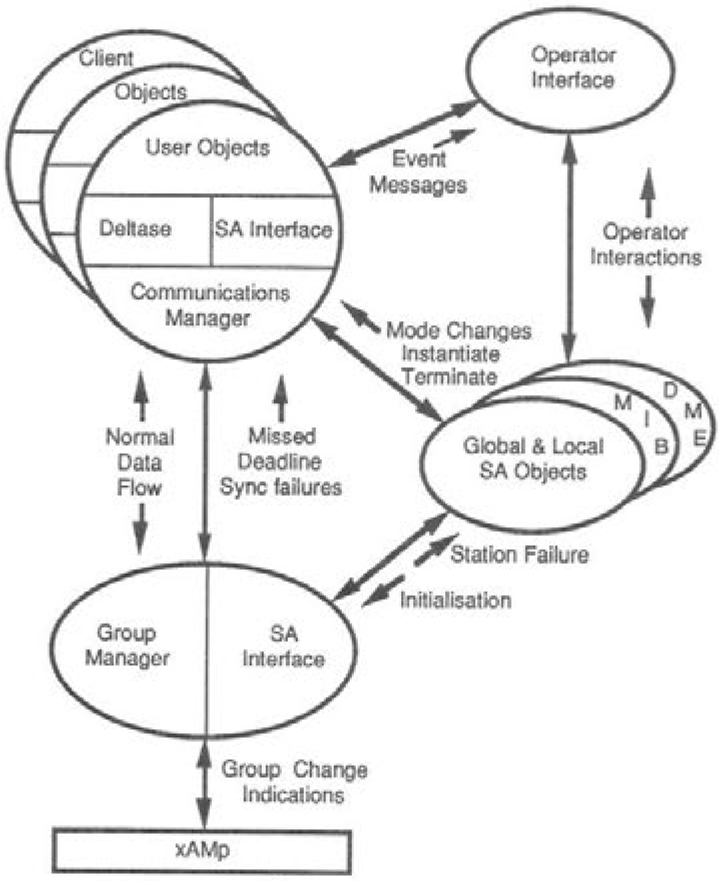

Fig. 7 - XPA System Administration Data Flow

9.7.2. SA Activities involving the XPA Group Manager

The group manager provides the interface between each capsule on the station and the xAMp communication system. The group managers on each station are not themselves replicas; they are replica coordination entities (cf. chapter 6). Each maintains its own local view of the identity and status (healthy, failed or quarantined, see section 9.7.8) of each station and communication group on the network, the identity and status (leader, follower, etc.) of each capsule on the stations and their replication domains.

The group manager on each station offers a management interface for processing requests for the instantiation and termination of the replica copies of each capsule on the system. The group managers ensure that each replica starts up in synchronism with the others and remains synchronized in the leader-follower role by the transmission of notification, acknowledgement and other messages.

The group manager is informed of communication group changes by the xAMp. Station failures are detected either as part of the normal message transmissions or by means of a multicast "Are You Alive?" request. The xAMp protocol for group change indications ensures that they are totally ordered at all recipients; see chapter 10.

All surviving GMs will therefore deterministically update their local opinion of the "takeover order" of each capsule. If the leader fails, the GM supporting the "first" follower will notify it to take over as new leader. (Note that the "takeover order" in XPA is maintained in the group managers. For leader-follower replicas in OSA, it is maintained in the Deltase envelopes of the capsules.)

This processing is carried out wholly by the system administration thread within each group manager, but the failed station event is also raised to an application-level system administration server, the XPA *Replication Domain Manager* (XPA-RDM).

The XPA-RDM is an XPA-Deltase object, unlike the OSA-RDM, and cloning of a new follower replica may be set in motion if this is required.

The group manager also detects most timing errors, excess desynchronism between leader and follower replicas and events caused by lack of resources, e.g., no thread or buffer space available for the onward transmission of a message. These are treated as fault events that may lead to failure unless the event handling can recover from them in time.

These events are handled as described in section 9.7.4 and passed to a local system administration object with an interface "presumed always to be present" for the purposes of event-logging. This local event logger is a prime candidate for evolution to allow presentation to a human administrator or an automated package capable of further action such as load balancing by replica migration.

9.7.3. SA Activities involving the Deltase Support Environment

The Deltase support envelope of each capsule, see figure 7, contains XPA-SA components and provides interfaces to the global and local XPA-SA server objects. Standard, or default, event handlers are provided in the Deltase envelope of each capsule, but provision is made to allow the use of event handlers within the application code which in turn may raise this or an alternative event with the client or an administration component (see section 9.7.4.4). The standard responses to timing failures and resource shortages are selected according to the real-time requirements of the capsule (see section 9.7.5).

The Deltase libraries also include the *Object Manager Entity* (OME) which, as for OSA-SA, is responsible for instantiating capsules under the control of the local factory in addition to handling capsule mode changes as described in section 9.7.8.

9.7.4. Event Management

In the context of XPA-SA, most events result from the detection of a failure to achieve either a correctness or a timeliness specification. This acts as a trigger for system administration event management action. Event management consists of any fault treatment and maintenance activities that enable the system to recover.

9.7.4.1. Event Types. The list of events that must be handled by XPA-SA is likely to change as XPA is developed. The following is a list of standard events handled by the prototype XPA-SA (note later discussion of the different implications of and treatment of events at leader and at follower):

- logical events:
  - station failure;
  - station overloaded for components of priority P or lower;
  - failure to find buffer space;
  - failure to find disc space or some other local resource;
  - failure to find a free processing thread[6];
- timing events:
  - missed deadline;
  - missed periodic activation;
  - excess desynchronization between leader and follower.

Any of these may occur on any station at any time whilst a Delta-4 system is operational. They are therefore "local events", affecting only one copy of a replicated capsule. Some initially local events may affect service provision. A failure to find a free processing thread ought to occur in all correct replicas. A deadline, if missed by a leader, will be missed by all. Where this occurs, a "replica event" is raised[7].

9.7.4.2. User Defined Events. These are defined in server interface definitions and by use of the "raise event" syntax of XPA-Deltase. An important example is conversion from another event type (e.g., deadline-elapsed) as the final act of the event-handler concerned. By this means, the event may be converted into some other event with greater semantic significance to its client (e.g., "cycle-abandoned"). Another example is some address-space-only computational condition (i.e., the classical "exception" concept). It could be useful for the programmer to abandon a computation for application-specific reasons and have nothing else to do but cause the client event-handler (if there is one) to run.

Note that all user-defined events are necessarily replica events; replica group determinism ensures that the computational circumstances that cause the event are encountered by all replicas. Replica transparency is preserved; the programmer never has to recognise or consider unwanted diversity.

9.7.4.3. Event Handling. The examples of real-time problems given in chapter 5 illustrate a tiny selection of the diverse requirements met in systems whose main characteristic is that their design must in some sense be assured to respect timeliness requirements. Generic mechanisms, such as are appropriate to incorporate into the fabric of the XPA support environment, must exhibit considerable adaptability and extensibility to be of value across a range of market requirements and to remain of value under rapid technological change. One of XPA's key mechanisms to address this requirement is that of event handling.

The main events of interest (consideration of which lead to the development of the model presented below) are those associated with timeliness — in particular, the elapse of deadlines. It is, however, noteworthy that the model appears to be applicable to other events and, in the spirit of an ODP recursively-defined architecture, allows a unified treatment of these, whether generated, or handled, at application-level or support-environment-level. Transparency is achieved by the provision of default event-handling.

---

[6] Failure to find a free processing thread occurs when all threads are in use and an input tries to preempt a lower-precedence thread in the capsule.

[7] A replica event is implemented like an input message: it is only passed to the follower at the same preemption point at which the corresponding event was passed to the leader, so the leader always sends an instruction for replica events.

System administration in XPA is considered to be part of the normal machinery available to fulfil application requirements. To achieve this, a mapping of events onto the computation models is desirable. The passage of control in the presence of an event must be defined; when an event is raised, an identified thread must be despatched. This may take a path through a series of objects; collectively, such objects make up XPA-SA. Some of these will have interfaces that are "presumed always to be present"; their implementation may, however, be standard or in some way system-specific. The path taken may therefore be more or less complex according to the particular system implementation choices taken.

In a symmetrical situation to that of user-defined events, user-defined handlers must be invoked only for replica events. Circumstances that are not replicated, such as a particular follower missing a deadline that the leader has met (and notified), are handled by invoking a local SA server ("presumed always to be present") that (at least) logs the occurrence. If the *leader* misses the deadline, then this will (as well as being locally logged) be notified to all followers by the normal XPA deterministic support mechanisms and will indeed result in invocation of all replica event handlers. Again, replica transparency is preserved; the programmer never has to recognise or consider unwanted diversity.

The event handling system must be sufficiently flexible to accommodate user-defined events and new requirements as they are identified, perhaps as a special case arising in a particular system implementation. It is not desirable to constrain the handling of events to a single outcome or even a small set of predefined outcomes.

9.7.4.4. A General Execution Model for Event Handling. In an ideal world, the machinery underlying an abstraction would be totally transparent to programmers. However, an abstraction that is elegant to use is not always elegant in its implementation; abnormal as well as normal combinations of circumstances must be addressed in a manner appropriate to the system's target application.

A suitable application-level abstraction and a matching infrastructure are required. A computational model of administration events is needed that can map easily to programming languages. A programmer should be granted the ability to ignore particular (or even all) events and have them receive default handling; in turn, the default should be selectable by a project engineer from a set of standard handlers during system construction. There is a strong resemblance between the mechanism needed here and that which has proved successful in a quite different context; that of handling exceptional circumstances arising in a pure programming context. We therefore propose to introduce into Deltase-XPA a generalized exception-handler.

A possible execution model, which may be employed in implementing some of the ideas outlined above, is a distributed exception handling mechanism based upon that of the Ada programming language. This is concerned with the passage of control in response to an event that causes suspension of normal program execution. When an exception is raised in an Ada subprogram:

- If a local handler has been provided, this is executed in place of the remainder of the current subprogram. Where there is no handler, or where a further exception is raised within the handler, the exception is propagated to the point of call of the subprogram.
- If there is no handler provided by the calling subprogram, the remainder of its execution is also abandoned and the same exception is propagated (re-raised) to its point of call, the next subprogram outwards and so on until the body of the originating task is reached.

- If the body of the task (which can be thought of as the origin of the thread of control) provides no handler, then the task is terminated.

This model can be mapped onto a distributed environment. As always, a distinction is drawn between inter-object and intra-object behaviour; the former can be described without elaborating the details of the latter. Thus, within an object, the local passage of control as a result of an event depends on the language mapping.

The model may be extended in various ways. For instance, the exception propagation can generate an "audit trail". This includes a description of the path that otherwise would be lost. These parameters are important in the case of onward propagation through a series of objects that do not provide handlers for the exception concerned. Eventually the top-level administrative object is reached where, instead of a simple termination (of the relevant thread), a handler is eventually provided in the form of a human administrator as the only remaining basis for handling the exception. This human will certainly require the path parameters (and much other) information, as well as an interactive service that permits what is necessary in the way of delicate or drastic action.

Another possible enrichment of the exception model, where a (human or machine) exception handler can successfully correct the circumstances giving rise to the event, is to provide for the return of control so that execution continues. An example involving a human administrator might be circumstances that interfere with or augment the mechanism for cloning a new replica. It could be argued that all such circumstances amount to design faults in the cloning service, but this view fails to take account of the sheer complexity of possibilities that can arise and the advantage in terms of simplifying the machinery involved of including a human "in the loop", represented by an SA server. Once the human has successfully created additional disc space, or resolved local naming difficulties, or determined a suitable destination host, or installed additional required servers, or whatever, it should be possible for control to return first to the machine cloning service and, when this is complete, to any thread that was diverted from its normal activity to pass the exception.

An elapsed deadline event occurring at a capsule does not remove the need to receive any outstanding RPC responses. Although the servers concerned inherit the deadline as part of the precedence and so may also experience the elapsed deadline event, their threads of control are returned, as below, immediately or eventually, either as a raised event or as provision of service, depending on those servers' own local handlers.

The exact form of object behaviour in the presence of a "deadline-elapsed" event is decided by the programmer:

- If the programmer chooses not to provide an event handler, then a default is provided (selected on component generation) which abandons computation, marks any outstanding remote services as orphaned and then returns a "server-deadline-elapsed" event to the invoking client concerned (or "top-level-deadline-elapsed" to the XPA-RDM event-catcher, if the object is top-level).
- If the application programmer provides a handler, it is this that determines how the component behaves. The next section discusses how event handlers are mapped into the programmer's language.

9.7.4.5. Language Mapping of Event Handlers. It is desirable to make a distinction between events occurring at an object and events raised with an object by (remote or local) servers. The former invoke a handler, if provided, in a manner similar to UNIX signal handling in that the handler is run as a preemptive parameterless procedure (with the important difference that the preemption is at an XPA preemption point and is therefore deterministic across replicas). The latter involve the return of a thread of control and therefore constitute an abnormal return *from* a procedure for which a clear semantics must be established.

For any particular RPC (or LPC) and server-event, return values must take a state that is definable with respect to the interface concerned. Any return parameters expected may indeed be returned if the event is raised by server code that is able to set them; this is not mandatory but a matter for the language mapping established. If server code does not or cannot set the parameters, then the Deltase envelope, which unmarshals parameters from the message representing the returned thread of control, is able to construct agreed "null" values for the type concerned (typically all 0's). If the event is raised elsewhere than within the server, for instance by a group manager protocol detecting network partition, then no such parameters can be returned even though the message is identified with a thread of control; the Deltase envelope of the client again constructs "null" values. This permits control to return eventually to the statement immediately following the RPC (or LPC); before doing so, a handler is run.

The means of mapping this to a language depends on the facilities of that language. "EVENT" is typically an enumerated type, with the enumeration of standard events predefined and extensions as necessary for particular interfaces. An event is raised using a standard procedure exported by the Deltase envelope:

```
RaiseEvent (Cycle_Abandoned, String);
```

where "String" is a legible identifier of the source and reporting path of the event, so that the event can be understood and maybe managed by a human administrator.

The event handlers discussed here can then be achieved by use of "case" or "select" statements, or by introducing new keywords and associated syntax such as:

```
interface declaration:
(1) EVENT Cycle_Abandoned; assignment of event to handler:
(2) EVENT Cycle_Abandoned : Catch_All();
(3) EVENT Time_Out : {
            Do_Something; /* using local variables */
            RaiseEvent (Cycle_Abandoned, Local_Name);
            };
```

which are preprocessed to construct "case" or "select" statements. The latter solution is more elegant and transparent in that it allows automatic insertion of defaults to preserve the equivalent appearance of local and remote procedure calls.

This allows handlers to be context-independent or context-sensitive. A context-independent handler is a parameterless global procedure assigned to all occurrences of a server event on a particular interface by declaring the assignment at the point of import of service as in (2), or to selected occurrences of an event by declaring the assignment at the point of server invocation. A context-sensitive handler is assigned to an event by declaring the assignment and positioning its code at the point of server invocation (3).

A different syntax is used in the body of code to catch standard events such as time-out occurring at an object. At the start of service, the default handler is assigned to all standard events. At any point in the code, a new handler may be assigned to a standard event using the standard procedure AssignEvent; this handler then remains ready to catch that event on that thread until an alternative is assigned or the service completes:

```
AssignEvent (Time_Out, Catch_All);
```

At present, this is restricted to assignment to a parameterless global procedure. The more difficult issues of how to permit context-sensitive handling of events occurring at an object are the subject of ongoing study.

It is the nature of such handlers and their interaction with SA that determines the *run-time* behaviour of the component in terms of the categories of real-time. The programmer is offered an opportunity to decide whether a particular event is to be treated by object application code or by a system-administered strategy or by some combination of both. This is particularly important in the case of events such as "missed deadline", where the nature and specification of the application are what necessarily determines how this event is treated. At one extreme it is considered a failure and therefore treated according to Delta-4's model for failures, whereas at the other extreme it is merely a convenient prompt to the application of circumstances that must be addressed.

9.7.4.6. Mechanisms to Support Event Handling. The group manager raises all GM-generated events to the Deltase envelope of the affected capsule. In some cases, such as replica desynchronization, the event is raised to both leader and follower replicas. However, not all events are generated by GM — both application objects and their Deltase envelopes may also raise events.

The Deltase envelope contains a default handler for each event, which may just re-raise the event, replacing the reply to some RPC with a standard event message containing named path information.

9.7.4.6.1. Local Events. Local events are received as input messages or as error replies to LEX calls (e.g., no free disc space). If the group manager has a local event for a follower, it may create an input message accompanied by a special pseudo-notification ("receive this message and invoke local event handler at next preemption point; afterwards perform any replica action otherwise notified, to be performed at this preemption point"), or may raise a signal (e.g., in UNIX).

A local event handler may itself raise a replica event. For example, if a leader runs out of disc space, the event handler in the leader generates the "leader failed" replica event. Indeed, an event is transformed from a local to a replica event only if it occurs at the leader capsule's station. For certain local events, for example persistent desynchronization, a follower capsule raises a replica event by notifying the leader capsule.

9.7.4.6.2. Replica Events. The transformation from local to replica event is implemented by either the leader's GM or local event handler raising a high-precedence input message to the capsule, which will therefore be deterministically received by all replicas. Deltase accepts the replica event at a preemption point and decodes it. When the event is decoded, it may be found to be directed at a particular thread; Deltase then runs the appropriate event handler in the context of the affected thread. Some events, such as "leader failed", concern the whole capsule rather than a particular thread and these are handled in the Deltase envelope itself.

Some computational events, e.g., "floating point exception" and "illegal instruction" will affect all replicas and are inherently deterministic since they occur immediately after the offending instruction (e.g., in UNIX they are presented as signals that preempt the next instruction). There is therefore no need to use the XPA determinism arrangements to treat these as replica events. From the point of view of the application, they are presented and may be handled in the same way as events arriving as input messages.

9.7.5. Hard and Soft Real-Time

"Hard" is a design-time classification (see chapter 5). At run time, a timing failure of a "hard" capsule is assumed to be an indication that the external world behaviour is no longer within the operational envelope. This assumption recognises that the entire responsibility for assuring that hard real-time requirements are met depends upon the system design process; there is nothing that can be done at run-time to tolerate such design faults.

The design process must therefore consider the "worst case" scenario. The "worst case" scenario conceivable is when all stations are in a non-operational mode. This drastic (and highly improbable) condition is, however, not useable for design purposes. The operational envelope is a definition of some less extreme worst case scenario with acceptably small probability of being exceeded, say in the lifetime or mission time of the system.

For a system subject to unboundable demands from the environment, the operational envelope may be exceeded and hard deadlines may not be met. What happens to "hard" components under such circumstances is not defined by the classification but by the application programmer; a "hard" component may have a missed-deadline handler defined for such an eventuality[8]. According to the design strategy, the passage of the event through programmed or default handlers and administration components may cause any action up to and including a controlled emergency shut-down of the entire Delta-4 system.

The following separation of concerns is made: the system designer is concerned to assure that the missed-deadline handler will not be invoked within the operational envelope and therefore the application programmer can assume this to be the case.

A capsule is classified as soft real-time when real-time deadlines are regarded as desirable targets. Unlike hard capsules, timing faults do not conflict with the requirements specification and can be tolerated. Delay or loss of instances of service can be said to occur "gracefully"; that is, any such degradation of system performance is not catastrophic and can be arranged to occur in stages. As the load on an XPA system is increased, the services of lowest priority are the first to suffer timing faults and the most important and valuable services are the last.

The real-time type classifications, "hard" and "soft" actually represent examples of a continuous spectrum of design assumptions that effectively define a function representing the cost of inability to achieve computational timeliness. The following paragraphs discuss intermediate type classifications made possible by the characteristics of the leader-follower model under XPA.

Consider the detection of a missed-deadline within a single capsule replica. This is considered to be a local event but if the affected replica is the leader, it will also constitute a replica event; the service deadline has not been met and there is reason to propagate this event to a client (and perhaps ultimately the top-level object, according to the mechanism described in the exception model). If the replica that misses the deadline is not the leader but a follower, then the ability of the system to meet service deadlines will be at risk if the leader should fail.

After leader failure, the follower should normally be able to perform the low-level protocol to become leader prior to deadline. The follower's missed deadline is evidence that if the leader fails, this may no longer be possible.

A strategy that cannot be used during the design process to achieve absolute assurance of hard behaviour, but may offer a high probability that such behaviour will nevertheless be exhibited, is as follows:

The follower's missed deadline causes SA to declare that the follower has failed because of station overload rather than station failure. This is then handled by cloning or migration of the capsule to a less loaded station, so as to reduce the risk of a future actual missed deadline. Note that there are several small contributions that must be taken into account when determining the

[8] If the "hard" deadline is also "critical" (in that if it is missed then the consequential damage is incommensurate with the benefits of the service provided by the system as a whole) then there may be no action that is reasonable to take in that eventuality or there may be no action that limits the scale or level of damage. To a first approximation, however, "criticality" is only compatible with "boundability"; only when the system is subject to boundable demands can the design process give an assurance that the catastrophe will never occur. Given such an assurance, a run-time missed-deadline handler is irrelevant.

probability that this will succeed. There may be pathogenic coincidences; the leader may fail coincidentally with the follower first missing its deadline, there may be no lighter-loaded station, there may be a failure during migration, etc. Even if such a calculation results in a probability that is close to 1 over the lifetime of the system, it is still possible for a pathogenic case to occur. This is very similar to the case discussed earlier of the possibility of all stations failing simultaneously.

9.7.6. Real-Time Events and Replica Group Determinism

Real-time events need to be handled efficiently, but without compromising replica group determinism. The implications of this are discussed below.

Section 9.3.5 lists some possible responses to a missed deadline event. The handler of a local event must not create replica divergence, or raise the event to a client, since the local event may yet be masked by the normal behaviour of other replicas of the server. The local event is therefore handled in the Deltase envelope and the handler runs in a diverted thread in a manner analogous to that of a UNIX signal handler (it cannot be given its own thread since the free thread resources in different replicas would then be different). The context of this handler is severely limited; it is, in effect, necessary to treat the execution as occurring in a separate virtual machine (although this may occupy part of the address space of the capsule). All interactions between this handler and elsewhere must be through remote service invocation unless or until drastic action is necessary, after which the capsule ceases to be a replica and terminates. Local events are normally handled by reporting them to the local SA event logging object, which may merely record the event, or may initiate some recovery or reconfiguration, or appeal to some human administrator. The local event handler may contain no preemption points of its own. Missed-deadline events are handled by the thread that missed the deadline, at a precedence higher than that thread, to ensure the event will preempt the thread. The missed deadline event can be a replica event, i.e., all replicas have missed the deadline, or a local event, i.e., only the local replica is known to have missed the deadline.

Missed-deadline events in a component that is known to be non-replicated cannot compromise replica group determinism. A Deltase handler might take advantage of this, but a handler in the application code itself should not assume any particular state of replication.

Missed-deadline events at the leader are necessarily transformed into replica events and are handled as follows. When the deadline elapses according to the local NAC clock (which is approximately synchronized with other local clocks, see section 9.6.6), the group manager local to the leader sends a replica event to all replicas. This is an input message of higher precedence than the thread that has missed the deadline, to ensure that thread will be preempted.

If a high-precedence thread and a low-precedence thread in the same component both miss a deadline at the same time, the high-precedence missed-deadline handler runs first in all replicas. As with the processing of normal input messages, the low-precedence thread cannot delay the high-precedence activity and replica group determinism is preserved.

The NAC clock is managed by a NAC clock server that maintains an ordered list of deadlines and the threads that are to be informed when these deadlines elapse. When a thread meets its deadline, it either terminates or sets a new deadline and the old deadline is removed from the list. A missed-deadline event may be produced in the NAC whilst the thread meets the deadline in the host; in this case, the event is discarded.

The handler can report the missed deadline to the local SA logger. This provides another example of the intended project engineering flexibility of the event handling model. If, in an XPA system where such mechanisms are important, the logger is constructed to further raise the local event to a special SA component[9] which is constructed to manage a load-balancing strategy, this might respond to the fact that a soft real-time follower has missed a deadline which its leader has met by migrating the follower to an underloaded node.

The missed deadline event is typical of the real-time events, signals, or interrupts that should be reported to XPA software components without compromising replica group determinism. One of the above methods of preserving replica consistency should be selected in each case. The need to preserve replica consistency can increase the latencies of event reporting or preemption handling. In [Le Lann 1989], it is argued that this normally has small real costs; so long as latencies are small compared with computation times, even a large increase in latency causes only a small increase in required processing power. Nevertheless, XPA seeks to minimize all such latencies and to ensure they have known bounds so that required processing power can be calculated.

9.7.7. XPA System Startup

This section discusses the initial startup of an XPA system. When it is switched on, an XPA system becomes fully operational after a sequence of steps, each of which depends only on its predecessors:

1) Initialization of GM on each station. This includes the initialization of message buffers for each group of priority levels to manage flow control and credit allocation (see section 9.6.4.2).
2) Creation of the GM group
3) Activation of Deltase on every station. This will lead to creation of the catalogue group or groups and RDM group, as well as local factories on each station, as with OSA. (The functions of the OSA MIB are fulfilled on XPA by the RDM database.)
4) A startup file contains instructions interpreted by a system startup component that invoke the necessary services on the RDM, and thence the local factories, to instantiate Deltase capsules, again as on OSA. The GM is also instructed to create communication facilities for each capsule.
5) Exports and imports are made via the trader, which uses the (possibly federated) catalogue (normally this is transparent to application code), again as on OSA.
6) Application-code startup activity, e.g., testing availability of and initialising I/O, setting up application-specific initial conditions within servers through invoking their services, etc. During this phase, timeliness properties cannot be assured of capsules; this is a convention that must be taken account of by the programmer.
7) After they have initialized, each component individually awaits initialization of its imported servers. A top-level component also raises its precedence.
8) Normal run-time operation. Servers await invocations of service, top-level components are autonomous (and usually cyclic). This is the phase with which the design-time assurances of timeliness are concerned.

In order to avoid individual entry to normal run-time operation causing conflict with its own timeliness and server availability requirements, the order of exit from startup is important and is controlled as follows:

Under OSA, if a client object invokes service on an imported server interface during the client's startup activity, then it will be blocked until the server object has completed its own

[9] Special components invoked by variants of standard SA components are themselves considered to be part of SA.

startup activity and is able to receive and service requests. This characteristic is made use of in XPA to assure that all of a client's servers are able to perform in a timely manner before the client itself enters normal run-time operation. This is done by introducing, for XPA, a "dummy" server invocation, recognised by the server envelope as being part of its clients' initialization and immediately returned, rather than being treated as a normal invocation requiring the allocation of a thread, or as an interfacing error requiring the raising of an event.

Each client must, after completing its startup activity but before entering normal operation, invoke this "dummy" service on all imported servers. This is done in the envelope and therefore is transparent to application code. In the case of a client that is itself a server, this is done within the envelope immediately prior to first waiting for service invocation.

In the case of a top-level client (i.e., one that does not export its service) the initialization code is in effect long-lived. Before entering the portion of this code considered to represent normal run-time operation, a real-time top-level client must raise its precedence, e.g., to a value determined in the object's environment string or the system startup file. A standard (envelope) procedure is invoked by application code to enter normal operation that both raises precedence and the necessary "dummy" service invocations. A non real-time top-level client need not raise its precedence or issue "dummy" service invocations, so need not call this procedure.

Since all objects during initialization have non-real-time precedence, then top-level clients completing initialization and gaining their normal precedence are in a better position to meet deadlines (i.e., have less competition) than under normal global run-time conditions. The above mechanism ensures that all their servers (to whatever degree removed) have preceded them into normal run-time operation. Therefore conflict with timeliness is avoided during the period of transition.

One restriction imposed by this mechanism is to prevent the mutual import of interfaces that would otherwise permit recursive interaction between objects. This is, in principle, permitted by OSA Deltase, but some language mappings (e.g., Ada) already impose this restriction. The restriction is not considered serious in XPA; recursive interactions would make timeliness predictions extremely difficult.

The group manager activities are initiated by a command from each host. (1) includes such operations as the initialization of the pool of free message buffers in the NAC. (2) and (3) involve distributed protocols described in the *Delta-4 Implementation Guide* [Delta-4 1990]; in (2) the group managers may discover that not all the expected stations are operational.

All capsules are generated with a connection to the local trader and binder; after stage (3), the local traders can access the catalogue, so that stage (5) becomes possible. Thus far, XPA startup differs little from OSA startup in that equivalent structures are created in an equivalent way; e.g., the GM group, used in the detection of station failure, is equivalent to the SMAP group.

However, XPA has to impose restrictions on the later startup of hard real-time activities at run-time. This is because the timeliness of hard real-time components is normally assured as part of the design process; therefore, to change the configuration of these components at run-time invalidates this assurance. All hard real-time activities must therefore be instantiated during the startup sequence. Soft real-time components may be instantiated later, but the appropriate SA service is only made available to authorized humans, e.g., a "real-time administrator". For non-real-time components, the service can be less restricted.

9.7.8. Maintenance/Fault Diagnosis

Facilities are needed to enable the human operator to start up a new station, or to re-introduce a station that has failed, into the Delta-4 communication network after it has been repaired. The new configuration will now support the cloning of those user capsules whose replication domains include the new station so as to restore the required level of capsule replica redundancy to pre-fault conditions. It is necessary to ensure that this is carried out in a controlled manner for two reasons:

- The cloning of a number of different user capsules to the new station must be scheduled to minimize the additional loading of the existing stations.
- The fault may not, in fact, have been discovered and repaired whilst off-line so that a restored station, or an individual capsule, may show the fault again either immediately or after a short period of operation. The effects of this on the rest of the Delta-4 system must be minimized.

An interesting method of controlling the re-introduction of the capsules on a restored station that will be studied for inclusion in the prototype XPA-SA is described below:

A policy might be that, on a new station or perhaps a station on which maintenance action has not revealed a suspected fault, none of the objects is trusted (they are prevented from becoming leader) until a period of "quarantine" (a mode, held as a node- or object-level status) has elapsed. During this period, behaviour comparison with the trusted leader is desirable, to discover whether the fault is still exhibited. The duration of this quarantine period, and the testing that is possible, are entirely under human control. Any necessary promotion of a quarantine object to leader, arising as a result of the failure of other nodes, is explicitly detected as part of the failure event handling. This is another example of an event appropriate to present to a human administrator "server" as described in section 9.7.4.4, in this case to decide whether to lift the quarantine or take some other action such as creating or migrating an additional replica.

Methods of comparison of the outputs of a suspect capsule replica against those of a healthy replica will be studied on the XPA prototype and will require additional services within the group manager and Deltase.

9.7.9. Cloning in XPA

Cloning of Software Components in XPA makes use of the same architectural components as cloning in OSA: that is the Object Manager Entity (OME), the Replication Domain Manager (RDM) and the factories, and their relationship is that described in section §8.2.4. These components for XPA differ in detail from those in OSA, due to the different communication system to which they interface, and the use of the leader-follower model of replication. This section describes the differences in implementation of cloning between OSA and XPA.

Phase 1: Reestablish communication context.

In OSA, messages queued in the NAC for the Software Component before and during this phase, and which will not arrive at the new replica, are always received by the Software Component before phase 2 starts, due to the ordering properties of the Atomic service of the xAMp (see section 10.4). In XPA, which uses the xAMp Reliable service, this cannot be guaranteed. Therefore in XPA there is an extra phase during which messages queued at the leader replica are transferred to the new replica by the group manager, and duplicate messages at the new replica are discarded. This new phase is carried out in parallel with phases 2, 3 and 4, but must be completed before phase 5 can commence.

Phase 2: Take a snapshot of the computational context.

This phase is essentially the same as in OSA.

Phase 3: Generate the global context.

In XPA, only the leader replica sends the checkpoint data (no voting takes place on checkpoint packets) and therefore the OME does not have to ensure identity of replicated checkpoint data before starting phase 4.

Phase 4: Transfer the global context.

This phase is essentially the same as in OSA.

Phase 5: Continuation at the execution checkpoint.

In XPA, before the new replica can continue, the OME and group manager must discard instructions generated and associated messages received during phase 1, i.e., messages which form part of the snapshot taken in phase 2. The instructions to discard are those whose preemption point counter is less than the value included in the snapshot.