Distributed Fault Tolerance: Lessons from Delta-4

Because they avoid extensive redesign of specialized hardware, software-implemented approaches to fault tolerance are very resilient to change. Europe's Delta-4 project argues persuasively for implementing fault tolerance in a distributed fashion. Our Delta-4 approach achieves fault tolerance by replicating capsules—runtime representations of application objects—on distributed, LAN-interconnected nodes. It can configure capsule groups to tolerate either stopping or arbitrary failures. Its multipoint protocols serve to coordinate capsule groups and for error processing and fault treatment.

David Powell

LAAS-CNRS

Most modern computing systems are distributed systems. Organizational considerations, as when sharing data in an integrated information system, often demand such arrangements. Physical constraints, as in process control, may also argue for distribution. Sometimes, simple economics motivates the choice, as when hardware resources must be shared.

Lamport[1] once jokingly defined a distributed system as "one that stops you getting any work done when a machine you've never even heard of crashes." More seriously, but similarly, Fischer[2] claims that the main distinction between distributed systems and parallel ones is the uncertainty they introduce through unreliable communication and faulty nodes.

Lamport's witticism and Fischer's distinction both underline dependability as an inherent concern of distribution. Consequently, distribution can be a motivation for fault tolerance, because users of distributed systems want, at worst, a slight degradation in performance should an element elsewhere in the system fail.

Of course, fault tolerance is essential for applications, even nondistributed ones, that require extremely high levels of availability or reliability. Achieving fault tolerance is impossible without redundancy, which often makes distribution the appropriate paradigm for defining redundancy and managing it to achieve fault tolerance. That is, fault tolerance can be a motivation for distribution.

To define an open system architecture with distributed fault tolerance, the Delta-4 project[3] investigated this symbiotic relationship between distribution and fault tolerance. (See the Delta-4 project box.) For our purposes here, I take distributed fault tolerance to mean software-implemented fault tolerance achieved solely by message-passing between the nodes of a distributed system. The nodes constitute the units of hardware redundancy and can communicate only by means of a message-passing communication system (in Delta-4, a local area network). The architecture is open in that it is

- based on open, distributed processing concepts and, whenever possible, open system interconnection communication standards; and
- uses standard, off-the-shelf processors and standard LAN technology with a minimum of specialized hardware.

Fault and failure assumptions

These two motivations for distributed fault tolerance—distribution-motivated fault tolerance versus fault tolerance-motivated distribution—correspond not only to different viewpoints but often also to divergent application contexts. In each context the reasons for fault tolerance are

0272-1732/94/$04.00 © 1994 IEEE

quite different, thus so are the relevant assumptions about faults and failures.

Distribution-motivated fault tolerance. This viewpoint is often the realm of networked computer systems, where a collection of autonomous nodes comprise the distributed system, each node under the control of a separate user or administrator. The applications in this context are often not critical. Given users will quite likely be lenient towards crashes of their own workstations—after all, they are probably the only ones to blame—but much less so if they are frequently bothered by failures of other nodes.

Application designers could implement fault tolerance mechanisms to hide the effects of quite mundane incidents affecting a remote node. Such incidents might include a power outage, a crash of its local operating system (perhaps due to a software bug), a scheduled maintenance intervention, a network incident leading to communication disconnection, or quite simply because its user turned it off before going home! Users elsewhere might perceive all these incidents as annoying faults if they prevent them from carrying out their work (remember the Lamport "syndrome").

Consequently, we often interpret "fault" tolerance in such systems as tolerance for the inaccessibility or unavailability of other nodes, where unavailable means "as if they weren't there." Designers of distributed fault tolerance mechanisms under this viewpoint implicitly assume that nodes are fail-silent—they either carry out their intended function or, as seen from the rest of the system, they simply disappear.

The unreliability of the underlying communication network presents one final consideration for this category of system. Other applications, or even other organizations, usually share such a network, so the designer of a distributed service or application has little influence on its design. The application designer therefore may need to devise specific fault tolerance techniques to deal with network partitioning.

Fault tolerance-motivated distribution. Such systems much more strongly emphasize very high levels of dependability compared to fault-intolerant (nondistributed) systems. The applications of such systems are typically much more critical than in the previous case—a failure here could have dire economic consequences or even lead to the loss of life. Consequently, considerable attention focuses on the definition of the basic units of distribution (redundant fault-containment regions), the faults the system can tolerate, the way components fail, and the means by which they are interconnected.

Distributed fault tolerance operates on the basic premise that faults affecting processes executed by different nodes are independent, so nodes are natural candidates for defining fault containment regions. This hypothesis is, of course, quite reasonable for physical faults (or, more formally, internal accidental faults[4]). Assuming independence in the manifestations of external accidental faults (or disasters), such as fire and floods, is also quite fair, provided the nodes are sufficiently far apart. Interestingly, the premise quite reasonably applies to some accidental design faults. Differences in local execution environments of processes operating on different nodes can cause some such faults to manifest themselves in different ways. Gray[5] refers to such faults as "Heisenbugs" because they go away when you look at them.

Not only must we consider the types of faults the system is to tolerate, we must also envisage the admissible effects these faults have on node (or process) behavior—that is, node failure modes. In fault tolerance-motivated distributed systems, we aim to distribute computation over a collection of redundant nodes, possibly in the same physical box, to provide the illusion of a single virtual machine that never fails. This single virtual machine may have multiple users, but operates under the control of a single administrator. We would not expect the nodes in such a machine to randomly disappear by users turning them off or rebooting them. Since high dependability is the primary aim of distribution in this case, we also usually globally protect the system against power failures and purposely design the interconnection network itself for fault tolerance.

Delta-4 project

The Delta-4 project began in March 1986 at the initiative of two major European companies—Bull SA in France and Ferranti International in the UK—as part of the European Strategic Programme for Research in Information Technology (ESPIRIT). The aim was to "Define and design an open dependable distributed architecture (whence the project's name), that is, a multivendor fault-tolerant platform for distributed applications." The project drew on the combined expertise of eight research and academic institutions: IEI-CNR (Italy), IITB-Fraunhofer (Germany), INESC (Portugal), LAAS-CNRS (France), LGI-IMAG (France), MARI (UK), a software house: Sema Group (France), and two potential user companies: Crédit Agricole and Renault (France).

Although the overall economic situation of the computing industry since the project ended has prevented wide commercial deployment of the project's results, it was technically very successful. Several fully integrated prototypes attest to the feasibility of distributed fault tolerance techniques for heterogeneous, off-the-shelf hardware. During the project's lifetime—March 1986 to January 1992—several hardware technology updates occurred with little or no impact on the system software. This strikingly demonstrates one of the major advantages of a software-implemented approach to fault tolerance. Other advantages of this approach include a certain degree of disaster tolerance and design-fault tolerance.

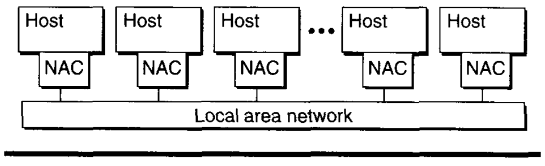


Figure 1. Hardware architecture.

Overall, we can consider the nodes in such a distributed, fault-tolerant machine as being individually much more reliable than those considered under the "computer network" viewpoint. However, the causes of node failure that we argue as being less relevant in a more tightly coupled, distributed fault-tolerant machine—shutdowns, reboots, power outages, network disconnections, operating-system crashes—are the very ones that largely justify the fail-silent node assumption in a computer network environment. Can we reasonably assume that the remaining causes of node failure result in nodes failing only by crashing?

The reasonableness of any such failure mode assumption is captured by the notion of assumption coverage,[6] defined as the probability of the assumption being true when a node fails. In this case, the coverage of a crash-only assumption equals the coverage of any self-checking mechanisms built into the nodes aimed at silencing the node should a fault occur. If the self-checking coverage is commensurate with the required application dependability, we can justifiably assume that nodes fail only by crashing. Otherwise, we must make weaker assumptions and devise distributed fault tolerance techniques that can accommodate more severe failure modes such as omission and timing failures, or even completely arbitrary failures.[6,7] As I describe next, Delta-4 follows what could be perceived as a "fence-sitting approach," whereby we can adopt both strong and weak failure mode assumptions according to the criticality of the considered application and knowledge about the available hardware.

Hardware architecture

The Delta-4 architecture is an open architecture that can employ off-the-shelf hardware to provide fault tolerance in money-critical (but not life-critical) applications. The replication of code and data on different computational nodes interconnected by a LAN provides for fault tolerance.

Since the self-checking coverage of off-the-shelf hardware often cannot justify the crash-only failure assumption, the architecture allows users to configure very critical applications to tolerate arbitrary failures. However, not all applications warrant the constraints and cost of tolerating arbitrary failures. Therefore, the Delta-4 architecture also provides simpler mechanisms when stronger failure assumptions are admissible. Users can follow both approaches, even in the same system, for different applications.

Off-the-shelf hardware is either fail-silent (crash-only) or fail-uncontrolled (admitting arbitrary modes of failure). Unfortunately, if complete nodes can fail arbitrarily, we must make the node interconnection scheme much more complex than with the single (or possibly, duplex) broadcast channel that would suffice for fail-silent nodes. For example, a fail-uncontrolled node connected to multiple channels could fail by saturating all channels, thus bringing down the complete system. Furthermore, protocols for ensuring agreement under such a failure mode assumption are notoriously complex and time consuming. The Delta-4 architecture therefore follows a hybrid approach that splits each node into two subsystems (Figure 1):

- an off-the-shelf computation component, called a host, that may be fail-uncontrolled;
- a communication component, called a network attachment controller (NAC), that we assume to be fail-silent.

The fail-silence assumption for the network attachment controllers alleviates the problems regarding the use of broadcast channels and agreement protocol complexity. A standard LAN (8802.4 or 8802.5) interconnects the NACs of each station. Duplex (or even simplex) channels can provide a very low probability of communication system failure in the maintainable environments for which Delta-4 is intended.[8] We therefore consider the communication system as hard core and make no attempt at the application level to tolerate physical network partitioning.

The NAC consists of a pair of piggy-backed cards that plugs into the host's backplane bus and interfaces the node with the physical communication channels. The NAC closely resembles any other standard LAN controller card. As described in the Implementing fail-silence box, the only difference is that it uses built-in hardware self-checking to substantiate the assumption that it is fail-silent.

Software architecture and communication system

The NACs are the only specialized hardware components in the Delta-4 open system architecture. System software implements the remainder of the Delta-4 functionality and the mechanisms for fault tolerance on top of either the hosts' local operating systems or the NACs' real-time kernels. The system software contains three main components (Figure 2, page 53):

- a host-resident infrastructure for supporting distributed applications;
- a computation and communication administration system (executing partly on the hosts and partly on the NACs); and
- a multipoint communication protocol stack (executing on the NACs).

Implementing fail-silence

Hardware self-checking mechanisms support the "fail-silence" assumption for the network attachment controllers. Figure A shows the functional diagram of a NAC for interfacing a VME-based host to an 8802.5 token ring. The figure distinguishes two functional parts: a protocol processor that executes all the high-level communication protocols (including the Inter-Replica protocol) and a LAN adapter that supports the LAN-specific layer 1 and 2 protocols.

Except for a few small devices such as a common clock and interrupt and bus synchronization circuits, the drawing completely duplicates the protocol processor section. The NAC copies all DMA-produced addresses and input data from the master system bus through a bridge to the shadow system bus. The master and shadow CPUs operate in lock-step; the controller compares their data buses on all memory accesses and most other CPU cycles, including CPU writes to the LAN adapter. Any fault causing an error on either bus forces both CPUs to halt until a reset occurs, and forces the LAN adapter to reset and disconnect from the ring. Any message that is aborted part way through its transmission will be rejected by the normal message validation checks performed at the receiving end.

The particular chip set used for this LAN adapter—the Texas Instruments TMS380—prevents self-checking of the LAN adapter part of

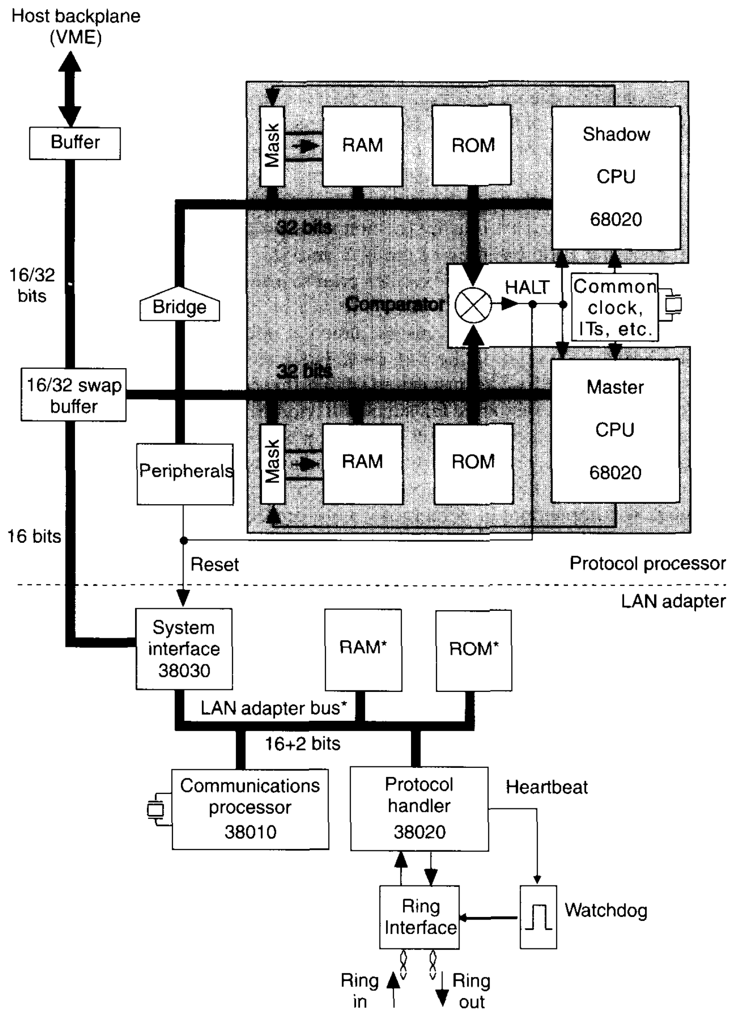


Figure A. NAC functional diagram. *Byte parity protected. *(Courtesy Ferranti Int'l, United Kingdom.)*

To support open object-oriented distributed computation, we developed a particular host-resident infrastructure 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. It provides the means for generating and supporting interactions between runtime software components called capsules (executable representations of objects). Deltase provides a runtime environment that can hide the differences in underlying local operating systems. In practice, though, all the implemented Delta-4 prototypes

Implementing fail-silence (continued)

the NAC by duplication and comparison due to the non-determinism in switching between the two clock sources of the protocol handler. Those sources are the local clock from the communications processor and the clock derived from the incoming data stream. However, this chip set does provide some built-in self-checking mechanisms: byte-wise parity-checking on all data transferred over the LAN adapter bus and a watchdog timer (a monostable timer retriggered every 10 ms by the adapter's program code) controlling the insertion of the NAC into the ring. In addition, a self-test program executes prior to inserting the NAC into the ring.

These self-checking mechanisms are aimed at protecting the NAC against faults in the NAC itself. However, we must still protect the NAC against external faults that could cause it to violate the assumed fail-silent behavior. In particular, we must ensure that a fault in the local host does not corrupt data in the NAC local memory that has already been validated by the Inter-Replica protocol. Consequently, we implemented a memory write-protection mechanism that prevents DMA write accesses to the NAC local memory except in areas to which the NAC software (executed by the self-checking CPU pair) has specifically and dynamically granted permission. The implemented protection mechanism provides two separate masks for write access from the host and from the LAN adapter. Protection comes in 128-byte granules.

This particular NAC design arose out of concern for the poor error-detection coverage achieved with an earlier prototype that did not use duplication-and-comparison in the protocol processor. We estimated coverage using physical fault injection.[1] The estimated 1-second latency error detection coverage for the early prototype with limited self-checking was only 30 percent. Even after 100 seconds' latency, the estimated error detection coverage was still only 75 percent.[2] The improved design described here led to a significant improvement as the 1-second and 100-second latency error detection coverages were an estimated 97 and 99 percent.[2]

References

1. J. Arlat et al., "Experimental Evaluation of the Fault Tolerance of an Atomic Multicast Protocol," *IEEE Trans. Reliability,* Vol. 39, No. 4, IEEE Computer Society Press, Los Alamitos, Calif., Oct. 1990, pp. 455-467.
2. J. Arlat et al., "Dependability Testing Report LA3—Fault-Injection on the Extended Self-Checking NAC," Report 91396, LAAS-CNRS, 1991 (available from the author).

are based on Unix whereby Deltase capsules map onto Unix processes.

The administration system 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. The administration system carries out three basic tasks:

- configuration management, which supports planning and integration of redundancy and distribution;
- performance management, which includes system status monitoring by event counting and polling; and
- fault management, including automatic fault treatment and support for maintenance interventions.

The notions of managed objects and domain managers form the basis for the management system. A managed object is a hardware or software component (such as a Deltase capsule) that, besides its normal functionality, is formally characterized by attributes, operations, and events by which domain managers can observe and control the component. A domain manager is a system server that is responsible for managing a set or domain of similar or related managed objects. Of particular relevance to fault tolerance is the notion of a replication domain manager for carrying out fault treatment (more on this later).

The multipoint communication protocol stack provides two major innovative features:

- multipoint associations for connection-oriented communication between *groups* of communication endpoints (for example, in different Deltase capsules); and
- the ability to coordinate communication to and from *replicated* communication endpoints (a special form of group).

The structure of the protocol stack follows the OSI reference model (see Figure 3) and incorporates compatibility with ISO protocol standards whenever possible. However, since there are not yet any standard protocols for multipoint communication, the Delta-4 stack implements a number of specific protocols (indicated by bold characters on Figure 3).

The core of the Delta-4 group communication mechanisms is the Atomic Multicast protocol (AMP),[9] which is implemented within layer 2 on top of a standard medium-access control protocol. This two-phase accept protocol allows data frames to be delivered to a group of logically designated gates. The protocol ensures unanimity—the controller delivers frames either to all addressed gates on nonfaulty nodes or to none of them. This can occur if the sender fails or if a recipient gate cannot accept a frame due to lack of receive credit. The protocol ensures that frames are delivered to all addressed gates in a consistent order. AMP makes sure as well that all members of a gate group are consistently notified of any changes in membership of that group (due to node failure or reinsertion). (Another version of AMP—called Turbo AMP—was specially targeted at the 8802.5 token ring and was implemented as a hardware and firmware extension of the existing token ring medium access control protocol.)

All communication entities execute on the real-time kernels of the fail-silent NACs. Therefore, we assume that each communication entity reacts in bounded time in the absence of NAC failure or otherwise crashes. If messages are never lost and delivered in bounded time, time-outs on responses to protocol frames can reliably detect remote crashes.

However, real networks can lose messages, so our system uses time redundancy by message retry to give the illusion of such a perfect network. AMP incorporates a frame retry mechanism that tolerates a predefined number (called omission degree) of successive omission failures. If a remote entity does not respond after the predefined number of retries, the sender assumes it has crashed. Such an assumed crash triggers the distributed election of an active monitor that ensures completion of any interrupted multicasts (to ensure unanimity) and informs all group participants of the new view of the group membership.

On top of AMP, the Inter-link protocol extends the AMP service to interconnections of LANs. The Multipoint transport protocol, a lightweight, connection-oriented protocol, adds segmentation, reassembly, and flow control functions to the underlying service. The next sublayer is the Inter-replica protocol (IRP), which is at the heart of the Delta-4 fault tolerance mechanisms. This protocol coordinates communication to and from endpoints that are replicated on different nodes such that replication is hidden from the sources and the destinations of messages sent to or by the considered endpoint. Hiding replication involves not only transparently delivering messages to all endpoint replicas (using the underlying multicasting), but also requires arbitration between send events across the set of endpoint replicas such that destinations only receive a single message. This arbitration can optionally

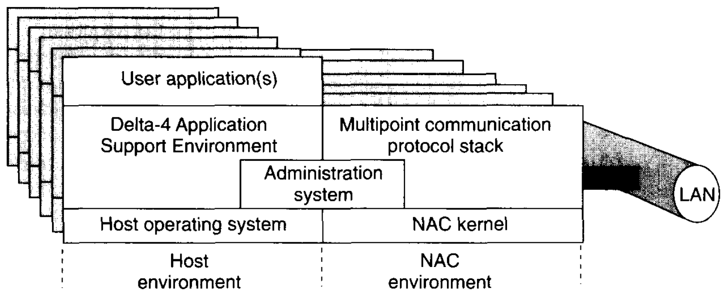
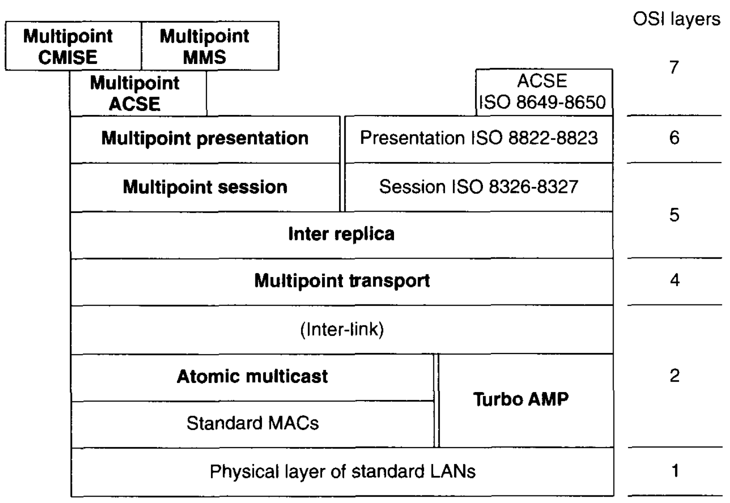


Figure 2. Software architecture.



ACSE	Association control service element
CMISE	Common management information service element
MMS	Manufacturing message specification
MAC	Medium-access control protocol

Figure 3. The multipoint communication protocol stack.

To specify and configure fault tolerance on a flexible, service-by-service basis, Delta-4 can replicate capsules independently.

include error-detection or error-detection-and-compensation by comparison of replicated messages.

The IRP services are available either to standard ISO upper-layer protocols or upper-layer multipoint protocols specific to Delta-4 that allow connection-oriented communication between groups of logically distinct communication endpoints.

Fault tolerance

Capsules, the runtime representations of Deltase objects, are the units of replication for achieving fault tolerance. To specify and configure fault tolerance on a flexible, service-by-service basis, Delta-4 can replicate capsules independently. Programmers of a Deltase object do not need to know that the capsule corresponding to the object they are programming—or indeed, of any other object with which it communicates—may be replicated.

Achieving fault tolerance by replicated processing involves two issues:

- How should we manage the interactions between a replicated capsule and other (possibly replicated) capsules so that the latter are unaware that some of the former's replicas may be faulty?
- How can we maintain (or restore) the number of capsule replicas at the level required to sustain this illusion, despite the occurrence of further failures?

These two issues relate, respectively, to error processing and fault treatment.

Error processing. To achieve fault tolerance on a capsule-by-capsule basis, error processing occurs entirely within the scope of each group of replicas without resorting, for example, to the use of transactions or conversations to coordinate error processing over logically distinct capsules. Consequently, Delta-4 must confine errors within replica groups. Any recovery or compensation action taken within a given group must not require actions by any other group.

We have investigated three different, but complementary, techniques and implemented them in the Delta-4 architecture: active, passive, and semi-active replication.

Active replication. With this technique, 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-instantaneous recovery from a node failure. Furthermore, because it can cross-check (in value and time) messages produced by different (active) replicas, it adapts to both the fail-silent and fail-uncontrolled node assumptions. However, active replication requires that all replicas be guaranteed to be deterministic in the absence of faults. If nonfaulty replicas process identical input message streams, the approach must guarantee that they produce identical output message streams. One way of achieving this guarantee is to oblige the programmer to ensure that capsules behave as state machines.[10]

Delta-4 directly manages active replication[11] by means of the IRP. The communication endpoints of actively replicated capsules are themselves replicated and configured either according to a fail-silent assumption (only late-timing errors are detected) or a fail-uncontrolled assumption (both timing and value errors are detected). If it detects any errors, the IRP causes the incriminated endpoint replica to abort and reports the errors to the administration system to initiate fault treatment.

Active replication has also been studied (under various names) in the Software-Implemented Fault Tolerance,[12] Circus,[13] Isis,[14] and Advanced Automation System[15] projects. To our knowledge, however, the Delta-4 implementation is the only one that manages replicated output messages at their logical source rather than at their destination and allows arbitrary (host) failures to be tolerated over a general (non-meshed) communication network. (To be fair, although Cooper[13] assumed that processors only failed by crashing, he did study the concept of a collator to vote on multiple replies from nondeterministic server replicas.)

Passive replication. In this technique, 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. Checkpoints systematically sent by the primary replica every time it sends a message do, however, regularly update the internal states of these standby replicas. We can envisage passive replication only if we assume that nodes are fail-silent. Unlike active replication, this technique does not require computation to be deterministic. However, the performance overheads of systematic checkpoints and rolling back for recovery may not be acceptable in certain applications—especially in real-time applications. (The use of periodic instead of systematic check pointing leads to a lower check pointing overhead but sacrifices the possibility of accommodating nondeterministic computation.)

Delta-4 manages passive replication by system code, called a *rep_entity*, included in a capsule when an application object is compiled.[16] The primary *rep_entity* atomically multicasts all data messages and associated checkpoints to the standby *rep_entities*. Both the primary and the standby *rep_entities* then forward data messages to their final destinations using the replicated endpoint mechanism of the IRP to ensure that exactly one message is sent, even if the primary fails.

Passive replication is essentially the same approach as the well-known process pair approach found in the Tandem Guardian operating system. In Delta-4, however, as in Isis[14] and AAS,[15] passively replicated groups can include more than one standby replica, so near-coincident failures can be handled.

In Delta-4, passively replicated groups can include more than one standby replica, so near-coincident failures can be handled.

Semi-active replication. In this hybrid of active and passive replication, only one replica (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 minicheckpoints) from the leader replica. Semi-active replication seeks to achieve the low recovery overheads of active replication while relaxing the constraints on computation determinism. Delta-4 can use notifications to force the followers to obey all nondeterministic decisions made by the leader replica.

Semi-active replication, or the leader-follower model as it is sometimes called, was initially developed in the context of the Delta-4 Extra Performance Architecture[17] since it has some interesting real-time properties. First, it can use notifications to inform follower replicas about the order in which the leader replica consumes messages. Therefore, messages no longer need to be sent by the totally ordered atomic multicast protocol, but can use a higher performance "reliable" multicast protocol (a protocol ensuring unanimity but not order). This model also allows high priority messages to jump the queue without introducing inconsistencies across the replica group.

Second, this model can use notifications to implement consistent preemption of computation across a replica group. For this, a precompiler must introduce preemption points that fix the points in computation at which execution may be preempted. When a follower replica reaches a preemption point, it waits for a notification from the leader replica that tells it whether to continue (to the next preemption point or to a later one) or to let a specified input message preempt computation at that point.

We can also use the semi-active replication technique for replicating large, off-the-shelf software components about which no assumption can be made concerning replica determinism and internal states (such as commercially available database management software). However, this technique cannot be a generic mechanism. The model must either identify potential sources of nondeterminism in the source code (if this is available). Otherwise, it must provide a purposely designed front-end to the replicas that maintains consistency by means of semantic-dependent leader-follower notifications. We used this technique to replicate an Oracle database in one of our prototypes.

At first sight, the semi-active replication technique requires nodes to be fail-silent because the notifications from a faulty leader could otherwise propagate errors to the followers. However, we could relax this restriction if each notification indicates the leader's choice among a predefined finite set of decisions. If the notification received from a faulty leader indicates a decision inside this finite set, although the leader may not have taken that decision, the followers can still follow it since it is a valid one. If the received notification is not in the finite set, all the followers will know and take a default decision.

In both cases, the IRP can detect and compensate for errors in the output messages from the leader and follower replicas. Even if we adopt the more restrictive fail-silent assumption (as we did for the implementations of this technique that we actually carried out), the IRP allows arbitration between messages sent by the leader and follower replicas to ensure that exactly one message is sent, even if the leader fails.

Fault treatment. The error processing techniques described just now allow capsule interactions to proceed even though a subset of replicas of each capsule may be faulty. However, if nothing more is done, subsequent faults could cause most or all of the replicas in a group to fail, thereby leading to system failure. Fault treatment, one of the roles of the administration system, avoids such exhaustion of redundancy.

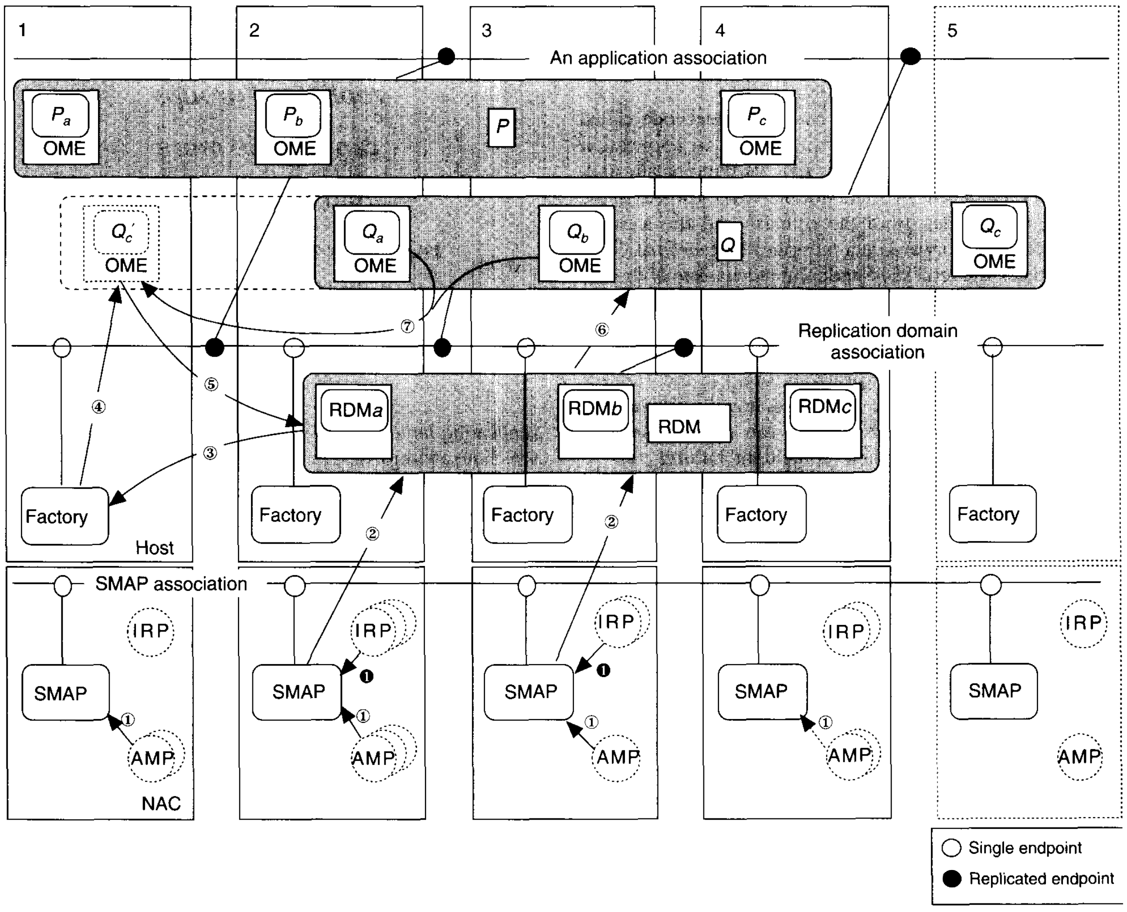


Figure 4. Error reporting and fault treatment.

For an explanation of fault treatment in Delta-4, let's look at an example. Figure 4 illustrates a simple Delta-4 system with five nodes, each split into its constituent host and NAC components. The figure shows two triplicated application capsules, *P* and *Q*, whose replicas have been initially allocated to node sets {1, 2, 4} and {2, 3, 5}. We will define the other entities as we look at the fault treatment actions that occur following the activation of a fault in node 5. Here, I assume that *P* and *Q* use active replication (they both have replicated endpoints on any application associations to which they are connected). The fault treatment actions for the other replication paradigms are similar.

Fault treatment involves fault diagnosis, fault passivation, and system reconfiguration. Fault diagnosis localizes the fault—the source of detected errors—and decides whether fault passivation is necessary to prevent the fault from causing further errors. We call faults judged not to require passivation soft faults; fault that do we call solid faults.[4]

By the very notion of fail-silence, a fault that results in node crash is automatically passivated. Because the fault changes group membership, Delta-4 makes its location known to all participants of any active AMP group that spans the failed node. Any such AMP group change event gets reported (①) to a local system administration component called a system management application process (SMAP). There is one SMAP executing on every NAC in the system. A multipoint association linking all the SMAPs consistently informs all other SMAPs of the group change.

Delta-4 can tolerate faults that produce a failure other than a node crash—faults that cause the host to violate the fail-silence assumption—only it employs active replication. In this case, replicated endpoint entities executing the IRP detect discrepancies in the values or timing of replica output messages. Any detected errors cause the incriminated endpoint replica to be aborted and are reported locally (❶) to the SMAPs of the stations on which the replicas reside. Delta-4 considers any faults revealed by IRP error detection to be solid faults. At the first such error, Delta-4 assumes the node to be faulty and passivates it by removing it from the system exactly as if it had crashed. However, we could easily extend the fault treatment strategy to consider soft faults.

After fault diagnosis and fault passivation, we can attempt system reconfiguration. This procedure entails the allocation and initialization of new replicas to replace failed ones and thereby restore the level of redundancy so that further faults can be tolerated. If new replicas cannot be allocated, Delta-4 may have to abandon some applications in favor of more critical ones. Otherwise, system reconfiguration will be delayed until failed nodes are repaired and re-inserted.

We refer to the allocation and initialization of new replicas as cloning, which proceeds according to the reconfiguration policy defined for a given set of nodes, called a replication domain. A Delta-4 system may have several, potentially overlapping, replication domains. An administration system component called a replication domain manager (RDM) applies the reconfiguration policy for a particular domain. Naturally, RDMs are very critical components, so they should be made fault-tolerant. The system handles fault tolerance of RDMs in exactly the same way as fault tolerance of application components. The five nodes in Figure 4 make up a single replication domain, so there is only one (triplicated) RDM.

In Figure 4, one or several SMAPs receive notification that node 5 has failed. These SMAPs then notify (②) the RDM of this failure event so the latter can orchestrate the reconfiguration actions defined by its reconfiguration policy. In this case, we will assume that a new replica of Q must be cloned on node 1 to replace replica Q_c on the failed node 5.

The cloning protocol involves two other system entities shown in Figure 4. The first of these is the object management entity (OME) linked to each capsule at compile time. An OME extends the user-defined functionality of a capsule to include management services for initializing and cloning replicas of the capsule. The second entity is a separate administration component, called a factory. A logically distinct factory is installed on every node in the system and takes responsibility for local instantiation of capsule replicas. The OMEs of each replicated capsule and the RDM have replicated endpoints on a domain-wide multipoint association (called the replication domain association in Figure 4). Each factory also connects to this association through a single (non-replicated) endpoint.

Active replication enables systems seeking high dependability with off-the-shelf hardware interconnected by standard LANs to tolerate arbitrary failures.

The cloning protocol proceeds as follows:

- The RDM requests (③) the factory on node 1 to create (④) a local template of Q from a local copy of the program file.
- During its initialization phase, the OME of this fledgling replica (Q_c') establishes the communication endpoints necessary for communication with the RDM and confirms (⑤) this to the RDM.
- The RDM instructs (⑥) the OMEs of Q_a and Q_b to transfer (clone) their process context to the new replica.
- The OMEs of Q_a and Q_b take a snapshot of their local context (data, stack, registers, ...) and initiate the transfer (⑦) of this context to the OME of Q_c'.
- When all the context data have been transferred, the OME of Q_c' substitutes the current process context with the received context data and continues execution with that context as a full-fledged replica (after synchronizing with the OMEs of Q_a and Q_b).

The actual transfer of context data falls subject to the same IRP error detection (message comparison) as any other communication from replicated components. Consequently, the system can confine near-coincident faults activated during cloning (or even tolerate them if it employs sufficient initial replication). However, any location-specific context data must be identified and equalized before transfer to avoid the natural differences in such data being perceived as errors.

With the fault treatment facilities outlined here, the level of redundancy of replicated capsules can be restored automatically, so long as the system has sufficient nonfailed resources. We can test off line any nodes that have failed and been disconnected from the system, repairing them if necessary and reinserting them into the system. However, the system views such reinserted nodes as entirely new nodes with total amnesia regarding their previous existence. When a node is (re-)inserted into the system, the replication policy can allow for a redistribution of replicas to balance the overall load.

AS THIS PROJECT SUCCESSFULLY DEMONSTRATED, active replication enables systems seeking high dependability with off-the-shelf hardware interconnected by standard LANs to tolerate arbitrary failures. Such tolerance does not come without a price; application programmers must respect certain rules to achieve deterministic, fault-free execution. What, however, is the alternative if we want off-the-shelf hardware to implement highly dependable systems? If the off-the-shelf hardware in question should only fail in a less severe fashion—for example, by crashing—we could adopt the alternative passive or semi-active replication techniques. However, we should justify any such assumption by an estimation of the accompanying coverage and a demonstration that this coverage is commensurate with the overall dependability objective.

Estimation of the actual coverage requires that we gather failure-mode statistics, either in operational life or by fault injection in prototypes. Note, however, that any upgrade in hardware technology will usually require a completely new estimation of the corresponding coverage. Also, note that the error detection coverage we can achieve with traditional, off-the-shelf computer hardware is often quite low unless we also employ extensive self-checking techniques. (For example, consider the coverage figures given earlier in the Implementing fail-silence box for the early prototype NAC, which could be considered as such traditional computer hardware with limited self-checking.)

When looking at active replication, we must also consider the interface between the actively replicated capsules and the outside world. Whereas active replication is eminently suitable for managing voting on multiple sensors (with multiple n-way multicasts to ensure interactive consistency), or for driving voting actuators, it works much less well when replication must be entirely hidden from the outside world. For example, consider setting up a connection over a public network to a non-Delta-4 site. In such cases, the necessarily unique Delta-4 site that manages the external connection becomes a hard core. Although we can implement application-specific procedures to tolerate crashes of such an I/O site, any failure of a more arbitrary nature could lead to failure of the complete Delta-4 system. Management of such a single-sited I/O forces adoption of the fail-silence assumption for the I/O site(s). The attendant lack of coverage will largely determine overall dependability, even if we make less severe assumptions for the other sites in the system.

The key to tolerating arbitrary failures without resorting to costly interconnection hardware or agreement protocols is the concept of a fail-silent NAC associated with each off-the-shelf host computer. The presence of low-level LAN-controller chips, which could not be duplexed and compared due to inaccessible, and therefore nonsynchronizable, internal clock circuitry, limited the error detection coverage of the NACs. Higher self-checking coverage would therefore have forced us to redesign some quite complex VLSI chips.

The Delta-4 approach to distributed fault tolerance relies heavily on the use of multipoint communication protocols. Such protocols are extremely useful for simplifying distributed computation in general, regardless of fault tolerance. For example, the availability of multipoint associations that could span all nodes, or all nodes in a particular replication domain (see Figure 4), significantly simplified the Delta-4 administration system.

We implemented the core group communication protocol, AMP, at the lowest possible level, at layer 2 of the OSI reference model. Consequently, node- or NAC-crash detection was reliable and usually quite fast. Sometimes, however, a crash can remain undetected by the rest of the system if none of the AMP groups that span the crashed node are active—that is, attempting to exchange messages. Imposing artificial, minimum frequency AMP traffic spanning all nodes could easily improve this situation. We could throttle back this artificial traffic to save bandwidth if real AMP traffic were sufficiently high to ensure low crash-detection latency.

The low-layer implementation of AMP proved to somewhat impede the distribution and uptake of the Delta-4 group communication facilities because it made it difficult to port Delta-4 to an existing networked environment. In contrast, the Isis[14] team first chose to implement multicasting on top of the widely available user datagram protocol (UDP), which enabled widespread distribution of the Isis tool kit in the research community and beyond. Note, however, that the Isis team is now considering putting its core mechanisms much closer to the network interface.[18]

Distributed fault tolerance techniques such as those developed in Delta-4 have a difficult time competing with the performance of hardware-intensive, tightly synchronized approaches. Still, I firmly believe that the economic and flexibility advantages of software-implemented, distributed fault tolerance techniques will soon emerge as a major option for highly dependable systems, especially in distributed workstation environments. In particular, the advent of microkernel technology opens new and exciting possibilities for high-performance, user-transparent, software-implemented fault tolerance.

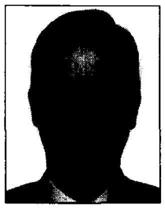
In the Dependable Computing and Fault Tolerance research group at LAAS-CNRS, we are now tackling a new challenge: how to design distributed systems that are not only highly available but also suitable for safety- and security-critical applications. We believe that distributed fault tolerance techniques like those developed in Delta-4—combined with modern microkernel technology—provide us with some of the answers.

Acknowledgments

The Commission of the European Community partially supported the Delta-4 project through the ESPIRIT program (Projects 818 and 2252). Many persons contributed to the Delta-4 project over its six-year lifetime. They are far too numerous to thank individually, but I would like to express my gratitude to them all for their professionalism, their enthusiasm, and the splendid *esprit de corps* that reigned within the project team. Special and very personal thanks must of course go to David, David, Doug, Gottfried, Marc, Pascal, Paulo, Peter, and Santosh, as well as all my compatriots in the Dependable Computing Group at LAAS.

References

1. S. Mullender, ed., *Distributed Systems*, ACM Press, Addison-Wesley, New York, 1989.
2. M. Fischer, "A Theoretician's View of Fault-Tolerant Distributed Computing," in *Fault-Tolerant Distributed Computing*, B. Simons and A. Spector, eds., Springer-Verlag, Berlin, 1990, pp. 1-9.
3. D. Powell, ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Springer-Verlag, Berlin, 1991.
4. J.-C. Laprie, ed., *Dependability: Basic Concepts and Terminology*, Springer-Verlag, Vienna, 1992.
5. J. Gray, "Why Do Computers Stop and What Can Be Done About It?" *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, IEEE Computer Society Press, Los Alamitos, Calif., 1986, pp. 3-12.
6. D. Powell, "Failure Mode Assumptions and Assumption Coverage," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing,* CS Press, 1992, pp. 386-395.
7. F. Cristian et al., "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," *Proc. 15th Int'l Symp. Fault-Tolerant Computing,* CS Press, 1985, pp. 200-206.
8. K. Kanoun and D. Powell, "Dependability Evaluation of Bus and Ring Communication Topologies for the Delta-4 Distributed Fault-Tolerant Architecture," *Proc. 10th Symp. Reliable Distributed Systems,* CS Press, 1991, pp. 130-141.
9. P. Veríssimo, L. Rodrigues, and M. Baptista, "AMP: A Highly Parallel Atomic Multicast Protocol," *ACM Computer Comm. Rev.*, Vol. 19, No. 4, Sept. 1989, pp. 83-93.
10. F. B. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* Vol. 22, No. 4, Dec. 1990, pp. 229-319.
11. M. Chérèque et al., "Active Replication in Delta-4." *Proc. 22nd Int'l Conf. Fault-Tolerant Computing Systems,* CS Press, 1992, pp. 28-37.
12. P.M. Melliar-Smith and R.L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault Tolerance Flight Control System," *IEEE Trans. Computers,* Vol. C-31, No. 7, July 1982, pp. 616-630.
13. E.C. Cooper, "Replicated Procedure Call," *ACM OS Rev.,* Vol. 20, No. 1, 1984, pp. 44-56.
14. K.P. Birman and T.A. Joseph, "Exploiting Replication in Distributed Systems," in *Distributed Systems*, S. Mullender, ed., ACM Press, New York, 1989, pp. 319-367.
15. F. Cristian, B. Dancey, and J. Dehn, "Fault Tolerance in the Advanced Automation System," *Proc. 20th Int'l Symp. Fault-Tolerant Computing,* CS Press, 1990, pp. 6-17.
16. N.A. Speirs and P.A. Barrett, "Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing," *Proc. 19th Int'l Symp. Fault-Tolerant Computing Systems,* CS Press, 1989, pp. 184-190.
17. P. A. Barrett et al., "The Delta-4 Extra Performance Architecture (XPA)," *Proc. 20th Int'l Symp. Fault-Tolerant Computing Systems,* CS Press, 1990, pp. 481-488.
18. R. van Renesse et al., "Reliable Multicast Between Microkernels," *Proc. Workshop on Microkernels and Other Kernel Architectures,* Usenix Assoc., 1992, pp. 269-283.



David Powell, scientific director of the Delta-4 project, is a member of the Dependable Computing and Fault Tolerance Research Group at LAAS-CNRS, Toulouse, France. His current research interests include distributed algorithms for software-implemented fault tolerance, stochastic Petri-net modeling for dependability evaluation, and the use of fault injection for estimating fault tolerance coverage.

Powell received a BSc from Southampton University, England, before joining LAAS-CNRS. He obtained his Speciality and State Doctorates, respectively, from Paul Sabatier University, Toulouse, and the National Polytechnic Institute, Toulouse. He is a member of the Association Française des Sciences et Technologies de L'information et des Systémes (AFCET) and IEEE.

Direct any questions concerning this article to the author at LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse, France; david.powell@laas.fr.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card

Low 165 Medium 166 High 167