Chapter 2

Overview of the Architecture

Despite the enormous improvements in the quality of computer technology that have been made during the last few decades, failures of computer system components cannot be ruled out completely. Some application areas therefore need computer systems that continue to provide their specified service *despite* component failures — such computer systems are said to be *fault-tolerant*. Distributed computing systems — in which multiple computers interact by means of an underlying communication network to achieve some common purpose — offer attractive opportunities for providing fault-tolerance since their multiplicity of interconnected resources can be exploited to implement the redundancy that is necessary to survive failures.

The Delta-4 fault-tolerant architecture aims to provide a computational and communication infrastructure for application domains that require *distributed* system solutions with various *dependability* and *real-time* constraints (e.g., computer integrated manufacturing and engineering, process control, office systems, etc.). The scale of distribution in the targetted application domains is commensurate with the distances that can be covered by local area networks, i.e., from a few metres up to several kilometres. The Delta-4 architecture also seeks to be an *open* architecture, i.e., one that can use off-the-shelf computers, accommodate *heterogeneity* of the underlying hardware and system software and provide portability of application software. However, in those application domains where *real-time* response is paramount, the heterogeneity and openness may need to be sacrificed to provide the appropriate assurance of timeliness. To this end, the Delta-4 architecture offers two variants (both based on sub-systems that present a high degree of commonality):

- the Delta-4 *Open System Architecture* (D4-OSA) which, as its name suggests, is an *open* architecture able to accommodate heterogeneity,
- the Delta-4 *Extra Performance Architecture* (D4-XPA) which provides explicit support for assuring timeliness.

This chapter sketches out the major characteristics of the Delta-4 architecture and its variants. The key attributes of the architecture — dependability and real-time — are discussed in the section 2.1 and section 2.2 outlines the Delta-4 approach to attaining these attributes. Section 2.3 presents the hardware architecture of Delta-4 and discusses the communication issues. The application software support environment and administration system are presented in section 2.4. Finally, section 2.5 describes the validation of the architecture.

2.1. Dependability and Real-Time Concepts

This section introduces some basic concepts and terminology regarding dependable and real-time computing (for a more detailed exposition of these concepts, see chapters 4 and 5).

2.1.1. Dependability

Dependability can be defined as the "trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers" [Carter 1982]. Dependability is thus a *global* concept that subsumes the usual attributes of *reliability* (continuity of service), *availability* (readiness for usage), *safety* (avoidance of catastrophes) and *security* (prevention of unauthorized handling of information).

When designing a dependable computing system, it important to have very precise definitions of the notions of faults, errors, failures and other related concepts. A system *failure* is defined to occur when the service delivered by the system no longer complies with its *specification*, the latter being an agreed description of the system's expected function and/or service. An *error* is that part of the system state that is liable to lead to failure: an error affecting the service, i.e., becoming visible to the user, is an indication that a failure occurs or has occurred. The adjudged or hypothesized cause of an error is a *fault*. An error is thus the manifestation of a fault *in the system*, and a failure is the effect of an error *on the service*.

Faults are thus the potential source of system undependability. Faults may either be due to some physical phenomenon (inside or outside the system) or caused (accidentally or intentionally) by human beings. They may either occur during the operational life of the system or be created during the design process. *Fault tolerance* — i.e., the ability to provide service complying with the specification in spite of faults — may be seen as complementary to *fault prevention* techniques aimed at improving the quality of components and procedures to decrease the frequency at which faults occur or are introduced into the system. Fault tolerance is achieved by *error processing* and by *fault treatment* [Anderson and Lee 1981]: error processing is aimed at removing errors from the computational state, if possible before a failure occurs; fault treatment is aimed at preventing faults from being activated again[1].

Error-processing may be carried out either by error-detection-and-recovery or by error-compensation. In error-detection-and-recovery, the fact that the system is in an erroneous state must first be (urgently) ascertained. An error-free state is then substituted for the erroneous one: this error-free state may be some past state of the system (backward recovery) or some entirely new state (forward recovery). In error-compensation, the erroneous state contains enough redundancy for the system to be able to deliver an error-free service from the erroneous (internal) state. Classic examples of error-detection-and-recovery and error-compensation are provided respectively by atomic transactions (see, for instance, [Lampson 1981]) and triple-modular redundancy or voting techniques (see, for instance, [Wensley et al. 1978]).

Fault treatment is a sequel to error processing; whereas error processing is aimed at preventing errors from becoming visible to the user, fault treatment is necessary to prevent faults from causing further errors. Fault treatment entails fault diagnosis (determination of the cause of observed errors), fault passivation (preventing diagnosed faults from being activated again) and, if possible, system reconfiguration to restore the level of redundancy so that the system is able to tolerate further faults.

When designing a fault-tolerant system, it is important to define clearly what types of faults the system is intended to tolerate and the assumed behaviour (or failure modes) of faulty components. If faulty components behave differently from what the system's error processing and fault treatment facilities can cope with, then the system will fail. In distributed systems, the behaviour of a node can be defined in terms of the messages that it sends over the network. The assumed failure modes of nodes are thus defined in terms of the messages that faulty nodes send or do not send. The simplest and most common assumption about node failures is that nodes are *fail-silent* [Powell et al. 1988], i.e., that they function correctly until the point of

1 In anthropomorphic terms, error processing can be viewed as "symptom relief" and fault treatment as "curing the illness".

failure when they *"crash"* and then remain forever silent (until they are repaired). The most severe failure mode that can be imagined is that of *fail-uncontrolled* nodes that fail in quite *arbitrary* or "Byzantine" ways [Lamport et al. 1982]. Such nodes can fail by producing messages with erroneous content, messages that arrive too early or too late, or indeed "impromptu" messages that should never have been sent at all. In between these two extremes it is possible to define failure modes of intermediate severity [Cristian et al. 1985, Powell 1991]. Generally, when the assumed failure modes of system components become more severe, then more redundancy (and complexity) must be introduced into the system to tolerate a given number of simultaneously active faults.

2.1.2. Real-Time

Real-time systems are those which are able to offer an assurance of *timeliness* of service provision. The very notion of *timeliness* of service provision results from the fact that real-time services have associated with them not only a functional specification of "what" needs to be done but also a timing specification of "when" it should be done. Failure to meet the timing specification of a real-time service can be as severe as failure to meet its functional specification. According to the application, and to the particular service being considered, the assurance of timeliness provided by a real-time system may range from a *high expectation* to a *firm guarantee* of service provision within a defined time interval[2].

At least three sorts of times can enter into the timing specification of a real-time service:

- a *liveline* and a *deadline* indicating respectively the beginning and the end of the time interval in which service must be provided if the service is to be considered timely,
- a *targetline* (or "soft" deadline) specifying the time at which it would somehow be "best" for the service to be delivered, i.e., the point at which maximum benefit or minimum cost is accrued.

The essential difference between a service deadline and a service targetline is that the former *must* be met if the service is to be timely whereas missing a targetline, although undesirable, is not considered as a failure. An alternative way to specify the timing constraints of real-time services is that of value-time or worth-time functions that indicate the benefit of service delivery as a function of the instant of service delivery (see, for example, [Jensen et al. 1985, Jensen and Northcutt 1990]).

In systems that are "real-time" in the sense defined above, the available resources (finite in any practical system) must be allocated to computation and communication activities so that, as long as specified "worst-case" environmental constraints (e.g., event occurrence rate < specified maximum) are satisfied, it is guaranteed that all critical deadlines are met or that the cost of not meeting targetlines is minimized. In some applications, even if the "worst-case" conditions are exceeded, it may be required that the system make a best effort to ensure that the number or cost of missed deadlines or targetlines is minimized.

Scheduling concerns the allocation of resources that must be time-shared; the resources in question could be processing power, memory, communication bandwidth, etc. In real-time systems, scheduling algorithms are concerned more with the respect of livelines, deadlines and targetlines than with ensuring fairness of resource usage. Consequently, most scheduling decisions need to take account of time explicitly when determining what activity should be scheduled next on a particular resource. Schedules can be calculated off-line (during the system design phase). This is often the case when it must be guaranteed that all critical deadlines be met under specified worst-case conditions. A set of such pre-calculated static schedules may be

2 Note that this view of real-time systems means more than just "high performance" or the ability to react "quickly" in response to asynchronous external events.

necessary if the system is to operate in several different modes (e.g., startup, production, normal shut-down, emergency shut-down, etc.). The performance penalties incurred by such a static approach can be partially mitigated if it can be arranged for non-critical computation or communication to make use of the "holes" that are left in the pre-calculated schedules. In complex real-time applications, on-line (dynamic) scheduling may be preferable to off-line (static) scheduling since the complexity of the application may be such that no *a priori* "worst-case" can be defined. Even if the worst-case is known, it may be desired that the system degrade gracefully whenever the "worst" case is exceeded (consider a radar-tracking system designed to track, at most, 100 aircraft — if 101 aircraft happen to be within range, it may be better to track as many as possible rather than none at all).

If consistent time-dependent decisions (see figure 1) must be made by different nodes of a *distributed* real-time system, then, if the expense of an agreement protocol is not to be incurred for every such time-dependent decision, each node must have a *consistent* view of time. Consequently, the local clocks of each node in a distributed real-time system must be synchronized to a specified *precision* so that all (non-faulty) nodes agree on an *approximate global time*. Furthermore, many applications require this global time to be synchronized to a known *accuracy* of some external standard of *physical time*. The precision and accuracy of clock synchronization determine the granularity at which different nodes can make consistent time-dependent decisions.

if (elapsed_time<10 s)	**then** (execute schedule A) **else** (execute schedule B)
if (alarm_A **before** alarm_B)	**then** (emergency shut-down) **else** (normal shut-down)

Fig. 1 - Examples of Time-Dependent Decisions

2.2. Dependability and Real-Time in Delta-4

The Delta-4 architecture is a distributed architecture that seeks to be both open and dependable, and capable of supporting real-time applications. To be able to satisfy a large range of application requirements in a cost-effective manner, the Delta-4 architecture can provide various degrees of dependability and real-time functionality. This section outlines the dependability and real-time features of the architecture.

2.2.1. Dependability in Delta-4

The Delta-4 architecture provides mechanisms for achieving dependability on a service-by-service basis in "money-critical" applications for which the relevant dependability attributes are reliability, availability and security. The architecture is *not* aimed at life-critical applications for which safety is the major concern. Note however, that reliability, availability and security are of very real interest in safety-*related* applications, e.g., systems that, if they fail, cause emergency shutdown and thus induce not only unavailability but also wear-out of the primary, safety-critical protection system (see annex A).

Delta-4 is concerned essentially with accidental physical faults, i.e., faults in the hardware components and, to a lesser extent, with accidental design faults in software.

The basic paradigm for tolerating hardware faults in Delta-4 is that of *replicated computations* executed by distinct nodes of a distributed system. The units of replication are *software components* — logical run-time units of computation and data encapsulation that communicate with each other by means of messages (only). Software components may be replicated to different degrees according to their degree of criticality and the assumptions made about the failure modes of the underlying node hardware (cf. section §2.1.1). For a given service, if a fail-silence assumption for the underlying hardware can be justified to a degree commensurate with the dependability objectives of that service, then the service can be made single-fault tolerant by duplication. Indeed, duplication of the software components providing the service is sufficient to allow recovery from a fault that causes a node to stop sending messages over the communication system. However, if the fail-silent assumption cannot be justified to a sufficient degree for a particular service, then triplication becomes necessary so that errors due to fail-uncontrolled behaviour can be masked by voting techniques.

It is of course assumed that hardware faults will occur independently in different nodes. In fact, any faults (of hardware or software origin) that manifest themselves *independently* in different nodes can be tolerated by straightforward replication techniques. Some design faults in system software at each node and, to a lesser degree, design faults in application software, can be expected to have such independence in their manifestations since the execution environments of replicas on distinct nodes are essentially different (due to loose synchronization and differing workloads). Such faults are commonly called "Heisenbugs" [Gray 1986]. In addition to this possibility of tolerating certain software design faults, distributed fault-tolerance techniques have the following advantages over tightly-coupled "stand-alone" fault-tolerance techniques:

- specialized hardware is kept to a minimum since distributed fault-tolerance is implemented primarily in software; the cost of specialized hardware design — or re-design when a technology update is required — is therefore minimized,
- geographical separation of resources does not have to be "added on" if disaster recovery is to be provided; the same distributed fault-tolerance techniques can be used regardless of whether the replicas are close to or distant from each other,
- loose-synchronization of replicas (through message-passing) leads to improved tolerance of transient faults that could otherwise simultaneously affect all redundant computations at the same point of execution [Kopetz et al. 1990].

Software design faults that do not manifest themselves in the independent "Heisenbug" fashion cannot be tolerated by replication. Diverse designs must be used to define *multiple variants*. These multiple variants may be encapsulated within a software component that can then be replicated for hardware fault-tolerance. Alternatively, the variants could be executed on distinct nodes in such a way as to tolerate simultaneously both hardware and software faults. The latter approach has been investigated in Delta-4 although it has not yet been implemented (see chapter 14).

In some multi-user application domains where data of a sensitive nature is manipulated, human faults of an intentional nature become a concern of considerable importance. There are two varieties of intentional faults: malicious logic and *intrusions*. The latter variety of fault has been explicitly addressed by the Delta-4 project in the form of *intrusion-tolerance* mechanisms aimed at supplementing conventional intrusion prevention schemes. Replication is of little help when trying to tolerate intrusions. From the confidentiality viewpoint, replication is in fact detrimental to security since the number of different places where sensitive information can be found is greater than in a system without replication. This therefore leads to more potential loopholes for an intruder to penetrate. The project has therefore investigated techniques based on *fragmentation-scattering*; sensitive information is split into fragments and scattered over different nodes so that intrusions only lead to access to partial information (see chapter 13).

The remainder of this sub-section is devoted to Delta-4 replication techniques for tolerating hardware faults and "Heisenbug" software faults.

2.2.1.1. Error Processing. In the context of replicated computation, error processing consists of those techniques for coordinating replicated computation that allow communication and computation to proceed despite the fact that some of the replicas may reside on faulty nodes (figure 2).

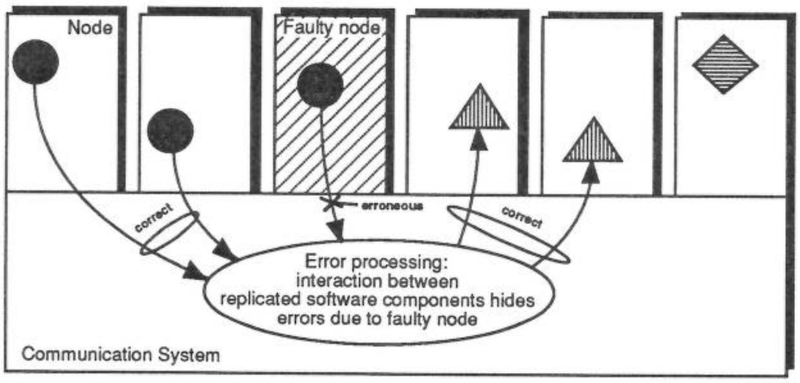
Node
Faulty node
correct
erroneous
correct
Error processing: interaction between replicated software components hides errors due to faulty node
Communication System

Fig. 2 - Illustration of the Principle of Error Processing in the Context of Replicated Computation

Delta-4 provides three different — but complementary — techniques for coordinating replicated computation: active, passive and semi-active replication.

Active replication is a technique in which all replicas process all input messages concurrently so that their internal states are closely synchronized — in the absence of faults, outputs can be taken from any replica. The active replication approach allows quasi-simultaneous recovery from a node failure. Furthermore, it is adapted to both the fail-silent and fail-uncontrolled node assumptions (cf. §2.1.1) since messages produced by different (active) replicas can be cross-checked (in value and time). However, active replication requires that all replicas can be guaranteed to be *deterministic* in the absence of faults, i.e., it must be guaranteed that if non-faulty replicas process identical input message streams, they will produce identical output message streams.

Passive replication is a technique in which only one of the replicas (the primary replica) processes the input messages and provides output messages. In the absence of faults, the other replicas (the standby replicas) do not process input messages and do not produce output messages; their internal states are however regularly updated by means of checkpoints from the primary replica. Passive replication can only be envisaged if it is assumed that nodes are fail-silent. Unlike active replication, this technique does not require computation to be deterministic [Speirs and Barrett 1989]. However, the performance overheads of transferring checkpoints and rolling-back for recovery may not be acceptable in certain applications — especially in real-time applications.

Semi-active replication can be viewed as a hybrid of both active and passive replication. Only one of the replicas (the leader replica) processes all input messages and provides output messages. In the absence of faults, the other replicas (the follower replicas) do not produce output messages; their internal state is updated either by direct processing of input messages or, where appropriate, by means of "notifications" or "mini-checkpoints" from the leader replica. Semi-active replication seeks to achieve the low recovery overheads of active replication while relaxing the constraints on computation determinism. A notifications can be used to force the followers to obey all non-deterministic decisions made by the leader replica [Barrett et al. 1990]. This possibility is particularly relevant for allowing replica-consistent preemption decisions. Like passive replication, this technique resides on the assumption that nodes are fail-silent. This technique is particularly suitable for replicating large "off-the-shelf" software components about which no assumption can be made on replica determinism and internal states (e.g., commercially available database management software).

Table 1 summarises the relative merits of each technique; further details are given in chapter 6.

Table 1 - Comparison of Replication Techniques

Replication technique	Recovery overhead	Non-determinism	Accommodates fail-uncontrolled behaviour
Active	*Lowest*	*Forbidden*	*Yes*
Passive	*Highest*	*Allowed*	*No*
Semi-active	*Low*	*Resolved*	*No*[3]

2.2.1.2. Fault Treatment. In the context of replicated computation, fault treatment consists essentially of three activities:

1) diagnosing the cause of error — i.e. finding out what entity (node or replica) is at fault,
2) if necessary, "passivating" the entity that is judged to be faulty (so that it will not cause further errors),
3) if possible, creating new replicas on fault-free nodes to restore the level of redundancy and thus be able to tolerate further faults (figure 3).

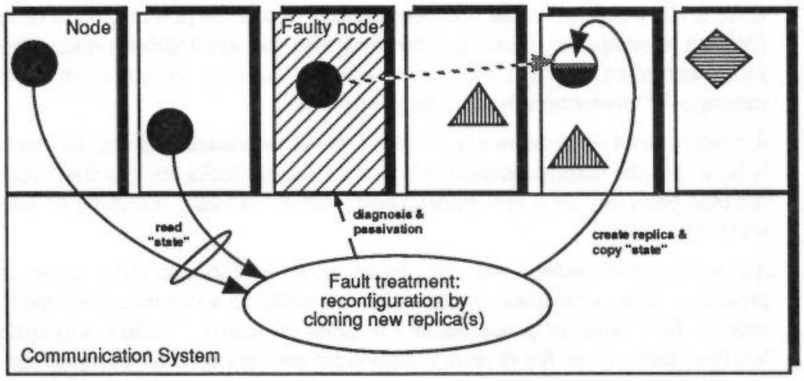
Node
Faulty node
read "state"
diagnosis & passivation
create replica & copy "state"
Fault treatment: reconfiguration by cloning new replica(s)
Communication System

Fig. 3 - Illustration of the Principle of Fault Treatment in the Context of Replicated Computation

3 An extension of the semi-active replication technique to accommodate fail-uncontrolled behaviour is presently being investigated.

The set of sites on which replicas of a given software component can be located is termed the component's *replication domain*. The creation of a new replica on a fault-free node is called *cloning*. The location at which a new replica is cloned may be:

a) the same site at which the original replica was located; this could be the case:
  - either because the fault was considered to a be soft fault (e.g., a Heisenbug) such that re-initialization of the state of the failed replica is sufficient to bring it back on line,
  - or when corrective maintenance of the faulty node has been carried out,

b) on any other site in the component's replication domain thus allowing corrective maintenance to be deferred.

Fault treatment is further detailed in section 2.4.3.

2.2.2. Real-Time in Delta-4

Support for heterogeneity and openness and support for real-time are antagonistic aims; in heterogeneous distributed systems, the local schedulers at each node may not even be the same, let alone time-dependent. This is one of the motivations for the homogeneous XPA (Extra Performance Architecture) variant of the Delta-4 architecture [Barrett et al. 1990] in which heterogeneity and openness are sacrificed to provide assurance of timeliness.

The XPA real-time variant of the Delta-4 architecture differs from the OSA (Open System Architecture) variant by the following features:

- *Nodes are homogeneous:* this eliminates the need, and therefore the performance penalties, of the translation of data representations during node interactions.
- *Nodes are fail-silent:* since openness has been sacrificed in order to achieve real-time performance, nodes are purpose-designed with built-in self-checking to support the fail-silence assumption. This eliminates the need, and therefore the performance penalties, of voting on the contents of transmitted messages.
- *Exclusive use of semi-active replication for fault-tolerance:* the semi-active replication technique (cf. §2.2.1.1) was pioneered during the development of XPA since it allows the potential non-determinism of process preemption to be resolved. Explicit preemption points are inserted into the application code; whenever a follower replica reaches a preemption point, it awaits a "continue" or "preempt by message *#n*" instruction from the leader replica.
- *A synchronized clock service is provided:* for performance reasons, the clock service is built into the communication sub-system. Local clocks are synchronized with an internal precision of a few milliseconds and an external accuracy of less than a second.
- *All nodes use a common real-time local executive:* this executive allows real-time processes to be scheduled dynamically according to a discipline of "most critical process first" and for processes of the same criticality, "earliest targetline first". Various techniques for deriving individual process targetlines from the overall targetline/deadline of a distributed service are being investigated as is the possibility of mapping an off-line pre-calculated schedule onto the dynamic run-time scheduling infrastructure.
- *The communication system is optimized for real-time performance:* in particular, a collapsed-layering philosophy is followed based on an atomic multicast protocol providing multiple "qualities of service" (see §2.3.2.3 below).

Note, however, that not all the Delta-4 target application domains require real-time response. As long as "sufficient" performance is attainable, the heterogeneity and openness criteria may be more important — in which case the OSA variant of the architecture is to be preferred. Some application domains may need features of both variants of the architecture. In this case, the overall system may contain sub-systems constructed according to either the OSA or XPA variants.

2.3. Architectural Sub-Systems

This section identifies the main hardware components of the Delta-4 architecture and outlines the communication facilities provided in the OSA and XPA variants of the architecture.

2.3.1. Hardware

The Delta-4 distributed system architecture has been designed so that it is capable of accommodating the arbitrary modes of failure of *fail-uncontrolled* hosts (cf. §2.1.1) by means of *active replication* techniques (cf. §2.2.1.1). To be able to use standard local area networks instead of resorting to the costly interconnection topologies normally required to accommodate arbitrary failures (see chapter 6), each node consists of two distinct parts: a *host* computer and a *network attachment controller* (NAC) (figure 4).

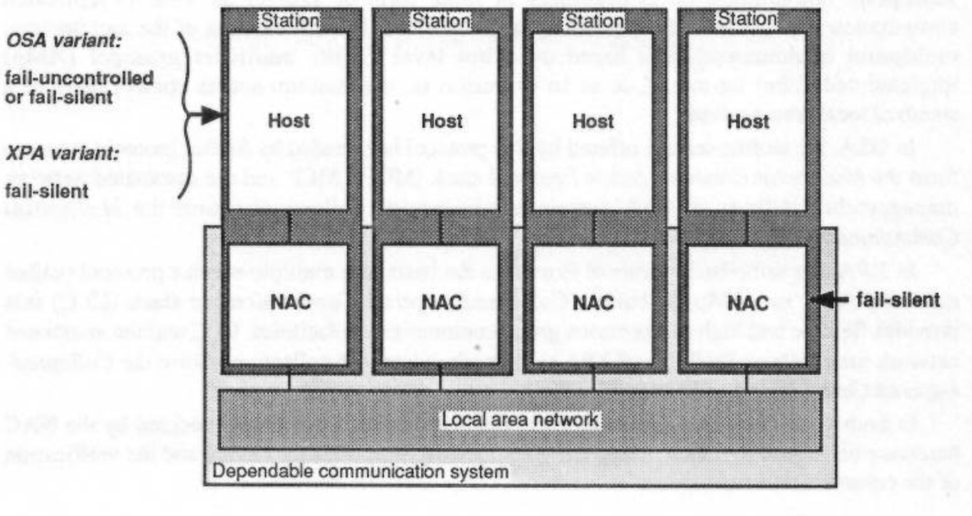

Fig. 4 - Delta-4 Hardware Architecture

Host computers (on which replicas are executed) may be fail-uncontrolled or fail-silent; however, NACs are assumed to be fail-silent. This assumption for the NAC is substantiated by the use of hardware self-checking techniques. The important consequence of this split in failure mode assumptions between host and NAC is that, even when a (fail-uncontrolled) host forwards erroneous data to its NAC, the latter will either process this data in a consistent manner or simply remain silent. The possibility of forwarding the data *inconsistently* to multiple destinations is effectively removed.

As mentioned in §2.2.1.1, active replication has the potential disadvantage of requiring computation to be deterministic. However, when either the *passive* or *semi-active replication* techniques are preferred, it is necessary to assume that hosts are fail-silent. This is possible when the coverage of the host self-checking mechanisms[4] is commensurate with the dependability objectives of the supported application. In the XPA variant of the architecture, where openness and heterogeneity are sacrificed to support real-time applications, all hosts are purpose-designed with built-in self-checking and are therefore assumed to be fail-silent.

To summarise, the OSA variant of the architecture can accommodate both fail-silent hosts and fail-uncontrolled hosts whereas the XPA variant accommodates only fail-silent hosts. Fail-silent NACs are used in both variants.

As for the local area network itself, implementations exist for token bus [ISO 8802-4], token ring [ISO 8802-5] and FDDI [ISO 9314]. For performance reasons, FDDI is preferred in the XPA variant of the architecture. FDDI is also of very real interest in the OSA context since it is able to accommodate a high number of interconnected sites spread out over quite long distances (up to 200 km). Since the local area network must also be dependable if the complete architecture is to be dependable, dual communication media can be employed in all LAN implementations.

2.3.2. Communications

An essential feature of Delta-4 communications is the provision of *multipoint* services. Multipoint communication is necessary in some form or another as soon as replicated computation is considered for providing fault-tolerance. In both variants of the architecture, multipoint communication is based on a low-level *atomic multicast protocol* (AMp) implemented either on top of, or as an extension to, the medium access control layer of a standard local area network.

In OSA, the atomic service offered by this protocol is extended by further protocol layers to form the *Multipoint Communication Protocol* stack (MCP). MCP and the associated network management facilities of OSA system administration collectively form the *Multipoint Communication System* (MCS).

In XPA, the same basic protocol is used as the basis of a multiple-service protocol (called *extended AMp*, or xAMp) to build a *Collapsed-Layered Communication* stack (CLC) that provides flexible and high performance group communication facilities. CLC and the associated network management facilities of XPA system administration collectively form the *Collapsed-Layered Communication System* (CLCS).

In both variants of the architecture, the communication software is executed by the NAC hardware (cf. figure 4) which, being fail-silent, greatly simplifies the design and the verification of the communication protocols.

2.3.2.1. Atomic Multicast Protocol. The Delta-4 atomic multicast protocol (AMp) allows data frames to be delivered to a group of logically-designated *gates*. The protocol ensures, for each frame, that either all addressed gates on non-faulty nodes receive the frame or none[5]. The protocol also ensures that frames are delivered to all addressed gates in a consistent order and that any changes in the membership of a gate group (due to node failure or re-insertion) are notified consistently to all members of that group.

The protocol is based on a centralized, two-phase accept protocol. In the first phase, the data frame is transmitted to all members of the group. The latter then inform the sender whether

[4] Or, equivalently, the conditional probability that, when a host fails, it fails by going silent.

[5] This occurs, for example, if any recipient cannot accept a frame due to lack of receive credit.

or not they are able to receive that frame. If all intended recipients can accept the data frame, the sender transmits an *accept* frame (if a participant perceives the data frame, but cannot accept it because of buffer limitations, a *reject* frame is sent). The protocol tolerates faults of both the sending and receiving nodes as well as transmission faults; it has been implemented in such a way that it can be ported to different underlying networks. It has been successfully implemented *on top of* the Medium Access Control layer of token bus [ISO 8802-4] and token ring [ISO 8802-5].

Another protocol, providing the same service, has been implemented as a *modification of* the Medium Access Control (MAC) layer of the 8802/5 token ring [Guérin et al. 1985]. This extension decreases the number of frames that need to be exchanged by making use of the "on-the-fly" bit flipping possibility of the token ring and is implemented partially in hardware.

Atomic multicasting in Delta-4 is described in more detail in chapter 10.

2.3.2.2. OSA Multipoint Communication Protocol Stack. The OSA multipoint communication protocol stack (MCP) provides two major innovative features:

- the ability to coordinate communication to and from *replicated endpoints*,
- the provision of *multipoint associations* for connection-oriented communication between groups of peer entities.

The *replicated endpoint* paradigm is used to implement the error processing associated with the *active replication* model. Replicated endpoints are provided by an *inter-replica protocol* (IRp) situated at the bottom of the session layer. This protocol coordinates the flow of information sent and received by the different replicas of a replicated software component. The information sent by each replica can be cross-checked for validity (in both the value and time domains) with that sent by the other replicas in the set. This is carried out in the (fail-silent) network attachment controllers (NACs) before sending the actual information over the network. The cross-check itself is based on the exchange of checksums of the data to be sent (for value domain errors) and on the comparison of inter-replica desynchronization (for time domain errors). The flow of information to a replicated software component is also controlled by this protocol; a buffer-status voting mechanism is used to ensure that flow control towards a replicated destination is ensured despite the fact that a faulty replica could choose to refuse messages from the network. The protocol can be configured for either fail-uncontrolled hosts or for fail-silent hosts.

Whereas the replicated endpoint facility provides for the transparent (or *invisible*) multicasting of information to a replicated destination, the MCP *multipoint association* facility allows *visible* multicasting between groups of peer software components (which may or may not be individually replicated). This service is delivered by the MCP multipoint session layer protocol. Facilities are provided that allow software components to join and leave multipoint associations and for information to be multicasted to all or some members of the association. A sending entity may choose to include or exclude itself from the set of destinations for a particular message. All information transferred over such a multipoint association is delivered in a consistent order to all overlapping destinations.

The MCP stack also provides a fully-conforming ISO session layer service so that applications conforming to ISO standards can be used. The replicated endpoint facility is offered both for the multipoint session-layer service and for the ISO-compatible bi-point service.

The MCP stack is further detailed in section §8.1.

2.3.2.3. XPA Collapsed-Layered Communications. The XPA communication subsystem provides the high performance that is a necessary (but not sufficient) condition for stringent real-time applications. To this end, a collapsed-layering philosophy is followed. Of course, removing layers also implies removing services so the functionality of the remaining layers needs to be increased to palliate this. In XPA, only four layers are defined — from the top down: the *group management layer*, the *group communication layer* and the (standard) LAN *medium access control* and *physical* layers.

The group management layer is responsible for choosing the group communication "quality of service" (see below) that is appropriate for the model of replication that is used within a particular group. It also implements an appropriate inter-replica protocol to ensure replica consistency and error-processing (in the semi-active replication or leader-follower model).

The group communication layer is based on an extension of the basic AMp protocol (xAMp) that provides multiple *qualities of service* (QOS) that can be selected according to the specific group communication requirements (see table 2).

Table 2 - Summary of xAMp Service Properties

Quality of service	Agreement	Total order	Causal order per clabel	Rx queue re-ordering
bestEffortN	*best effort to N*	*no*	*FIFO*	*no*
bestEffortTo	*best effort to list*	*no*	*FIFO*	*no*
atLeastN	*assured to N*	*no*	*FIFO*	*no*
atLeastTo	*assured to list*	*no*	*FIFO*	*no*
Reliable	*all*	*no*	*FIFO*	*no*
Atomic	*all or none*	*yes (same gate)*	*yes*	*no*
Tight	*all or none*	*yes (same gate)*	*yes*	*yes*

The *bestEffortN* and *bestEffortTo* services ensure that the frame is received by a number N, or a specified list, of the addressed recipients, *if* the sender does not fail. The *atLeastN* and *atLeastTo* qualities of service enforce stronger agreement, assuring that a given number or subset of the participants will receive the frame, even if the sender fails. The *Reliable* service ensures that, if any recipient received the frame, *all* addressed recipients receive it, even if the sender fails. In all of these first three services, there is no guarantee of order at the receivers and no control of receive credit; frames are forwarded directly to the service users as soon as they are received. The *Atomic* service is the same as that provided by the basic AMp protocol: it guarantees, even if the sender fails, that all recipients, or none of them, receive the frame. This service ensures both consistent ordering (across multiple receive queues) and causal ordering (within a receive queue). The *Tight* service extends on the *Atomic* service by providing the possibility to allow more urgent frames to overtake less urgent ones in the receive queues. Of course, this can only be done if the resulting frame deliveries to users are still consistently and causally ordered.

2.4. Software Environment and Management Issues

Figure 5 provides an abstract view of the overall Delta-4 architecture. The left-hand "slice" of the diagram recapitulates the hardware architecture discussed in the previous section.

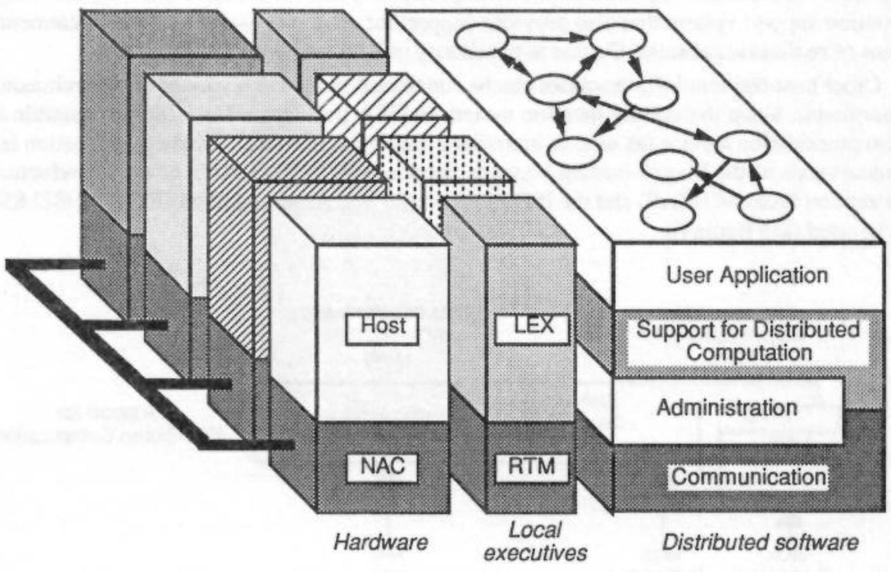

Fig. 5 - Abstract View of the Delta-4 Architecture

The middle slice of the figure represents the local executives residing on the host and NAC hardware. The local execution environments (LEXes) of the hosts are shown shaded differently (like the hosts) to underline that, in the OSA variant of the architecture, heterogeneous host hardware and executive software may be accommodated. In practice, the present implementations all use different flavours of UNIX. In principle, however, the design philosophy of the OSA variant of the architecture would allow an implementation in which both UNIX and non-UNIX systems could co-exist. In the XPA variant of the architecture, LEXes (and hosts) are homogeneous; today, a specially-developed real-time version of UNIX (RT-UNIX [SVC200]) is used but future implementations may adopt some other real-time operating system. In both variants of the architecture, the NACs use a real-time monitor (RTM) that may be homogeneous across all NACs (although this is not of course mandatory).

The right-hand slice of the figure represents the *distributed Delta-4 software* that can be represented in four parts:

- the distributed user application software represented as a set of "software components" (logical units of distribution) that communicate by messages (only),
- the host-resident infrastructure for support of distributed computation,
- the computation and communication administration software (executing partly on the host computers and partly on the NACs),
- the communication protocol software (executing on the NACs).

A particular host-resident infrastructure for supporting open *object-oriented* distributed computation has been developed for the Delta-4 architecture: the *Delta-4 Application Support Environment* (Deltase). According to the philosophy of "open" distributed processing, Deltase facilitates the use of heterogeneous languages for implementing the various objects of a distributed application and allows the differences in underlying LEXes to be hidden (see section §2.4.1 below). Deltase provides the means for generating software components called "capsules" (executable representations of objects). Capsules are coordinated by the Deltase execution support system that also provides support for error processing and fault treatment by means of replicated capsules. Deltase is mandatory in XPA and is optional in OSA.

Other host-resident infrastructures can be considered in the OSA variant of the architecture. In particular, since the communication system in OSA provides a fully ISO-compatible (bi-point) presentation service (as well as innovative multipoint services), standard application layer protocols such as the Manufacturing Message Service (MMS) [ISO 9506] of the Manufacturing Automation Protocol (MAP) and the ISO File Transfer and Access Method (FTAM) [ISO 8571] can be used (see figure 6).

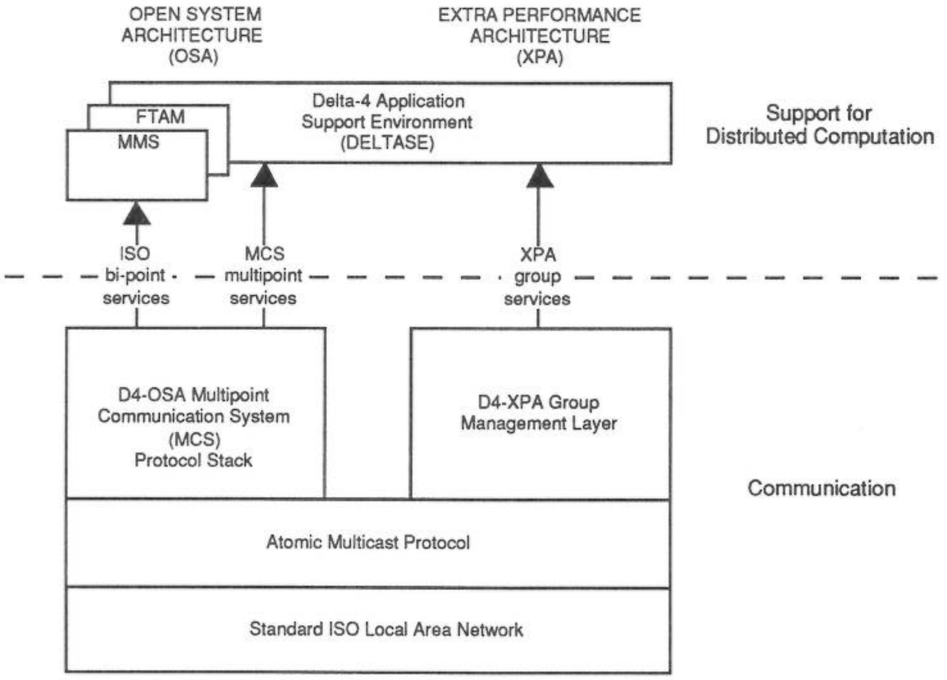
OPEN SYSTEM ARCHITECTURE (OSA)
EXTRA PERFORMANCE ARCHITECTURE (XPA)
FTAM
MMS
Delta-4 Application Support Environment (DELTASE)
Support for Distributed Computation
ISO bi-point services
MCS multipoint services
XPA group services
D4-OSA Multipoint Communication System (MCS) Protocol Stack
D4-XPA Group Management Layer
Communication
Atomic Multicast Protocol
Standard ISO Local Area Network

Fig. 6 - Computation and Communication Support in OSA and XPA

In addition to the communication software (already outlined in the previous section), a further important sub-system in the distributed software slice of figure 5 is that of *system administration*. Delta-4 system administration provides the mechanisms for managing a Delta-4 system: it consists of both support for *network management* in the classic sense as well as support for managing the computation system (see section 2.4.2 below).

In conjunction with Deltase/XEQ, the Delta-4 administration system carries out the automatic *fault treatment* functions mentioned in §2.2.1.2: this is detailed in section 2.4.3 below.

2.4.1. Application Support Environment

The purpose of *Deltase*, the *Delta-4 Application Support Environment*, is to provide a *single virtual machine* for the support of modular applications on Delta-4 systems. This single virtual machine conceals the (possibly-heterogeneous) underlying hardware and software, consisting of computers, local operating systems, language systems and the communication system.

The *computational model* provides the model to be used for application programs that are to be supported by Deltase. This model has its origins in work on *Open Distributed Processing* (ODP) within the ANSA (later ISA) project [ANSA 1989] and related work on standardisation of a *Support Environment for ODP (SE-ODP)* within ECMA [ECMA TR/49 1990]. ODP is concerned with the definition of a generic architecture for distributed processing systems. The current implementation of Deltase is a prototype Support Environment for Open Distributed Processing extended to include the Delta-4 approach to fault-tolerance and real-time.

The decomposition of an application into a number of *language-level* program modules, which interact with one another by means of procedure calls, is a widely-used method for applications that are implemented as a single program and executed on a single machine. Deltase enables this application structure to be supported on Delta-4 systems; that is, to extend, to distributed fault-tolerant systems, a widely-used application structure.

Since fault-tolerance and distribution are made *transparent* to the application programmer, an application written for use with Deltase can be used, without change to the application code, on a small-scale system, consisting of a single host machine with a suitable implementation of Deltase.

Deltase supports this computational model through a combination of:

- Generation Support — referred to as Deltase/GEN.
- Run-Time Support — referred to as Deltase/XEQ.

The language-level program modules are independent of the underlying computing environment, and can be ported between different computing environments where that language is supported. Thus Deltase provides vendors, and users, with a natural and economic way to migrate their software investment to new technology.

2.4.1.1. Concepts

2.4.1.1.1. Computational Objects and Services. For use with Deltase, the program modules (which together constitute an application) must be *computational objects*. Each one encapsulates a private *state*, and provides one (or more) clearly-defined *service interface(s)* for operations on that state; each service interface offers a set of *services*. These services provide the only way for one computational object (the one invoking the service) to read or modify the private state of another computational object (the one offering the service). The specification of a service interface is independent of its many possible implementations; each service is defined by its parameters, results, effects and constraints.

2.4.1.1.2. Remote Procedure Call. The *Remote Procedure Call (or RPC)* mechanism is a direct extension, to distributed systems, of the widely-used procedure call mechanism provided by many conventional programming languages. RPC provides a means of programming the interactions between objects, using existing language constructs, with each service treated as a separate procedure. The same language construct is therefore used both for the invocation of a service that is provided internally (by a local procedure), and for the invocation of a service provided externally (by another object). RPC is an abstraction away from messages, offering a familiar language-level construct for programming the interactions between computational objects. With language-based type-checking mechanisms at both the calling and the called objects, the use of RPC provides an assurance that code at both ends conforms to the same interface specification.

2.4.1.1.3. Interface Trading. Before two computational objects can interact by way of RPC, a *logical path* between them must be established, through *interface trading*. An object (service provider) may *offer* its services for use by other objects, by *exporting* that service interface; this offer will be added to a catalogue of such offers. An object requiring the use of a particular set of services (service user) may find a service provider by invoking a search of that catalogue for offers of service, based on the specified interface type name. The effect of *importing* a service interface is to establish a logical interaction path; such a path would enable the service user to invoke the services provided by the service provider, through that particular interface.

2.4.1.1.4. Threads. To allow a service provider to process a number of service requests concurrently, Deltase supports the use of multiple threads of control within an object. Each thread handles one service request at a time; such threads are called *server* threads, and are managed by Deltase. Server threads are not directly visible to the applications programmer, but the mutual exclusion of access to shared internal data remains the responsibility of the application programmer. Within an object, a server thread may create distinct subsidiary (*forked*) threads that disappear when their activity is complete. Typically, such forked threads are used to allow local processing to continue in parallel with service requests invoked by RPC (whose threads are blocked); computational progress may then occur on several hosts on behalf of a single original service request.

2.4.1.1.5. Transformers. To interact with alien computational worlds, Deltase uses the concept of a *transformer*. This is a "half-object"; it appears from one side as a conventional Deltase object, but from the other side, interacts according to the rules and conventions of some other computational world. A transformer object can be used to provide access from the Deltase world to an existing (non-Deltase) software package. Typically, the software accessed in this way would be proprietary and commercially important. Conversely, a transformer object may provide access, from the non-Deltase world, to a service provided in the Deltase world (with the consequent benefits in terms of representational transparency and Delta-4 dependability), whilst preserving non-Deltase interface conventions and standards. More generally, transformer objects provide a means of interworking with other proprietary or standard computational worlds, presenting these to the Deltase world in a consistent, and preferably generic, manner. An important use for transformer objects is in interfacing to input-output devices, either directly or by way of existing drivers.

2.4.1.2. Implementation. Deltase/GEN is used to generate executable software components from the language-level computational objects; the term *capsule* is used for a software component that is generated from a computational object, using Deltase/GEN. A capsule is directly executable on a particular type of host and under the chosen local operating system; for example, for operation under UNIX, a capsule would be generated (and then handled) as a UNIX process. A capsule consists of the compiled source code of one or more computational objects, together with additional code that provides the environment to map the computational object(s) onto the local operating system. This additional software is referred to as the *envelope*; each capsule contains one envelope, which represents all the code necessary to support all the computational objects within that capsule. The envelope is generated automatically by Deltase/GEN and includes the following:

- *interface modules* corresponding to the services exported or imported by the computational objects; interface modules are generated automatically from the service interface specifications,
- a set of environment-dependent library procedures,
- a thread scheduler, for scheduling of threads local to that capsule,
- object-dependent support for error-processing and fault-treatment (see below).

2.4.1.3. Impact of Fault-Tolerance. Deltase combines ODP concepts with the Delta-4 approach to fault-tolerance based on user-transparent replicated computation. From the application programmer's viewpoint, fault-tolerance is transparent. Deltase/GEN can automatically generate capsules and provide them with the additional run-time code necessary to support the Delta-4 replication models.

The error processing associated with the *active replication* model is carried out by the underlying communication system. However, this model requires that the observed behaviour of each replica be identical in the absence of faults (cf. §2.2.1.1). From the viewpoint of Deltase/XEQ, this means that each replica of a capsule must process requests *deterministically* despite any possible internal parallelism. One means of ensuring this determinism is to structure each capsule as a *state machine* [Schneider 1990] such that capsule replicas process requests and supply responses in a strictly identical order. Adherence to the state machine model means that the object envelope must schedule threads in the same sequence and that the transfer of control from one thread to another must be at the same point in computation at all replicas.

In the case of the *passive replication* model, Deltase/GEN generates an object envelope that allows a capsule replica to act either as the *primary* replica or as a *back-up* replica. It is this special software in the object envelope that is responsible for sending and receiving checkpoints at appropriate times and updating the state of passive replicas. In the *semi-active replication* model, Deltase/GEN must again generate an object envelope that allows a capsule replica to act either in *leader* mode or *follower* mode.

Deltase also provides support for the *cloning* mechanism associated with fault treatment (see section 2.4.3 below).

2.4.2. System Administration

System administration provides the mechanisms for managing a Delta-4 system: it consists of both support for *network management* in the classic sense as well as support for *computation management.*

2.4.2.1. Management Functions. The term "administration" covers the set of management functions concerned with planning, organising, supervising and controlling a Delta-4 system. In particular, these management functions are concerned with maintaining a specified level of service and enabling the system to evolve by providing the means to add new facilities. Management of distributed systems is a complex and often ill-defined topic, for the following reasons [Sloman 1987]:

- distributed systems are large and complex,
- their components, very diverse in nature, all need to interact and be managed,
- there are many different facets to management; management has to deal not only with configuration, designation, performance, faults, security and accounting, but also with people and computers,

- there is a floating boundary between management and the normal functionality of the system.

The management of distributed systems is currently under intense discussion in the academic field as well as in the immense ISO standardisation work, international multi-vendor initiatives (MAP, CNMA, OSI/Network Management Forum) and network management product developments (IBM's Netview, HP's Open View, DEC's EMA, etc.). An important consequence of the sophisticated Delta-4 fault-tolerance facilities is that system administration must cover not only the management of communication resources but also those necessary for computation. Presently, as is the case in MAP, three categories of management functions are supported [MAP]:

- management of system configuration and naming,
- performance management,
- fault and maintenance management.

The latter category of functions is particularly important in Delta-4 because it includes all aspects of fault treatment mentioned earlier. A further category of management functions concerned with security is also being investigated (see chapter 13).

2.4.2.2. Management Model and Design Principles. Delta-4 management is based on the ISO/OSI concept of "managed objects", which has been consistently extended such that it can be applied to objects outside the OSI scope, for instance hardware and software components. Apart from their normal functionality, managed objects are characterized by:

- *attributes* such as their version identification, their state and their operational parameters, information relative to their error and performance statistics, their dependencies with respect to other managed objects, etc.,
- specially-defined management *operations* to allow access to, and manipulation of, object attributes, e.g., create, clone, delete, re-initialize, set_value, get_value, etc.,
- *events,* which are a means by which a managed object delivers management information asynchronously (events are a special form of attribute).

In accordance with the current state-of-the-art in distributed systems management [ANSA 1989, Sloman 1989], management specific to a *set* of managed objects is termed *domain management* and the set of objects, a *domain*. Examples of domains are sets of nodes, sets of replicas or sets of software components. Since replicas of a given software component may only be located on certain nodes (i.e., those possessing the resources necessary for their execution), the set of such nodes is termed the software component "replication domain".

Each management domain in Delta-4 is assigned an architectural component called a *domain manager*. A domain manager may consist of a single *domain manager process* (which may be replicated) or a set of *peer domain manager processes* that cooperate within the domain boundary to carry out domain-specific management tasks. Domain managers of different domains cooperate to fulfil common management policies.

Domain managers make use of two sorts of management information: a) management information *about* managed objects that is integrated within the domain manager, and b) management information integrated within the managed objects themselves that is closely related to their normal functionality. Both kinds of management information are conceptually summarised under the term *Management Information Base* or *MIB*, which is therefore, by its very principle, distributed.

The Delta-4 implementation presently comprises three types of domain managers:

- a manager of communication objects within a communication domain — in accordance with the MAP terminology, a domain manager process of the communication domain manager is termed a "Systems Management Application Process" or SMAP,

- managers of application objects within replication domains, or "Replication Domain Manager" or RDM,
- a manager of a particular object within a replication domain, that contains management information, called the Global-MIB ("MIB Domain Manager").

Communication domain managers in Delta-4 manage types of communication objects beyond those defined in present ISO, MAP and IEEE standards. Furthermore, these communication objects may be replicated. By essence, the SMAPs cannot be replicated themselves. Consequently, SMAPs are executed by self-checking hardware — the fail-silent network attachment controllers (cf. §2.3.1). A special protocol (M-CMIP: Multipoint — Common Management Information Protocol) is used for the exchange of information between SMAPs. To manage replicated communication objects, SMAPs on different Delta-4 nodes must have a consistent view of the non-local features of such objects. To achieve this, the corresponding communication management information is stored in a separate global management data base, the "Global-MIB". As this is critical information, the Global-MIB is replicated and thus managed by a (replication) domain manager of its own (the MIB Domain Manager).

The architectural approach has also been applied to objects on the application level (see section §2.4.3):

- *Processes:* They form the basis on which (existing) applications are built. In the present Delta-4 implementation, a Deltase capsule is represented by a UNIX process.
- *Files:* Global files may be useful in certain applications, so a server for managing replicated global files also been implemented.

These kernel management components, which support the Delta-4 fault-tolerance approach, are supplemented by a set of management application tools with graphical human interfaces to allow the control and the visualisation of the Delta-4 system behaviour. Among these are a configuration toolbox for the Global-MIB and various status, utilization and performance monitoring tools.

2.4.3. Cloning

Cloning techniques have been investigated and implemented for both Deltase capsules and files. Cloning of higher level objects, for instance file servers or databases, may be based on the cloning mechanisms of these basic managed objects.

The implemented cloning machinery for both capsules and files comprises:

a) A *replication domain manager* (RDM), which applies a reconfiguration strategy to a set of replicated processes or files that have the same replication domain. The reconfiguration strategy defines when and where new replicas are to be instantiated, for instance:

   - restoration of the replication degree of objects which have lost a replica due to node failure by cloning them to nodes offering spare redundancy, or
   - migration of all objects from a node, e.g., due for maintenance.

b) *Object manager entities* (OMEs), local to the managed application objects, that perform the cloning protocol on request of the replication domain manager.

The cloning of Deltase capsules is carried out by three generic components: a capsule RDM, capsule OMEs and *factories*. A factory exists on each node and is responsible for instantiation of capsule replicas on that node. To instantiate a capsule replica at a particular node, the replication domain manager makes use of the services offered by the factory resident at that node. Each capsule includes an OME as part of its envelope that is responsible for carrying out those operations that can only be done from within the capsule. The capsule OME consists of a set of library procedures, which are included in the envelope as part of the capsule generation process.

The initial state of a new capsule replica is obtained from the file generated by the capsule generation process. The current state of the replicated capsule is obtained from the OMEs of the existing replica(s) by what is essentially the same activity as the checkpointing activity associated with the passive replication model. Further local state information is handled locally by the OME at the node where the new replica is created. This information consists of data specific to the local environment of the new replica such as references to local resources.

✣ ✣ ✣

Further details on Deltase can be found in chapter 7, and on system administration and cloning in section 8.2 for OSA and in section 9.5 for XPA.

2.5. Validation

For an architecture to be worthy of the epithet "dependable", it is necessary that users of the architecture may *justifiably* place their confidence in the architecture. Consequently, such an architecture must undergo extensive validation both from the verification viewpoint (removal of faults in the specification, design and implementation) and the evaluation viewpoint (quantification of the provided dependability and performance).

Validation can and should be carried out at each step in the process of producing a system. At the specification stage, validation consists essentially of verifying that the specifications of the future architecture are mutually consistent with each other and with the requirements of the intended application domains. This "informal" verification is carried out in Delta-4 by a "peer review process" either explicitly during scientific and technical committee meetings or implicitly, as in the production of this book describing the architecture's concepts.

More tangible validation activities are carried out during the design and implementation phases. Ideally, all components of a system should be extensively validated. However, for the money-critical (as opposed to life-critical) applications for which Delta-4 is intended, it was decided to restrict the validation to the most important (or the most critical) sub-systems.

2.5.1. Design Validation

Design validation is centred on descriptions or models of the future implementation. Its purpose is: a) to verify that these models are consistent with the specifications or b) to evaluate (predict) some characteristics (e.g., performance, dependability) of the future implementation. Two design validation activities have been carried out.

- *Protocol verification* aimed at removing faults in the protocol design has been carried out on various versions of the essential atomic multicasting protocol, AMp. This work employed temporal logic specifications of the required properties of AMp and verification that a formal description of the protocol, in Estelle/R, satisfied these properties [Baptista et al. 1990]. Present protocol verification work is centred on the inter-replica protocol (IRp) of the session layer of the OSA variant of the architecture.
- *Dependability evaluation* work is being carried out with a view to quantifying the dependability actually achievable by the Delta-4 architecture. The work carried out to date has centred on the communications infrastructure and has shown the importance

of coverage of the network attachment controller self-checking mechanisms (to substantiate the fail-silence assumption) and identified the conditions under which redundant communication media should be employed [Kanoun and Powell 1991]. Present dependability evaluation work is aimed at evaluating the availability and reliability of applications making use of the various Delta-4 fault-tolerance models.

2.5.2. Implementation Validation

Implementation validation is centred on *testing* actual prototype versions of the architecture instead of on models. Like design validation, its purpose is twofold: a) to verify that the implementation provides the specified functionality and b) to evaluate (measure) some characteristics of the actual implementation. Implementation validation has centred on two aspects.

- *Fault injection* (into the prototype hardware) has been used as a means for validating: a) the self-checking mechanisms of the Delta-4 network attachment controllers (NACs), and b) the implementation, on these NACs, of the atomic multicasting protocol (AMp) [Arlat et al. 1990]. This activity contributes to the verification of the absence of implementation faults (and residual design faults) — in particular, it verifies that the system works as intended *in the presence of the very faults it is meant to tolerate*. Fault-injection also enables the measurement of the effectiveness of the built-in error detection and fault-tolerance mechanisms by means of coverage, dormancy and latency estimations.
- *Software reliability evaluation* is being carried out on many of the major software subsystems of the architecture. Static testing tools are being used to identify important characteristics of the implemented software. In addition, failure data is being collected during the software development and testing phase to predict the rate at which it can be expected that residual design or implementation faults will cause the system to fail when in operational use.

✣✣✣

Further details on the validation of the Delta-4 architecture can be found in chapter 15.