The Delta-4 Approach to Dependability in Open Distributed Computing Systems

D.Powell
LAAS-CNRS
7 avenue du Colonel Roche
31077 Toulouse
France

P.Verissimo
INESC
Rua Alves Redol 9
1000 Lisbon
Portugal

G.Bonn
IITB-Fraunhofer
Sebastian-Kneipp-Str 12-14
D-7500 Karlsruhe 1
West Germany

F.Waeselynck
BULL-MTS
1 rue de Provence
38432 Echirolles
France

D.Seaton
Ferranti Computer Systems Ltd.
Withenshawe
Manchester M22 5LA
United Kingdom

ABSTRACT

As part of the European Strategic Programme for Research in Information Technology (ESPRIT), the Delta-4 project is seeking to define an open, fault-tolerant, distributed computing architecture. This paper presents the overall Delta-4 framework for open, fault-tolerant, distributed computing systems and sketches the current implementation which is based on a local area network with specific atomic multicasting and error-processing protocols for communicating between replicated software components.

Keywords: Fault-tolerant distributed systems, open distributed processing, replicated processing, atomic multicast

Introduction

The Delta-4 project seeks to define a distributed real-time computing architecture that is (a) *open*, i.e. that allows integration of heterogeneous computing elements and (b) *dependable*, i.e. that provides a quality of service on which users can have justified reliance. Open distributed computing implies not only the definition of inter-node communication protocols that are independent of the particular hardware and local executives at individual nodes of the distributed system but also the definition of appropriate distributed computational and administrative techniques.

The primary aim with respect to dependability, is to define a homogeneous set of techniques for user-transparent tolerance of hardware faults* by means of replicated software components residing on distinct host computers. This homogeneous set of techniques allows different systems and applications, with different dependability and performance requirements, to be configured without redesign of the individual software components. Furthermore, the degree of fault-tolerance is incrementally specifiable on a component-by-component basis. Replicated software components are similar to the notion of k-resilient objects in the *ISIS* [6] and object groups in *ANSA* [3]. However, Delta-4 seeks to provide fault-tolerance under a wider set of fault assumptions and with extended performance properties.

The first part of this paper presents the overall Delta-4 framework for open distributed fault-tolerant computing systems and part two outlines the current implementation.

1. Overall framework

This section of the paper first describes the general approach to providing tolerance of hardware faults in an open distributed system and then outlines how such a system is viewed from an administrative viewpoint.

* Work is also being carried out within the project on the tolerance of accidental design faults in software and intrusion-tolerance techniques for security.

1.1 Fault-tolerance

1.1.1 Error-processing and fault-treatment. Fault-tolerance is carried out by *error-processing* and *fault-treatment* [2, 16]. Error processing is aimed at removing errors from the computational state, if possible before the occurrence of a failure in the service delivered by the system; fault treatment is aimed at preventing faults from being activated again.

In Delta-4, the basic units of fault-tolerance are the nodes or host computers of a distributed computing system. Each *individual* host may be a classical mono-processor, a multi-processing system or indeed a specialized system such as an array processor. The distributed software system that is executed by such interconnected nodes can be globally viewed as a set of *software components* that share no common memory and which communicate by means of explicit *messages*. Each software component can be seen as possessing one or more input and output ports through which messages are received from and transmitted to other software components. Bindings between ports on different components can be statically or dynamically created or destroyed.

Replication of individual software components on separate hosts provides the redundancy that is necessary for error-processing (figure 1). In this context, error-processing involves the management of interactions between replicated software components by error detection/recovery or error compensation in order to mask the fact that one (or more) of the underlying nodes may be faulty. Subsequent fault-treatment can be seen as self-repair facility that identifies and passivates faulty nodes and, by creating new replicates, can allow software components to survive further faults (within the limits of available hardware resources).

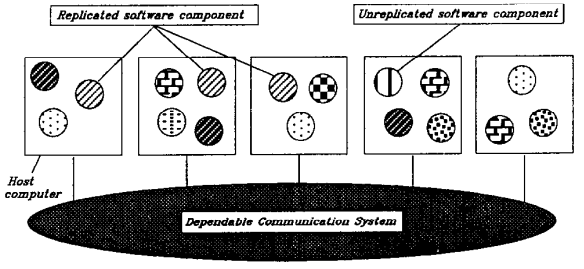

Figure 1 : Abstract view of Delta-4 architecture

1.1.2 Fault assumptions. Two extreme assumptions about faulty hosts are allowed: *fail-silent* hosts and *fail-uncontrolled* hosts. The set of fault-tolerance techniques provided must allow either assumption to be made depending on the available host hardware and/or the criticality of the application. Intermediate host-

fault assumptions may be introduced depending on the properties of the underlying communication system (assumptions concerning the inter-host communication system are deferred to paragraph 1.1.5).

1.1.2.1 Fail-silent hosts. Numerous implementations of fault-tolerance in distributed systems make the simplifying (and simplistic) assumption that host computers fail "cleanly" by just stopping to send messages, i.e. that hosts are "fail-silent" (this failure mode corresponds to the "halt-on-failure" property of fail-stop processors [21], the notion of "fail-fast" hardware [5] and "crash" failures [7]). Fail-silent hosts send only correct messages and error-detection may be achieved relatively simply by means of background "I'm alive" messages. Replication of software components on such hosts serves solely for recovery from errors; error-processing protocols can be implemented so that a software component can tolerate t active faults using just t+1 replicates.

However, the fail-silent assumption implies that hosts possess perfect, zero-latency self-checking mechanisms. Clearly, such an assumption cannot be justified in an open system that must accomodate heterogeneous, off-the-shelf computers, especially in critical applications and it would be judicious to allow a much more relaxed asssumption.

1.1.2.2 Fail-uncontrolled hosts. The most unrestrictive fault assumption about host computers is that they are fail-uncontrolled, i.e. that they do not possess any local error-detection mechanisms and can thus produce quite random or even malicious behavior. A fail-uncontrolled host may:
- omit to send (some) messages,
- send extra messages,
- send messages with erroneous content,
- refuse to receive messages.

Note that "sending" and "receiving" in this context means forwarding messages to and accepting messages from the communication system.

With fail-uncontrolled hosts, the replication of software components and the associated error-processing protocols must serve not only for recovery from errors but also for their detection. In order to tolerate t active faults, at least 2.t+1 replicates are necessary for there to be a majority of replicates executing on non-faulty hosts.

1.1.3 Error-processing strategies for fail-silent hosts. Since replication on fail-silent hosts serves solely for error recovery and not for error-detection, back-up replicates can be either *active* or *passive*.

1.1.3.1 Active replicates. The provision of active replicates allows for quasi-instantaneous recovery from host hardware faults if it can be guaranteed that all replicates of a given software component remain *consistent* so as to produce the *same output messages* (on each output port). Sufficient conditions for replicate consistency are discussed in section 1.1.5. Selection of a particular output message from the set of replicated messages can be carried out either cyclicly among all replicates (a virtual ring mechanism must be implemented) or just by choosing the first available message. Note however, that the latter technique implies that some other synchronization must exist so that slow replicates do not lag permanently behind fast ones.

1.1.3.2 Passive replicates. A passive replicate consists of a copy of the code of the software component together with a copy (checkpoint) of some previous state of the component from which execution can be resumed should the active replicate fail. Passive back-up replicates allow host processing capacity to be be economized but at the expense of a *permanent* communication overhead to provide back-ups with checkpoints for backward error recovery and a *temporary* processing overhead, when a fault occurs, for a back-up to re-execute from its last checkpoint.

The well-known *domino effect* [20] can be avoided if the checkpointing technique is such that resumption of activity (by a previously-passive replicate) from the last checkpoint avoids the need to request re-sending of previously processed input messages and prevents duplicate output messages from being sent.

In order to avoid having to request re-sending of previously processed input messages, the passive replicate(s) must have access to either the messages processed by the active replicate or the corresponding state modifications. This can be done by ensuring that the passive replicate(s) maintain a queue of input messages processed by the active replicate since the last checkpoint was taken.

Duplicate output messages can be avoided by systematic or periodic checkpointing. *Systematic checkpointing* involves the creation of checkpoints whenever the active replicate communicates some of its internal data to the outside world, i.e. whenever a message is sent. Thus, rollback to the last checkpoint never requires re-sending of an output message. *Periodic checkpointing* decreases the number of checkpoints that are taken and can be carried out in a similar way to that described in [4]. During recovery, any output messages generated by the previously-passive replicate are checked against a log of previously sent messages and only sent over the network if no equivalent message is found.

Correct recovery using periodic checkpointing requires that replicates remain *consistent* (as for the active replicate strategies) so as to produce the same output messages. Furthermore, if it can be guaranteed that the order of output messages is identical, then the log of output messages can be replaced by a simple tally of the number of previously-sent messages. In the case of systematic checkpointing, a back-up replicate need not necessarily produce the same messages (either in content or order) as those that would have been produced by the previously healthy active replicate. This relaxation of the requirements concerning replicate behavior is an interesting advantage of systematic checkpointing to be weighed against the disadvantage of higher overhead.

1.1.4 Error-processing strategies for fail-uncontrolled hosts. Replication on fail-uncontrolled hosts must serve not only for recovery from errors but also for their detection. Whereas errors resulting from faults in fail-silent hosts are, by definition, time-domain errors (permanent lack of response), errors due to faults in fail-uncontrolled hosts may be either in the time-domain (late or missing messages or extra messages) or in the value-domain (messages with erroneous content). The error-processing protocols that control the interactions between replicated software components must take account of all error-types. In this case, it is the error-processing protocol entities that must signal detected errors to the administration system (instead of the converse) so that the latter can carry out fault treatment.

1.1.4.1 Time-domain errors. Time-domain errors are particularly difficult to process in an open distributed environment. Even if local clocks are kept approximately synchronized [17, 22], this is of no direct use unless the local scheduling of replicates at each node explicitly uses the resulting global time reference to determine the instants at which messages should be sent or received and if communication delays are bounded [25, 15]. Neither is necessarily true in an open distributed system. Scheduling techniques on heterogeneous hosts cannot be assumed to be the same, let alone time-dependent. Furthermore, the total message traffic between components of a dynamically-evolving system cannot in practice be assumed to be deterministic; consequently, delays over shared communication resources cannot in general be bounded.

We are forced to conclude that, in practice, the only viable basis for dealing with time-domain errors in an *open* system is the use of time-outs. However, it is important to underline that although expiration of a time-out is indeed an error, this does not necessarily mean that the sending node is faulty and will be irrevocably removed from the system. Properly-designed error-processing protocols will mask such errors but report them to the administration system. The latter can first try to alleviate the incriminated node by load-balancing and will only passivate the node if it diagnoses that the number of such reported errors has exceeded a given threshold.

1.1.4.2 Value-domain errors. Value-domain errors are detected by comparing equivalent output messages from the different replicates of a software component. For such a comparison to be possible, it is of course necessary that replicates on non-faulty nodes remain *consistent* so as to produce the same output messages. It is also necessary that there be some way of identifying *equivalent* messages, i.e. messages that should be the same and can thus be compared. The comparison itself can be carried out either before or after propagating a message to the destination component(s).

(a) Validate-before-propagate. Before propagating any message to the destination component(s), the validate-before-propagate technique requires that messages

from at least m out of n different replicates be first compared and found to agree. If it is required to tolerate up to t active faults, then it is necessary that m > t and n > 2.t.

In absolute terms, as soon as m messages agree, then one can be sure that the consensus message that is propagated is error-free. However, to prevent faults in the remaining n-m nodes from remaining dormant, it is necessary to ensure that:

- either, messages from a specific set of m replicates be compared and that the set of compared replicates is periodically changed to ensure activation of all n replicates,
- or, that an attempt be made to systematically compare the messages from all n replicates - it is however possible in this case to compare the remaining n-m messages after having propagated the consensus value.

(b) Propagate-before-validate. With the previous technique, even if the first replicate produces an error-free message, propagation must wait until m-1 slower replicates are found to agree. The aim of the propagate-before-validate technique is, on the contrary, to allow computation to proceed at the rate of the fastest replicate as long as no fault occurs. Message comparison is in effect pipelined with the useful computational process.

However, it is quite evident that as it stands and on the contrary to all the previous techniques for fail-silent or fail-uncontrolled hosts, such a technique only allows for error-detection and some further mechanism must be implemented to ensure error-confinement. One way of achieving this is by creating global checkpoints so as to allow computation to be carried out tentatively and to be undone if any message used in part of that computation is subsequently found to be erroneous. A transactional model of computation provides a suitable framework for such a global checkpointing facility. Before commiting a transaction (i.e. before making results of computation permanent and visible outside the transaction), all messages sent by replicates of components involved in the transaction must have been checked against at least m-1 other replicates. Furthermore, to ensure that no faults remain dormant, an attempt must be made to compare all messages from all n replicates of each component - if only a minority of slower replicates are subsequently found to disagree with the messages that were propagated over the network, then, although no computation need be undone, the administration system must be notified of the nodes that are suspected so that fault-treatment can be carried out.

1.1.5 Replicate consistency. Error-processing using active replicates requires that replicates residing on fault-free hosts must produce the same output messages. Although consistent ordering of output messages is not strictly essential, it does greatly simplify the mechanisms for transparently identifying "equivalent" output messages. We therefore stipulate the following constraint for all active replicate techniques:

- *Output consistency:* for each software component, all replicates executing on non-faulty hosts must produce the same output messages in the same order over the same output ports.

Note that this output consistency condition is also necessary in order to implement the passive replicate strategy using periodic checkpointing but *not* with systematic checkpointing (cf. 1.1.3.2). Sufficient conditions for output consistency are:

- *Input consistency:* each replicate must receive the same messages.
- *Processing consistency:* each replicate must process the input messages deterministically in what appears to be the same order.

In the input/output consistency conditions, the "*same* messages" necessarily imples "the same *semantics*". However, with fail-uncontrolled hosts it is necessary to be able to compare such messages and declare them as being the same. In the interests of generalization and openness, it thus further required that the "same messages" should mean "the same *syntax*". Furthermore, this can allow optimizations of the comparison mechanisms by means of message signatures.

Input consistency implies that any messages sent to a particular software component be atomically broadcasted to all replicates of that software component. The required assumption for the communication system is thus that it provides *atomic multicasting* in that messages must be sent to all or none of the fault-free nodes possessing a replicate of the destination software component.

Processing consistency is difficult to ensure in a truly heterogeneous environment. The locations of replicates must be restricted to a sub-set of nodes which, if fault-free, guarantee that processing consistency is maintained. The set of such nodes is termed a *software component replication domain.*

1.2 Administrative view of system

Open distributed systems can be extremely complex and require powerful management facilities to support planning and system integration, to assist in daily operations and to help with fault-treatment and maintenance. The set of such facilities is referred to here as *system administration.* Administration of distributed systems is currently under vivacious discussion and is still not widely well understood [24]. Two main reasons for this situation are:

- the diversity of system components that need to be managed, their different functional management requirements and interrelationships,
- the often ill-defined distinction between "management" and the normal functionality of the system.

A requirement for enhanced fault tolerance further increases the complexity of system administration. This section of the paper attempts to classify the various administrative tasks that must be performed in a fault-tolerant open distributed system.

Existing literature on "network management" identifies a variety of ways for categorizing administrative tasks, e.g.:

- by function classes [19] (configuration-, fault- and performance-management),
- by order of magnitude of real time constraints [9] (short, medium, and long term management),
- by classes of human administration users (designers, operators, maintenance staff).

A more general view is that of defining three overall administrative tasks (ATs):

AT1 : Planning and integration of redundancy and distribution.
AT2 : Monitoring of system behavior.
AT3 : Fault treatment.

A complete illustration of the ATs is beyond the scope of this paper and is in the sequel restricted to the salient points of the Delta-4 approach. The overall management model and the implementation principles (outlined later in section 2.2) are however designed to provide support for all aspects of system administration.

1.2.1 AT1: Planning and integration of redundancy and distribution. The first step to configuring a distributed application is that of identifying application-specific characteristics such as:

- topological configuration (e.g. process control interface constraints),
- services to be made dependable (replicated software components),
- degree of possible redundancy (software component replication domains),
- error-processing modes dependent on host fault assumptions, required dependability attributes, expected workload and tolerable realtime conditions,
- contingency plans for application-specific reconfiguration strategies.

The system integration phase in the real or a simulated application environment is used to validate the design decisions and to tune the systems' operating parameters. In particular, administrative information and actions are required to support the following:

- *Testing the fault tolerance mechanisms* by using online fault injection to stimulate the use of the provided redundancy. Stimuli should cover all expected operational conditions such as failures of host and communication resources, buffer overflows and general overloading.

- *Performance evaluation of hosts and the communication system* requires actions to initiate time-based measurement to obtain appropriate statistical data of a real or simulated workload under different operational (fault) conditions.

- *Dimensioning the system parameters* such as communication buffer and window sizes, priorities and schedule frequences derived from the offered statistical data is an important and difficult task. Of particular interest is the first rough layout of the timeout parameters required by the various error-processing protocols (cf. 1.1.3 and 1.1.4). This must be done on a per replication-component basis (realtime constraints, workload conditions) and is an iterative process that can be performed with human interaction or automatically by systems management. In either case, during the system integration phase, "slow" hosts are treated as being non-faulty until timeouts have been tuned to an acceptable compromise between spurious error detection and acceptable realtime responsiveness.

1.2.2 AT2: Monitoring of system behavior. By the nature of the Delta-4 fault-tolerance approach, the required human interactions to preserve the expected system behavior should be minimized. The error-processing protocols of the communication system must be supplemented by powerful system administration facilities for fault treatment.

AT2 provides the essential input for AT3 in order to carry out successful fault treatment and maintenance by gathering appropriate system information such as actual configuration data (redundancy used, current allocation of replicates, etc.), performance statistics or counters and the numbers of (recovered) errors. The monitoring activity may be carried out by periodic polling, event-triggered polling, and/or by counting events such as error reports from the error-processing protocol entities. Additional tests might be invoked to check the absence of dormant faults within unused redundant system components.

1.2.3 AT3: Fault treatment. The traditional "network management" view of AT3 is that of providing long-term management functions for system maintenance staff. Within a truely *fault-tolerant* distributed system environment this task is much more critical since it pertains to the prevention of serious consequences when faults cumulate. It should therefore be automated as far as possible. Fault treatment is achieved by supporting fault diagnosis, fault passivation, system reconfiguration and system maintenance.

Fault diagnosis is necessary to decide if a fault is permanent (e.g. judgement of the significance of timeout occurrence within the assumed and actual workload profile) and to assist in fault localization. If fault diagnosis should conclude that a permanent fault has occurred, then fault passivation must be carried out and system reconfiguration envisaged.

Fault passivation of fail-uncontrolled hosts must be done automatically by the integrated administration system, i.e. manual intervention should not be required. Note that fail-silent hosts, by definition, carry out automatic and autonomous fault passivation.

System reconfiguration can be envisaged if there are sufficient redundant resources. It entails re-allocation of the software component replicates that were resident on failed hosts in order to restore the level of redundancy required for the error-processing protocols to function correctly despite further faults. If re-allocation is not possible, then some software components may either have to be abandoned in favor of more critical ones or, at least, fault-tolerant operation is degraded to fail-safe operation in order to ensure safety and/or integrity of the distributed application(s).

Re-allocation of software component replicates is achieved by means of a *cloning* operation that creates a new replicate on a specified node. Three sub-operations can be identified:

- creation of a template of the software component at the new location; this can be done in advance of an actual cloning request in accordance with application-specific contingency plans specified by AT1,
- creation of a copy of the component's persistent data or "state" at the new location,
- activation of the new replicate whilst ensuring replicate-consistency; this involves the automatic management of the dynamic, configuration-dependent associations between replicated components.

Two techniques for cloning with different performance characteristics are being studied that are termed recursive transfer and snap-shot transfer. *Recursive transfer* necessitates the continuous identification and tagging of structures that are modified in the active replicate(s) while a state transfer is attempted. The state transfer sub-operation is then repeated using only the tagged data. This is carried out recursively until no data structures are tagged during state transfer. *Snap-shot transfer* involves the creation of a local copy (or "snap-shot") of the component's state on the host of the active replicate(s). While this snap-shot is transferred to the new location, a log of messages sent to the software component is constructed (at the new location). When the snap-shot transfer is complete, the new replicate processes the stored messages.

The basic fault treatment administrative facilities outlined above are supplemented by facilities for configuration and maintenance management. *Configuration management* facilities include remote initializing/loading of nodes, station/node passivation, version control etc. *Maintenance management* facilities are provided in order to minimize repair time. Such facilities include tools for post mortem analysis, remote access to the host's local operating system diagnostics and support for remote initialization or loading of off-line tests.

2. Implementation

The present implementation of the framework outlined in section 1 is based on the use of a local area network with specific (i.e. non-standard) protocols for communicating between replicated software components and in which network partitioning is not allowed. In particular, a high-performance atomic multicasting protocol with logical designation of multiple destinations has been implemented that allows the input consistency constraint of 1.1.5 to be respected. Furthermore, the validation of messages sent by replicated software components is based on the comparison of message signatures so as to minimize network traffic. All processing relating to communication and message comparison is carried out on specific network attachment controllers (NACs) that are self-checking and thus assumed to be fail-silent (note however, that the attached host computers may be fail-silent or fail-uncontrolled) (figure 2).

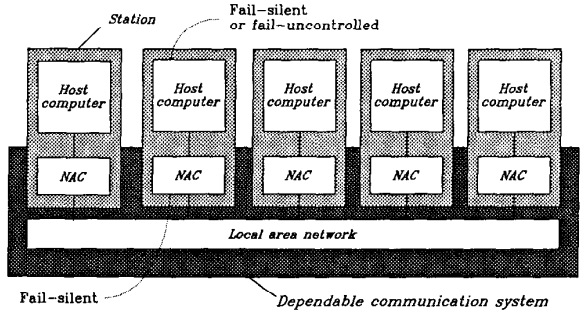

Figure 2 : Simplified Delta-4 hardware architecture

2.1 Communication system

The dependable communication system of the current Delta-4 implementation is called MCS (Multicast Communication System). MCS provides reliable multi-endpoint communication based on multi-endpoint connections [10]. Two sorts of multicasting are provided, termed *visible* and *invisible* multicasting. Visible multicasting pertains to the communication of data to one or more *user-specified* destinations. Invisible multicasting refers to *implicit* multiple destinations (or *replicated* destinations) resulting from the replication of software components for fault-tolerance. This section first overviews the layered structure of MCS and then details the atomic multicast protocol on top of which the higher-level services are built.

2.1.1 MCS layered structure. Although work has commenced at within ISO on multipoint communication [14], no standard broadcast protocols yet exist. Consequently, Delta-4 proposes a hierarchy of specific protocols that are nevertheless in conformance with the ISO/OSI reference model. Despite the specificity of these protocols, there is a total compatibility with ISO standards for the physical layer, with the services offered by the MAC sub-layer and by the application layer. In the present MCS implementation, both the presentation and network layers are null-layers.

2.1.1.1 Application layer. The application layer contains the MCS Association Control Service Element (ACSE) which is an extension of ISO ACSE to include multi-endpoint associations. The ACSE provides basic facilities for the control of an application-association between several application entities that communicate by means of bi-endpoint or multi-endpoint session-connections.

2.1.1.2 Session layer. This layer coordinates the dialogues between entities of the next upper layer by means of bi-point or multi-point session connections. The bi-point dialogue service is in conformance with the ISO 8326 service. The multipoint "dialogue" (or colloquium) generalises the half-duplex and full-duplex dialogues of the bi-point service.

The session layer manages the interactions between replicates, according to the error-processing strategies outlined in 1.1.3 and 1.1.4. Both source and destination may be replicated but they have no knowledge of the replication degree of each other. Consequently, the management of sets of replicated messages must be carried out at their source rather than at their destination.

In the active replicates model, three basic services are offered: messages may be sent by the fastest replicate (competitive), or by each replicate in turn (round-robin), or when all replicates have reached the same point of computation (inter-replicate rendez-vous). A message is propagated only once from the source to the destination(s) (except in the "propagate-before-validate" strategy if the propagated message is detected as being erroneous). Validation of messages sent by replicates residing on fail-uncontrolled hosts is carried out by means of a bit-to-bit comparison of message signatures. A message is *valid* if a majority of signatures are identical. A replicate is *erroneous* if its message signature is not identical to the agreed signature.

In the passive replicates model, only the active replicate sends messages. The session layer provides basic facilities to insert checkpoints in the communication, and to resume the communication from the last checkpoint or to start another communication.

2.1.1.3 Transport layer. The transport layer provides for a reliable data transfer over multi-endpoint transport connections. It guarantees multicast transfer of messages with or without total order, without message loss, alteration or duplication. It offers two sorts of data flows (normal and reserved) with separate flow controls. The "reserved" data flow is used for transferring the service messages generated by the session layer in order to manage replication. A simple "stop-and-go" flow control technique is used due to the complexity of using credit windows in the context of multi-endpoint connections.

2.1.1.4 Link layer. The link layer is split into three sub-layers:

a) the *Inter Link* sub-layer (IL) provides for interconnection of LAN segments, by extension of the service provided by the next sub-layer (LLC) on a single segment to an interconnection of LAN segments,

b) the *Logical Link Control* sub-layer (LLC) ensures reliable multicast transfer with total order on a single LAN segment based on the atomic multicast protocol of the MAC sub-layer,

c) the *Medium Access Control* sub-layer (MAC) provides two Service Access Points to the upper sub-layer: the first one can be used by LLC 802.2 in ISO-compatible communication systems; the second one is used by the MCS LLC sub-layer and provides the atomic multicasting service described in 2.1.2.

2.1.1.5 Physical layer. The physical layer is fully compatible with selected LAN standards. The current implementation uses the 4Mbit/s twisted pair option of the 802.5 token ring. Future implementations will include 802.4 baseband token bus and both 802.5 and FDDI fiber-optic token rings.

2.1.2 Atomic multicast protocol. The fundamental atomic multicasting service on which all the facilities of MCS are built is provided by a high-performance atomic multicast protocol (AMP) implemented in the MAC sub-layer. Multicasting at this sub-layer is a frame transmission mode in which the addressee is a logical group of MAC entities. Each logical group is associated with a communication-object called a *gate* that is identified by a unique *name*. The destination address of every MCS frame is thus a *logical address* giving the name of the gate to which frames are to be sent. Every station in which there is an instantiation of a gate belonging to the addressed group is thus a destination of such frames. More than a million different gates can be created in a Delta-4 system.

The atomic multicast protocol possesses the following set of properties (here, "correct receiver" refers to those MAC entities in the logical group addressed in the frame that reside on fault-free NACs):

P1) *Unanimity.* All correct receivers deliver the same frames to their local LLC entity.
P2) *Non-triviality.* If the sending MAC entity (entities) is (are) correct, the delivered frames are frames that were sent.
P3) *Order.* Frames are delivered to LLC in the same order by all correct receivers.

P1 and P2 define the basic Byzantine Agreement [18, 7] and P3 states that AMP achieves consistent order of delivery, which may not be the order of sending [8]. A corollary of P1 is that if at least one of the correct receivers cannot accept a frame then that frame is accepted by none (atomicity). Such cases of failure may happen whenever one or more addressed MAC entities cannot copy the incoming frame into its receive buffers due to transmission errors, lack of receive buffers... To the authors' knowledge, no previously published results on reliable broadcasting have addressed the problem of the availability of receive buffers.

Referring to distributed transaction systems, the atomic multicasting of frames is achieved using a centralized two-phase commit method [11, 23]: (a) transmission of frame F, (b) acknowledgements returned to the sending MAC entity, and (c) upon analysis of the responses, commitment by the sending MAC entity (reject or confirm). Being a centralized commit protocol, AMP is subject to failures of the coordinator, in this case, the sending MAC entity. Failures of the sending MAC entity are recovered by a "termination protocol" [23] that effectively blocks all further communication over the LAN until consistent decisions have been taken by all correct receivers.

There are two implementation versions of AMP within the MAC sub-layer. In the *first version*, existing LAN VLSI circuits are used and AMP is implemented on the controlling microprocessor. Logical address recognition is also managed on the controlling microprocessor, and acknowledgements are returned by the receivers within specific MAC frames. In the *second version*, although AMP is integrated within and around existing LAN VLSI circuits whenever possible, advantage is taken of the 802.5-type token-ring LAN topology. Logical address recognition and computation of the acknowledgements are performed "on the fly". Additionally, the commit information is implicitly forwarded to the receivers while sending a token or another frame. In case of multicast failure, an explicit "reject" MAC frame is transmitted by the sender prior to release of the token.

Version 1 of AMP has been implemented and found to work correctly in the presence of a restricted set of injected faults (NAC power-downs). A formal design verification of AMP has been initiated [12] and an extended fault-injection campaign using the MESSALINE injection tool [1] is being carried out. The latter will experimentally verify both the self-checking capabilities of the NACs and the robustness of the atomic multicast protocol and its implementation.

2.2 Administration system

The inherent complexity of the various administrative tasks outlined in section 1.2 requires the development of a practical management model, unified design principles and a stepwise implementation strategy aimed to conform and coexist with related (draft) standards.

2.2.1 Management model and design principles. The basis for the model is to treat a system as consisting of a set of (interacting) typed components. A manageable system consists of a set of manageable components (MCs). Without going into the model's details, one main aspect is to extend normal components (NCs) of some granularity relevant to administration towards MCs, i.e. to become manageable. In addition to the designed normal functionality, MC's are characterized by their:

- static and dynamic management-related attributes such as version identification, operational state and parameters, error and performance-related statistical information, designed fault-tolerance behavior, various management relationships to other MCs etc., and

- performable management operations such as create, clone, delete and reset, change state, get information and set parameters or wait for events triggered by the MC.

The attributes representing MCs are stored in the management information base (MIB) which can be distributed (for performance) and replicated (for fault-tolerance). A further major concept is a set of communicating management processes called agents to access the MIB, to exchange management information and to call MC-operations. This conceptual layout serves as a generic administration-support basis which is instantiated by an application-specific description of the designed system's behavior (AT1, cf. 1.2.1), represented only by the MCs and their attributes.

2.2.2 Status. The currently proceeding stepwise implementation of the Delta-4 administration system comprises:

- the detailed specification of the MCs and the implementation of their representation templates within the MIB,
- the implementation of the agents for some kernel functionality,
- the implementation of services and protocols used by the agents to exchange management information, and
- the implementation of a primitive administrative user interface (management application) to instantiate the MIB (system decription) and to display its dynamic behavior.

The OSI-related work concerning management is completely covered by the developed model. Standardization activities restrict their view of components to those involved in communication (communication resources). Within Delta-4 these resources are treated as MCs and their management is implemented in conformance and coexistence with expected standards.

However, the management requirements of fault-tolerant applications (*replicated_software_component*-MC) and the multicast communication system itself requires extensions beyond the current draft standards' viewpoints. Furthermore, overall system administration does not allow a complete separation between the requirements of communications management (OSI-view) and that concerning applications. Current work is aimed at clarifying these extensions and interdepencies.

3. Conclusion and future work

A prototype Delta-4 system with limited administration facilities has been built and used to demonstrate the various fault-tolerance techniques by means of a replicated database application. Further developments concern both extensions and upgrades of the MCS communication system and the implementation of a complete administration system.

Throughout this paper, we have purposely made no mention of the *programmer*'s view of the computational environment provided by the Delta-4 architecture. The set of generic features provided by the "replicated software component" concept and the associated multicast protocols and administration system could in fact be used to develop a variety of computational environments.

However, in the expected lifetime of the architecture, a standard Reference Model for Open Distributed Processing (ODP) will emerge ([13, 3]). This activity is of particular significance to the Delta-4 project since it deals with many correlated issues, which implies a need both to demonstrate the compatibility of Delta-4 dependability theory with ODP and to discover any constraints which may apply to ODP concepts in relation to the building of dependable ODP systems. With this aim, a prototype ODP support environment (called DELTASE) for the programming of ODP applications is also being designed and implemented.

References

[1] J.Arlat, Y.Crouzet, "Messaline: a fault-injection tool for dependability evaluation of fault-tolerant computing systems", LAAS report no.86356, December 1986.

[2] T.Anderson, P.A.Lee, "Fault tolerance - principles and practice", Prentice Hall, 1981.

[3] "The ANSA reference manual", Release 00.03 (draft), Advanced Networked System Architecture, Cambridge, UK, June 1987

[4] A.Borg, J.Baumbach, S.Glazer: "A message system supporting fault tolerance", Proc. 9th. ACM Symp. on Operating System Principles, October 1983, pp.90-99.

[5] J.Bartlett, J.Gray, B.Horst, "Fault tolerance in Tandem computer systems", in Dependable Computing and Fault-Tolerant Systems, vol.1, Springer-Verlag, 1987, pp.55-76.

[6] K.P.Birman, "Replication and fault-tolerance in the ISIS system", TR 85-668, Dept. of Computer Science, Cornell University, September 1985, 24p.

[7] F.Cristian, H.Aghili, R.Strong, D.Dolev, "Atomic broadcasts: from simple message diffusion to Byzantine agreement", Proc. 15th Fault-Tolerant Computing Symp. (FTCS-15), Ann Arbor, Michigan, USA, June 1985, (IEEE), pp.200-206.

[8] J.Chang, N.Maxemchuk, "Reliable Broadcast Protocols", ACM TOCS, vol.2, no.3, August 1984.

[9] "Management of Local Area Networks", Part 2 of Final Report of COST11 BIS Local Area Network Group, October 1984.

[10] C.Guérin, H.Raison, P.Martin : "Procédé de diffusion sûre de messages dans un anneau et dispositif permettant la mise en oeuvre du procédé", French Patent no. 85.002.02, January 1985.

[11] J.Gray, "Notes on Database Operating Systems", in Lecture Notes in Computer Science, Springer Verlag, 1978.

[12] S.Graf, J.Sifakis, J.Voiron: "Protocol validation methodology", Delta-4 report R7.1, IMAG/LGI, January 1988

[13] ISO/TC97/SC21 WG1 N 363, "Proposed Technical Assumptions for Open Distributed Processing - a contribution from ECMA/TC32-TG2", December 1986.

[14] ISO/TC97/SC21 WG1 N2031, "Working draft addendum to ISO 7498-1 on multipeer data transmission", June 1987.

[15] L.Lamport, "Using time instead of timeout for fault-tolerance in distributed systems", ACM TOPLAS, vol.6, no.3, April 1984, pp.254-280.

[16] J.C.Laprie, "Dependability: A unifying concept for reliable computing and fault tolerance", LAAS report no.86357, December 1986 (to appear in "Resilient Computing Systems, vol.2", Collins and Wiley).

[17] L.Lamport, P.M.Melliar-Smith, "Synchronizing clocks in the presence of faults", Journal of the ACM, vol.3 no.1, January 1985, pp.52-78.

[18] L.Lamport, R.Shostak, M.Pease, "The Byzantine Generals Problem", ACM TOPLAS, vol.4, no.3, July 1982, pp.382-401.

[19] "MAP/TOP 3.0 Network Management Requirements Specification", MAP Chapter C11, 1987.

[20] B.Randell, "System structure for software fault-tolerance", IEEE Trans. on Software Engineering, vol.SE-1, no.2, June 1975, pp.220-232.

[21] F.B.Schneider, "Byzantine generals in action: implementing fail-stop processors", ACM TOCS, vol.2, no.2, May 1984, pp.145-154.

[22] F.B. Schneider, "A paradigm for reliable clock synchronization", Proc. Advanced Seminar on Real-Time Local Area Networks, Bandol, France, April 1986, (INRIA) pp.85-104.

[23] M.D.Skeen, "Crash recovery in a distributed database system", Ph.D. Thesis, University of California at Berkeley, May 1982.

[24] Sloman, M. (ed.), "Distributed Systems Management", Imperial College Research Report DOC 87/6, 1987.

[25] J.H.Wensley, L.Lamport, J.Goldberg, M.W.Green, K.N.Levitt, P.M.Melliar-Smith, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control", Proc. IEEE, vol.66, no.10, October 1978, pp.1240-1255.

Acknowledgements: Delta-4 is partially financed by the CEC European Strategic Programme for Research on Information Technology. The work described in this paper results from the collective effort of all thirteen partners of the Delta-4 project consortium: BULL-MTS (F) (Prime Contractor), BASF (D), Ferranti CSL (GB), GMD-First (D), IEI-CNR (I), IITB-Fraunhofer (D), INESC (P), Jeumont Schneider (F), LAAS-CNRS (F), LGI-IMAG (F), MARI (GB), Teletra (I), University of Bologna (I). Particular thanks must go to Pascal Martin, Marc Chéreque, Christian Guérin and Bruno Séhabiague of BULL-MTS, Paulo Ciompi, Luca Simoncini and Lorenzo Strigini of IEI-CNR and Neil Speirs of MARI, as well as the conference reviewers, for their constructive comments on previous versions of this paper.