Towards an integrated approach to fault tolerance in Delta-4

To cite this article: P A Barrett and N A Speirs 1993 Distrib. Syst. Engng. 1 59

View the article online for updates and enhancements.

Related content
- Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes
  S Tao, P D Ezhilchelvan and S K Shrivastava
- The Voltan application programming environment for fail-silent processes
  D Black, C Low and S K Shrivastava
- Consul: a communication substrate for fault-tolerant distributed programs
  S Mishra, L L Peterson and R D Schlichting

Recent citations
- Fabian Scheler et al.
Towards an integrated approach to fault tolerance in Delta-4

P A Barrett and N A Speirs
Department of Computer Science, University of Newcastle upon Tyne, Newcastle upon Tyne, UK

Received 15 February 1993

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 addresses other areas of dependability, including software fault tolerance. This paper describes the architecture of the Delta-4 project and considers methods of achieving tolerance to software design failures in addition to hardware failures. The incorporation of these methods into the overall Delta-4 architecture are also described.

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 has been the Definition and Design of an Open, Dependable, Distributed computer system architecture (hence the project’s name). This architecture is intended for use in application areas such as Computer Integrated Manufacture, Office Automation etc., and the project has demonstrated its results via instances of the architecture in a Unix environment, running applications at Credit Agricole (a credit card authorization system) and at Renault (a process control application within an integrated manufacturing cell).

Dependability is defined as being a quality of delivered service such that reliance can justifiably be placed on that 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.

Openness, in the context of Delta-4, has a number of implications. Firstly, the Delta-4 architecture is capable of co-existing with, and inter-working under, standards conforming to the Open Systems Interconnection (OSI) reference model. Delta-4 can 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) conforms 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 have been published [1].

Delta-4 has introduced mechanisms which support explicitly the requirements of real-time systems with respect to both throughput and response. The real-time concepts of priorities and deadlines are supported and these concepts are reflected in its communication protocols. The Delta-4 project has also addressed the issue of design errors and has incorporated software fault tolerance mechanisms into the architecture. The integration of software fault tolerance mechanisms into the existing Delta-4 architecture, in order to provide a consistent, coherent approach to fault tolerance, is the subject of this paper.

We begin by presenting an overview of the Delta-4 architecture in section 2. Section 3 describes the three different hardware fault tolerance mechanisms implemented within the Delta-4 project. In section 4 we describe how Software Fault Tolerance techniques may be incorporated into 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 nodes of the system as shown in figure 1. Each software
A Barren and N Speirs

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 NACs are specialized communications processors which together implement a dependable communication system allowing multi-point communication between replicated computational entities. NACs 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, 3]. A fail-silent component exhibits the halt-on-failure property of fail-stop processors [4]. The fail-silence assumption for the network attachment controller is substantiated by the use of built in hardware and software 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 Delta4 software running on each node may be classified as follows.

1. The communications system software which executes on the NACs. The communication system of the Delta-4 open systems architecture is called the Multicast Communication System (MCS). The range of services supported by MCS ranges from unreliable unicast through to totally ordered atomic multicast. MCS provides these services by utilizing multi-endpoint connections and using a low level Atomic Multicast protocol (AMP) [5].

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. The administration software is, for example, responsible for failure detection of a node and for ‘cloning’—making a copy of an existing object on a new node [1].

3. The Delta-4 Application Support Environment (Deltase) which provides a framework for the construction of object-oriented, dependable, distributed applications. Deltase conforms with emerging standards for open distributed processing.

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 (RSRs). RSRs may be synchronous (in which case they are equivalent to Remote Procedure Calls) or asynchronous. The latter case allows a thread of control in a client object to continue to execute concurrently with a requested service; i.e. the service request represents the forking of a new thread of control which executes in parallel with the original thread until some point of mutual convenience at which the threads ‘join’ and the service results are passed back to the client.

3. Hardware fault tolerance in Delta-4

In the Delta-4 system, a number of techniques are available for providing tolerance to hardware faults. Each involves the use of replicated software components as shown in figure 2. Three different replication strategies are provided by Delta-4.

3.1. Active replication

Active replication is a technique in which several identical replicas of a process execute in parallel on different nodes of a system. Should one replica fail, the others continue to provide the required service. In active replication, replicas are required to behave deterministically with respect to one another. In order to achieve replica determinism we must satisfy the following conditions.

Input consistency. Each replica of a process must receive the same set of input messages in the same order. In Delta-4, input consistency is achieved through the use of ordered atomic multicast protocols (AMP) for message delivery.

Deterministic replicas. When starting from identical states, each replica of a process must, in processing a consistent set of input messages, produce an identical, and
identically ordered, set of outputs. This property is usually achieved by structuring processes as state machines [6].

In addition, active replication requires that the outputs from all components are compared and the majority decision is selected for use (voting) [7]. In general $2f + 1$ processors are required in order to tolerate $f$ node failures. For most applications, one failure is permitted ($f = 1$) and software components are replicated over three nodes.

In Delta-4, 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 replication model detects that a node is faulty when a message originating on that node fails to agree with the majority of its replicas, is produced late, not at all, or is produced in error; the latter three cases being determined through time-outs 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 [8].

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 two, the communication mechanisms 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.

3.2. Passive replication

A second technique used within Delta-4 to enhance reliability is passive replication. 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.

The sending of duplicate output messages can be avoided by means of either systematic or periodic checkpointing. Systematic checkpointing involves the creation of checkpoints whenever the primary replica 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 [9].

Periodic checkpointing is a strategy whereby the number of checkpoints is reduced by only taking them say, every $n$ output messages [10, 11]. During recovery, any output messages generated by the substitute replica 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 checkpoints requires that replicas be deterministic and that messages be received by all replicas in the same order (as for the active replica strategies) so that the substitute replica produces the same messages as those that were produced by the primary replica before its failure.

Although systematic checkpointing entails more overhead than periodic checkpointing, its capacity to accommodate non-deterministic processing is an important advantage.

3.3. Leader/follower replication

Each of the replication strategies outlined above 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, with the outputs of the fastest replica 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 replica 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 [6], or by implementing replica-deterministic objects using some form of agreement on execution path [12]. If active replica 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 replicas, since each replica 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 replica 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 replica is active, processing requirements are minimized; checkpoint capture will generally use less resource than replicated execution. Passive replication does, however, require the use of fail-silent hosts, and when a primary replica 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 in which all copies of an object are active in that they all execute the same code [13]. One copy is designated the leader, however, and is responsible for taking all decisions which affect replica determinism; such decisions are propagated from leader to follower via synchronization 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 replication model but uses additional processing to update the states of backups (or followers) rather than checkpoints.

Two forms of synchronization message are used in the leader/follower model: input synchronization messages and pre-emption synchronization messages. It is necessary in real-time applications that the notion of 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 precedence. However, all copies of an object must consume the same messages in exactly the same order, otherwise their paths may diverge and replica 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 synchronization 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 replica 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 replica 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 synchronization mechanism which imposes a very small overhead to ensure that replica 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 synchronization 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 replica, 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 of which this will take place is selected (the current counter value plus 1 represents the next pre-emption point) and assigned, and a synchronization 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 replica 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 synchronization 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 synchronization 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 variety of applications, a range of software fault tolerance techniques of varying cost and utility is required; see for example [14].
Also, of course, the techniques used must be compatible with the Delta-4 hardware fault tolerance techniques described above. We begin by reviewing software fault tolerance mechanisms and then examining their applicability within the Delta-4 framework. As we shall see, a number of techniques for dealing with software fault tolerance fit particularly well within the Delta-4 architecture.

4.1. Recovery blocks

A recovery block ([15] 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. The syntax of a recovery block is as follows:

```
ENSURE acceptance test
BY primary module
ELSE_BY first alternate module
ELSE_BY second alternate module
... 
ELSE_BY n-th alternate module
ELSE_ERROR.
```

Thus, a recovery block is a program module which offers tolerance to design faults. Consider, however, the implications of inter-process communication occurring whilst a program is executing within a recovery block. Any information divulged by a process whilst within a recovery block would be rendered invalid should that recovery block (or, rather, the executing alternate within the recovery block) fail, and those processes which had consumed that information must also be considered to have failed. One way around the problem is to ban inter-process communication whilst a process is executing within a recovery block. For many applications, however, this may be considered unreasonably restrictive, and an alternative method is required. One such mechanism is the dialogue [16].

Dialogues 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. Dialogues cause recovery to be propagated from a failed process to any other processes with which the failed process has communicated whilst within the dialogue. Recovery points, i.e. the state to which the affected processes will be recovered should a recovery block fail, are actually associated with dialogues rather than recovery blocks. Conditions for the termination of a dialogue are similar to those for the committal of a transaction; all processes must confirm successful termination of all processing which is to be carried out within the dialogue before a dialogue may be terminated. Should one of the processes within the dialogue be unable to terminate due to a failure, all processes within the dialogue must recover.

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 replication 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 optimization; replica outputs are required for voting, and thus all replicas must run the recovery blocks in parallel. The replica determinism property will ensure that all replicas 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 3. 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.

![Figure 3. Recovery blocks with passive replication.](image-url)
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 optimized in such a way that the followers never need to worry about failing with a software error, and thus do not 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 replica 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 synchronization messages to followers during the execution of recovery block alternates (synchronization messages are used at points of potential non-determinism, such as consumption of a message, to ensure that replica states do not diverge). In the example shown in figure 4, the leader sends three synchronization 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 synchronization 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 synchronization messages 4–7 are sent. Finally, the acceptance test succeeds, and synchronization message 8 instructs the followers to continue, executing alternate 2, and acting according to synchronization messages 4–7.

4.2. N-version programming

The second major technique available for software fault tolerance is N-version programming [17, 18]. 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 an adjudicator which constructs the final result. Because the code of each version is different, the outputs of the various different versions are likely to be different too. Thus, the adjudicator 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 comparator used by the Delta-4 active replication 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 replicas) becomes considerable. An alternative approach is for each replica of an object to run a different version of the N-version modules. Outputs of the different replicas, and thus the different versions, are sent to an adjudicator which, in this case, takes 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 general structure is shown in figure 5.

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

![Figure 4. Recovery blocks with leader/follower replication.](image)

![Figure 5. Software fault tolerance using N-version programming in Delta-4.](image)
occurrence, is to 'clone' a new replica of the object to replace the one which has failed. Delta-4 already contains mechanisms for cloning new replicas of objects which have failed copies 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. Many closed loop applications (e.g. aircraft control systems) possess this property.

(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 limited to specific applications.

(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.

A further consideration for N-version programming (not only within Delta-4) is ensuring that different software versions do not use the same underlying library or application routines without consideration of the reliability of such routines. The problem can be managed in Delta-4 since all object interaction is handled through the Delta-4 Application Support Environment. The support environment could then report any instances of different software versions of an object using the same underlying facility.

5. Conclusions

The Delta-4 architecture has been designed to meet the requirements involved in the implementation of dependable open systems. The dependability models for hardware fault tolerance (active replication, passive replication and leader/follower) have all been implemented within the project. We have shown that the problem of incorporating software fault tolerance into the Delta-4 architecture can readily be overcome. We have shown that both recovery blocks and N-version programming techniques 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.

Acknowledgments

The work of the Delta-4 project has been carried out by a consortium of thirteen partners: BULL-MTS (F), Credit Agricole (F), Ferrari International (GB), IEI-CNR (I), ITB-Fraunhofer (D), INESC (F), LAAS-CNRS (F), LIGI-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

[12] Tully A and Shrivastava S K 1990 Preventing state divergence in replicated distributed programs Proc. 9th


