Chapter 5

Real-Time Concepts

Real-time systems are those which are able to offer some assurance of the timeliness of service provision. This assurance may range from a firm guarantee of service provision within a defined time interval to a high expectation of service provision before a defined time elapses. In order to meet the timing constraints of the more important real-time services, a graceful degradation in the timeliness of less important services may be necessary.

This chapter is concerned with the definition of real-time concepts, rather than with the Delta-4 techniques to support real-time systems. A detailed account of these techniques is given in chapter 9.

5.1. Concepts and Definitions

5.1.1. Definitions

It is common practice to begin a discussion of real-time with a set of definitions of terms, in particular, the term "real-time" itself. As an example, consider: "If a real-time system misses a deadline, it has failed". Definitions like this exclude from analysis many issues that clearly involve the timeliness of computation. Consider, for example, control systems based on periodic sampling; here it is cumulative rather than individual service untimeliness that leads to risk to control stability, so deadlines are not an appropriate basis to distinguish correct from incorrect control. One conventional practice is to notice that a parameter other than time has exceeded a defined bound.

It is not the purpose of this chapter (or of the architecture) to reject all but a convenient sub-domain of these real-time issues; instead, we seek inclusive models. All-inclusive models are unachievable, so we shall instead examine models for their generality, looking for evident opportunities to solve a large class of commonly-encountered problems. Under such an intent, definitions are entities to be examined, rather than representations of truth.

The following definitions are derived from [PDCS 1990], since they are perhaps more inclusive than most. They are offered as a starting point for the evolution of the necessary concepts.

A *real-time service* is a service that is required to be delivered within time intervals dictated by the environment. (Note that to deliver a service may mean to sample an input rather than to produce an output.)

A *real-time system* is a system that delivers at least one real-time service.

A distinction is normally made between hard and soft real-time. Whilst it is universally agreed that this distinction exists, its meaning is subject to subtle differences in interpretation. The following are typical definitions; note that they express service requirements rather than properties of an implementation:

A *hard* real-time system is a real-time system in which at least one of the timing failure modes is costly or damaging to the environment.

A *soft* real-time system is a real-time system for which all possible timing failures are benign, i.e., system service at the wrong time may be useless, but it is not catastrophic.

Similar, but not equivalent, definitions are to be found in [Jensen 1991]:

A *real-time service* is one that produces results having time constraints that are part of the results' correctness, rather than performance, criteria.

A *hard* real-time service is one that has zero or negative utility if it is delivered outside a certain time interval.

A *soft* real-time service has positive but sub-optimal value if delivered outside a certain time interval, and maybe zero or negative value if delivered outside a wider time interval.

These definitions reflect a change in perspective: Jensen is concerned to address what are termed "mission-critical systems", where a condition of overload exists when services are needed most and timeliness assurances are in general impossible. The notion of the value or utility of a service is introduced to develop the concept of dynamically maximising the overall utility of system behaviour. The design goal is to achieve a particular form of behaviour optimization, rather than the behaviour assurance that is implied by the earlier definitions.

XPA is intended to support hard and soft real-time services according to either perspective and within the same distributed system. This leads directly to one important restriction in the applicability of the architecture; such support is incompatible with the constraints necessary to address safety-critical systems. This is discussed further in annexe A.

Whereas definitions such as these are in common use, they must be used carefully as they appear to draw a sharp distinction among three classes of service: *hard-*, *soft-* and *non-real-time*. The boundary between classes is indistinct; in practice there are many compounding factors. To accept the definitions too literally results in polarization and inflexible architectures.

This point is well illustrated through use of a graphical representation due to [Jensen et al. 1985], which is able to capture quite subtle characteristics of services and service interactions. In particular, it reveals that definitions such as those given above represent only special cases amongst many others that it might prove equally useful to recognise. Figures 1 to 4 give examples based on Jensen's graphical representation of a utility function of service-delivery against time, i.e., the benefit or cost of providing a particular service at a particular instant in time[1].

In figure 1, the cost (or "negative utility") of delivering the service outside a window defined by the "liveline" and "deadline" is unacceptably large, so this example corresponds to both definitions of hard real-time. In figure 2, the cost just outside a similar window is small, so this corresponds to the first definition of soft real-time. Note, however, that if service is seriously delayed, the cost eventually becomes as unacceptably large as in figure 1. However, there is a difficulty in defining a window that allows this service to be classified as "hard" according to the first definition above, since service provision just *inside* any such window would be almost as costly.

If the above definitions were to be cast in terms of the rate of change in utility at significant points in time, the two cases illustrated would be distinguished, but other cases are then rendered anomalous. Is the service requirement of figure 3 harder (because greater rate-of-change of utility occurs) or softer (because no negative utility occurs) than that of figure 2?

---

1 Note that Jensen proposes that such utility functions can both define a service requirement and be internally represented and used by a dynamic scheduling mechanism. Such scheduling has not been implemented on Delta-4, but it should be noted that a local execution environment using such a scheme could be included under the architectural mechanisms described in chapter 9.

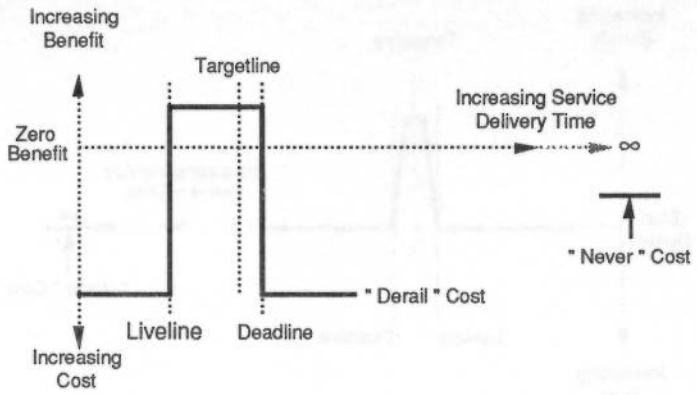

Fig. 1 - Benefit of a Hard Real-Time Service against Delivery Time

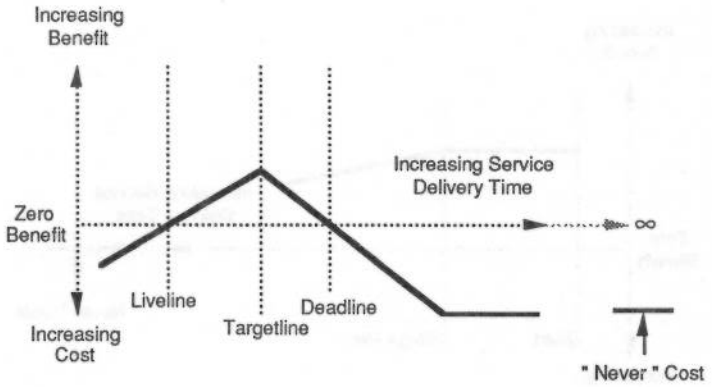

Fig. 2 - Benefit of a Soft Real-Time Service against Delivery Time

These issues are further discussed in [Burns 1990b], through use of the same graphical representation.

Many commercial Local Execution Environments (LEXes) claiming to offer real-time facilities provide a single mechanism (e.g., a priority- or deadline-based scheduler), and allow its shortcomings to be addressed through making various "escape" facilities available to the programmer (e.g., priority or deadline may be manipulated explicitly from the application code). In this chapter, we will assume the use of such off-the-shelf "real-time" nodes, and discuss the distributed aspects of linking these together under the Delta-4 model of dependability.

We therefore define *precedence* to be a generic representation of the necessary timeliness characteristics of the service as seen by the system designer. A system component is presumed to exist on each node to transform and interpret the precedence of each service, from the form chosen to represent the system requirements into the form required by the local execution environment. The distributed environment does not prescribe the scheduling philosophy that will be used.

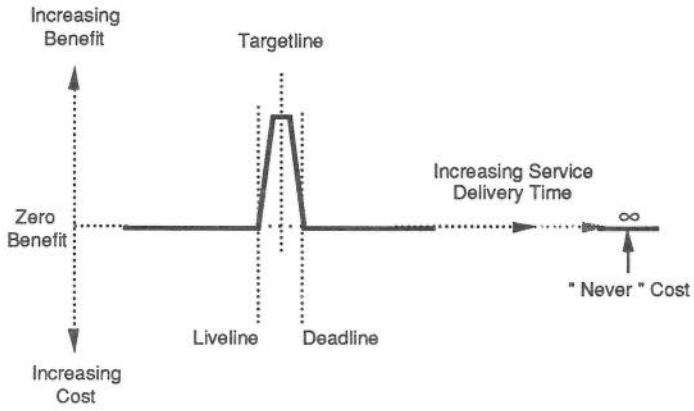

Fig. 3 - Benefit of Anti-Aircraft Gun Service against Delivery Time

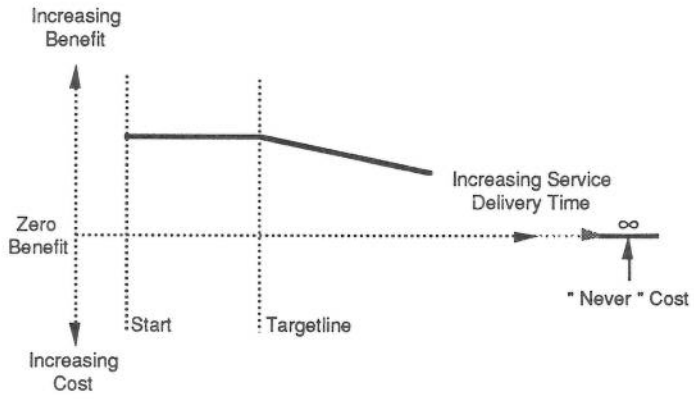

Fig. 4 - Benefit of a Non-Real-Time Service against Delivery Time

Since it is difficult to examine instances of real-time issues exclusively in terms of such a general abstraction as precedence without some prescription for its implementation, we will map this, where example is needed, onto a set of common concepts as defined below. The reader is reminded that alternative mappings are possible, through re-implementation of the precedence interpreter.

Notice that the graphs have been extended to show the cost of *never* delivering the service. In figure 1, which shows the benefit or cost of switching railway points at certain times, the cost of never switching the points is actually less than the cost of doing so when it will derail a train. Some of the graphs are further discussed in section 5.1.3.

The *priority* of a real-time service is a measure of the cost of missing its timing constraints. Hard real-time services are therefore given the highest priority values and non-real-time services the lowest priority.

The priority of a component that supports provision of a real-time service is naturally derived from that of the service. If the timing of component C can cause a timing failure of service S, and C can cause no more costly timing failures, C's priority is exactly the priority of

S. This is called "priority inheritance" (see [Sha et al. 1987]). The "ceiling protocol" [ibid] is a refinement of simple priority inheritance that bounds the number of times a high priority process is blocked by other components.

A *deadline* is the end of a time interval within which a real-time service is required to be delivered.

So if the timing of component *C* can cause service *S* to miss its deadline, *C* should be assigned a deadline earlier than or equal to the deadline of *S*. Alternative strategies for this "deadline inheritance" are discussed in chapter 9.

A *liveline* is the start of a time interval within which a real-time service is required to be delivered.

The *targetline* is the time at which the system designer aims to deliver the service. The targetline is normally chosen as the time of maximum benefit, if known. For a real-time component, it is chosen between the liveline and deadline. The system designer might equate the targetline of a periodic process to the start of the next period, or to an earlier time if required. The targetline of a non-real-time process might be equated to the time at which the user expects to collect the output.

The *precedence* of a real-time service will therefore be assumed in this chapter to be a combination of priority and targetline.

At or before a deadline, a software component may receive an *event* to notify it that it should take some special action, e.g., interact with the environment at once, before it misses the deadline, or abandon itself, because it has missed the deadline. In figure 3, for example, the event might instruct the gun controller to fire, and the event time might coincide with the targetline, if that was chosen as the optimum time to fire.

5.1.2. Systems and their Environments

A control system is acted upon by, and in turn acts upon, its enclosing environment. Events in the enclosing environment that are presented to the system are termed *input events*. Note that, by design, a system might not always change its internal state as a result of these. However, if it does so, these are then called *observed events*. Events initiated by the system on the enclosing environment are termed *output events*. Note that, by design, the environment might not always be affected by such events. However, if it does so, these are termed *control events*.

An *operational envelope* is a part of the universe of all possible behaviours of the enclosing environment, specified in such a way as to permit design-time assurances that the system will behave in a timely manner within the envelope.

Although a requirements specification is treated as if it captures facts about the environment with which a system interacts, this is not quite what it does. It captures assumptions, which in the judgement of the specifier resolve two conflicting issues:

- The assumptions must be useful. They establish a basis for or assist in the design process. They must, for example, establish sharp, finite bounds that permit a design to be constructed and design assurances to be made.
- The assumptions must be realistic. They do not misrepresent reality to the point where a system designed to those assumptions cannot give useful service when presented with reality.

For many requirements in many disciplines, the real world is convenient and permits satisfactory resolution of this conflict.

For some, however, the real world is inconvenient in that, although its parameters are finite, they are generally not sharply bounded, instead exhibiting probability distributions with extended tails of uncertain extent. On the surface of the Earth, what are the limits on

temperature, humidity, wind-speed, earthquake-magnitude, lightning-strikes? How will an enemy behave in sending attack aircraft? How predictable are future alarm conditions?

We may not like it, but even where the extent is known, perhaps by appeal to physics, such limits may not be useful as design bounds, being orders-of-magnitude beyond what is practical or economic in the market concerned. A lesser, artificial limit must then be set.

Thus, bridge and skyscraper builders set boundaries for wind-speed and the magnitude of earthquakes. Electronic systems designers bound such parameters as temperature, humidity, supply voltage, duty cycle and so on. Command-and-control systems are based on assumed minima for the inter-arrival times of enemy aircraft. Electricity supply and other process control systems are based on bounded inter-arrival times for input events.

On what basis are artificial bounds plausible? A room containing electronic equipment may be air-conditioned, or the climate may be considered by the customer to be sufficiently well-behaved that he can accept responsibility for the consequences of abnormal weather. Probabilistic evidence may also be used in other cases where external control cannot be applied, such as the command-and-control example.

Generally, a commercial contract between supplier and customer is involved. This raises several issues that directly influence the choice of technology. Who takes the risk in specifying or making use of an artificial bound? What is the nature of the risk: is safety involved, or lost production, or merely inconvenience? What is the cost of not taking the risk? In a competitive world, an over-cautious supplier can lose an order completely with a marginally more expensive solution. What is the cost of taking the risk? A supplier, or customer, or insurance company, might carry the cost of lost production or worse.

The simplest systems-environment relationship is such that first, all output events are control events, and second, all input events result from previous control events; the design can therefore arrange to ensure that all input events are observed and that all necessary computation is completed in a timely manner.

Most systems-environment relationships are not so simple. In many, the environment can present the system with events that do not result from previous control events but from *autonomous behaviour*. The system-environment relationship is still simple if such input events are sufficiently predictable, e.g., with a minimum duration between consecutive events, such as might occur with manufactured parts delivered by a conveyor belt. The design can again ensure that these are observed and that all computation is timely. In this case, the operational envelope specifies the entire universe of environment behaviour.

If such is not the case, as with *aperiodic events* (i.e., events with no known minimum inter-arrival time) such as the attack of enemy aircraft, operator actions, the occurrence of alarm conditions in all their diverse forms, changes of operating mode, the failure of nodes, etc., then *there is a possible behaviour of the environment for which timely system behaviour cannot be assured by the design process*. This chapter will discuss this issue at some length, since it is the experience of the industrial partners of Delta-4 that it cannot be ignored in the target markets for the architecture.

Since aperiodic events are incompatible with design-time assurances, a designer may try to transform the system-environment relationship as follows. Instead of receiving aperiodic events, the system periodically samples environmental state information, and is sized so that it can always process each periodic sample before the next sample is received. Between samples, some aperiodic events will not be observed, but in some applications this is acceptable, e.g., because it only entails the loss of intermediate values of a continuous variable. Even when it is possible, however, the transformation does not always lead to practical and economic limits on the system resources required.

In the context of Delta-4, the failure of system hardware components is viewed as autonomous environment behaviour, for which specialized sensors and event management

mechanisms are constructed within the Delta-4 support environment itself. Such events are only statistically predictable; it is possible (but of infinitesimal likelihood) for a cascade of failures to fall outside any defined operational envelope.

Within the operational envelope, a system is required to meet timeliness criteria needed for correct system operation. This assurance must be provided during the system design process. The designer must demonstrate that at run time, providing the enclosing environment complies with the specified operational envelope, timeliness requirements will be met.

If the resulting system subsequently encounters circumstances and events that fall outside this envelope, then no such absolute guarantee can be given. One may, however, still be able to prove that timeliness criteria will be met with very high probability, or that the timeliness of the higher priority system functions can be guaranteed. Systems offering such qualified assurances can still be useful in the target market areas.

5.1.3. Some Examples

To assist the discussion, we now consider a few examples of real-time problems. A hard real-time problem is described first.

5.1.3.1. Example 1. Imagine a process control plant that uses motorized rail-trolleys to transport materials from one part of the plant to another. One of these trolleys approaches a set of points; the time when the trolley will reach the points can be calculated. If the points are currently in the wrong state, they must change state before that time; this is a hard deadline in the real world (see figure 5).

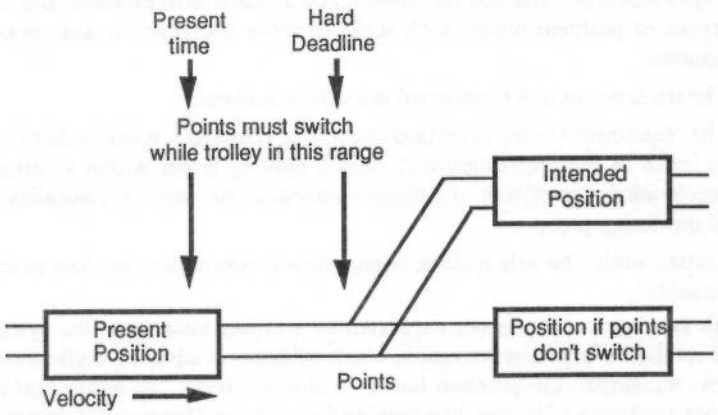

Fig. 5 - Rail-Trolley Example

The example is of course simplified. A wealth of practical details complicate the issue. For instance, the deadline might be estimated from position and velocity; velocity might be determined from the times at which front and rear axles pass fixed track-side position sensors; there might be a velocity-changing phenomenon such as friction that can only be estimated. The time when the points will be reached cannot be predicted exactly so that an earliest bound must be established. Also, the points take some time to move: this too must be bounded.

These bounds perhaps establish the actual deadline, the latest time by which the points must have started to move. If the points can jam, this must be detectable and it might be necessary to

determine another deadline, the latest time at which the trolley brakes may be applied. Events could be sent to the controlling capsule at either or both these deadlines, instructing it to take the appropriate action.

None of these elaborations alter the requirement for the controlling component to complete some action by some point in time. If it fails to do so, the trolley can be derailed, which could have catastrophic costs; therefore by definition this is indeed hard real-time. As such, design time assurances of timeliness will normally be required.

This raises the question discussed in section 5.1.2. Can the rail-trolley system encounter environmental circumstances and events outside any definable operational envelope? If so, we cannot provide absolute design time assurances of timeliness, but should still seek to provide qualified assurances that timeliness will be maintained with a high quantified probability. If deadlines are missed, the design must include measures to limit the damage.

Thus, the rail-trolley example has a short-term and a long-term consequence to missing the deadline (see figure 1). In the short term, there is a deadline after which the component must no longer change the state of the points, or the trolley will derail, at considerable cost. In the long term, if the points are never switched, the trolley may be on the wrong track, which has lesser costs, but requires longer term management to get the trolley onto the right track by a longer route. In other words, *although the component controlling the points has a hard deadline, the containing system does not.*

5.1.3.2. Example 2. As a second example, consider the controller of an anti-aircraft gun that tracks the predicted trajectory of an enemy aircraft, i.e., points to where it is estimated the aircraft will be when the shell reaches it. The anti-aircraft gun is required to hit as many enemy aircraft as possible. Figure 3 shows the benefit or cost of firing the gun at different times.

Most people would describe the anti-aircraft gun as a real-time problem, and it is certainly one of the types of problem which XPA seeks to solve. However, it has several possibly surprising features:

- The requirement is not expressed as a timing constraint.
- The requirement is not expressed as a timing constraint. It can perhaps be expressed in terms of the probability of hitting a moving target within a certain velocity, acceleration, rate-of-turn, and distance envelope. Any timing constraints arise as part of the design process.
- Consequently, the only realistic requirement is best-effort: "as close to the aircraft as possible".

Although the requirement is not expressed as a timing constraint, the system can still calculate and set itself a targetline for action, which is known in advance, unlike the liveline and deadline. This transforms the problem into a "real-time" form, but notice that missing the targetline is not a criterion of failure, like missing the deadline. One possible criterion of failure would be a loss of service due to counterattack from the aircraft!

Having moved the gun to the required position, the gun controller might suspend itself waiting for the order to fire, which might take the form of an event at the targetline.

5.1.3.3. Example 3. Consider a 3-term digital control loop based on a periodic process calculating and responding to a so-called "error_term" so as to control an environmental variable within acceptable bounds. The requirement is to remain within a predefined range of the (changing) ideal value at each instant, otherwise there will be a costly consequence (in exactly the sense of section 5.1.1). The process implements a control theory prescription to minimize the absolute value of the "error_term", and can check that this remains less than some

predefined amount (i.e., ABS(error_term) < δ). It is therefore able to discover any violation of this condition, and signal this as a failure. It is likely to fail if it suffers long-term delayed cycle activation or execution time instability.

Again the requirement is not expressed in terms of time and the failure is not a failure to meet a specified deadline, although conceivably it might have been predicted that another delayed activation would cause failure. In this case, however, the requirement is "hard" rather than "best-effort". Again we can transform the non-time requirement into a real-time form, e.g.:

"Each cycle activation time should be within an interval on either side of a targetline, and it will then follow that ABS(error_term) < δ."

This transformation has in fact strengthened the original requirement, but if such a targetline and interval can be defined, we can schedule the component in the same way as other periodic real-time tasks, e.g., by sending it an event at the targetline to activate another cycle. So we might take the design decision to adopt the transformed requirement as a basis for scheduling; but notice that the only criterion of failure is still the original requirement: "ABS(error_term) < δ".

If we try to express the transformed requirement as a graph, we get something like figure 2, where only a little cost is assigned to missing the deadline, because one missed deadline will normally not lead to failure. Notice we are here reaching the limitations of these graphs, which have no way of expressing the real failure criterion.

Such a 3-term controller is certainly required to operate in "real-time", even though its failure criterion does not mention time. It is of considerable interest to XPA, and its implementation requires components such as clocks, which are unarguably "real-time".

5.1.3.4. Example 4. Consider the updating of an operator's display; this might meet some desired delay requirement of the form: "the screen must be updated on average within 0.5 seconds over any 30 second period, except when the system is in emergency mode". The only cost of exceptional delay may be temporary inconvenience to the operator.

This is a typical soft-real-time timing constraint. Notice that the failure criterion is statistical: there are no precise deadlines. The designer might decide to schedule each update operation by assigning it a targetline 0.5 seconds after the start, but this does not mean it has failed if it occasionally takes 0.6 seconds.

5.2. Delta-4 Approach to Real-Time

5.2.1. Approaches to System Design

If a system is presented with circumstances outside the operational envelope, then it is not possible to assure that deadlines will be met. Two views can be taken on this, which roughly speaking identify two schools of thought on design, called here the "bounded-demand" school and the "unbounded-demand" school.

5.2.1.1. The Bounded-Demand School. This school considers that to take the possibility of failing to meet deadlines into account during design is to propose a method of tolerating an inadequate system size. A system in which some deadlines might not be met is not, in this view, a real-time system. Arrangements made to tolerate missed deadline faults are uninteresting since the design process is concerned to assure that this cannot happen.

In this view, an operational envelope is determined which bounds environmental behaviour, and then the system is sized so that deadlines will be met within this envelope. For

all relevant input events, bounds must be determined which are commensurate with the control system technology. They must be *periodic* (occurring at regular intervals), or *sporadic* (occurring at irregular intervals, but in such a way as to impose a bounded system load in any finite duration [Burns 1990a]). The bounds must be absolutely assured, perhaps by the laws of physics. since events beyond the operational envelope will either be ignored or result in "undefined behaviour".

Bounds on internal behaviour must also be established. Unfortunately, combinatorial explosions (e.g., of state space) can, even in quite modest sized systems, exceed the practical limits of existing formal methods and design tools. Some techniques for reducing the state explosion are discussed in section 15.2, but despite these techniques there is a limit on the size of bounded-demand systems established by the state of the art.

5.2.1.2. The Unbounded-Demand School. This school owes its existence to the fact that requirements exist where an "adequate" system size, in the above sense, is simply not able to be established in design. Although finite, the bounds are not discoverable, are too extreme to be reached or supported in practice, or for some other reason have not been specified. "Unbounded" is therefore used here as shorthand for "without applicable bounds" rather than for "without intrinsic bounds". Input events may be *aperiodic* (with no specified minimum inter-arrival time, so that no upper bound can be specified for the system load imposed by the environment). The consequences of output events may be too complex for complete analysis. System size calculations then depend on a large number of assumptions. Even where they are fully understood, these assumptions are often inseparably cross-dependent and resist formal analysis.

In practice, these assumptions are probable rather than certain, and if any of them prove invalid, events may move outside the operational envelope in which timeliness can be guaranteed. If so, the consequence must not be "undefined behaviour", which may in fact be the loss of the system's most important function. Some form of "graceful degradation", i.e., some effort to control and define behaviour outside the operational envelope, is preferable and is seen by this school as responsible defensive engineering.

A similar situation arises when an "adequate" system size *can* be calculated, but is found to imply a noncompetitive system price. A commercial decision may then be taken to use a smaller system size, assuming that the probability of this size proving inadequate is negligible in the lifetime of the system, or its mission time. This assumption may be justified by statistical arguments (see §5.2.2), but there is still a need for defensive engineering in case it is violated.

Even the execution time of a single component can be unbounded. For example, an algorithm for inverting a matrix may converge in a known duration for nearly all possible input values, but may exceed this limit, or never converge at all, for a few input values. Whatever the system size, we must therefore design for the possibility that the matrix inverter will miss its deadline.

5.2.1.3. Argument between the Schools. A major criticism of the "unbounded-demand" concept from the perspective provided by the "bounded-demand" school of thought can be summarised as follows:

- To design for behaviour outside an operational envelope, a wider operational envelope (or several envelopes) must be defined, since one cannot design anything to work reliably in an unbounded environment. So an "unbounded-demand" system is in fact a "bounded-demand" system with several operational envelopes.

This is certainly true, and it follows that neither type of system is predictable outside the widest imaginable operational envelope, i.e., if all the hardware fails. However, as the

environment grows more hostile, the two design approaches lead to different system behaviours:

- The "bounded-demand" designer may argue that, since an operational envelope is expected to bound the environment in all but the most drastic circumstances, all one can do outside this envelope is abandon all real-time services and perform a safe shutdown, provided one is still within the wider envelope assumed by the shutdown routine. The result is a clearly-defined but rather brittle degradation.
- The "unbounded-demand" designer expects the "inner" operational envelopes to be violated more often; so he abandons or postpones system services one by one, starting with the least important or urgent. The result is a graceful degradation that provides a service probabilistically even when it cannot be guaranteed. This is more compliant with a specification stating that the service is desirable but not essential.

The first approach certainly applies to hard real-time services; if they cannot be guaranteed, the whole system has failed, and should stop, preferably in a safe and consistent state. The second approach is more appropriate to soft real-time (or non-real-time) services that can be delivered belatedly without damaging the environment.

This suggests that both design approaches could be used in a system that provided both hard real-time and non-hard real-time services. This appears to be possible in the special case where the following conditions are met:

- The demand for hard real-time services is bounded, so that the system can be sized to assure them.
- The demand for the non-hard services is unboundable.
- The hard real-time subsystem is always able to preempt the resources it requires in a bounded time, so that it can be designed as if the whole system is available to it.
- In particular, the hard real-time subsystem must be able to preempt any input or output channel. It must be possible to ignore or lock out the (possibly unbounded) inputs to non-hard components without losing inputs to hard real-time components.

In such a system, the environment stays within a wide operational envelope in which the hard real-time services can be guaranteed. However, it sometimes violates a narrower operational envelope, outside which the non-hard services have to be progressively postponed or abandoned.

Since Delta-4 is concerned to support the whole range of real-time systems, with and without bounded environmental behaviour, it must make use of both bounded-demand and unbounded-demand approaches, as appropriate.

5.2.2. System Sizing

To determine the required system size, a bounded-demand designer must bound all execution and communication times for all combinations of components in all circumstances. For example, 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.

An unbounded-demand designer has the less stringent requirement of determining bounds that will only be exceeded with a known probability, so that the risk of graceful degradation can be quantified. It should therefore be possible to size the system by arguing from observed probability distributions.

Distributions based not on theory but on observation are modelled by mathematics that is more able to reflect regions of high probability than low-probability tails. Such models typically display infinite tails even though these may be known to be unrealistic, and still be useful for

many purposes. However, establishing a reasonable artificial bound requires low probability behaviour to be quantified; what is being said is something like the following:

- "It is believed that the probability of an event exceeding this bound is negligible in the lifetime of the system, (or mission, or critical period,...)."

A good mathematical model of high probability behaviour may exhibit order-of-magnitude error in quantifying low probability behaviour. Setting an artificial bound must be done on some additional basis. One traditional way is to allow an adequate "safety-margin"[2]; this leaves unanswered the question of how to establish what is adequate.

Instead, a contract may specify a bound, a maximum probability for exceeding the bound, and a behavioural requirement if the bound is exceeded. The behavioural requirement is typically that the contractual requirements for service timeliness no longer apply but that some form of graceful degradation must be exhibited by the technology concerned. In a competitive world, the form this takes is of some importance.

5.2.3. Types of Algorithm

Given a choice between algorithms exhibiting good average but poor worst-case execution times and those exhibiting medium average and medium worst-case execution times, which should be chosen? Note that such a choice is quite common, if we consider algorithms pertaining to all levels in the systems concerned. Memory access can be augmented normally, but slowed down on "misses", by cacheing algorithms. Preemption benefits the preemptor but disadvantages the preempted. Speeding up the sorting of disordered data can slow down the sorting of data in an unfortunate order. The benefits of recognising special cases incur costs to other cases. See, for many examples, [Knuth 1973].

This question is easy to answer in an exclusively hard real-time context. The choice of "medium average and medium worst case" execution times minimizes the system power required to provide a-priori assurance of timeliness, all else being equal. However, in a context in which hard real-time activity is a subset of the total activity, the issue is much less clear. As an example, consider two systems, C being based on "good average but poor worst-case" algorithms and D being based on "medium average and medium worst-case" algorithms. Clearly the timeliness of the hard real-time activities can be a-priori assured in D with less computing power than in C.

However, we are also interested in the expected timeliness of other, desired but soft real-time activities that are allowed execution resources when the hard real-time activities are satisfied. A posteriori, timely behaviour will normally be exhibited for a larger set of soft real-time activities in C than in D, assuming C and D have equal power. Given a sufficient proportion of soft real-time activity, the conclusion is therefore reversed. C is the preferred choice.

Moreover, the uncertainty concerning the proportion of the activity population that will exhibit timeliness depends on the size of this population and the independence of its members. A large, busy system is less variable than the execution-time variance of individual activities would suggest. The subset of activities exhibiting timeliness appears to vary less (in timing or set-membership) than its members, by a significant and useful factor. This is the same sort of mass-statistical effect as permits the hypotheses of large-scale physics or chemistry to be accepted as laws, despite the uncertainty that exists on the microscopic scale (the reasons for these observed effects are discussed in annexe C). These statistical effects are particularly significant in large and complex systems.

---

2 Even with a large safety margin, a bound can still be artificial in that it is less than that which theory can deliver.

The preference for good average but poor worst-case algorithms is a fortunate conclusion, since in practice we may choose such algorithms for other reasons. We may be obliged, because of their power, cost and commercial acceptability, to build systems with microprocessors that use pipelining, cacheing and busses with contention algorithms. Similar comments apply to operating systems and the more advanced language compilers and applications.

5.2.4. Types of Scheduler

We now examine the nature of run-time support for the assumptions taken above; for example, how to grant activities execution resource preference.

A *scheduler* is a resource allocator that affects the timing of a real-time component. The resource may be processing power, network bandwidth, memory, access to a monitor, etc.

An *off-line scheduler* is executed during the system design phase, when it works out fixed schedules for all anticipated component execution sets. The result is a timetable; "slots" are reserved for every process execution and message transmission, and collisions cannot occur. When an event occurs which requires a change of behaviour, this is discovered in a prearranged "slot" and an alternative predetermined schedule is installed. The fixed schedules take account of dependencies between components, which are assumed to be predictable at design time, so that they do not cause delay.

An *on-line scheduler* is executed at run-time, using heuristics that must not themselves impose a significant system load. Dependencies and deadlocks must either be avoided by component design or the resultant delays must be minimized by priority inheritance protocols [Sha et al. 1987] and distributed deadlock detection techniques (see chapter 6 in [Moss 1981]).

Note that both types of scheduler can be combined on one system. An off-line scheduler can calculate a fixed schedule at design time, and encode this schedule by assigning suitable precedence parameters to each component. An on-line scheduler can then reproduce the fixed schedule at run-time by interpreting the precedence parameters. In the free slots of the fixed schedule, the on-line scheduler can run additional components not anticipated by the off-line scheduler.

The requirement that hard real-time components have provable timeliness, despite the presence of lower precedence components, leads to a requirement for *preemption*. This means that when a higher precedence component becomes executable because of an external stimulus or internal event, and a lower precedence component is using the resources it requires, and an unbounded number of lower precedence components are waiting for the resources, the scheduler nevertheless gives the resources to the higher precedence component after a bounded delay.

An on-line scheduler preempts the lower precedence component by waiting for the next interruptible state in its execution — perhaps as soon as the end of the current machine instruction. An off-line scheduler may take longer to preempt, but still schedules the higher precedence component in a bounded time, typically at a pre-arranged point in a fixed cyclic schedule.

There is a large body of research into on-line schedulers that are guaranteed to find a feasible schedule dynamically, if one exists (e.g., [Halang 1986, Sha et al. 1988]). This research assumes that feasible schedules can separately be proved to exist, at least within some operational envelope.

Others, such as [Kopetz 1986], working with systems of the type characterized above as "bounded-demand", use off-line schedulers to generate static cyclic schedules for both computation and communication that assure the required timeliness of behaviour within each operational envelope. The construction of a static schedule can itself be regarded as a proof of

the existence of a feasible schedule, again provided the real world does indeed stay within the operational envelope.

These approaches lead to different types of run-time delay: those of an on-line scheduler include schedule calculation times and preemption delays when components collide; those of an off-line scheduler include delaying the start times of components until their pre-arranged slots begin, when in practice they could often have started sooner. Off-line schedulers normally reserve one slot in the cycle for switching from one schedule to another; so the latency before such a switch may be a whole cycle.

However, if we neglect the difference between the run-time delays of on-line and off-line schedulers, then within an operational envelope, the schedules they produce will both exhibit acceptable (but not necessarily identical) behaviour.

A "bounded-demand" system may have several operational envelopes, sporadic external stimuli, variable execution times and different operating and failure modes. So an off-line scheduler may face a combinatorial explosion of scenarios leading to a similar number of different static schedules, or to one "worst-case" schedule that is vastly more pessimistic than the normal case. With the on-line approach, the scenarios can often be collapsed into a single set of dynamic control rules, or at worst one set of rules for each user-defined operational mode. (For example, a military command and control system may have "combat" and "training" modes in which the same component is assigned different priorities.)

When environmental demand is unbounded, outside the operational envelope, a system based on static schedules continues to sample inputs periodically, so that it may (nonselectively) ignore events. If execution times are unbounded, it may also miss deadlines. If the static slot and cycle times are exceeded, and there is inadequate defensive programming, the system may even crash. A dynamic system is less vulnerable outside the operational envelope; provided its on-line scheduling algorithm is carefully chosen, then the most valuable functions are the last to be placed at risk. The choice of on-line scheduling algorithms is discussed in chapter 9.

Again, a set of dynamic control rules apply in each of a relatively modest number of user-defined operational modes. These rules amount to a set of ordered lists of latent activities, a subset of which are likely to be demanded in each mode. The ordering represents preference between activities that cannot all be carried out, but for which it is desirable to carry out as many as possible. Since the normal object is to minimize the cost of timing faults, this will be a priority order in the sense of section 5.1.1.

5.3. Real-Time Communications Protocols

Given the characteristics of real-time systems, just described, communications protocols for these systems should ensure timeliness properties, apart from other useful engineering features. We shall focus on timeliness properties of real-time broadcast protocols, which are associated with their synchronism, as defined below. The emerging requirements, together with engineering features concerning real-time, will be detailed in the appropriate chapters (chapters 9 and 10).

Building blocks, to manage real-time replicated objects, are *reliable broadcast or multicast* protocols, *group management* protocols, and *measures of the passage of time*. Broadcast protocols reliably disseminate information; group management protocols help to determine to whom or by whom the information is disseminated, how participants cooperate, and how groups of replicas are managed; measures of time (like clocks and timers) assist in establishing timeliness properties. A complete solution has normally a flavour of each of these three building blocks.

There are essentially two classes of approaches to build reliable broadcast and group membership services: the *clock-driven* [Cristian et al. 1985] and the *clock-less* [Birman and

Joseph 1987] approach. The former rely on the existence of a global timebase, which constitutes an absolute time reference, whereas the latter do not, although they may rely on timers, i.e., relative time references. With regard to synchronism, clock-driven and clock-less protocols have often been classified as equivalent to "synchronous" and "asynchronous" protocols, respectively. That analogy is mostly due to the "synchronized" nature of clock-driven protocols, which progress at pre-defined times. The clock as an implementation tool is, however, not mandatory to achieve synchronism, as we discuss below.

The importance of this argument stems from the fact that while clock-less protocols are a useful alternative to clock-driven protocols in a number of situations, their suitability for real-time fault-tolerant applications has been questioned, on the grounds of their "asynchronism". The two attributes relevant to this question are: achieving *execution times with a known bound*, i.e., being able to fulfil not only liveness, but also timeliness properties, and respecting *temporal orderings of events*. It is established in [Veríssimo 1990] that clock-less protocols do indeed have these attributes, and are therefore capable of supporting real-time systems. Having clarified what "clock-less" means, let us now introduce a metric for "synchronism" that applies equally to clock-driven and clock-less protocols. This way, we extend the criteria of suitability for real-time and fault-tolerant applications, to clock-less protocols.

5.3.1. Synchronism

Let us start by making some definitions. Consider a reliable broadcast execution, invoked through a *send* primitive, whereby message m_i is delivered to recipients through a *receive* primitive. Let us call *Execution Time* (T_e), the time between *send*(m_i) request and the last *receive*(m_i) indication. Similarly, *Inconsistency Time* (T_i), will be the maximum time between *receive*(m_i) indications at any two different recipients.

If these variables have a variation, two broadcasts that start at slightly different times, may deliver their messages in diverse orders, according to the relative magnitude of those variations. These variations of T_e and T_i from execution to execution are paramount to characterize protocol timeliness, and in consequence, we define the following two attributes:

- **Tightness** (τ) — *Tightness of a protocol, is the measure of* $\Delta T_i = T_{imx} - T_{imn}$.
- **Steadiness** (σ) — *Steadiness of a protocol, is the measure of* $\Delta T_e = T_{emx} - T_{emn}$.

The *synchronism* of a reliable broadcast protocol is a measure of its tightness and steadiness. An asynchronous protocol is neither tight nor steady. A protocol is synchronous when its steadiness is known and bounded[3]. Synchronism can have several grades, depending on how tight and steady a protocol is.

Clock-less and clock-driven methods normally yield different grades of protocol synchronism, but they can both be measured as defined above. This establishes that both of them are capable of real-time operation, if only for requirements up to their possibilities.

Any protocol implementation that can be translated to a sequence of events forming a chain in time (coarse though it may be), can yield a synchronous protocol, if the maximum length of that chain is known and bounded. This provides the foundation for building real-time clock-less protocols, and the approach taken in the Delta-4 LAN-based protocol, AMp:

> MAC entities[4] cooperate with each other to fulfil timeliness of LAN operation. In some networks, it is possible to build a timing equation representing a frame transmission delay, show that it is bounded, and determine the upper bound, for

---

3 Since $T_i < T_e$, this is a sufficient condition for tightness to be also bounded.

4 Medium Access Control layer entities, which are responsible for exchanging frames and keeping the network in operation.

assumed load and fault patterns. Similarly, it is possible to impose a performance specification on the hardware and software of the communication adapters (CPU, kernel, etc.) so that processing times of protocol actions are also bounded, and known for the worst case traffic pattern specified. We will show that the protocol architecture can be designed so that these requirements are respected (chapter 10). In conclusion, if a protocol displays an execution time with a known bound, it is synchronous.

5.3.2. Temporal Order

The second attribute of real-time capability is respecting the temporal order of events. This is an extension of the logical causal ordering requirements discussed for decentralized operation of distributed algorithms [Lamport 1978], to real-time systems, mainly those with input/output — and possibly replicated — actions, which must be correctly ordered in time. Note however that temporal order is not always required; some of the required ordering properties can instead be ensured by appropriate computational and replication models. An example of this is discussed in chapter 9.

One way to enforce it, is by using real time clocks, approximately synchronized with a precision ε. Running a clock-driven protocol, all participants timestamp their messages, so that they are delivered by timestamp order. The fault-tolerant synchronization method proposed in [Lamport 1984] achieves such an order, which is also guaranteed in the atomic broadcast of [Cristian et al. 1985]. A number of methods exist to maintain local clocks approximately synchronized [Cristian 1989, Schneider 1986, Srikanth and Toueg 1987]. Since these protocols deliver messages by order of their timestamps, this also implies that messages are delivered in the same order everywhere, i.e., a consistent order, one of the requirements to ensure replica group determinism (see chapter 6).

Again, it can be shown that temporal order is intimately related to the grade of synchronism. When τ and σ are not negligible, compared with T_e, we may say a protocol is *loosely-synchronous*, to distinguish from the tightly-synchronous ones, an example of which are clock-driven Δ-protocols such as those of [Cristian et al. 1985]. As a rule, synchronous clock-less protocols are not able to achieve the fine degree of steadiness and tightness, that clock-driven ones are, and this is also true of AMp. However, their ability to fulfil temporal order properties is definable, and in consequence, so are the classes of applications for which a given protocol provides a correct implementation, vis-a-vis ordering of events.

It is worthwhile noting that these concepts apply to general distributed processes. In fact, synchronism of distributed — and in particular replicated — computations, i.e., their steadiness and tightness, is an important issue in real-time systems. We recall that they are related to familiar concepts like lock-step, action rate, simultaneity of replicated I/O, etc.

The approach used in Delta-4, for real-time distribution, relies on *synchronous clock-less* protocols, on local area networks. Although clock-driven protocols can be built on top of the basic communication mechanism (AMp, see chapter 10), clock-less protocols achieve the best possible average execution time. So this approach optimizes the provision of different qualities of service to support complex systems, in which for example periodic hard real-time tasks may coexist with sporadic and possibly aperiodic soft real-time and non-real-time ones.

5.4. Summary of Real-Time System Requirements

From the basic real-time requirement to offer design-time assurances of timeliness, and the arguments above, there follow a number of real-time system requirements, which are summarised below.

- There must be a requirements specification for a particular real-time system, defining the timeliness requirements and the operational envelope(s) in which they are to be achieved. The assumptions made should be useful and realistic. Probabilities should be estimated where assumptions are not certain, and the action to be taken for each timing fault defined.
- The design philosophy should cater for both "bounded-demand" and "unbounded-demand" requirements. Outside an operational envelope, it should still ensure, with high probability, as many as possible of the services that are guaranteed within the operational envelope.
- To this end, designers should choose algorithms exhibiting good average behaviour, to maximize the total activity normally achieved.
- To ensure a graceful degradation as each operational envelope is exceeded, and each priority level loses its guaranteed timeliness, on-line schedulers should be used. To prove the existence of a feasible schedule, or to satisfy simple bounded-demand requirements, off-line schedulers may also be used.
- Bounded-demand systems should be sized so that the kernel of hard real-time activity is still achieved within the worst-case operational envelope. Unbounded-demand systems where even this envelope can be exceeded should be sized so that the probability of this is acceptably small; if it occurs, timing failures should be detected, and predefined actions triggered to minimize the resultant damage.
- There a need for tools to measure maximum execution times and system latencies, and assist in the sizing of systems and the verification of the timeliness specification. For unbounded requirements, the probability that the system size will prove inadequate must also be estimated.
- All system latencies and support mechanisms must be time-bounded, notably communication metrics and the precision of clock synchronization. There must be support for bounded preemption, so that hard real-time components can co-exist with lower priority components without loss of timeliness.

The consequences of these requirements are analysed in more detail in chapter 9, so as to derive a set of techniques for supporting real-time systems in Delta-4.

5.5. Conclusions

The theory underlying bounded-demand systems that are amenable to a complete worst-case analysis has been well-explored and many techniques of design are now mature. These can be used successfully for some classes of system, characterized by limited complexity and a well-behaved environment.

These design techniques are, however, inappropriate for various other possible system requirements, i.e.:

- The environment imposes unboundable demands.
- The demands of the environment can be bounded, but the bounds do not lead to an economically competitive system size. The processing power required may even exceed the bounds of technology.
- Worst-case analysis at design-time may face combinatorial explosions that render it unfeasible or impossibly costly. Design validation is then only possible on the basis of simplifying assumptions that sacrifice the possibility of an absolute assurance.

- Absolute assurances may not be required, especially for soft real-time services, or the customer may choose to forgo them because of the cost and time required to provide them.

These cases seem to require the approach of an older but less mature school of design, called here the "unbounded-demand" school. This school recognises the limitations on scale and complexity imposed by the present state of the art in achieving mathematically exact and complete design assurance. Instead it uses statistical method and probability theory, in which large scale and diversity are an advantage rather than a disadvantage.

Delta-4 aims to support a wide range of real-time systems, including large-scale complex applications, and therefore admits the use of both design philosophies. In one special case, where the demand for hard real-time services can be bounded but the demand for soft real-time services is unboundable, it may be possible to use both design approaches in one system.

In the Delta-4 Open Systems Architecture (OSA), because of its openness and generality, it is difficult to achieve real-time objectives such as the bounding of all system latencies and application execution times. This is a major motivation for the development of the Extra Performance Architecture (XPA) described in chapter 9.