Active Replication in Delta-4

Marc Chérèque[1] David Powell[2] Philippe Reynier[1] Jean-Luc Richier[3] Jacques Voiron[3]

[1] Bull SA
B.P. 208
38432 Echirolles (France)

[2] LAAS-CNRS
7, avenue du Colonel Roche
31077 Toulouse (France)

[3] LGI-IMAG
B.P. 53x
38041 Grenoble (France)

Abstract

Delta-4 is an open dependable distributed computing systems architecture, in which fault-tolerance is achieved by means of replication of run-time software components on host computers interconnected by a local area network. Several replication techniques have been designed and implemented; this paper concentrates on the coordination of active replicas executing either in a fail-silent host computer environment, or in a fail-uncontrolled one. This coordination is carried out by a specific protocol, the Inter Replica protocol.*

1. Introduction

Delta-4 is a collaborative project of the European ESPRIT program that has defined, implemented and prototyped an open, dependable, distributed computing architecture [1]. Fault tolerance in Delta-4 is achieved by replicating software components on distinct host computers interconnected by a local area network. Software components are active logical entities that communicate by means of messages and share no common memory.

Distributed fault-tolerance techniques based on software component replication have several advantages over tightly-coupled "stand-alone" fault-tolerance techniques:

- specialized hardware is kept to a minimum since distributed fault-tolerance is implemented primarily in software; the cost of specialized hardware design — or re-design when a technology update is required — is therefore minimized;
- geographical separation of resources does not have to be "added on" if disaster recovery is to be provided; the same distributed fault-tolerance techniques can be used regardless of whether the replicas are close to or distant from each other;
- loose-synchronization of replicas (through message-passing) leads to improved tolerance of transient faults that could otherwise simultaneously affect all redundant computations at the same point of execution [2];
- certain software design faults can be tolerated without resorting to diverse-design since slight differences in the local environment of the replicas can lead to design faults manifesting themselves independently in different replicas [3].

The Delta-4 architecture aims to accommodate host processors that can exhibit two extreme modes of failure: they can be either *fail-silent* or *fail-uncontrolled*. A fail-silent processor is one that either operates in conformance with its specification or remains silent [4]. In particular, any message sent by a fail-silent processor is a message that is correct in both value and time. Fail-uncontrolled processors can exhibit arbitrary types of failures, in both the time and value domains. Such hosts can send messages with erroneous contents, send messages too late or not at all, send extra or "impromptu" messages or refuse to accept (some) incoming messages.

Several replication techniques have been developed in Delta-4 — this paper concentrates on the *active replication technique*, which can be used for tolerating faults according to both the fail-silent and fail-uncontrolled host failure mode assumptions. In this technique, all replicas process concurrently all input messages so that their states are closely synchronized and, in the absence of faults, produce the same output messages in the same order. The coordination of active replicas is performed within the Delta-4 *Multipoint Communication System* (MCS). Error detection and compensation are handled by a special protocol, the *Inter Replica protocol* (IRp), which resides at the bottom of the session layer and uses the services of an underlying multipoint transport protocol.

The IRp protocol has been implemented and experimented in various Delta-4 prototypes. The design of the protocol is being verified using formal methods: the

* Supported by CEC through ESPRIT project Delta 4, n° 2252

protocol, the service it delivers and the environment assumptions are formally specified and the specifications have been formally verified for consistency by model checking techniques. The implementation is also being tested by fault-injection techniques. The main aspects of this validation are summarized in [5].

The paper is organized as follows. Section 2 discusses general issues of fault-tolerance implemented by replication in distributed systems. Section 3 describes the active replication technique implemented in Delta-4. Section 4 is devoted to the formal specification and verification of the IRp protocol and section 5 gives some implementation details and results of performance tests. Section 6 concludes the paper by a brief comparison with related work.

2. Fault-tolerance by replication of software components

Replication of data and computation on different nodes is the only means by which a distributed system may continue to provide non-degraded service in the presence of failed nodes. Even though stable storage within nodes can be used to allow the system to recover (eventually) from node failures, such techniques used alone[1] do not allow distributed systems to achieve better availability than a non-distributed system.

We define a *software component* to be an elementary run-time unit of distributed computation and data encapsulation that, in the absence of replication, resides on a single node. Even when several software components co-reside on a single node, they do not share common memory. The data encapsulated by a software component is referred to as its *state*. Software components are active logical entities that may communicate with each other by means of messages through one or more *ports*. A *replicated software component* is defined as a software component that possesses a representation or *replica* on two or more nodes. Unless stated otherwise, the term *software component* will refer to the logical entity as a whole, i.e., the *group* of replicas.

2.1. Replica failure mode assumptions

If only hardware faults are considered, it can be assumed that the replicas executing on a given host fail in a way that is defined by the assumed failure behavior of that host. If a fail-silent host fails, then all replicas that were executing on that host will appear to have failed silently or to have "crashed". Conversely, if a fail-uncontrolled host fails, then any or all replicas that were being executed can fail arbitrarily, i.e., send messages too early or too late, omit to send some messages, send messages with incorrect content or send extra or "impromptu" messages.

However, it is possible to refine these assumptions and at the same time use a finer failure granularity than that of a complete host [6]. For instance, an incident in a node's local operating system could cause a single replica to crash. Alternatively, if a host becomes overloaded (due to a *configuration fault*), then although the host hardware may be completely fault-free, buffer overflow may cause replicas on that node to fail by omitting to respond (*omission failures* [7]). Of course, if a software component contains a residual *design fault*, then all replicas could fail in a quite arbitrary fashion. Any faults that manifest themselves *independently* on different hosts can be tolerated by means of active replication — some host configuration faults and software component design faults fall in this category.

A wide spectrum of failure modes with different severities can be defined (e.g., see [8]). However, for the particular case of interest here, i.e., software component replicas executing in a distributed message-passing environment, just a few modes are sufficient. Replicas that are assumed to suffer only crash, omission or late-timing failures can be collectively termed *fail-restrained* replicas since, by assumption, they only ever send messages that are of correct value; replicas that can fail arbitrarily are termed *fail-uncontrolled* replicas.

Note that whereas replicas may be assumed to be fail-restrained *or* fail-uncontrolled when executing on fail-silent hosts, all replicas executing on fail-uncontrolled hosts must be assumed to be fail-uncontrolled.

2.2. Replica determinism and coordination techniques

A *replica* (of a given software component) is said to be deterministic if, in the absence of faults, any execution of the replica starting from the same initial state and consuming the same ordered set of input messages leads to the same ordered set of output messages.

A *replica group* is deterministic if, in the absence of faults, given the same initial state for each replica and the same set of input messages, every replica in the group produces the same ordered set of output messages. If all replicas in a group consume identical input messages in the same order, then replica determinism is a sufficient condition for replica group determinism.

The design of software components must be considered very carefully if replica determinism is to be ensured. For example, the use of preemption, local timers or local random number generators must be forbidden. Replica determinism can be further complicated by heterogeneity.

[1] We do not consider cross-strapping of stable storage mechanisms to several nodes since this is inappropriate for nodes that are not co-located.

If a software component is compiled with different compilers (for different machines, or even for the same machine), the resulting replicas may not behave consistently.

If replicas are not deterministic, then replica group determinism can only be achieved by negotiation between replicas [9]. Alternatively, the potentiality for replica non-determinism can be removed by adopting a restrictive model of computation based on *state machines* [10].

It is assumed in the remainder of this paper that due attention has been paid to the design of software components so that, in the absence of faults, replicas are deterministic. Consequently, a sufficient condition for ensuring replica group determinism is to ensure that all replicas receive the same input messages in the same order.

There are two main issues in the replication of software components for fault-tolerance:

- *group membership management*: how is a software component instantiated as a group of replicas and how is group membership updated as a consequence of failures and repairs (fault treatment)? This issue will not be discussed in this paper; for details, see [11].
- *inter-replica coordination:* how is the activity of the group of replicas coordinated in order to process errors and give the illusion to other software components of a single (fault-free) software component?

Three basic replication techniques have been studied in the Delta-4 architecture:

- *passive replication*: one of the replicas (the *primary* replica) processes the input messages and provides output messages — in the absence of faults, the internal states of the other replicas are regularly updated by means of checkpoints from the primary replica [12];
- *semi-active replication*: one of the replicas (the *leader* replica) processes all input messages and provides output messages — in the absence of faults, the other replicas (the *follower* replicas) do not produce output messages; their internal state is updated either by direct processing of input messages or by means of "notifications" from the leader replica [13];
- *active replication* is a technique in which all replicas process all input messages concurrently so that their internal states are synchronized — in the absence of faults, outputs can be taken from any replica.

The choice for a particular software component replication technique depends on the replica failure mode assumption and on the existence or not of replica determinism (see table 1).

Table 1 - Delta-4 replication coordination techniques.

Replication technique	*Error-processing overhead*	*Replica non-determinism*	*Fail-uncontrolled behavior*
Active	Lowest	Forbidden	Tolerated
Passive	Highest	Allowed	Forbidden
Semi-active	Low	Resolved	Forbidden[2]

2.3. Replica coordination entities and node architecture

It is desirable to be able to program a software component without taking into account the fact that it may be instantiated as a group of replicas. The programmer should be able to concentrate on the logical problem or function that the software component is meant to solve or to provide without having to deal with the intricacies of replica coordination.

It is therefore useful to separate this replica coordination functionality and let it be provided by one or more standard system entities. Each software component replica can be thought of as having one or more local "replica coordination entities" (abbreviated hereafter to *rep_entities*) acting on its behalf[3] (figure 1).

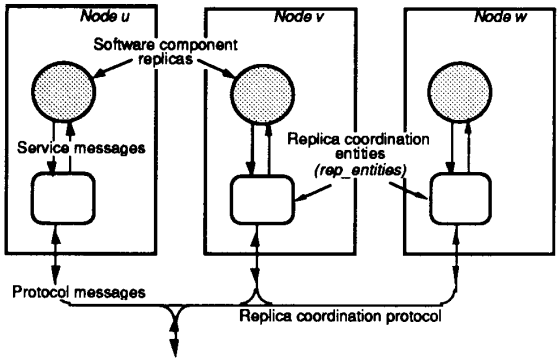

Fig. 1 - Software component replicas and replica coordination entities

In Delta-4, *rep_entities* are assumed to be fail-restrained even if their corresponding replicas are fail-uncontrolled; this important distinction in assumed failure modes allows a considerable simplification of the distributed error-processing protocols for the fail-uncontrolled case. It means, however, that *rep_entities* should be executed on fail-silent hardware — either on the host, if the latter is

[2] An extension of the semi-active replication technique to accommodate fail-uncontrolled behavior has also been investigated.

[3] The overall management of resources and, in particular, replica installation, also requires appropriate administration facilities (see [14]).

fail-silent, or on a purpose-designed, fail-silent *network attachment controller* (NAC) [4].

Each Delta-4 node is composed of a host processor (that can be either fail-silent or fail-uncontrolled) with an associated fail-silent NAC supporting the *rep_entities* of replicas executing on the local host. NACs are implemented with enhanced self-checking mechanisms based on dual processors and comparators to substantiate the assumed fail-silent behavior. It is this important distinction between the failure modes of the replicas and the failure modes of their *rep_entities* that enables the Delta-4 architecture to achieve tolerance of the arbitrary faults of fail-uncontrolled host processors without resorting to a costly, meshed host interconnection structure (for example, see [7, 15]).

3. Active replication

In the following, we concentrate on the coordination of active replicas. With this technique, quasi-instantaneous recovery from detected errors can be achieved if it can be guaranteed that all correct replicas produce the same output messages in the same order over the same output ports — this is referred to as the *output consistency* condition. Sufficient conditions for output consistency are:

- *input consistency:* the sets of input messages delivered to correct replicas are identical;
- *replica group determinism.*

The *input consistency* condition implies that the communication protocol used to transmit messages to an actively-replicated software component must be a *reliable group communication* protocol that ensures *unanimity* between correct recipients.

For *replica group determinism*, the active replication technique adopted here requires replica groups to be made deterministic by (cf. §2.2):

- ensuring that correct replicas receive the same messages in an *identical order*;
- enforcing replica determinism by structuring software components as *state machines*.

Support for active replication in Delta-4 is provided by the Inter Replica protocol (IRp) layer. It includes services needed to handle communication between *rep_entities*, such as:

- cross-checking the service requests issued by the group of replicas;
- propagation on the network of a single message corresponding to each successfully cross-checked replicated "data send" request;
- support for dynamic changes in the replica group.

The IRp protocol is responsible for detecting errors due to failures of replicas and host processors. IRp also ensures error recovery whether errors are detected by the underlying multipoint transport protocol or by IRp itself.

IRp uses the service provided by the Delta-4 multipoint transport protocol, which allows the reliable, ordered and time-bounded dissemination of information in a group of nodes. All the communication software is executed on fail-silent network attachment controllers (NACs). A NAC will stop processing on self-detected errors, or when it detects a host processor crash. This results in a node crash that is detected by all other nodes by means of the multipoint transport service.

3.1. Message transmission processing

Different philosophies for processing errors in output messages may be followed according to the underlying replica failure mode assumption.

Fail-restrained active replicas: In this case, since any output sent by any replica of the group can be assumed to be of correct value, it is possible to choose any of the outputs and to discard the others. Data and/or code replication techniques for continued operation in the presence of t faulty fail-silent processors need only rely on $t+1$ replicas.

Since any output is a correct-valued output, it would be possible to relax the output consistency condition and optimize the use of individual replicas by, for example, sending requests that do not modify the state of the component to just one replica of the group. If no response is forthcoming, the request can be re-submitted to another replica. Such "read optimization" (e.g., see [16]) allows a decrease in node workload and in message traffic over the network at the expense of imposing a read-write serialization mechanism to ensure replica consistency However, we do not consider such an optimization, since:

- in the general "software component" paradigm, outputs cannot necessarily be paired with a corresponding "read-request" input (e.g., a software component could be programmed to periodically transmit some internally computed value);
- we wish to provide similar mechanisms for managing replicated active replicas with both the fail-restrained and fail-uncontrolled assumptions.

From the output message viewpoint, the *rep_entity* processing activity for managing fail-restrained replicas consists of a simple arbitration between the local rep_entities to decide which of them will send the message.

Fail-uncontrolled active replicas: If replicas are fail-uncontrolled, the set of replica outputs must be considered as a whole. Since failures of fail-uncontrolled host processors can manifest themselves by messages

being sent with erroneous content, replication techniques for continued operation in the presence of t faulty fail-uncontrolled processors must be based on at least $2.t+1$ replicas so that a minority of erroneous messages can be masked.

To process value errors in messages, the *rep_entities* must cross-check each message sent by the local replica with equivalent messages sent by remote cohort replicas. This cross-checking is referred to here as *message validation.*

The message validation mechanism is built into the output message selection protocol executed by *rep_entities*. As soon as $t+1$ messages are found to agree, then, since there are only supposed to be t errors, it can be safely assumed that the consensus message that is propagated is error-free. It is important to underline that the *rep_entity* that propagates the validated message must not alter the message in any way — rep_entities must be fail-restrained (cf. §2.3) so that it can be assumed that any message that they do send is a correctly-valued message.

To process time domain errors, *rep_entities* use timers to check that at least $t+1$ replicas send equivalent messages within a specified time interval T_d.

To prevent faults in the remaining replicas from remaining dormant, it is also necessary to ensure that all replicas are regularly activated and checked. This can be done either by rotation of the $t+1$ replicas whose messages are compared or by systematic comparison of messages from all $2t+1$ replicas. The latter approach is simpler to implement and can provide acceptable performance if the last t messages are compared after having propagated the consensus value.

3.2. The validation protocol

The principle of inter-replica coordination is that each replica forwards all its output messages to its local *rep_entity*. The *rep_entities* use the multipoint transport service to notify the whole group of *rep_entities* (including themselves) of a local change of status (such as reception of a request, or time-outs). Since this transport service guarantees that the same information is delivered to all receivers in the same order, all *rep_entities* can consistently monitor the state of all the other *rep_entities*.

Most of IRp is based on a common kernel, the validation protocol, which carries out the cross-checking of replicated requests. We concentrate in the following on the IRp data send protocol, which is an illustration of the use of the validation protocol. A detailed presentation can be found in [17].

The first aim of the validation protocol is to detect, mask and report errors. This includes: data validation (detection of value errors), synchronization (detection of timing errors by limitation of the desynchronization between replicas), loss of replicas and error reporting. The second aim is to ensure that at most one valid message per (replicated) request is sent. This covers two problems, sender selection and data propagation.

The protocol is executed in two phases. The first phase copes with data validation and sender selection. The second phase copes with synchronization and data propagation. The loss of replicas must be handled during both phases.

Data validation and sender selection phase: For performance reasons, requests are validated by comparing the *signatures* of requests issued by each replicated entity. The *rep_entities* exchange the signatures of the local requests and as soon as the number of matching signatures is greater than or equal to a given threshold V, the request is validated.

The threshold V depends on the replica failure mode assumption. If replicas are fail-uncontrolled, the threshold is usually set equal to $\lfloor N/2 \rfloor+1$, where N is the number of replicas. If replicas are fail-restrained, all issued requests have a correct value, so the threshold is set equal to 1.

Since only one message must be propagated, it is necessary to select exactly one *rep_entity* to send the validated message. This may be done in two different ways:

The so-called *competitive mode* proceeds as follows. When a *rep_entity* dequeues a data message from its replica, it multicasts a *claim* protocol message to the group of *rep_entities* (including itself); the *claim* message includes the signature of the data message received from the local replica. Since the underlying multipoint transport service ensures ordered delivery of protocol messages to all *rep_entities*, the *claim* messages from the group of *rep_entities* will be received by all in the same order. Each *rep_entity* compares the signatures of the sequence of claim messages until V signatures are found to be identical; this point in the sequence is termed the *validation point.* The unique *rep_entity* that reaches its validation point by means of the signature in its own *claim* message, is elected to forward the locally-received data message to its destination(s).

In the *cyclic* or *round-robin mode*, the group of *rep_entities* is configured as a logical ring with an associated token. When it receives a local message, the *rep_entity* that owns the token sends the message signature to all members of the group by means of a *turn* protocol message. This *turn* message implicitly forwards the token to the rep_entity's successor in the ring. The token circulates around the ring until it reaches a *rep_entity* for which the signature of the local message is identical to that contained in V-1 previous *turn* messages. This *rep_entity* has thus reached its validation point and is

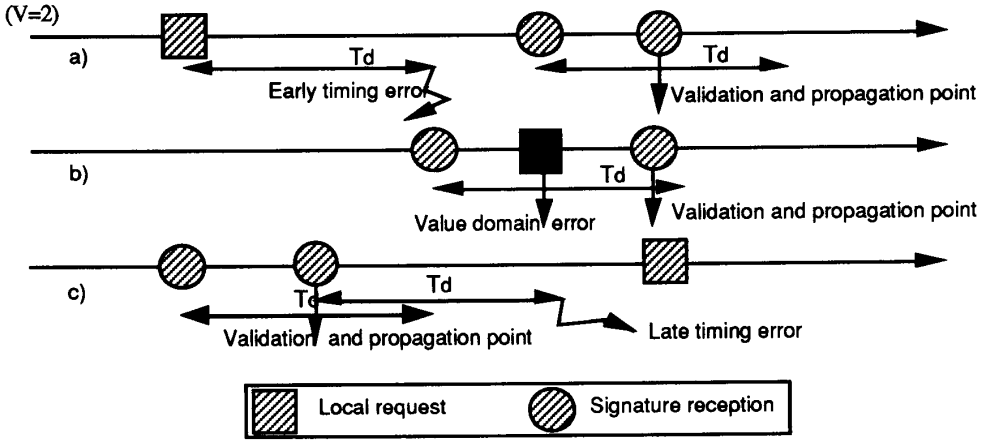

Fig. 2 - Different types of errors detected by the inter replica validation protocol
(as seen by *rep_entity* of faulty replica)

elected to forward the corresponding data message to its destination(s).

To ensure token recovery, the interval between receipt of the local data message and receipt of the token is monitored. If time-out occurs, then the *rep_entity* reverts to the competitive mode by issuing a *claim* message.

Early-timing and late-timing errors (including omission and silence) are detected and masked by monitoring the time interval between receipt of the local data message and the validation point or vice versa (see figure 2). Impromptu errors are detected as value errors if an impromptu message is received *within* the time interval (and of course if its signature is different to that of V other signatures) or as early-timing errors if no timing window has been opened.

Synchronization and data propagation phase: The *rep_entity* elected during the first phase forwards the data message to the (possibly replicated) destination entity together with an *ack* protocol message to the other source *rep_entities*, in the *same atomic multicast transfer*. The other *rep_entities* keep their local message until either receiving an *ack* message from another *rep_entity* (indicating that the data message has been forwarded) or receiving the indication of failure of the *rep_entity* that should send the data message. In this case, a new *rep_entity* is selected to carry out the data transfer.

The competitive mode gives preference to the V fastest replicas of the group and can allow other replicas to lag further and further behind. The amount of desynchronization is explicitly limited by requiring the *rep_entities* to carry out a periodic rendezvous during which they wait for all *claim* messages to be received before one of them forwards the corresponding data message. The rendezvous is again time-limited to detect and recover from replica silence, omission errors and late-timing errors. In practice, the fastest replica slows down because it *must* send a *claim* message whereas the slower replica(s) will not need to send one if the fastest replica's *claim* message has already been received.

Error reporting and loss of replicas: The inter-replica protocol allows detection and compensation of both value domain and time domain errors. In both cases, the service provided to the *faulty* replica is *aborted.*

Replicas can be lost either because they have crashed (e.g., due to a node failure) or because they have been aborted by the IRp protocol. If the loss occurs during the validation phase, two cases need to be handled:

- the number of replicas still alive is greater than or equal to the threshold V,
- the number of replicas still alive is lower than V.

The second case reports a global error and all the replicas are aborted. Otherwise, the validation protocol continues (with the same threshold value V).

In the round-robin mode, if the token owner is lost, the token is given to the next *rep_entity*. If the loss occurs during the data propagation phase, a new token owner is chosen among the valid *rep_entities* (i.e., the *rep_entities* that agree with the majority).

4. Formal specification and verification

The IRp data send protocol is the critical kernel of active replica coordination so it has been formally specified in order to carry out both design and implementation validation. The complete specification is based on the set of state transition tables defining the protocol, the description of the service expected from IRp and on the service delivered by the underlying multipoint transport

protocol. Only the service specifications are given here since space prevents us from giving the protocol state transition tables (15 states, 10 events and 12 local conditions to be tested). The full specification can be found in [17].

4.1. Specification

Let us consider a set of *N rep_entities*, receiving requests from *N* replicas. A replicated request corresponds to a request issued by the software component, i.e., each replica issues an instance of this request. Globally, the set of *rep_entities* receives and processes a sequence of replicated requests in the following way:

- Each *rep_entity* receives one instance of the considered replicated request.
- There is a window of one request for the service: replicas must wait for the confirmation of the previous request before issuing a new one.
- In the absence of faults, a replicated request is a set of *N* identical request instances occurring within an interval of at most *T1* local time units, where *T1* is the specified maximum inter-replica skew[4].
- The expected actions resulting from a replicated request are:
  - the delivery of a single correct message to its intended recipient(s);
  - and either the positive confirmation to each replica or the abort of the associated *rep_entity*, unless it fails.
- A *rep_entity aborts* when value-domain or time-domain errors are detected:
  - value-domain errors may occur if request instances have different data fields;
  - time-domain errors may occur if occurrences of different request instances are too far apart.
- A *rep_entity fails* when the node fails, or when failures occur in the underlying multipoint transport service.

The IRp service: The following notation is used:

- *V* is the "validation threshold",
- "Matching" means for a set of messages or requests: same parameters, same data fields (this predicate is generalized to a set of sequences of messages or requests).

The service that should be delivered by the IRp data send protocol is defined by the following set of properties:

- *Ordering*: For each replica, the sequence of delivered messages matches the sequence of positively confirmed replicated requests (this property is a consequence of the other properties and of the "one request window" constraint of the IRp protocol);
- *Non-triviality*: Any message delivered matches a request instance that was received by at least *V rep_entities* (the "non-validated" protocol case, for fail-restrained replicas, is characterized by $V = 1$);
- *Unicity*: For each replicated request, at most one message is delivered;
- *Delivery*: If matching request instances are received by *V rep_entities* within an interval of *T1* local time units, then a matching message is delivered unless one of the *V rep_entities* fails before the delivery;
- *Bounded time*: For any message delivered, each *rep_entity* either received a matching request instance at most *T2* local time units before this delivery or will receive a matching request at most *T3* local time units after this delivery unless the *rep_entity* fails, otherwise the *rep_entity* aborts (a "rendezvous" synchronization phase is characterized by $T3 = 0$);
- *Termination*: Within a delay of *T4* local time units, unless the *rep_entity* fails, either a received request is positively confirmed to the local replica after the delivery of the message, if the request matched the delivered message, or the *rep_entity* aborts.

The multipoint transport service: IRp uses the service delivered by an underlying *multipoint transport layer*. This protocol extends the very strong properties of Delta-4's atomic multicast protocol [1, 18] to long messages by providing suitable data fragmentation and reassembly procedures.

The service needed by the IRp data send protocol is defined by the following set of properties:

- *Unanimity*: any message delivered to a correct participant[5] is delivered to all correct participants;
- *Non-triviality*: any message delivered was sent by a correct participant;
- *Accessibility*: any message delivered was delivered to a participant correct and accessible for that message;
- *Delivery*: any message from a correct sender is delivered;
- *Consistent causal order*: any two messages delivered to a participant are delivered to all correct participants in the same order, which obeys causality;
- *Consistent group view*: changes to group membership are indicated to all correct group participants and obey *consistent causal ordering*;
- *Unicity*: any message is delivered at most once to a participant;
- *Termination*: messages and notifications of group membership changes are delivered within a known bounded time.

The primitives used are requests and indications: the service is not acknowledged. If the transport layer is unable

[4] Replica cloning protocols (see [17]) initialize each replica in such a state that the sequence of replicated requests is consistent.

[5] A correct participant is a participant residing on a fault-free node.

to deliver the service, the *rep_entity* issuing the request is aborted. All other *rep_entities* will then be notified through a group membership change indication.

4.2. Verification

The consistency of the formal specifications has been verified by *model checking* techniques, using the verification tool *Xesar* developed at LGI [19]. Model checking techniques consist in automatically constructing a model from the description of the distributed system, and checking on this model properties describing the expected service. These methods have been chosen because they can be automated to a large extent, and are very effective for revealing design faults since they are based on a complete search of the model of all possible behaviors of the system.

The service properties to be checked are described by temporal logic formulas and Büchi automata.

The model represents the behavior of a communicating system defined by interconnected nodes, each one executing the protocol to be validated. This system is described by a finite set of processes, forming a *closed system*, i.e., for each possible communication, the complete description of both transmission and reception must be provided. A complete description of the environment is not needed, but the process(es) describing it must allow all behaviors satisfying the *hypotheses on which the protocol is based.*

Thus, the model is a *state graph* obtained by translation and composition from the formal specifications:

- For each process, *states* represent the values of variables and *transitions* correspond to atomic steps. The latter are either *internal* steps (occurring inside a process independently of the others) or *communications* with other processes.
- The model of the whole system is a composition of the individual state graphs taking into account the timing constraints of the environment [1].

The validity of the formal verification depends on the *soundness* of the model, in the sense that all properties formally verified on it must be true in the real world, or that each error detected on it must correspond to an error of the protocol. If the description of protocol and its environment is successfully verified, it can be deduced that the protocol works correctly in any environment satisfying the hypotheses.

Effective verification consists of complete checking of particular configurations of networks (*scenarios*). The whole set of scenarios must cover all possible protocol behaviors. These formal specifications and verification techniques are discussed with more detail in [1, 20].

Verification of the scenarios defined for IRp has shown inconsistencies in the first formal specifications. As a result, several design inadequacies were identified, leading to deadlock or early termination of the protocol, and improvements proposed and implemented. The consolidated implementation has been derived from these formal specifications. The interested reader should refer to [21].

5. Implementation and performance

Various prototypes of the Delta-4 architecture have been developed, including both hardware and software support for active replication. Specific network attachment controllers (NACs) able to run the complete communication stack have been implemented. They possess enhanced self-checking mechanisms based on dual processors and comparators to substantiate the assumed fail-silent execution of *rep_entities*.

The complete software support for active replication includes a communication stack (including the IRp protocol) and application libraries. IRp has been implemented as a sub-layer of the session layer. The mapping between the software components and the various entities of the IRp sub-layer can be described as follows. There is a one-to-one link between a software component and an inter-replica service access point (SAP). Like the software components, the SAPs are replicated entities. A single *rep_entity* is attached to each SAP instance. It is initialized by an administrative operation that defines the initial group membership. The input and output ports described earlier are mapped onto connection endpoints. The endpoints are part of the SAP entity. Each endpoint is both an input and an output port. The endpoints are also replicated entities. Independent validation protocol machines are attached to each endpoint instance. Each validation protocol machine processes a single request flow issued out of the connected output port. At the first reception of an event concerning the request (either local or remote event), an independent protocol machine is initialized with all the needed information (list of replicas, validation threshold, value of timers). This protocol machine performs the validation of a single request and terminates when the request is fully processed. Any detected error causes the abort of the SAP instance used by the faulty replica, which is thus deprived of its communication resources.

Performance tests have been carried out on the Delta-4 prototype using a "null" remote procedure call in which a single client sends a message to a replicated server that then replies with the same message. The hardware prototypes were Bull DPX/2-340 workstations (Motorola 68030 rated at 33MHz). The communication software including the IRp and the multipoint transport protocol was implemented on a self-checking NAC based on a

duplicated Motorola 68030 rated at 25MHz. The network was a 16Mb/s token ring.

Table 2 shows, for different message sizes, the measured null-RPC round-trip delay for a non-replicated server (*N=1*) and the overheads in the round-robin (RR) mode for duplicated and triplicated servers. In the latter case (*N=3*), the 'Tripl.' column indicates the triplication overhead when no validation is required (no signature computation and validation threshold *V=1*). The 'Valid.' column indicates the additional overhead when the threshold *V* is set to *2*.

Table 2 - Round-robin mode null-RPC delays (in milliseconds)

Size (bytes)	*N=1*	*N=2, RR, V=1*		*N=3, RR, V=2*		
		Total	Dupl.	Total	Tripl.	Valid.
6	14.6	22.4	7.8	35.8	21.2	8.6
100	15	22.8	7.8	36.4	21.4	9
1000	20	28	8	41.8	21.8	9.4
5000	41.6	49.6	8	66.4	24.8	12.1
10000	68.4	76.4	8	96.8	28.4	15.8
15000	95.6	103.6	8	130.6	35	22.2
25000	159.8	171	11.2	202.8	43	26.8

Figure 3 represents the overhead of each option when the message size varies. The round-trip delay for a null-RPC to a non-replicated server is compared to the overhead of each option (replication, validation) when the server is replicated. It can be seen that the overhead of the various options is only slightly dependent on message size. This is essentially due to the use of message signatures by the validation protocol, since the overhead of signature computation is low compared to the overhead that would be involved by the exchange of complete messages. The consequence is that the replication protocol overhead becomes proportionally low when message size increases.

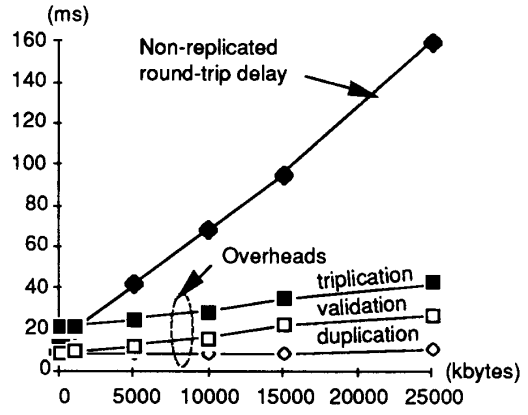

Fig. 3 - Basic delay and various overheads according to message size

Another lesson to be learnt from these tests concerns the comparison of the round-robin and competitive modes. At first sight, it would appear that the competitive IRp mode should have a performance advantage over the round-robin mode in that it allows message propagation at the earliest possible opportunity. However, this must be weighed against the higher protocol message traffic that is incurred due to the systematic sending of *claim* messages. The test campaigns have demonstrated unambiguously that the round-robin mode is more powerful. Another natural advantage of the round-robin mode is that it provides automatic synchronization of replicas. For example, the results given in table 2 should be compared to those given in table 3 when the competitive mode is used.

Table 3 - Competitive mode null-RPC delays (in milliseconds)

Size (bytes)	*N=1*	*N=2, CP, V=1*		*N=3, CP, V=2*		
		Total	Repl.	Total	Repl.	Valid.
6	14.6	34	19.4	43	28.4	0.2
100	15	34.4	19.4	43.4	28.4	0.2
1000	20	39.6	16.6	49.2	29.2	1
5000	41.6	61	19.4	73.6	32	3.6
10000	68.4	88	19.6	104	35.6	7.2
15000	95.6	115.2	19.6	134.8	39.2	10.4
25000	159.8	182.8	23	209.2	49.4	17.6

This comparison shows the considerable advantage of the round-robin mode in terms of replication overheads, because of the systematic sending of *claim* messages by all *rep_entities* in the competitive mode (when replicas are closely synchronized, which was the case for the tests described here). This is not compensated by the lower validation cost of the competitive mode (which represents only the overhead of signature computation); this can be explained by the fact that in the round-robin mode, it is the local request that allows the elected *rep_entity* to reach the validation point (and thus only *V-1* turn messages are transmitted on the network), but in the competitive mode, *V* identical claim messages are required.

6. Conclusion

Replicating computation over a distributed system has been shown to be a viable technique for achieving fault-tolerance. In contrast to most other previously-published work (see, for example, [22-24]), our approach does not assume that replicas are only subject to crash or timing

failures; it allows for the detection and compensation of messages with erroneous content due to arbitrary failures. Other similar schemes, such as those presented in [15, 25], rely on voting on replicated messages sent over multiple communication paths in order to accommodate arbitrary node failures. In our approach, voting is carried out on signatures *before* sending full data messages over the network. This was made possible by the fact that Delta-4 nodes are split into two distinct sub-systems: an off-the-shelf host computer (that can fail in arbitrary fashion) and a purpose-built fail-silent network attachment controller. Consequently, our active replication technique can be, and has been, implemented using standard local area networks with limited (or no) redundancy. Furthermore, performance measurements have shown that the percentage overhead due to replication and source-based voting decreases with increasing message size.

References

[1] D. Powell (Ed.), *Delta-4: a Generic Architecture for Dependable Distributed Computing,* Research Reports ESPRIT, 484p., Springer-Verlag, Berlin, 1991.

[2] H. Kopetz, H. Kantz, G. Grünsteidl, P. Puscher and J. Reisinger, "Tolerating Transient Faults in MARS", in *Proc. 20th. Int. Symp. on Fault-Tolerant Computing (FTCS-20),* pp.466-473, IEEE, Newcastle upon Tyne, June 1990.

[3] J. Gray, "Why do Computers Stop and What can be done about it?", in *Proc. 5th. Symp. on Reliability in Distributed Software and Database Systems,* pp.3-12, IEEE, Los Angeles, CA, Jan. 1986.

[4] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", in *Proc. 18th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* pp.246-251, IEEE, Tokyo, June 1988.

[5] K. Kanoun, J. Arlat, L. Burrill, Y. Crouzet, S. Graf, E. Martins, A. MacInnes, D. Powell, J. L. Richier and J. Voiron, "Delta-4 Architecture Validation", in *Proc. Esprit Conference,* pp.234-252, CEC-DGXIII, Brussels, November 1991.

[6] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Comm. ACM,* 34 (2), pp.56-78, Feb. 1991.

[7] F. Cristian, H. Aghali, R. Strong and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", in *Proc. 15th. Int. Symp. on Fault-Tolerant Computing (FTCS-15),* pp.200-206, IEEE, Ann Arbor, MI, June 1985.

[8] D. Powell, "Failure Mode Assumptions and Assumption Coverage", in *Proc. 22nd. Int. Symp. on Fault-Tolerant Computing (FTCS-22),* IEEE, Boston, MA, July 1992.

[9] A. Tully and S. K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", in *Proc. 9th. Symp. on Reliable Dist. Systems (SRDS-9),* pp.104-113, IEEE, Huntsville, AL, Oct. 1990.

[10] F. B. Schneider, "Implementing Fault Tolerant Services using the State Machine Approach: a Tutorial", *ACM Comp. Surveys,* 22 (4), pp.229-319, Dec. 1990.

[11] U. Bügel and B. Gilmore, "Process Cloning in Delta-4", in *ESPRIT Information Processing Systems and Software - Results and Progress of Selected Projects,* XIII/372/91, pp.179-189, CEC-DGXIII, Brussels, 1991.

[12] N. A. Speirs and P. A. Barrett, "Using Passive Replicates in Delta-4 to provide Dependable Distributed Computing", in *Proc. 19th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-19),* pp.184-190, IEEE, Chigago, MI, June 1989.

[13] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues and N. A. Speirs, "The Delta-4 XPA Extra Performance Architecture", in *Proc. 20th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20),* pp.481-488, IEEE, Newcastle upon Tyne, June 1990.

[14] G. Bonn, U. Bügel, F. Kaiser and T. Usländer, "Management of the Delta-4 Open, Distributed and Dependable Computing System", in *Proc. IFIP TC6/WG6.6 Symp. on Integrated Network Management,* pp.573-584, North-Holland, Boston, MA, May 1989.

[15] P. M. Melliar-Smith and R. L. Schwartz, "Formal Specification and Mechanical Verification of SIFT: A Fault-Tolerance Flight Control System", *IEEE Trans. Computers,* C-31 (7), pp.616-630, July 1982.

[16] P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987.

[17] *Implementation Guide IG2,* Delta-4 Project Consortium, Delta-4 Document, Dec. 1991.

[18] P. Veríssimo and J. A. Marques, "Reliable Broadcast for Fault-Tolerance on Local Computer Networks", in *Proc. 9th. Symp. on Reliable Distributed Systems,* pp.54-63, IEEE, Huntsville, AL, Oct. 1990.

[19] S. Graf, J. L. Richier, C. Rodriguez and J. Voiron, "What are the Limits of Model Checking for the Verification of Real Life Protocols", in *Proc. Workshop on Automatic Verification Methods for Finite State Systems,* LNCS 407, pp.275-285, Springer-Verlag, Grenoble, June 1989.

[20] M. Baptista, S. Graf, J.-L. Richier, L. Rodrigues, C. Rodriguez, P. Veríssimo and J. Voiron, "Formal Specification and Verification of a Network Independent Atomic Multicast Protocol", in *Proc. 3rd. Int. Conf. on Formal Description Techniques (FORTE'90),* North-Holland, 1990.

[21] J.-L. Richier and J. Voiron, *Verification of the IRp Protocol,* LGI, Report RTC 31, November 1991.

[22] K. P. Birman, "Replication and Fault-Tolerance in the ISIS System", *ACM Op. Sys. Review,* 19 (5), pp.79-86, 1985.

[23] A. Borg, W. Blau, W. Graetsch, F. Herrmann and W. Oberle, "Fault Tolerance under UNIX™", *ACM Trans. Computer Systems,* 7 (1), pp.1-24, Feb. 1989 1989.

[24] F. Cristian, B. Dancey and J. Dehn, "Fault-Tolerance in the Advanced Automation System", in *Proc. 20th. Int. Symp. on Fault-Tolerant Computing (FTCS-20),* pp.6-17, June, IEEE, Newcastle upon Tyne, UK, 1990.

[25] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso and U. Voges, "The UCLA DeDiX System: A Distributed Testbed for Multiple-version Software", in *Proc. 15th. Int. Symp. on Fault-Tolerant Computing (FTCS-15),* pp.126-134, IEEE, Ann Arbor, MI, June 1985.