Towards an integrated approach to fault tolerance in Delta-4

P.A. Barrett and N.A. Speirs

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. The Delta-4 approach to fault-tolerance is based upon the replication of software components on distinct host computers. It deals primarily with hardware fault tolerance (i.e. the tolerance of the system to hardware failures), but also address other areas of dependability, including software fault tolerance.

This paper describes the architecture of the Delta-4 project, considers methods of achieving tolerance to design failures, and how these methods fit into the overall Delta-4 architecture.
Bibliographical details

BARRETT, Peter Adrian
Towards an integrated approach to fault tolerance in Delta-4
[By] P.A. Barrett and N. A. Speirs
Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1991.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 344)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 344
SPEIRS, Neil Alexander

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. The Delta-4 approach to fault-tolerance is based upon the replication of software components on distinct host computers. It deals primarily with hardware fault tolerance (i.e. the tolerance of the system to hardware failures), but also address other areas of dependability, including software fault tolerance.

This paper describes the architecture of the Delta-4 project, considers methods of achieving tolerance to design failures, and how these methods fit into the overall Delta-4 architecture.

About the author

Mr. P.A. Barrett has been a principal research associate in the Computing Laboratory since August 1991.

Dr. N.A. Speirs has been a lecturer in the Computing Laboratory since December 1987.

Suggested keywords

DISTRIBUTED SYSTEMS
FAULT TOLERANCE
RELIABILITY
REPLICATED PROCESSING

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 621.381968
U.D.C. 519.687 681.322.02
TOWARDS AN INTEGRATED APPROACH TO
FAULT TOLERANCE IN DELTA-4

by

P.A. Barrett and N.A. Speirs

Computing Laboratory, University of Newcastle Upon Tyne, Newcastle upon Tyne, England.

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. The Delta-4 approach to fault-tolerance is based upon the replication of software components on distinct host computers. It deals primarily with hardware fault tolerance (i.e. the tolerance of the system to hardware failures), but also address other areas of dependability, including software fault tolerance.

This paper describes the architecture of the Delta-4 project, considers methods of achieving tolerance to design failures, and how these methods fit into the overall Delta-4 architecture.

Keywords: Reliability, Fault-tolerance, Distributed systems, Replicated processing.
1. Introduction

Delta-4 is a collaborative project carried out within the framework of the European Strategic Programme for Research in Information Technology (ESPRIT). Its aim is the Definition and Design of an Open, Dependable, Distributed computer system architecture (hence the project’s name).

Dependability is defined as being a quality of delivered service such that reliance can justifiably be placed on this service. It embraces the attributes of, for example, reliability, availability, maintainability, safety, integrity and security, each of which is seen as a different perception of the same attribute, which can be addressed by the same underlying support mechanisms. The Delta-4 architecture allows the user to obtain specifiable levels of dependability on a service-by-service basis without assuming any special attributes of participating hosts.

The range of application areas addressed by Delta-4 include Computer Integrated Manufacture, Office Automation, Process Control etc. These are demonstrated by instances of the architecture in a Unix environment, running applications at Credit Agricole (a credit card authorisation system) and at Renault (a process control application within an integrated manufacturing cell). The project also has a demonstration facility at LAAS in Toulouse.

Openness, in the context of Delta-4, has a number of implications. Firstly, the Delta-4 architecture must be capable of co-existing with, and inter-working under, standards conforming to the Open Systems Interconnection (OSI) reference model. Delta-4 will accommodate existing proprietary computer systems, and the connection of heterogeneous equipment via one or more inter-connected local area networks. Secondly, the Delta-4 Application Support Environment (Deltase) will conform to the emerging standard for a Support Environment for Open Distributed Processing proposed by the European Computer Manufacturers Association (ECMA) working group TC32/TG2. Finally, the results of the project are published [1, 2, 3, 4].

For certain critical, real-time applications, the openness and generality which is otherwise one of the major attributes of the project, becomes a constraint. The Delta-4 Multi-point Communication System (MCS) implements a wide variety of services which superset those of OSI, and successfully offer the functionality needed for a broad spectrum of Delta-4 applications. Precisely because of its generality and openness, however, MCS cannot offer assurances of timeliness required or the performance demanded in the real-time sector. Thus the Delta-4 project has developed an Extra Performance Architecture (XPA) alongside its Open Systems Architecture (OSA) [4].

The Delta-4 XPA architecture introduces mechanisms which support explicitly the requirements of real-time systems with respect to both throughput and response. The architecture
supports the real-time concepts of priorities and deadlines and reflects these concepts in its communication protocols. The XPA architecture inherits as much of the OSA architecture (both concepts and implementation) as is possible within the constraints of its performance requirements.

In this paper we present an overview of the open systems architecture in section 2. Section 3 describes the Hardware Fault Tolerance mechanisms implemented by the Delta-4 project. In section 4 we describe how Software Fault Tolerance techniques may be added to the architecture in order to tolerate design failures and to provide an overall integrated approach to fault tolerance. Finally, section 5 describes the conclusions drawn from the work.

2. An Overview of the Delta-4 Open Systems Architecture

A Delta-4 system consists of a number of computers (possibly heterogeneous), interconnected by a Dependable Communication System. Application programs are structured as software components distributed among the hosts of the system as shown in Figure 1. Each software component may be replicated, however, copies of software components must be located on homogeneous (functionally identical) machines. Each node consists of a host computer together
with a Network Attachment Controller (NAC). The NAC's are specialised communications processors which together implement a dependable communication system allowing multi-point communication between replicated computational entities. NAC's are assumed to be fail-silent; i.e. they will fail in such a manner that they will become silent and will never send out erroneous messages to nodes functioning correctly [2, 5]. A fail-silent component exhibits the halt-on-failure property of fail-stop processors [6]. The fail-silence assumption for the network attachment controller is substantiated by the use of built in hardware self checking techniques.

The Local EXecutives (LEXs) of the hosts may be heterogeneous. Each NAC has its own local executive in the form of a Real-Time Monitor (RTM), and these, too, may be heterogeneous. The distributed Delta-4 software running on each node may be classified as follows:

1) The communications system software which executes on the NAC's. The communication system of the present Delta-4 open systems architecture is called the Multicast Communication System (MCS). MCS provides reliable multi-end point communication based on multi-endpoint connections using a low level Atomic Multicast protocol (AMp) [1, 7].

2) Administration software which provides management of both computational and communications elements of the system, and which executes partly on the host computers and partly on the NACs.

3) The Delta-4 Application Support Environment (Deltase) provides a framework for the construction of object-oriented, dependable, distributed applications.

4) The user applications software itself. This will consist of a number of software components (objects), potentially written in different languages, and communicating via Remote Service Requests (RSR's), an RPC-like mechanism for inter-object communication.

3. Hardware Fault Tolerance in Delta-4

In the Delta-4 system, a number of techniques are available for providing fault tolerance. Each involves the use of replicated software components.

One technique involves the parallel operation of a number of identical copies (at least 3) of a software component, with the outputs from all components being compared, and the majority decision being used (voting) [8]. This is the active replicates model. Comparison of 'signatures' (a form of checksum) is used to validate messages. If two components produce an identical signature, MCS selects one copy of the message to forward to its (possibly replicated) destination. The active replicates model detects that a node is faulty when a message originating on that node fails to agree with the majority of its replicates, is produced late, not at all, or is produced in error; the latter three cases being determined through timeouts that monitor the relative skew between
replica output messages. This allows the system to tolerate fail-uncontrolled hosts, i.e. hosts which can fail in an arbitrary way.

A second technique used within Delta-4 is passive replication [3]. In this model, each software component is replicated but only a single copy is active. This component periodically copies (checkpoints) its state to the others (its backups). If the computer hosting an active software component fails, a backup is awoken and begins to execute from its most recent checkpoint. In order to detect failure, the assumption is made that all processors are fail-silent.

If software components run on fail-silent nodes, active replication may be used without voting, since any message generated by a process may be assumed to be correct. As a result, the minimum replication degree is reduced to 2, the communications mechanism required are simplified, and better performance is achieved since results may be propagated immediately they are generated, rather than being held pending the voting process.

Each of these replication strategies has benefits and drawbacks. A benefit of active replication is that there is no interruption in service when a failure occurs. Further, if fail-silent hosts are used, the requirement to validate outputs by voting may be avoided, the outputs of the fastest replicate being propagated to the network. On the debit side, it is essential to support active replication with some form of atomic multicasting; the protocols to achieve this are necessarily complex. Further, active replicate objects must behave identically with respect to messages consumed and produced. The necessary properties are conferred by the use of an appropriate applications model, i.e. by implementing deterministic objects using the State Machine model [9], or by implementing replicate-deterministic objects using some form of agreement on execution path) [10]. If active replicate objects are required to respond quickly to external events through some form of pre-emption, the difficulty is compounded. Pre-emption is complex and costly to synchronise between active replicates, since each replicate must be pre-empted at exactly the same point in its processing. In practice this is likely to lead to unacceptably large maximum pre-emption times.

The passive replication model suffers from no such problem with pre-emption; since only one replicate is active at any time and all external consequences of its activity are accompanied by capturing a checkpoint, pre-emption may occur at any time. Further, passive replication requires relatively simple communications support and does not require objects to be deterministic. This is a major advantage for many applications. Since only one replicate is active, processing requirements are minimised; checkpoint capture will generally use less resource than replicated execution. Passive replication does, however, require the use of fail-silent hosts, and when an active replicate fails there is a delay in the provision of service while recovery and re-execution is carried out. Such a delay may not be compatible with the achievement of demanding real-time deadlines which must be met in spite of failure. In addition, there is a systematic overhead
incurred during fault free operation of the system due to the communication activities required for checkpointing.

A variant of active replication provides a further technique for error processing. This is the leader/follower model of replication where all copies of an object are active in that they all execute the same code. One copy is designated the leader and is responsible for taking all decisions which affect replicate determinism; such decisions are propagated from leader to follower via synchronisation messages. System nodes are assumed to be fail-silent, thus output message validation is not required; messages may be sent by the leading copy of an object immediately they are generated and when the followers generate the same messages, they are discarded automatically by the communications system. This model is similar to the passive replicates model but uses additional processing to update the states of backups (or followers) rather than checkpoints.

Two forms of synchronisation message are used in the leader/follower model; input synchronisation messages and pre-emption synchronisation messages. It is necessary that notions of the precedence of one global computation over another may be propagated from a client, via the communications system, to its servers. Thus, objects must be able to consume input messages in an order which respects their instantaneous precedence. However, all copies of an object must consume the same messages in exactly the same order, otherwise their paths may diverge and replicate determinism will be lost. Therefore, when the leader selects a message (according to a precedence rule applied at some instant to its local set of messages), it also constructs a synchronisation message containing the identity of that input message and sends this to its followers. The followers consume messages in the order dictated by their leader, and replicate determinism is preserved. In practice this mechanism is embedded in the communications protocols and made totally transparent to the applications programmer.

Certain objects must be constructed to be pre-emptable very quickly should certain events, such as alarm conditions, occur. As with active replication, pre-emption can result in replicate non-determinism unless every copy of an object is pre-empted at exactly the same point in its processing. The leader/follower model incorporates a pre-emption synchronisation mechanism which imposes a very small overhead to ensure that replicate determinism is preserved. This mechanism makes use of the concept of a pre-emption point; this is a predefined point in its processing at which an object may be pre-empted. Pre-emption synchronisation is achieved as follows:

Each time the leader reaches a pre-emption point, a counter is incremented. (Note that there is one counter per object replicate, not one per pre-emption point). When a message arrives at the leader, a check is made to determine whether this message requires the leader to be pre-empted. If so, the pre-emption point at which this will take place is selected (the current counter value plus 1
represents the next pre-emption point) and assigned, and a synchronisation message containing this value and identifying the message is constructed and dispatched to the followers. On arriving at the assigned pre-emption point (i.e. when their counters match the assigned value), each replicate begins to process the pre-emption. Since pre-emption point code must be executed more often than the maximum allowable pre-emption delay, it is essential that the normal, non-pre-empting path through this code be efficient.

In order for this mechanism to work, the followers must always execute at least one step behind the leader, where a step constitutes the receipt of a synchronisation message due either to a pre-emption, or to the consumption of an input message by the leader. To avoid followers falling too far behind their leader (a figure which is determined by the time permitted for recovery following leader failure), dummy synchronisation messages, which also double as 'I am alive' messages, may be sent periodically by the leader.

4. Extensions for Tolerance of Software Faults

Delta-4 is not intended for use in safety-critical applications, therefore the consequences of failure would normally be measured in terms of financial loss rather than loss of life and limb. However, such losses may still be considerable, and for some applications the use of software fault tolerance mechanisms will be considered worthwhile. Examples of systems to which software fault tolerance may be applied include industrial planning and resource management (where errors may cause inefficiency in the allocation of resources, disruptions to the production process etc.), process control (where errors may cause damage to the process being undertaken), office automation, and banking, electronic funds transfer, etc (where the risks are obvious and the potential losses huge, both financially and in terms of credibility, which ultimately means financially as customers go elsewhere).

To meet the requirements of such a range of applications, a range of software fault tolerance techniques of varying cost and utility is required. Also, of course, the techniques used must be compatible with the Delta-4 hardware fault tolerance techniques described above.

4.1. Recovery Blocks

A recovery block [11] is a module in which a number of alternates, independently developed to the same specification, are combined with an acceptance test for error detection to form a software fault tolerant module. When a recovery block is executed, the first alternate is run, then the acceptance test. If the acceptance test indicates that the alternate completed successfully, the block exits. If, however, the acceptance test indicates that, following the execution of the alternate, there is an error in the state of the process, then the state is restored to that which existed when the recovery block was entered, and a different alternate is tried. The alternate / acceptance test
process continues until either the acceptance test is passed successfully, or there are no more alternates left to try, in which case the recovery block as a whole has failed. Recovery blocks may be nested one within another, so the failure of one recovery block may trigger recovery within an enclosing block, or separate exception handling mechanisms may be provided to deal with the failure in another way.

Dialogues [12] are structures which may enclose a number of processes and data structures in such a way as to allow communication between them to take place without compromising the ability to recover from errors. Communication occurring within a recovery block will be invalidated should recovery be required within that block; dialogues provide a means of propagating recovery in such a way that, not only the failed block, but also any entity with which it has communicated, may be recovered together, thus returning the system to a consistent state.

Recovery blocks and dialogues map onto the Delta-4 architecture very easily with Delta-4 objects corresponding to processes. The checkpointing mechanism which is part of the Delta-4 passive replicates model may (with a slight modification) be used to provide the backward recovery needed when a recovery block alternate fails. Similarly, dialogues may be implemented via an extension to the Delta-4 transaction mechanism.

Delta-4 objects are, of course, replicated for the purpose of hardware fault tolerance, and recovery blocks, on the whole, fit in well with this replication. With Delta-4’s active replication technique there is little scope for optimisation; replicated outputs are required for voting, and thus all replicates must run the recovery blocks in parallel. The replicate determinism property will ensure that all replicates finally exit the recovery block having succeeded in executing the same alternate.

With passive replication, the backups do not execute the code of the application, therefore they do not themselves execute recovery blocks. They do, however, need the recovery points taken on entry to each recovery block to be available to them in case a hardware failure should occur whilst the primary is executing a recovery block, and a software error should then cause the recovery block to need to be recovered. Thus, on entry to a recovery block, the primary must send a copy of its recovery information to its backups. On leaving the recovery block, a message is sent instructing the backups that the recovery point may be discarded as shown in Figure 2. Note that recovery points and the checkpoints which form part of the passive replication mechanism are independent of one another. An incoming checkpoint automatically supersedes the previously stored checkpoint; recovery points, however, must be retained until explicitly deleted in response to a message from the primary.

Using recovery blocks in conjunction with the leader/follower model of replication is appealing in that it can be optimised in such a way that the followers never need to worry about failing with a
software error, and thus don't need to maintain recovery points, or execute multiple alternates. On arrival at a recovery block, the followers suspend awaiting instructions from the leader. The leader runs through the recovery block and, on completion, sends a message to the followers instructing them which alternate to execute. The followers simply execute that alternate and exit the recovery block. The replicate determinism requirements of the leader/follower model will ensure, in the absence of hardware faults, that the alternate succeeds in the follower as it did in the leader.

The scheme is complicated by the need to allow leaders to send synchronisation messages to followers during the execution of recovery block alternates (synchronisation messages are used at points of potential non-determinism, such as consumption of a message, to ensure that replicate states do not diverge). In the example shown in Figure 3, the leader sends three synchronisation messages to its followers during execution of the first alternate of a recovery block. The acceptance test then fails, and the state of the object is recovered. This recovery invalidates the three synchronisation messages which have been sent (but not consumed since the followers are currently suspended). Therefore, when the recovery takes place, the followers must be instructed to discard these messages. Alternate 2 is then executed, and synchronisation messages 4-7 are sent. Finally, the acceptance test succeeds, and synchronisation message 8 instructs the followers to continue, executing alternate 2, and acting according to synchronisation messages 4-7.
4.2. N-Version Programming

The second major technique available for software fault tolerance is N-version programming [13, 14]. As with recovery blocks, a number of alternate modules (called versions) are constructed to the same specification. When the N-version module is entered, all versions are executed concurrently, and the results of all versions are passed to a voter which constructs the final result. Because the code of each version is different, the outputs of the various different versions is likely to be different too. Thus, the voter is application dependent, and must be able to cope with results which, although different, may all be correct. This, unfortunately, makes it incompatible with the voter used by the Delta-4 active replicates model which works by comparing message signatures, and thus requires the outputs being compared to be completely identical. So, N-version programming in Delta-4 requires a separate voting mechanism to be incorporated into the application.

There are a number of ways in which N-version modules may be combined with Delta-4's hardware fault tolerance mechanisms. Clearly, one could simply incorporate N-version modules into Delta-4 objects. However, the overheads of this technique (a number of versions running in each of a number of replicates) becomes considerable. An alternative approach is for each replicate of an object to run a different version of the N-version modules. Outputs of the different replicates,
and thus the different versions, are sent to an adjudicator which is simply a voting mechanism in the form of a Delta-4 object. The adjudicator may itself be replicated for dependability using the standard hardware fault tolerance mechanisms. The adjudicator represents the N-version object to the rest of the system, and having carried out its voting process, sends the appropriate output message. In order to gain the symmetry required by a remote procedure call mechanism, messages to an N-version object also go via the object’s adjudicator.

The main problem with this scheme is what to do when a version fails to deliver a correct result. (This is a problem with the basic N-version scheme as well as a Delta-4 implementation of it). Unless each version is essentially stateless: that is, unless versions retain no state between calls and return results based simply on the information supplied to them, the failure of a version will result in an inconsistent state in the object, and the object as a whole must be considered to have failed.

One way of dealing with the failure of a version, bearing in mind that such failures will hopefully be a rare occurrence, is to 'clone' a new replicate of the object to replace the one which has failed. Delta-4 already contains mechanisms for cloning new replicates of objects which has a failed copy due to hardware failure [1]. Cloning new versions of an object is harder, as each version is likely to have a different internal representation of the state of the computation being performed. Nevertheless, since many Delta-4 objects will be long-lived, the problem must be addressed. There are three possible solutions:

1. If N-version objects are stateless, then cloning is straightforward. For example, if a server carries over no state information from one computation to the next, returning to its initial state between each computation, then a new variant may be started from its initial state and begin operating with the next computation. Unfortunately, this is somewhat restrictive and few objects can reasonably be constructed in this way.

2. In some instances, it may be possible for objects to deduce the correct state by interrogating other components within the system. Once again, however, this mechanism is likely to be of limited use.

3. The most general, and most complex, mechanism is for new versions to acquire the necessary state information from existing, correct versions of the module via a standard interface. One (or more) of the existing versions carries out a translation of its state into a standard form. This is then passed to the new version, which uses it to build its own internal state. This mechanism does impose some constraints on state representations. However it does mean that versions may retain the ability to maintain their states in different forms whilst allowing new versions of objects to be created following failure.
5. Conclusions

The Delta-4 Open Systems Architecture was designed to meet the requirements involved in the implementation of dependable open systems. The dependability models for hardware fault tolerance (active replicates, passive replicates and leader/follower) were described. All three of the basic dependability models have been implemented within the Delta-4 project - active and passive replicates within the open systems architecture and leader/follower within both the open systems and extra performance architectures.

The problems of introducing software fault tolerance into the Delta-4 architectures were then addressed. It was shown that both recovery blocks and N-version programming can easily be incorporated into the existing framework without significant overheads. In conclusion, the Delta-4 system provides a comprehensive set of mechanisms from which an integrated fault tolerant system may be built.

Acknowledgements

The work of the Delta-4 project is carried out by a consortium of thirteen partners: BULL-MTS (F), Credit Agricole (F), Ferranti International (GB), IEI-CNR (I), IITB-Fraunhofer (D), INESC (P), LAAS-CNRS (F), LGI-IMAG (F), MARI (GB), NCSR (GB), Renault (F), Sema Group (F), University of Newcastle upon Tyne (GB). The authors would like to thank all members of the Delta-4 project. We would also like to thank the Commission of the European Communities, Information Technologies and Telecommunications Task Force for providing funding in support of this work under the ESPRIT programme.
References


