lly for soft real-time services, or
of the cost and time required to

ıt less mature school of design,
ɔognises the limitations on scale
ieving mathematically exact and
and probability theory, in which
ıtage.

s, including large-scale complex
ilosophies. In one special case,
ɔut the demand for soft real-time
approaches in one system.

of its openness and generality, it
ling of all system latencies and
· the development of the Extra

Chapter 6

Distributed Fault-Tolerance

Distribution and fault-tolerance are tightly related. Should a single element of a distributed system fail, users expect at worst a slight degradation of the service that is offered; distributed systems must thus at least have *some* built-in fault-tolerance. On the other hand, most fault-tolerant systems can, at some level or another, be seen as a distributed system due to their redundant processing resources. Distributed fault-tolerance is used here to refer to that class of techniques suitable for ensuring fault-tolerance in an architecture consisting of a set of processing elements (called *nodes* or *stations*) interconnected by a message-passing communication network (figure 1). The distributed fault-tolerance techniques discussed here are focussed towards distributed systems in which the communication network consists of one or more local area networks. In particular, the existence of high-bandwidth broadcast channels allowing efficient multicast communication is assumed.

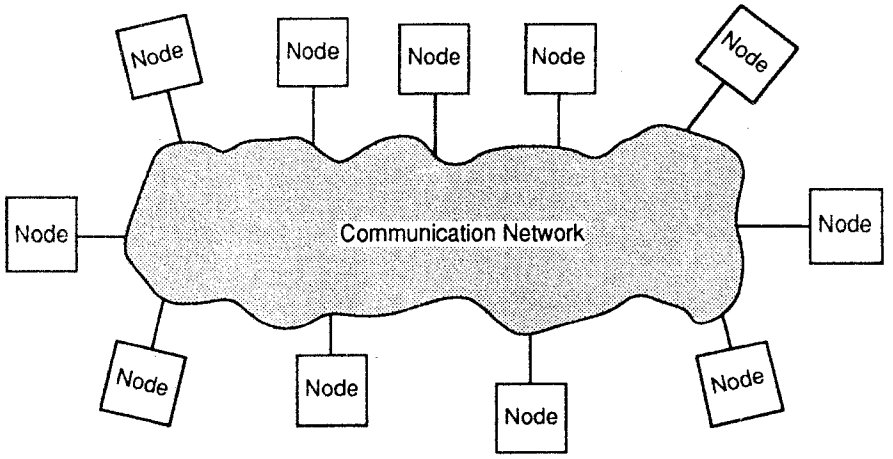


Fig. 1 - Distributed System

The chapter is organized as follows. After discussing some of the related work in this field, section §6.3 discusses the characteristics of the computational nodes of the considered class of distributed systems. In particular, this section details the assumed node failure modes and their impact on the design of distributed fault-tolerance techniques. The assumed models of distributed computation are sketched out in section 6.4, then section 6.5 considers various aspects relating to the replication of computation for the purposes of fault-tolerance.

The two issues of *error-processing* (preventing errors from affecting service delivery) and *fault treatment* (preventing faults from inducing further errors) are dealt with separately. Error-processing is described in sections 6.5 through to 6.7 which describe three classes of techniques for coordinating replicated computation, called active, passive and semi-active replication. Fault treatment is discussed in section 6.8.

Finally, section 6.9 is devoted to the definition of group communication facilities that the underlying communication system should offer to support the various distributed fault-tolerance techniques.

6.1. Related Work

Fault-tolerance is often implemented in single "stand-alone" machines rather than in the distributed fashion considered here. Such stand-alone fault-tolerant machines have several disadvantages:

- much special-purpose hardware must be designed — and re-designed when a technology update is required; our software-implemented distributed approach allows the use of standard, off-the-shelf machines;
- geographical separation of redundant resources has to be "added on" if disaster recovery is to be ensured; geographical separation of redundant resources is just a configuration parameter in our distributed approach — the same techniques are used whether the redundant resources are close to or distant from each other;
- stand-alone fault-tolerant machines often resort to tight synchronization in order to ensure that the fault-tolerance mechanisms are as transparent as possible to application software; such tight synchronization leads to a reduced tolerance of transient faults that could simultaneously affect all redundant channels at the same point in computation [Kopetz et al. 1990];
- tightly-coupled redundancy does not provide the same tolerance to software-design faults as a loosely-coupled approach; in particular, restart and recovery mechanisms must also be included to provide tolerance of operating system crashes.

Distributed approaches to fault-tolerance via message-passing replicated computation similar to those presented here have also been used in a number of other fault-tolerant systems, many of which are also aimed at supporting real-time applications [Birman 1985, Borg et al. 1983, Cooper 1984, Cristian et al. 1990, Gunningberg 1983, Kopetz and Merker 1985, Mishra et al. 1989, Shrivastava et al. 1988, Walter et al. 1985, Wensley et al. 1978]. Some of these are described briefly here and compared to Delta-4.

One of the first distributed fault-tolerant architectures was the SIFT multi-computer machine developed for critical flight-control applications [Melliar-Smith and Schwartz 1982, Wensley et al. 1978]. The MAFT system is a more recent example of the same type of architecture [Kieckhafer et al. 1988, Walter et al. 1985]. The SIFT architecture is based on general-purpose processors or nodes interconnected by a set of broadcast serial busses. Due to the criticality of the intended application domain (civil aircraft flight control), no restrictive assumption is made about node failure modes, i.e., nodes may be fail-uncontrolled (see section §6.2.2). To mask the arbitrary behaviour of such nodes, each node is connected to every other node by its own private broadcast bus. Single-source data is broadcast to all nodes by means of a clock-synchronous, phased Byzantine Agreement Protocol [Melliar-Smith and Schwartz 1982]. Such an interconnection structure was feasible in SIFT since the architecture was designed to accommodate at most eight nodes geographically located in the same equipment bay. It was required that the Delta-4 architecture be able to accommodate several tens of nodes spread out over quite considerable distances (i.e., a factory or several large neighbouring

buildings) so such an interconnecti
SIFT is based on majority voting of
Tasks are executed according to a st
4 is intended to be an *open* archite
local operating systems and serv
synchronous frame-based schedulir
node failure is relatively simple in
tasks can be maintained in every n
contain little or no internal state (p
be as large as a complete databas
create (or *clone*) new replicas who
replicas on non-faulty nodes and t
replicas are created.

MARS [Kopetz et al. 1988, K
tolerant system for real-time appl
goes beyond that of a single equip
(nodes are designed to be self-che
baseband bus. The local clocks of
designed clock synchronization cl
synchronization (less than 10μs) t
of time-division multiplexing. Sir
tasks based on a static (off-line)
application scenario. However, M
are assumed to be fail-silent; nod
the time domain). Since all compu
the instants of communication and
the communication system doe
overflow. Furthermore, due to th
schedule, reliable broadcasting c
mask *k* transmission errors. Like
XPA architecture, see chapter
overheads of voting. However,
architecture adopts a dynamic ev
computation and communication
on the system may suddenly v
applications, no *a priori* worst-ca
scheduling philosophy of XPA a
Like MARS, the clocks of node
reasons, clocks are synchronized

The ISIS system [Birman
presents many similarities to De
fault-tolerance in a general-purp
and MARS systems that are tai
system provides a flexible *tool*-
build a distributed application t
assumes that nodes or proces
provides a single mechanism f
cohort scheme. This scheme i
technique (see section 6.7). T
the issue of resolving replica nc
implementation — by the a

rom affecting service delivery) and
rs) are dealt with separately. Error-
7 which describe three classes of
d active, passive and semi-active

p communication facilities that the
e various distributed fault-tolerance

ne" machines rather than in the
lt-tolerant machines have several

ned — and re-designed when a
plemented distributed approach

has to be "added on" if disaster
n of redundant resources is just a
n — the same techniques are used
ant from each other;

tight synchronization in order to
as transparent as possible to
leads to a reduced tolerance of
l redundant channels at the same

me tolerance to software-design
restart and recovery mechanisms
ing system crashes.

passing replicated computation
r of other fault-tolerant systems,
tions [Birman 1985, Borg et al.
opetz and Merker 1985, Mishra
y et al. 1978]. Some of these are

was the SIFT multi-computer
lliar-Smith and Schwartz 1982,
example of the same type of
SIFT architecture is based on
broadcast serial busses. Due to
t flight control), no restrictive
e fail-uncontrolled (see section
de is connected to every other
dcast to all nodes by means of
[Melliar-Smith and Schwartz
T since the architecture was
cated in the same equipment
nmodate several tens of nodes
r several large neighbouring

buildings) so such an interconnection structure is not economically viable. Error processing in SIFT is based on majority voting of the results of tasks that are replicated across several nodes. Tasks are executed according to a static frame-based cyclic schedule (calculated off-line). Delta-4 is intended to be an *open* architecture designed to accommodate heterogeneous nodes and local operating systems and serve a wider range of applications; systematic use of such synchronous frame-based scheduling was thus precluded. Finally, system reconfiguration after node failure is relatively simple in SIFT. Copies of the data necessary for the execution of all tasks can be maintained in every node of the system since the elements of computation (tasks) contain little or no internal state (persistent data). In Delta-4, the elements of computation may be as large as a complete database system. The reconfiguration mechanism must be able to create (or *clone*) new replicas whose internal state is initialized by copying the state of existing replicas on non-faulty nodes and transferring it across the network to the nodes on which new replicas are created.

MARS [Kopetz et al. 1988, Kopetz and Merker 1985] is an example of a distributed fault-tolerant system for real-time applications in which the geographical distance between nodes goes beyond that of a single equipment bay. The nodes in MARS are assumed to be fail-silent (nodes are designed to be self-checking, see section 6.2.1) and are interconnected by a serial baseband bus. The local clocks of each node are closely synchronized by means of a specially-designed clock synchronization chip [Kopetz and Ochsenreiter 1987] that achieves such a tight synchronization (less than 10μs) that it can be used to control access to the serial bus by means of time-division multiplexing. Similarly to SIFT and MAFT, MARS uses cyclic scheduling of tasks based on a static (off-line) schedule taking into account the worst-case or peak-load application scenario. However, MARS does not need to resort to majority voting since nodes are assumed to be fail-silent; node failure only results in the absence of messages (detected in the time domain). Since all computation and communication in MARS are statically scheduled, the instants of communication and the quantity of transferred data are pre-established; therefore, the communication system does not have to worry about flow control to prevent buffer overflow. Furthermore, due to the fail-silent node assumption and the static communication schedule, reliable broadcasting can be achieved by systematic $k+1$ repetition of messages to mask k transmission errors. Like MARS, the real-time variant of the Delta-4 architecture (the XPA architecture, see chapter 9) also adopts a fail-silent node assumption to avoid the overheads of voting. However, unlike the static time-triggered approach of MARS, the XPA architecture adopts a dynamic event-triggered approach. This allows a more economical use of computation and communication resources for dynamic applications in which the load imposed on the system may suddenly vary due to the occurrence of asynchronous events. In many applications, no *a priori* worst-case load scenario can be determined; in such cases, the dynamic scheduling philosophy of XPA allows a best-effort approach to meeting application deadlines. Like MARS, the clocks of nodes in XPA are globally synchronized; however, for commercial reasons, clocks are synchronized without resorting to special-purpose hardware.

The ISIS system [Birman and Joseph 1987, Birman 1985, Birman and Joseph 1987] presents many similarities to Delta-4. Like Delta-4, ISIS is aimed at providing user-transparent fault-tolerance in a general-purpose distributed computing environment (as opposed to the SIFT and MARS systems that are tailored to clock-synchronous, real-time applications). The ISIS system provides a flexible *tool-kit* of basic primitives that allow an application programmer to build a distributed application that is made fault-tolerant by replication of code and data. ISIS assumes that nodes or processes fail only by crashing (i.e., that they are fail-silent) and provides a single mechanism for fault-tolerance at the process level based on a *coordinator-cohort* scheme. This scheme is similar in some respects to Delta-4's semi-active replication technique (see section 6.7). The ISIS coordinator-cohort scheme does not, however, address the issue of resolving replica non-determinism. The tools provided by ISIS could also allow the implementation — by the application programmer — of actively-replicated processes

(restricted to a fail-silence assumption) and passively-replicated processes (by making coordinators systematically transfer their state to cohorts when a service request is completed). In Delta-4, many applications can be designed as if they were to run on a system that never fails. Since fault-tolerance is managed by built-in system facilities, the issues of replication can be entirely hidden from the application programmer and only specified at configuration time. ISIS also provides the basic *state transfer* mechanism [Birman and Joseph 1987] necessary to ensure the cloning of replicas for system reconfiguration during fault treatment. However, since ISIS assumes fail-silence, the state transfer tool does not provide a facility for error-detection during state transfer by cross-checking states copied from multiple source replicas. Like Delta-4, ISIS makes use of special communication facilities supporting *multicast* protocols based on a clock-asynchronous approach rather than the clock-synchronous techniques of SIFT and MARS. However, the ISIS multicast protocol suite is implemented on top of TCP/IP such that each multicast results in a number of point-to-point TCP/IP messages. In Delta-4, the basic atomic multicast protocol is implemented on top, or as extension of, the medium access control protocol of selected local area networks (see section 6.9 and chapter 10); this allows hardware broadcasting opportunities to be exploited for increased performance.

A system that resembles Delta-4 quite closely is IBM's Advanced Automation System (AAS) for the US Air Traffic Control network [Cristian et al. 1990]. The AAS concept of "server groups" is equivalent to that of a "replicated software component" developed here. In AAS, both active and passive replication techniques are available[1] but only for the case of server replicas that fail by responding late, by omitting to respond or by crashing. In our approach, the case of fail-uncontrolled (active) replicas — ones that can fail in quite arbitrary fashion — is also accommodated by means of a built-in voting mechanism. The Delta-4 atomic multicast protocol allows replicated entities to be logically addressed such that messages are delivered (with low overhead) to all replicas; this results in somewhat simpler management of active replicas than in AAS. Processors in AAS are organized in distinct *processor groups* that can each support replicas of a given set of servers. There is a *group service availability manager* (gSAM) for each processor group, with replicas on all processors of that group. The gSAM is responsible for ensuring that the number of replicas of all servers supported by the processor group is maintained according to the server group's replication policy (specifying the minimum number of required replicas). If a server group can no longer execute according to its replication policy then, in some cases, it may be moved to another processor group under the control of a *global service availability manager* (GSAM). Each processor group supports a group membership service and a group broadcast service that enables gSAM replicas to maintain a consistent view of the processor group's global state. A routed multicast facility is also provided for communication between server groups residing in the same or different processor groups. In Delta-4, management is based on the open distributed system concepts of "managed objects" and "management domains". The nearest equivalent to the AAS processor group and gSAM is that of a software component *replication domain* and its corresponding *replication domain manager* or RDM (see sections 6.4.3 and 8.2.3.3.1). However, since nodes in Delta-4 are not *a priori* split into groups, replication domains for different software components may overlap. Indeed, some software components may have a replication domain that spans all nodes in the system. A global processor membership service is provided over all nodes in a Delta-4 system and is built into the Delta-4 atomic multicast protocol.

6.2. Node Hardware Characteristics

This section identifies the different failure modes that can be assumed for nodes and sketches out the communication network topologies that are appropriate for achieving fault-tolerance

[1] Referred to in AAS as *close* and *loose* synchronization of replicas.

(formal definitions of assumed fa
a processor, local memory and s
the internal node architecture ar
fault-tolerance techniques.

6.2.1. Fail-Silent Nodes

A fail-silent node is defined
communication network, either
[Powell et al. 1988]. In particul
correct in both value and time.

Some authors describe suc
silent" is preferred here since "fa
only the "halt-on-failure" chara
informed of the failure [Schlic
accessing the node's stable sto
explicit the necessary *external* p
of the node disconnecting itsel
carry out, for example, some lo
a fail-silent node exhibits "crash

Numerous implementations
failure mode such as that emb
particular, an essential assumpt
as a structuring concept for a
§6.3.2).

There are several important

First, since fail-silent node
any "two-faced" behaviour of a
the simplest possible [Fischer 1
achieve consensus in the prese
(the problem is vacuous for n

Second, since such a node
correct values or not at all, the
has failed by means of a simp
regularly transmitted "I'm aliv
also bounded [Fischer et al. 19

Third, data and/or code re
faulty nodes need only rely on

Fourth, since such an
communication network bein
faults in the communication ne
Thus, simple communication
be envisaged (see figure 2).

However, a 100% guaran
implemented with perfect, zer
are available for implementin
off-the-shelf general purpose c

Consequently, although c
coverage of this assumption m

(formal definitions of assumed failure modes are given in annexe E). A node consists of at least a processor, local memory and some sort of communication network interface; refinements of the internal node architecture are discussed later that enable simplifications of the distributed fault-tolerance techniques.

6.2.1. Fail-Silent Nodes

A fail-silent node is defined here to be a processing element that, viewed from the communication network, either operates in conformance with its specification or remains silent [Powell et al. 1988]. In particular, any message sent by a fail-silent node is a message that is correct in both value and time.

Some authors describe such nodes as being "fail-stop" (cf. chapter 4); the epithet "fail-silent" is preferred here since "fail-stop processors" have been previously defined to include not only the "halt-on-failure" characteristic implied here but also the fact that the other nodes are informed of the failure [Schlichting and Schneider 1983] or even that they are capable of accessing the node's stable storage [Schneider 1984]. Moreover, the term fail-silent makes explicit the necessary *external* perception of node activity and does not preclude the possibility of the node disconnecting itself from the network (i.e., going silent) but remaining active to carry out, for example, some local testing activity. In the terminology of [Cristian et al. 1985], a fail-silent node exhibits "crash" failure semantics.

Numerous implementations of fault-tolerance in distributed systems assume a "clean" node failure mode such as that embodied in the above definition of a fail-silent node. It is, in particular, an essential assumption in all previous work that we know of that uses transactions as a structuring concept for achieving fault-tolerance in distributed systems (see section §6.3.2).

There are several important simplifications that result from the fail-silent node assumption.

First, since fail-silent nodes never send any messages that are incorrect (thus eliminating any "two-faced" behaviour of a failed node), solutions to the distributed consensus problem are the simplest possible [Fischer 1983]. In particular, the minimum number of nodes necessary to achieve consensus in the presence of t faulty nodes, is given by $n \geq t+2$ [Lamport et al. 1982] (the problem is vacuous for $n \leq t+1$).

Second, since such a node either sends messages within specified time delays and with correct values or not at all, the other nodes of the distributed system can detect whether a node has failed by means of a simple interrogation and time-out mechanism or by timing out on regularly transmitted "I'm alive" messages (subject to the fact that communication delays are also bounded [Fischer et al. 1985]).

Third, data and/or code replication techniques for continued operation in the presence of t faulty nodes need only rely on $t+1$ replicas (see section 6.4).

Fourth, since such an assumption effectively precludes any possibility of the communication network being saturated by spontaneously-produced "garbage" messages, faults in the communication network can be dealt with independently from faults in the nodes. Thus, simple communication architectures using shared multipoint transmission channels can be envisaged (see figure 2).

However, a 100% guarantee that nodes are indeed fail-silent implies that the nodes are implemented with perfect, zero-latency self-checking mechanisms. Although many techniques are available for implementing self-checking hardware [Wakerly 1978], it is not easy to buy off-the-shelf general purpose computers that use these techniques extensively.

Consequently, although one can *assume* that an off-the-shelf computer is fail-silent, the coverage of this assumption may not be very high (see annex F). To attain a higher degree of

dependability, a less stringent assumption with higher coverage can be adopted, e.g., that of "fail-uncontrolled" nodes.

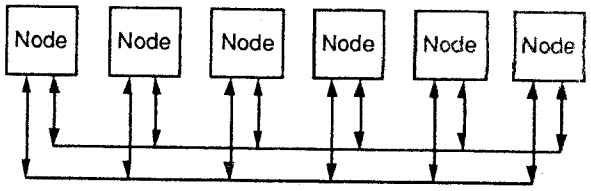


Fig. 2 - Interconnection Topology suitable for Fail-Silent Nodes

6.2.2. Fail-Uncontrolled Nodes

A fail-uncontrolled node represents the opposite extreme of the failure mode spectrum, i.e., a node that may fail in a quite arbitrary fashion. In particular, fail-uncontrolled nodes can:

a) send messages that are late (including completely omitting to send messages);

b) send messages sooner than expected;

c) send messages with erroneous content;

d) send unspecified or "impromptu" messages [Powell 1991].

Since this worst case assumption (see annexe E, expression 3) is — by essence — true, the probability of the assumption being satisfied in a real system (the assumption coverage, see annexe F) can be set equal to 1. The resulting ease with which it is possible to quantify the dependability achievable with such a worst-case assumption must however be weighed against several important disadvantages.

First, a fail-uncontrolled node can exhibit quite malicious behaviour[2]. For example, when relaying a message received from one processor, it could relay different copies to different destinations (the so-called "Byzantine" faults). It can be shown that the minimum number of nodes necessary to achieve consensus in the presence of t faulty nodes exhibiting such two-faced behaviour is given by $n \geq 3.t + 1$ (e.g., 4 nodes for 1 fault) [Lamport et al. 1982].

Second, it is impossible to use a simple interrogation and time-out mechanism to detect whether a fail-uncontrolled node has failed since a faulty node can reply correctly to an interrogation yet still send erroneous messages to other nodes. Node failures can only be revealed by comparing the activity of different nodes (and of course assuming that nodes fail independently).

Third, since node failure can manifest itself by messages being sent at the wrong time or with erroneous content, data and/or code replication techniques for continued operation in the presence of t faulty nodes must be based on at least $2.t+1$ replicas so that a minority of erroneous messages can be masked (see §6.4).

Fourth, since a faulty fail-uncontrolled node may generate an arbitrary number of "impromptu" messages, any (or all) transmission channel(s) to which it is attached may be saturated by garbage messages that prevent the channel(s) from being used by other nodes. This is a simple illustration of the fact that, from the viewpoint of error propagation, there is no built-in "error containment barrier" at the node interface like the one that exists for a fail-silent

2 Any assumption to the contrary would not be the worst case and would need to quantified by an appropriate less-than-unity assumption coverage.

node. Furthermore, since the c
be sent with erroneous source a
to know where they came from
non-faulty nodes and thus foil
to rely on the node interconne
authenticated messages (whic
less than arbitrary and consequ
Consequently, the only viable
are those in which nodes do no
Melliar-Smith and Schwartz 1
suitable for constructing a fault
assumption. The first of the
unidirectional multi-drop bus
[Melliar-Smith and Schwartz 1
is connected to each of its ne
implied in [Cristian et al. 1985
limited to the fault-containm
channel(s) (examples are show

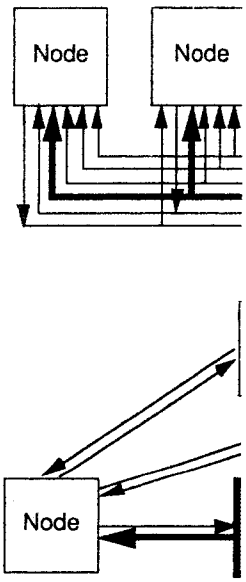
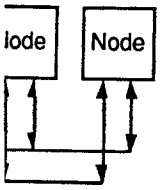


Fig. 3 - Interconn

The main disadvantages of
are that of high cost and lack of
a new (set of) transmission cha
or (b) to consider an intermedia

rage can be adopted, e.g., that of



ail-Silent Nodes

the failure mode spectrum, i.e., a
l-uncontrolled nodes can:
mitting to send messages);

l 1991].
sion 3) is — by essence — true,
m (the assumption coverage, see
ich it is possible to quantify the
ust however be weighed against

behaviour[2]. For example, when
lay different copies to different
vn that the minimum number of
ulty nodes exhibiting such two-
ult) [Lamport et al. 1982].
d time-out mechanism to detect
node can reply correctly to an
des. Node failures can only be
course assuming that nodes fail

being sent at the wrong time or
s for continued operation in the
replicas so that a minority of

erate an arbitrary number of
to which it is attached may be
om being used by other nodes.
of error propagation, there is no
e one that exists for a fail-silent

l need to quantified by an appropriate

node. Furthermore, since the content of erroneous messages may be arbitrary, messages may be sent with erroneous source address fields that would make it impossible for a receiving node to know where they came from. A faulty node could thus masquerade as an arbitrary number of non-faulty nodes and thus foil any attempt at a consensus by the latter. It is therefore necessary to rely on the node interconnection topology to identify the source of all messages or to use authenticated messages (which essentially defines a *different* failure mode assumption that is less than arbitrary and consequently has an assumption coverage less than 1, [Powell 1991]). Consequently, the only viable fault-tolerant architectures with such a node failure assumption are those in which nodes do not all share the same transmission channels ([Lamport et al. 1982, Melliar-Smith and Schwartz 1982]). Figure 3 gives two possible interconnection topologies suitable for constructing a fault-tolerant distributed system based on the fail-uncontrolled node assumption. The first of these shows each node connected to every other node by a unidirectional multi-drop bus (this is in fact the architecture of the SIFT multiprocessor [Melliar-Smith and Schwartz 1982]). The second shows a meshed network in which each node is connected to each of its neighbours by a unidirectional channel (such an architecture is implied in [Cristian et al. 1985]). In both cases, the immediate consequence of node failure is limited to the fault-containment domain constituted by the node itself and its associated channel(s) (examples are shown in heavy lines on figure 3).

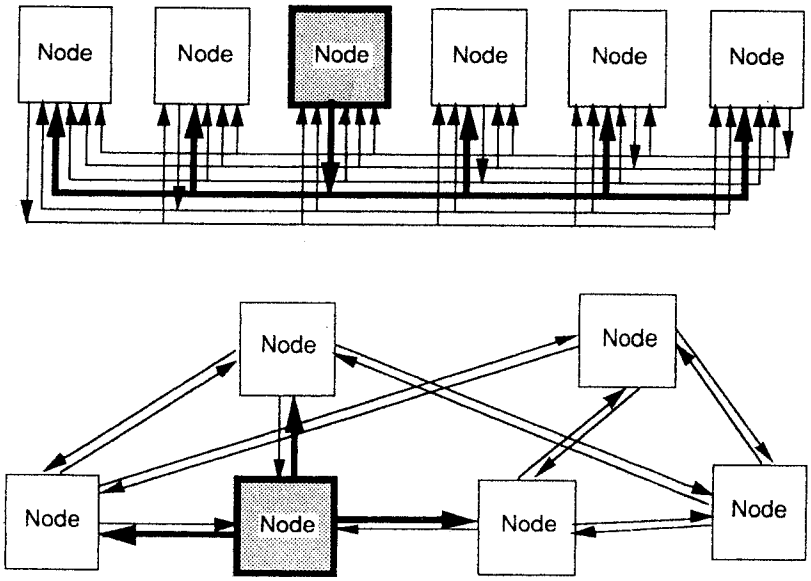


Fig. 3 - Interconnection Topologies suitable for Fail-Uncontrolled Nodes

The main disadvantages of the architectures for fail-uncontrolled nodes shown on figure 3 are that of high cost and lack of extensibility; the addition of a new node implies the addition of a new (set of) transmission channel(s). The alternatives are thus (a) to assume fail-silent nodes or (b) to consider an intermediate solution such as that proposed in the next section.

6.2.3. Network Attachment Controllers

A compromise between the restrictive fail-silent node assumption and the worst-case fail-uncontrolled node assumption can be envisaged whereby each node is considered as consisting of two components (figure 4):

- an off-the-shelf computational component, called a *host*, that may be fail-uncontrolled;
- a purpose-built communication component, called a *network attachment controller* (NAC), that is assumed to be fail-silent.

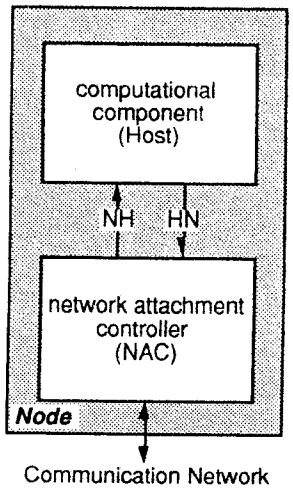


Fig. 4 - Node split into Host and NAC Components

Since the network attachment controller is a purpose-built component, the fail-silence assumption can be substantiated by built-in hardware self-checking techniques.

The intent of such a node architecture is to allow the use of simplified node interconnection topologies and less elaborate consensus protocols, while at the same time allowing for arbitrary failures of the off-the-shelf host component. Consider now the host/NAC interface that can be viewed as two links over which each perceives the other's behaviour:

- HN: the host to NAC link over which the host sends the NAC the following sorts of information (which, for the benefit of this description, will be termed "service items"):
  - messages to be sent over the communication network (resulting from computation on the host);
  - handshakes (requests or acknowledgements) for service items sent over the NH link;
- NH: the NAC to host link over which the NAC sends the host the following sorts of service items:
  - messages received from the communication network (that will affect future computation on the host);

- handshakes (r
  HN link.

Whereas the behaviour of
messages sent over the networ
items on the HN link. For a fa
uncontrolled host behaviour nee

a) send messages or ha
b) send messages or ha
c) send messages or ha
d) send unspecified or '

In the case of handshakes,
controlling the flow of informa
(see §6.5.3) since it difficult
handshake being late: (i) the r
means that the NAC must requ
flow of information, or (ii) the
be able to continue the flow of n

Possibilities b), c) and d) a
fail-silent assumption, protecti
unexpected or incorrectly-value
a master role in the interactior
transfer information to the NAC
NAC itself.

The NH link of the host-NA
delivery of incorrect NH service
itself failing. In practice, howev
techniques to substantiate its fa
silent behaviour can be assumec

In the remainder of this c
assumed. Each node is split int
fail-silent, whereas hosts may b
shown in figure 5 may contain
also chapter 10). Management
assumed that the resulting local
network partitioning is not cons

6.2.4. Stable Storage

In database applications, there
database to be stored while so
condition cannot be fulfilled (e
computation may need to be a
intention is to allow computatio
extension, the mechanism (stora
states to be stored is referred to a

This concept has frequent
computation cannot complete
Intermediate states (or *checkpoi*
if an error should be detected du

- handshakes (requests or acknowledgements) for service items sent over the HN link.

Whereas the behaviour of fail-silent or fail-uncontrolled nodes was defined in terms of messages sent over the network, the behaviour of *hosts* must be defined in terms of service items on the HN link. For a fail-uncontrolled host, the following possibilities for faulty fail-uncontrolled host behaviour need to be considered:

a) send messages or handshakes that are late (or completely omitted);

b) send messages or handshakes sooner than expected;

c) send messages or handshakes with erroneous content;

d) send unspecified or "impromptu" messages or handshakes.

In the case of handshakes, possibility a) above is of particular importance with regard to controlling the flow of information being multicasted to several, possibly failed, destinations (see §6.5.3) since it difficult to distinguish between the two possible causes of a host handshake being late: (i) the receiving host entity could blocked for a logical reason, which means that the NAC must request the "network" (ultimately, the sending nodes) to stop the flow of information, or (ii) the host could have failed, in which case the sending nodes should be able to continue the flow of multicasted information.

Possibilities b), c) and d) above clearly indicate that, if the NAC is to be able to respect a fail-silent assumption, protection mechanisms must be built into the NAC to shelter it from unexpected or incorrectly-valued HN service items. This militates in favour of the NAC playing a master role in the interactions across the host-NAC interface whereby the host may only transfer information to the NAC at times, and to locations in the NAC memory, dictated by the NAC itself.

The NH link of the host-NAC interface presents fewer problems since the non-delivery or delivery of incorrect NH service items is (will eventually be perceived as) equivalent to the host itself failing. In practice, however, since the NAC needs to be implemented using self-checking techniques to substantiate its fail-silent behaviour with respect to the network, a similar fail-silent behaviour can be assumed for the NH link.

In the remainder of this chapter, the hardware architecture shown in figure 5 will be assumed. Each node is split into host and NAC components; NACs are always assumed to be fail-silent, whereas hosts may be either fail-silent or fail-uncontrolled. The local area network shown in figure 5 may contain redundant communication channels or *media* (cf. figure 2, see also chapter 10). Management of these redundant channels is not considered here — it is assumed that the resulting local area network is (internally) fault-tolerant; in particular, physical network partitioning is not considered.

6.2.4. Stable Storage

In database applications, there is a need for a mechanism that allows a consistent state of the database to be stored while some new tentative computation is carried out. If some logical condition cannot be fulfilled (e.g., due to a bank account being insufficiently funded), such computation may need to be aborted and the previous state of the database restored. The intention is to allow computation to be carried out as a series of atomic steps or *transactions*. By extension, the mechanism (storage device and atomic update procedures) allowing intermediate states to be stored is referred to as *atomic storage*.

This concept has frequently been extended to allow a previous state to be restored if computation cannot complete due to a *fault* condition (as opposed to a logical condition). Intermediate states (or *checkpoints*) of a node's computation are stored in a "safe" place so that, if an error should be detected during subsequent computation, a previous error-free state can be

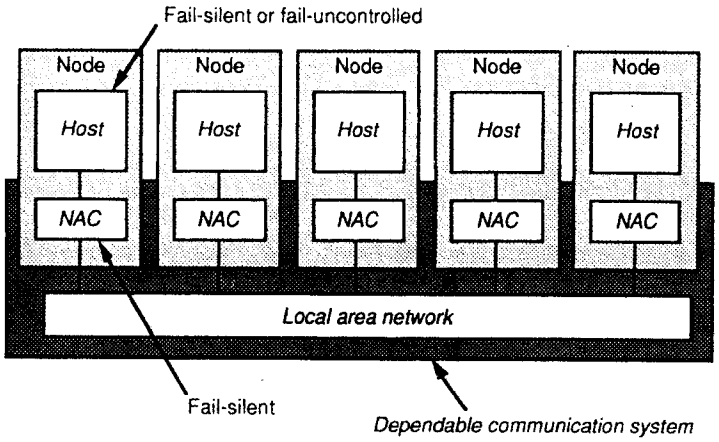


Fig. 5 - System Hardware Architecture

retrieved. As a means for fault-tolerance, the "safe place" in which checkpoints are stored must in itself be fault-tolerant; in particular, since it should be able to survive power outages, such a storage mechanism is referred to as *stable storage*. Stable storage is typically implemented using magnetic disc technology, although semi-conductor *stable memories* have also been implemented [Banâtre et al. 1986, Banâtre et al. 1988].

From the viewpoint of fault-tolerance, it is important to distinguish between these two different motivations for storing the intermediate states of a computation, i.e., a) as a means in transactional applications to allow tentative computation to be aborted should some logical condition not be fulfilled, or b) as a means to restore an error-free state after detection of an error due to a fault. In transactional applications, an atomic storage mechanism is needed even if the underlying system is completely fault-free. System fault-tolerance may or may not be based on the implementation of atomic storage as stable storage.

If the nodes in a distributed system possess stable storage, they can be "repaired" (either manually or automatically) after failure and re-inserted into the network with the assurance that the stored "stable data" is still in a state that is identical to that before node failure. Note however, that such a backward error recovery scheme inevitably leads to a time overhead (which, in the case of manual repair, could be quite long) that can lead to an unacceptable decrease in service availability. Redundancy of data and code is necessary if computation is to proceed while failed nodes are being repaired.

Alternatively, if nodes do not possess stable storage, then the design of the system-level fault tolerance techniques must be based on the assumption that all data stored locally is lost should the node fail. This means that if the node is re-inserted into the network, it must be assumed to have suffered total amnesia; the re-inserted node is thus equivalent to a totally new node. It can only be re-introduced into the system after its local storage has been re-initialized by copying information across the network from other (non-failed) nodes.

It is worthwhile considering the stable storage abstraction in the context of the two extreme assumptions made for hosts in sections 6.2.1 and 6.2.2:

a) *Fail-silent host with stable storage*. Seen from the communication system, such a node either delivers correctly-valued and timely messages or stops sending messages until repair is carried out. After repair, information may be retrieved from stable storage that is identical to that stored there before failure.

Since the host is fa
non-volatile storage

b) *Fail-uncontrolled* such a node may d
written to stable
information retriev
consistent state bef

Since the host ma
volatility of storage
mechanisms are n
failure [Banâtre et a

In conclusion, although st
computation, see section 6.
distributed systems, replicatior
be structured as transactions o
error recovery. However, the
initialization of repaired nodes

6.3. Models of Distri

Before presenting the various p
section §6.4, this section introc
a *distributed* computation.

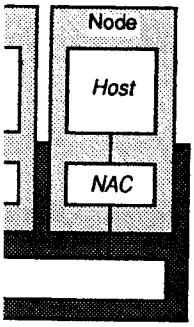
6.3.1. Software Compo

We define a *software compon*
and data encapsulation that, i
encapsulated by a software c
active logical entities that may
when several software compc
common memory. A distribut
multiple software components.

A software component ma
output ports and receive mes
output port and a single input p

The concept of a run-time
overloaded term "object"; it is
specific notions such as sing
[Almes et al. 1985], CONIC
Emerald [Black et al. 1987], A

Since software component
and allocated over the nodes o
the basic units by which compu
components is discussed in sec



munication system

:h checkpoints are stored must
survive power outages, such a
is typically implemented using
: *memories* have also been

istinguish between these two
putation, i.e., a) as a means in
aborted should some logical
ree state after detection of an
: mechanism is needed even if
ance may or may not be based

hey can be "repaired" (either
:twork with the assurance that
at before node failure. Note
)ly leads to a time overhead
can lead to an unacceptable
iecessary if computation is to

ie design of the system-level
all data stored locally is lost
into the network, it must be
us equivalent to a totally new
torage has been re-initialized
) nodes.
ie context of the two extreme

mmunication system, such a
messages or stops sending
nation may be retrieved from
failure.

Since the host is fail-silent, an implementation of atomic storage using redundant non-volatile storage media is a reasonable approximation of stable storage.

b) *Fail-uncontrolled host with stable storage.* Seen from the communication system, such a node may deliver incorrectly-valued or untimely messages. However, data written to stable storage is unaffected by host processor failures; therefore, information retrieved from stable storage after node repair is identical to *some* consistent state before failure.

Since the host may fail in an arbitrary fashion, atomicity of updates and non-volatility of storage are insufficient to ensure the stable storage abstraction. Further mechanisms are needed to protect against data corruption due to host processor failure [Banâtre et al. 1988].

In conclusion, although stable storage (together with a transactional model of distributed computation, see section 6.3.2) can be viewed as one possible basis for fault-tolerance in distributed systems, replication of code and data is more appropriate for applications that cannot be structured as transactions or which cannot support the time overheads induced by backward error recovery. However, the stable storage concept does provide an *option* for simplified re-initialization of repaired nodes (see §6.8).

6.3. Models of Distributed Computation

Before presenting the various possible approaches to distributed fault-tolerance by replication in section §6.4, this section introduces some elementary concepts for expressing what is meant by a *distributed* computation.

6.3.1. Software Components

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. 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 (only). Even when several software components co-reside on a single node, they do not explicitly share common memory. A distributed application can be viewed as any activity coordinated across multiple software components.

A software component may send messages to other such components thourgh one or more *output ports* and receive messages through one or more *input ports*. For simplicity, a single output port and a single input port will be assumed unless explicitly stated otherwise.

The concept of a run-time *software component* is introduced here to avoid using the very overloaded term "object"; it is used here to reason generally about the run-time view of more specific notions such as single-threaded or multiple-threaded "processes", Eden "Ejects" [Almes et al. 1985], CONIC "modules" [Loques and Kramer 1986], the "active objects" of Emerald [Black et al. 1987], ANSA [ANSA 1989] or Deltase "capsules" (see chapter 7), etc.

Since software components are the basic units into which a computation may be partitioned and allocated over the nodes of the distributed system, it is also convenient to consider them as the basic units by which computation can be replicated to tolerate faults. Replication of software components is discussed in section 6.4.

6.3.2. Transactions

Transactions were first introduced in the field of data-base systems (see [Bernstein et al. 1987] for a full bibliography) to ensure that updates to multiple items of data (or the states of multiple software components) are executed atomically:

- the refusal of an operation on one data item (e.g., a debit from an account is refused if the account has insufficient funds) may imply that related operations on other data items need to be cancelled (or "un-done");
- the potentiality for concurrent multiple access to shared items of data requires that operations on individual data items be scheduled to avoid mutual interference due to interleaving.

The mechanisms that are necessary to enable operations to be undone due to a purely logical reason can also be used to implement backward recovery following a failure or a conflict between operations that was not avoided by the control mechanisms. For this reason, the transaction concept has been extended beyond the database world and is often used as a basis for providing fault-tolerance in distributed systems by means of backward error recovery.

It has in fact been shown that the transaction mechanism is a dual of the *conversation* scheme that was introduced as a structuring mechanism for fault-tolerance in concurrent systems [Mancini and Shrivastava 1989, Randell 1975]. The aim of the conversation scheme is to control the "*domino effect*" that may be caused when, after detection of an error, communicating processes (software components) are rolled back and re-executed from some previously saved state (a checkpoint or recovery point). The domino effect may occur when the state of a process A is restored (rolled back) to some previous state that existed prior to a communication between A and some other process B. If A had sent a message to B before initiating roll-back then, when re-executing, it will send another message to B that, in the general case of non-idempotent messages, will cause B's state to become incorrect[3]. Similarly, if A had received a message from B, then when re-executing, it would require B to re-send the message it had already sent. In either case, B would also have to be rolled back. This could require a further roll-back of A if A and B had interacted after B's last checkpoint yet before that of A.

Conversations provide a means for restricting this domino effect. Once a process has entered a conversation, it is not allowed to communicate with processes not in the same conversation. If each process takes a checkpoint on entering a conversation then the roll-back of any process in the conversation causes the roll-back of only those processes in the same conversation. The mechanisms necessary for controlling the entering and leaving of conversations by processes are analogous to those used by transactions for locking and unlocking data items [Mancini and Shrivastava 1989].

6.4. Replicated Software Components

Replication of data and/or 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 and can thus be thought of as a means for providing fault-tolerance, such techniques used alone do not allow distributed system architectures to achieve better dependability than a non-distributed system. In fact, if a computation is spread over multiple nodes without any form of replication, distribution can only lead to a *decrease* in

[3] It is however possible to envisage the tolerance of repeated input messages by using sequence numbers.

dependability since the compu
operational.

The basic unit of replicatic
A *replicated software comp*
representation on two or more r
software component (even thou
given instant in time). Unless s
logical entity as a whole (i.e., th

There are two issues to repl

- *inter-replica coordin*
  order to process erro
  group is a single (fau
- *group membership*
  group of replicas ar
  failures and repairs?

For clarity, we shall first c
that software components have
membership is restricted to that
creation of groups and the mana
section §6.8.

The degree of replication o
degree of criticality of the comp
to an existing group (to replac
varying size, even though the d
i.e., no replication, for non-critic

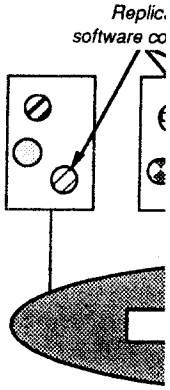


Fig.

Three basic techniques for re
of replica synchronization:

- *active replication* is
  concurrently so that th
  of faults, outputs can b

dependability since the computation may only proceed if each and every node involved is operational.

The basic unit of replication considered here is that of a software component (cf. §6.3.1). A *replicated software component* is defined as a software component that possesses a representation on two or more nodes. Each representation will be referred to as a *replica* of the software component (even though the actual representations may in practice be different at a given instant in time). Unless stated otherwise, the term *software component* will refer to the logical entity as a whole (i.e., the *group* of replicas).

There are two issues to replication of software components:

- *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 that the group is a single (fault-free) software component?
- *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?

For clarity, we shall first concentrate on the replica coordination issue: we shall suppose that software components have been instantiated as groups and that management of group membership is restricted to that of replicas *leaving* the group as a consequence of failure. The creation of groups and the management of replicas joining existing groups will be considered in section §6.8.

The degree of replication of software components in the system depends primarily on the degree of criticality of the component but also on how easy (and fast) it is to add new members to an existing group (to replace failed replicas). In general, it is wise to envisage groups of varying size, even though the degree of replication may often be limited to 2 or 3 (or even 1, i.e., no replication, for non-critical components) (see figure 6).

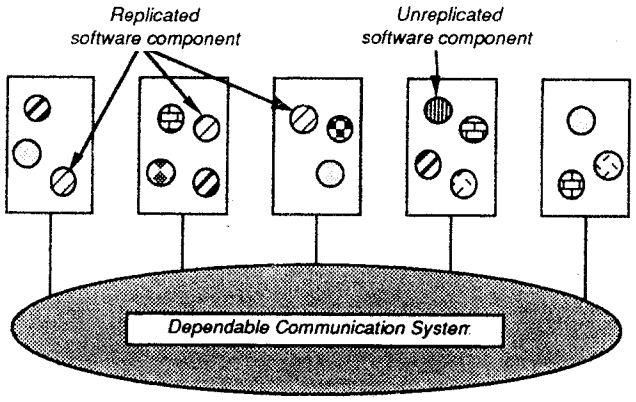


Fig. 6 - Replicated Software Components

Three basic techniques for replicated computation can be identified according to the degree of replica synchronization:

- *active replication* is a technique in which all replicas process all input messages concurrently so that their internal states are closely synchronized — in the absence of faults, outputs can be taken from any replica,

- *passive replication* is a technique in which only one of the replicas (the *primary* replica) processes the input messages and provides output messages — in the absence of faults, the other replicas (the *standby* replicas) do not process input messages and do not produce output messages; their internal states are however regularly updated by means of checkpoints from the primary replica,
- *semi-active replication* can be viewed as a hybrid of both active and passive replication; only 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, where appropriate, by means of notifications or "mini-checkpoints" from the leader replica.

These three different replication techniques will be described in the following sections; the remainder of this section is devoted to concepts that are common to all replica coordination techniques.

6.4.1. Replica Determinism and Replica Group Determinism

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, each 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.

Replica determinism is difficult to ensure in a truly heterogeneous environment. For each software component, it is necessary to restrict the locations of replicas to a sub-set of nodes that, if non-faulty, guarantee that each replica is deterministic. Even in a homogeneous sub-set of nodes, there remain potential causes for non-determinism, e.g., preemption, use of non-deterministic language constructs, decisions based on site-specific information, etc. If replicas are not deterministic, then replica group determinism can only be achieved by negotiation between replicas [Tully and Shrivastava 1990]. Alternatively, the potentiality for replica non-determinism can be removed by adopting a restrictive model of computation based on *state machines* [Schneider 1990].

If replica determinism cannot be ensured then it is necessary to impose strong assumptions on allowable replica failure modes and to restrict the choice of possible mechanisms for error processing.

6.4.2. 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 behaviour 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 early or late, omit to send some messages, send messages with incorrect content or send extra or "impromptu" messages (see 6.4.2.3 below).

However, it is possible to refine these assumptions and at the same time use a finer failure granularity than that of a complete host [Cristian 1991]. 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 an inappropriately-dimensioned system configuration, i.e., a *configuration fault*), then although the host hardware may be completely fault-free, buffer overflow may

cause replicas on that node to
failures [Cristian et al. 1985]. S
late-timing or *performance* fa
design fault, then all replicas c
value failures. Certain replicati
faults, are capable of tolerating
if such faults manifest themsel
reasonable for host configuratic
since, by definition, all replicas
However, it has been observec
different replicas due to slight c
their execution. In [Gray 198(
away when you look at them".

Consideration of the vario
the value domain can lead to tl
severities (e.g., see [Powell 1
software component replicas e
few categories of failure mode
failures, only omission failure:
silent, fail-omissive and *fail-tar*
fail-restrained replicas since, t
value; replicas that can fail arbit

It is tempting to consider tl
omitting to send some message
the other possible consequence
a reply because it "lost" a re
probably also cause the state
failure in the value domain. F
intermediate between sudden si
the mechanisms necessary to
complex than those necessary t
assumptions are provably less
can only be higher.

Note also that whereas rep
when executing on fail-silent h
assumed to be fail-uncontrolled

6.4.2.1. Time-Domain Erro

fail-uncontrolled, detection of
errors are particularly difficult
that been proposed to simplify
synchronized [Lamport and M
direct use unless the local sched
time reference to determine the
1984]. This cannot be assum
heterogeneous hosts cannot be
the only viable basis for dealin
outs.

Nevertheless, the use of ti
replica execution times and cc

ne of the replicas (the *primary*
les output messages — in the
replicas) do not process input
eir internal states are however
primary replica,

rid of both active and passive
ca) processes all input messages
f faults, the other replicas (the
; their internal state is updated
where appropriate, by means of
eplica.
ed in the following sections; the
mon to all replica coordination

rminism

listic if, in the absence of faults,
nd consuming the same ordered
ssages.
, given the same initial state for
in the group produces the same
me identical input messages in
1 for replica group determinism.
geneous environment. For each
f replicas to a sub-set of nodes
Even in a homogeneous sub-set
, e.g., preemption, use of non-
ific information, etc. If replicas
ly be achieved by negotiation
he potentiality for replica non-
of computation based on *state*

y to impose strong assumptions
possible mechanisms for error

e replicas executing on a given
our of that host. If a fail-silent
appear to have failed silently or
ien any or all replicas that were
t to send some messages, send
nessages (see 6.4.2.3 below).
ie same time use a finer failure
stance, an incident in a node's
lternatively, if a host becomes
figuration, i.e., a *configuration*
ult-free, buffer overflow may

cause replicas on that node to fail by omitting to respond; such failures are called *omission failures* [Cristian et al. 1985]. Similarly, replicas may respond too late; these failures are called *late-timing* or *performance* failures. Of course, if a software component contains a residual *design fault*, then all replicas could fail in a quite arbitrary fashion, including both *timing* and *value* failures. Certain replication techniques, although primarily designed to tolerate hardware faults, are capable of tolerating host configuration faults and software component design faults if such faults manifest themselves *independently* on different hosts. Although this may seem reasonable for host configuration faults, this is less evident in the case of software design faults since, by definition, all replicas of an incorrect software component will be identically incorrect. However, it has been observed that some design faults manifest themselves independently in different replicas due to slight differences in the local environment of the replicas at the time of their execution. In [Gray 1986], such faults are referred to as "Heisenbugs" since they "go away when you look at them".

Consideration of the various ways by which components can fail in the time domain and the value domain can lead to the definition of a wide spectrum of failure modes with different severities (e.g., see [Powell 1991]). However, for the particular case of interest here, i.e., software component replicas executing in a distributed message-passing environment, just a few categories of failure modes are sufficient. Replicas that are assumed to suffer only crash failures, only omission failures or only performance failures can be termed respectively *fail-silent*, *fail-omissive* and *fail-tardy* replicas. Replicas that fail thus can also 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.

It is tempting to consider the possible phenomena that could cause a replica to fail *only* by omitting to send some messages or by sending some messages too late and then to reason about the other possible consequences of those phenomena. For instance, a replica might fail to send a reply because it "lost" a request message. In this case, the lost request message would probably also cause the state of the replica to be erroneous — thus leading ultimately to a failure in the value domain. However, the concept of considering "abstract" failure modes intermediate between sudden silence and totally uncontrolled behaviour is a useful one because the mechanisms necessary to tolerate such restrained forms of failure are not much more complex than those necessary to tolerate total silence; yet, since the corresponding failure mode assumptions are provably less restrictive than total silence, the resulting assumption coverage can only be higher.

Note also 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.

6.4.2.1. Time-Domain Errors. Whether replicas are considered to be fail-restrained or fail-uncontrolled, detection of timing errors is a fundamental part of error processing. Such errors are particularly difficult to process in an *open* distributed environment. One technique that been proposed to simplify processing of timing errors is to keep local clocks approximately synchronized [Lamport and Melliar-Smith 1985, Schneider 1986]. However, this is of no direct use unless the local scheduling of replicas at each node explicitly uses the resulting global time reference to determine the instants at which messages should be sent or received [Lamport 1984]. This cannot be assumed an *open* distributed system. Scheduling techniques on heterogeneous hosts cannot be assumed to be the same, let alone time-dependent. In practice, the only viable basis for dealing with time-domain errors in an open system is the use of time-outs.

Nevertheless, the use of time-outs does not avoid the requirement for upper bounds on replica execution times and communication delays (if such upper bounds did not exist, it is

impossible to distinguish between a component that has stopped or is infinitely slow [Fischer et al. 1985]). Since it is particularly difficult to estimate execution times and communication delays, especially in complex dynamically-evolving systems, one is faced with the inevitable problem of dimensioning time-outs sufficiently high so as to achieve an acceptable rate of late-timing failures yet sufficiently low to allow speedy detection.

However, it is important to underline that, although from an error-processing viewpoint, expiration of a time-out can only be attributed to a replica late-timing failure, this does not necessarily mean that the sending node as a whole is faulty and will be irrevocably removed from the system. Properly-designed error-processing protocols will mask such errors but report them to the administration system's fault-treatment facility. The latter can first try to alleviate the incriminated node by moving (some of) its software component replicas to other nodes (load-balancing) and will only passivate the incriminated node if it diagnoses that the number of such reported errors has exceeded a given threshold (that could be dynamically adjusted to account for varying system load).

6.4.2.2. Value-Domain Errors. Value-domain errors only need to be considered when replicas are assumed to be fail-uncontrolled. The only way to detect value errors is to compare equivalent output messages from different replicas. This requires of course *active* replicas that satisfy the replica group determinism condition (cf. §6.4.1). To mask t value errors, then there must be at least $2t+1$ replicas in the replica group. The comparison itself can be carried out either on complete messages or, for performance reasons, on hash-coded representations of messages — called hereafter, *signatures*. Note that in the latter case, it must be assumed that two different messages do not produce the same signatures; there is thus an attendant assumption coverage to take into account when evaluating the achieved dependability.

6.4.2.3. Impromptu Errors. An impromptu error occurs when a replica spontaneously produces an unspecified message. Impromptu errors may be detected in either the value domain or in the time domain. For example, if a time window has been opened in which messages are expected, then any impromptu message that occurs *inside* this time interval will appear to be correct in the time domain. However, the impromptu message will be detected as value-erroneous if compared with the values of messages from other replicas as in §6.4.2.2 above. If an impromptu message is received *outside* any expected-message time window, then the message will be detected as timing-erroneous. Since impromptu errors affect both the time and value domains, it is necessary that there be at least $2t+1$ replicas in the replica group to mask t errors.

6.4.3. Replication Domains

A software component *replication domain* is defined to be the set of nodes on which replicas of that software component are allowed to reside.

Replicas of a given software component can, of course, only be executed on nodes that possess the necessary resources. However, there are often other reasons for restricting a component's replication domain. For example, it may be further restricted to those nodes that not only possess the necessary resources but which guarantee replica determinism (cf. §6.4.1).

Another factor affecting the definition of a component's replication domain could be that the chosen replication technique relies on a particular replica failure mode assumption. If the dependability requirements of the application dictate that the coverage of that assumption be greater than some minimum value, then replicas may have to be confined to nodes with features that support that assumption. For example, if replicas must be fail-silent with a high degree of

confidence, then replicas should
checking hardware.

Finally, equivalence of execu
of a replication domain. Even th
when executed on a given set of r
the detection of timing errors. It s
anyway force all replicas to proce

6.4.4. Replica Coordinatio

It is desirable to be able to progr
that it may be instantiated as a gr
on the logical problem or functio
without having to deal with the in

It is therefore useful to separ
by one or more standard system e
having one or more local "replic
acting on its behalf (figure 7).

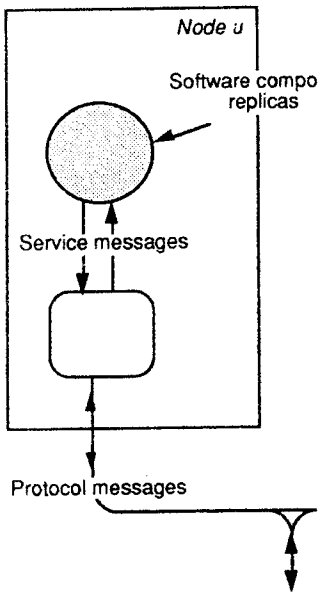


Fig. 7 - Software C

Rep_entities are assumed tc
uncontrolled; this separation of
error-processing protocols for tl
executed on fail-silent hardwa
associated NAC (cf. figure 5).

Replicas and their correspor
messages corresponding to the u

confidence, then replicas should be restricted to hosts that have been implemented using self-checking hardware.

Finally, equivalence of execution speed is yet another criterion entering into the definition of a replication domain. Even though replicas may have identical value domain behaviours when executed on a given set of nodes, a large dispersion in execution speed would complicate the detection of timing errors. It should also be noted that inter-replicate synchronization would anyway force all replicas to proceed, on average, at the speed of the slowest replica.

6.4.4. Replica Coordination Entities

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 (figure 7).

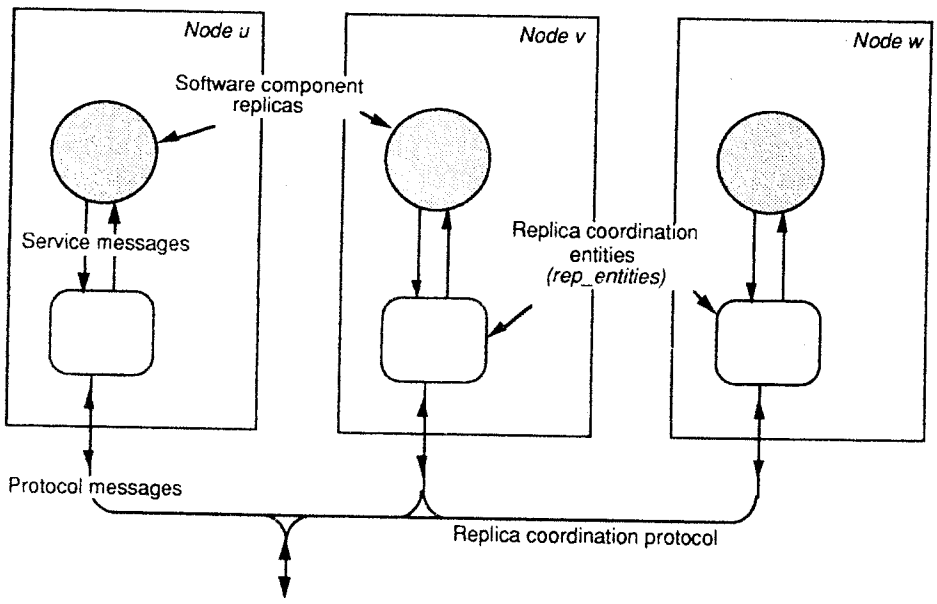


Fig. 7 - Software Component Replicas and Replica Coordination Entities

Rep_entities are assumed to be fail-restrained even if their corresponding replicas are fail-uncontrolled; this separation of function allows a considerable simplification of the distributed error-processing protocols for the fail-uncontrolled case. This means that rep_entities must be executed on fail-silent hardware — either on the host, if the latter is fail-silent, or on the associated NAC (cf. figure 5).

Replicas and their corresponding rep_entities exchange two sorts of *service messages*: *data messages* corresponding to the user-programmed data exchange between software components,

and *handshake messages* for the purposes of flow-control. Messages exchanged between rep_entities are called (replica coordination) *protocol messages*, which may or may not contain embedded data messages.

6.5. Active Replication

Active replication is a technique whereby a software component is installed on multiple nodes such that at all times each replica in the group may, in the absence of faults, provide a service that is equivalent to any of the other replicas in the group. 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* (cf. §6.4.1): when starting from identical initial states and processing identical sets of input messages, each replica produces identical output messages in the same order.

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

The *replica group determinism* condition is more subtle: it does not necessarily imply that messages be received and processed by replicas in an identical order — one can imagine scenarios by which replicas could process messages in different orders yet still remain consistent and produce output messages in the same order. However, this would require the semantics of input messages to be taken into account when deciding what would be admissible orders. To avoid the potential complexity of such an approach, the active replication technique proposed here requires replica groups to be made deterministic by:

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

Two different error-processing philosophies may be followed according to the underlying failure mode assumption:

a) if replicas are fail-restrained, then any output sent by any replica of the group can be assumed to be of correct value; it is therefore possible to choose any of the outputs and discard the others,

b) if replicas are fail-uncontrolled, then the set of outputs must be considered as a whole so that value errors and unexpected outputs may be masked.

In case a), since the output from any fail-restrained replica can only be a correct output, it is possible to relax the output consistency condition and optimize the use of individual replicas by, for example, only sending requests that do not modify the state of the component to just one replica of the group (e.g., the "nearest"). If no response is forthcoming, the request can be re-submitted to another replica. In database terms, such requests are called read-requests. Of course, any inputs to the software component that *do* modify the internal state (i.e., "write" requests) must be delivered to all replicas. Such "read optimization" (see [Bernstein et al. 1987]) 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 consistency of the replicas.

The management of replicas in thi
approach to distributed computatio
since:

- in the general "software
  cannot necessarily be
  software component
  periodically transmit so
- similar mechanisms ca
  restrained and fail-unco

The error processing protocol
presented in the next two sub-sect
Note however that these separatic
single *parameterized* protocol ca
modes of operation.

6.5.1. Fail-Restrained Acti

As mentioned earlier, the philoso
logical whole. Any messages sent
the same order so that, if the repl
faults, each replica will produce t
protocol, carried out by the rep
themselves when the replicas atte

Since replicas are assumed t
assumption, of correct value. Th
activity is reduced to a simple arb
that only one copy is delivered t
implemented in two ways:

a) for each output mess
   executed in order to d

b) all the rep_entities
   corresponding destina

For long messages, the firs
activity. Each replica forwards al
the various rep_entities must m
destination(s). The output me
multicasting service that ensures
other rep_entities on non-failed
message selection protocol may

6.5.1.1. Competitive Propa

receives a data message from it
sent by another rep_entity. If n
group of rep_entities (includi
message before any others, it
discarded. As it stands, this si
omission errors and late-timing
errors must not only be mask
mechanism. If a rep_entity rece

rol. Messages exchanged between
ages, which may or may not contain

onent is installed on multiple nodes
absence of faults, provide a service
Quasi-instantaneous recovery from
ll correct replicas produce the same
ports — this is referred to as the

; delivered to correct replicas are

ting from identical initial states and
h replica produces identical output

unication protocol used to transmit
ust be some sort of *reliable group*
ect recipients.
: it does not necessarily imply that
entical order — one can imagine
different orders yet still remain
. However, this would require the
deciding what would be admissible
ch, the active replication technique
ic by:
essages in an identical order, i.e., a

; software components as *state*

lowed according to the underlying

by any replica of the group can be
ssible to choose any of the outputs

outputs must be considered as a
; may be masked.
a can only be a correct output, it is
e the use of individual replicas by,
state of the component to just one
rthcoming, the request can be re-
ests are called read-requests. Of
ify the internal state (i.e., "write"
timization" (see [Bernstein et al.
e traffic over the network at the
ensure consistency of the replicas.

The management of replicas in this way can be carried out in the framework of a transactional approach to distributed computation. However, we prefer not to consider such an optimization, since:

- in the general "software component" paradigm, outputs from a software component cannot necessarily be paired with a corresponding "read-request" input (e.g., a software component need not have *any* inputs but could be programmed to periodically transmit some internally computed value),
- similar mechanisms can be used for managing active replicas with both the fail-restrained and fail-uncontrolled assumptions.

The error processing protocols for fail-restrained and fail-uncontrolled active replicas are presented in the next two sub-sections. In both cases, two "modes" of operation are described. Note however that these separations in explanation are for clarity only and that, in practice, a single *parameterized* protocol can be used to cover both failure mode assumptions and both modes of operation.

6.5.1. Fail-Restrained Active Replicas

As mentioned earlier, the philosophy followed here is to treat the group of active replicas as a logical whole. Any messages sent to the software component must be delivered to all replicas in the same order so that, if the replica determinism condition is fulfilled then, in the absence of faults, each replica will produce the same output messages in the same order. The inter-replica protocol, carried out by the rep_entities, must mask and detect errors that may manifest themselves when the replicas attempt to send or to receive messages.

Since replicas are assumed to be fail-restrained, any message sent by such a replica is, by assumption, of correct value. Thus, from the output message viewpoint, the error-processing activity is reduced to a simple arbitration between the multiple copies of the output messages so that only one copy is delivered to the intended destination(s). This arbitration activity can be implemented in two ways:

a) for each output message, an arbitration protocol between the local rep_entities is executed in order to decide which of them will send the message,

b) all the rep_entities forward every output message; the rep_entities of the corresponding destinations discard all but one of the messages that they receive.

For long messages, the first approach is obviously less demanding in communication activity. Each replica forwards all its output messages to its local rep_entity. For each message, the various rep_entities must mutually decide which of them is to forward the message to its destination(s). The output message selection protocol is built on top of an underlying multicasting service that ensures that messages multicasted by any rep_entity are received by all other rep_entities on non-failed nodes (including the sender) and in the same order. The output message selection protocol may operate in either a competitive or a cyclic mode.

6.5.1.1. Competitive Propagate Mode. In the *competitive mode*, when a rep_entity receives a data message from its replica, it first verifies that the message has not already been sent by another rep_entity. If not, the rep_entity multicasts a *claim* protocol message to the group of rep_entities (including itself). Therefore, if a rep_entity receives its own *claim* message before any others, it may forward the data message; if not, the data message is discarded. As it stands, this simple competitive message selection protocol allows silence, omission errors and late-timing errors to be masked. However, to initiate fault treatment, such errors must not only be masked, they must be detected. This is achieved by a time-out mechanism. If a rep_entity receives a *claim* protocol message corresponding to a data message

that it has not yet received from its local replica, a timer is armed. Replica silence, omission errors and late-timing errors are declared if this timer should expire before the local replica has produced the corresponding data message.

The competitive mode gives preference to the fastest replica of the group and can allow other replicas to lag further and further behind. The amount of desynchronization may be implicitly limited by controlling the flow of information to the replicas (when the input queue to the slowest replica is full, flow control will limit the rate of delivery of input messages to the rate at which the slowest replica dequeues input messages). Alternatively, the desynchronization may be explicitly limited if the rep_entities periodically carry out a "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 in order to detect and recover from replica silence, omission errors and late-timing errors.

In practice, it has been observed that re-synchronization need not be carried out frequently; the fastest replica automatically 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.

6.5.1.2. Round-Robin Propagate Mode. In the *cyclic or round-robin mode*, the rep_entities are configured in a logical ring with an associated token. When a rep_entity receives a data message from its replica that has not already been sent by another rep_entity then, if it possesses the token, the message is forwarded to its destination(s) and the token is transferred to the rep_entity's successor in the logical ring. Rep_entities not possessing the token must store messages until they receive the token; then, all messages already sent (identified by a *last_message* identifier contained in the token) are discarded. Since the round-robin mode treats all replicas "fairly", no further inter-replica synchronization mechanism is necessary.

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 (the rep_entity also reverts to the competitive mode should it receive a *claim* message from a peer rep_entity).

Omission and late-timing errors can be detected by monitoring the time interval between the receipt of a (multicasted) protocol data message by a peer rep_entity and receipt of the local service data message if the former should occur before the latter.

6.5.2. Fail-Uncontrolled Active Replicas

When replicas are no longer assumed to be fail-restrained, the error-processing activity must take account of not only silence, omission errors and late-timing errors but also *value* errors, *early-timing* errors and *impromptu* errors (cf. §6.4.2.3).

To mask t early-timing errors, $2t+1$ replicas are necessary. Even if t replicas of the group send a data message to their local rep_entities at approximately the same time, the latter cannot know immediately whether these messages are the first t messages of a set of $2t+1$ or if they are messages being sent too early or indeed impromptu messages. Each rep_entity must therefore arm a timer and await notification that equivalent messages have been sent within a specified time interval by at least t other replicas.

To process value errors in data messages, the rep_entities must cross-check each data message sent by the local replica with equivalent data messages sent by remote replicas. This cross-checking is referred to here as *message validation*. To mask t value errors, equivalent data messages from $t+1$ replicas must be compared and found to agree before propagating a validated message to its destination(s); since there can be t messages with erroneous values, a

total of $2t+1$ messages is necess
propagation-*before*-validation, is

The message validation me
executed by rep_entities. As soo
supposed to be t errors, it can be
is error-free. It is important to
message must not alter the messa
can be assumed that any message

To prevent faults in the rema
ensure that all replicas are regula
whose messages are compared o
The latter approach is simpler to
messages are compared after hav

As for the fail-restrained rep
operate in either a competitive o

6.5.2.1. Competitive Valid
when a rep_entity dequeues a d
has not already been sent by a
protocol message to the group o
signature of the data message
communication service ensures
claim messages from the group
rep_entity compares the signatu
found to be identical; this point i

The unique rep_entity that r
claim message, forwards the loc
its peers that this has been done
the *ack* message are sent togethe

Early-timing and late-timi
masked by monitoring the tim
validation point or vice versa.
reached within a specified inter
detected if the validation point i
the local data message is not rec

An impromptu error is de
within the time interval (and of
as a timing error if no timing w

As in the case of fail-restr
fastest replica of the group so
time-limited in order to detect t

6.5.2.2. Round-Robin Va
robin mode, the group of rep_
When a rep_entity receives a
rep_entity then, if it possesses

4 Note also that it is possible to e
group configured in this way ef

total of $2t+1$ messages is necessary (cf. §6.4.2.2) (an alternative approach based on message propagation-*before*-validation, is discussed in annexe G)[4].

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 so that it can be assumed that any message that they do send is a correctly-valued message.

To prevent faults in the remaining t replicas from remaining dormant, it is also necessary to ensure that all replicas are regularly activated and checked 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.

As for the fail-restrained replica case, the message-sending error-processing protocol may operate in either a competitive or round-robin mode.

6.5.2.1. Competitive Validate-before-Propagate Mode. In the *competitive mode*, when a rep_entity dequeues a data message from its replica, it first verifies that this message has not already been sent by another rep_entity. If not, the rep_entity 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 group communication 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 $t+1$ 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, forwards the locally-received data message to its destination(s) and indicates to its peers that this has been done by means of an *ack* message (the forwarded data message and the *ack* message are sent together in a single atomic operation).

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. An early-timing error is detected if the validation point is not reached within a specified interval after receipt of the local data message. A late-timing error is detected if the validation point is first reached by means of messages from the other replicas and the local data message is not received within the specified time interval.

An impromptu error is detected as a value error if the impromptu message is received *within* the time interval (and of course if its signature is different to that of t other signatures) or as a timing error if no timing window has been opened.

As in the case of fail-restrained replicas, the competitive protocol gives preference to the fastest replica of the group so resynchronization is necessary. The resynchronization is again time-limited in order to detect timing errors.

6.5.2.2. Round-Robin Validate-before-Propagate Mode. In the *cyclic or round-robin mode*, the group of rep_entities is configured as a logical ring with an associated token. When a rep_entity receives a local data message that has not already been sent by another rep_entity then, if it possesses the token, the signature of the local data message is sent to all

[4] Note also that it is possible to ensure *detection* of t value or timing errors with just $t+1$ replicates; a replica group configured in this way effectively constitutes a fail-silent software component.

members of the group and the token is forwarded to the rep_entity's successor in the ring by means of a *turn* protocol message. The token circulates round the ring until it reaches a rep_entity for which the signature of the local data message is identical to that contained in t previous *turn* messages. This rep_entity has thus reached its validation point; it forwards the corresponding data message to its destination(s) and informs its peers that this has been done by means of an *ack* message (containing the majority signature). The other replicas reach their validation point after having received the *ack* message and a concording local data message.

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 (the rep_entity also reverts to the competitive mode should it receive a *claim* message from a peer rep_entity). An early timing error is declared if, after reverting to the competitive mode, the validation point is not reached within a further specified interval. Late timing errors are treated in exactly the same way as in the competitive protocol — by monitoring the time interval between occurrence of the validation point and receipt of the local data message if the former should occur before the latter.

6.5.3. Message-Reception Error Processing

Any message sent to a group of active replicas must, in the absence of faults, be delivered to all replicas to ensure that all replicas can produce the same outputs. Therefore, all messages sent to a software component must be multicasted *atomically* to the corresponding rep_entities who must then forward these messages to their local replicas.

However, to ensure end-to-end flow control, replicas may refuse to accept data messages from their rep_entities if their receive buffers are full — replicas thus have to send explicit handshake messages to their local rep_entities to indicate their willingness to accept the incoming data message. Replicas must unanimously accept all data messages to ensure that they remain synchronized. The logic of atomic multicasting with end-to-end flow control would normally dictate that, if any replica cannot accept an incoming data message, then all replicas must refuse the message.

Now, seen from the rep_entities, the flow-control handshake messages are just another sort of service message (cf. figure 7) that a *faulty* replica could omit to send or send too late. We are thus faced with contradictory requirements in the two following situations:

a) if a non-faulty replica cannot accept an incoming data message, then the flow of data messages to all replicas of the group must cease;

b) if a replica fails by omitting or delaying to send handshake messages: data message flow to the non-faulty replicas should continue.

The solution to this contradiction is to limit the time for which situation (a) can persist before declaring that the replica in question is faulty.

Whenever a data message is to be forwarded to the local replica, the rep_entity sets a timer to await the corresponding handshake message and thus limits the time for which a local transient overflow condition may persist. If time-out occurs then the rep_entity requests its peer rep_entities to send back their local status. The way in which these replies are interpreted depends on the replica failure assumption. If replicas are assumed to be fail-restrained, then all handshake messages sent are correct handshake messages. Therefore, if *any* remote rep_entity has received the handshake message, then the message-refusal situation is judged abnormal and the local replica is declared as failed. If replicas are fail-uncontrolled, then the possibility of an erroneously-produced handshake message must be envisaged. Therefore, when a message-refusal condition exists, it must be assumed to be a normal overflow situation until a majority of remote rep_entities have received their local handshake messages.

6.5.4. Error-Reporting

The inter-replica protocol for acti
number of replicas in a group:

- for fail-restrained repli
  by a group of $t+1$ active
- for fail-uncontrolled re
  can be masked by a gro

The inter-replica protocol alsc
by its corresponding rep_entity. Tl

- the rep_entity can abor
  group (by a *leave* proto
- the error can be reporte
  the group until it is tol
  soft faults to be toler
  component is re-initi
  existent.

6.5.5. Performance Consid

A major advantage of the active
error is quasi-instantanous sir
synchronization. Of course, the
must be increased by at least the

At first sight, the competiti
mode in that it allows message p
However, this must be weighed a
to the systematic sending of *clain*

6.6. Passive Replicatio

When replicas are executed onl
replication technique that econon
only when they are needed to e
consists of a group of replicas ir
input messages and provides al
replica produces output messag
(i.e., replicas that only fail by
messages late).

The other replicas in the gro
the software component togethe
which execution can be resum
standby replicas must be regul
checkpointing. Standby replica
processing other than house-k
previously-passive back-up repli

5 Whence the alternative name for

6.5.4. Error-Reporting

The inter-replica protocol for active replicas masks errors resulting from failures of a certain number of replicas in a group:

- for fail-restrained replicas: t crash, omission or performance errors can be masked by a group of $t+1$ active replicas;
- for fail-uncontrolled replicas: t arbitrary errors (value, timing or impromptu errors) can be masked by a group of $2t+1$ active replicas.

The inter-replica protocol also ensures that any error caused by a replica is detected locally by its corresponding rep_entity. The latter can then follow two different strategies:

- the rep_entity can abort itself immediately after informing the other members of the group (by a *leave* protocol message) and reporting to the fault treatment facility,
- the error can be reported to the fault treatment facility but the rep_entity remains in the group until it is told to remove itself by the fault treatment facility; this allows soft faults to be tolerated in the case where the internal state of the software component is re-initialized after each output message or, equivalently, is non-existent.

6.5.5. Performance Considerations

A major advantage of the active replication technique is that recovery from a detected replica error is quasi-instantanous since the all replicas in the set are maintained in close synchronization. Of course, the price that is paid for this is that the overall processing power must be increased by at least the degree of redundancy.

At first sight, the competitive mode has a performance advantage over the round-robin mode in that it allows message propagation at the earliest possible opportunity (see annexe G). However, this must be weighed against the higher protocol message traffic that is incurred due to the systematic sending of *claim* messages.

6.6. Passive Replication

When replicas are executed only by fail-silent hosts, it is possible to envisage an alternative replication technique that economises host processor utilization by activating redundant replicas only when they are needed to ensure recovery. A passively replicated software component consists of a group of replicas in which one replica, termed the *primary* replica, processes all input messages and provides all output messages. Since, in the absence of errors, only one replica produces output messages, this technique is only suitable for fail-restrained replicas (i.e., replicas that only fail by crashing, by omitting to send some messages or by sending messages late).

The other replicas in the group are *standby* replicas[5]; each consists of a copy of the code of the software component together with a copy of some previous state of the component from which execution can be resumed should the primary replica fail. The internal state of the standby replicas must be regularly updated by the primary replica: this operation is called *checkpointing*. Standby replicas are passive since, in the absence of faults, they carry out no processing other than house-keeping operations following reception of checkpoints. A previously-passive back-up replica attempting recovery is termed a *substitute* replica.

---

5 Whence the alternative name for passive replication — the *primary/standby* technique.

6.6.1. Checkpointing Strategies

Various strategies are possible for the taking of checkpoints. One technique is to implement *transactional checkpoints* whereby interactions between groups of software components are structured as transactions (cf. §6.3.2) and the primary replica of each software component involved in a transaction checkpoints its state to its back-up replica(s) only when changes to this state are committed (i.e., if and when the transaction terminates successfully).

In the absence of a transactional model of computation, recovery can be ensured on a component-by-component basis if checkpointing is organized in such a way as to prevent the domino effect (cf. §6.3.2). Checkpointing must be carried out in a way such that a substitute replica re-executing from the last checkpoint does not need to request re-sending of previously received input messages and avoids sending duplicate output messages.

To avoid having to request re-sending of previously received input messages, either the back-up replicas must maintain a queue of input messages identical to those received by the primary replica since the last checkpoint was taken or, each time the primary replica receives a message, a new checkpoint must be taken. The former approach has the advantage that checkpointing is less frequent and the communication overhead is thus usually lower, especially if the input message queues are created concurrently with the normal inter-component message flow, e.g., by means of a reliable group communication protocol exploiting broadcast channels.

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.

Periodic checkpointing is a strategy whereby the number of checkpoints is reduced by only taking them say, every n output messages [Borg et al. 1983]. 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 checkpointing requires that replicas be deterministic (cf. §6.4.1) 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. In the case of transactional and systematic checkpointing, this requirement for replica determinism and identical order of input messages is unnecessary since *any* execution based on any order of input messages that respects causality is a valid execution.

Although systematic checkpointing entails more overhead than either periodic or transactional checkpointing, its capacity to accommodate non-deterministic processing and its suitability for the implementation of independently fault-tolerant software components (unlike transactional checkpointing) are very important advantages. It is thus this technique that has been implemented in Delta-4.

In the systematic checkpointing technique, it is important that the transfer of checkpoints to the standby replicas and the sending of data messages by the primary replica to their destination(s) be carried out as a single atomic operation to avoid the domino effect. This could be done by using an atomic multicast protocol to send a combined data and checkpoint message to both the standby replicas and the designated data-message destination(s). However, since checkpoints may be quite large, this would put an unnecessary load on the latter who would have to unpack the checkpoint only to discard it. An alternative approach is illustrated on figure 8 which shows a passively replicated group of three replicas: the primary replica and two back-up replicas.

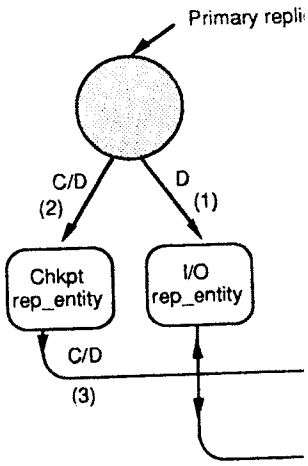


Fig.

In this scheme, there are tw
I/O rep_entity[6]. Whenever the
it also forwards the message to
be taken (2). The data message
checkpoint rep_entities of the s
I/O rep_entities and update the
thus in possession of the data
out a message-sending arbit
replication technique for fail
protocol is used in a "rendez
synchronized before the data
the primary I/O rep_entity
checkpointing is carried out, it
are ready to compete. Conseq
claim messages will be sent.
rendezvous will ensure that th

6.6.2. Taking Checkpo

A checkpoint consists of a sn
internal state is inevitably spe
representation of replicas. I
system information specific t
etc. Furthermore, if input m
include information as to w
since the last checkpoint w
queues.

6 For the purposes of the ex
processes.

One technique is to implement
ps of software components are
a of each software component
ica(s) only when changes to this
successfully).

recovery can be ensured on a
in such a way as to prevent the
in a way such that a substitute
equest re-sending of previously
essages.

ived input messages, either the
ntical to those received by the
e the primary replica receives a
proach has the advantage that
is thus usually lower, especially
ormal inter-component message
l exploiting broadcast channels.

y means of either systematic or

its whenever the primary replica
e., whenever a message is sent.
f an output message.

checkpoints is reduced by only
. During recovery, any output
gainst a log of previously sent
ge is found.

at replicas be deterministic (cf.
e order (as for the active replica
messages as those that were
of transactional and systematic
ntical order of input messages is
essages that respects causality is

head than either periodic or
deterministic processing and its
nt software components (unlike
t is thus this technique that has

at the transfer of checkpoints to
the primary replica to their
id the domino effect. This could
ed data and checkpoint message
destination(s). However, since
y load on the latter who would
approach is illustrated on figure
e primary replica and two back-

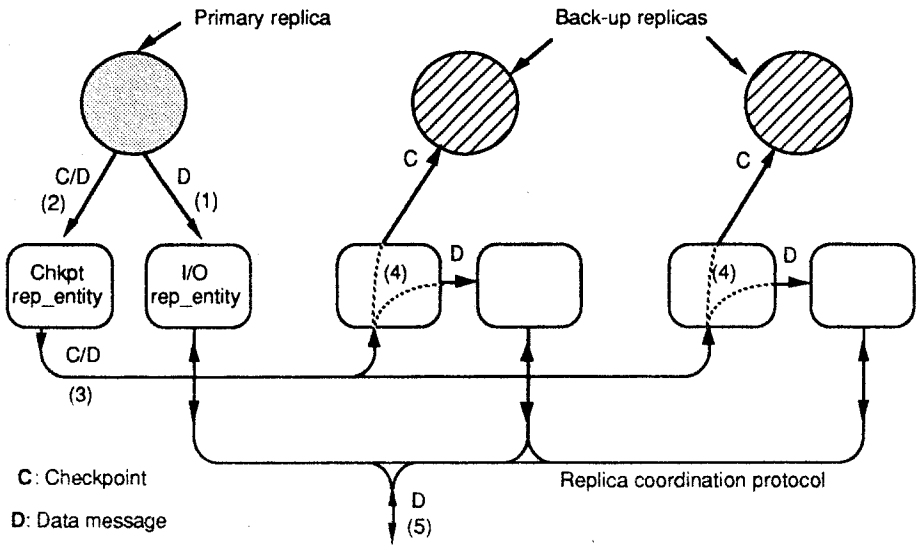


Fig. 8 - Systematic Checkpointing Technique

In this scheme, there are two rep_entities for each replica: a checkpointing rep_entity and an I/O rep_entity[6]. Whenever the primary replica forwards a data message to its I/O rep_entity (1), it also forwards the message to the local checkpoint rep_entity to indicate that a checkpoint must be taken (2). The data message and the checkpoint are atomically multicasted together (3) to the checkpoint rep_entities of the standby replicas. The latter forward the data message to their local I/O rep_entities and update the internal state of the standby replicas (4). All I/O rep_entities are thus in possession of the data message produced by the primary replica. They can thus carry out a message-sending arbitration protocol (5) in exactly the same way as in an active replication technique for fail-restrained replicas (cf. §6.5.1). If the competitive propagate protocol is used in a "rendezvous" mode, then it is ensured that all I/O rep_entities will be synchronized before the data message is sent to its destinations. Since, in the absence of faults, the primary I/O rep_entity can enter the arbitration competition before or while the checkpointing is carried out, its *claim* message will be sent long before the other I/O rep_entities are ready to compete. Consequently, the former will win the competition outright and no other *claim* messages will be sent. If the primary I/O rep_entity should fail, the time-limits on the rendezvous will ensure that the data message is sent by one of the other I/O rep_entities.

6.6.2. Taking Checkpoints

A checkpoint consists of a snap-shot of the "internal state" of the primary replica. Access to this internal state is inevitably specific to host type, local execution support system and the run-time representation of replicas. It must consist of the data space of the replica together with all system information specific to the replica: processor registers, stack pointer, status information, etc. Furthermore, if input messages are multicasted to all I/O rep_entities, checkpoints muwt include information as to which input messages have been processed by the primary replica since the last checkpoint was taken and should thus be discarded from the back-up input queues.

---

6 For the purposes of the explanation, it can be imagined that replicates and rep_entities are separate processes.

Some of the internal state of the primary replica concerns local resources that are significant only in at the site of the primary, e.g., local file descriptors, context identifiers, etc. Such local descriptors have no meaning on remote sites; the primary checkpointing rep_entity must use them to build a global context that is included in the transferred checkpoint. This global context must then be used by the remote checkpointing rep_entities to establish equivalent local resources if recovery is required.

Files can be managed in two different ways. First, files could be considered as being part of the internal state of a replica group. In this case, the content of the files opened by the primary replica must be included within the checkpoint, together with global descriptors for global-local context mapping as mentioned above. Alternatively, the use of a separate, fault-tolerant *global* file system may be enforced on application programmer's (see section 8.2.4.2).

A UNIX interpretation of the checkpointing mechanism is given in [Speirs and Barrett 1989].

6.6.3. Error Detection and Recovery

The passive replica fault-tolerance technique is primarily used only for the tolerance of hardware faults (of fail-silent hosts). With this assumption, a fault causes the failure (by silence) of *all* entities executing on the faulty host. Therefore, error detection can be reduced to the detection of silence of any entity executing on a given host. A set of system entities can implement a node *group membership protocol* based on the exchange of "heartbeat" or "I'm alive" messages. Alternatively, an equivalent node group membership service can be offered as a facility of the communication system (see section 6.9.2). With such a node group membership service, each checkpoint rep_entity of a passive replica group can be informed of the failure of any of the hosts supporting a replica of the group. If the host supporting the primary replica fails then a substitute replica must be selected to carry out recovery from the last checkpoint. This selection may be carried out by a dynamic "election" (for instance, by a competitive protocol such as that used for active replica groups, cf. §6.5.1) or be based on a pre-established ordering between the back-ups. In the latter case, the pre-established ordering must be updated whenever any of the hosts supporting a member of the group fails.

A reduction in failure granularity from that of a complete host down to individual replicas can only be achieved if the failure of an isolated primary replica can be detected. The very principle of passive replication precludes the mutual observation techniques used for replica-level error-detection in active replica groups. Detection of primary replica crash, omission or performance failures could be achieved by several techniques:

a) by monitoring the time interval between a service request and the corresponding reply (a *client-server* model of computation is necessary for this to be possible) — the monitoring can be carried out either by (a representative of) the client issuing the request or by the I/O rep_entity of the primary server who forwards the client's request to the primary server replica[7];

b) by requiring an "I'm alive" response to a periodic local interrogation from the primary's checkpoint rep_entity (this enables the detection of crashes only);

c) by relying on facilities built into the local operating system (e.g., process termination signals, interrogation of the table of active processes, etc.) (this enables the detection of crashes only).

[7] Note that by the very principle of passive replication, the time needed for carrying out roll-back recovery implies that a performance failure of the primary server replica can only be tolerated if such a failure is defined with respect to a replica response delay set to less than half the maximum response delay (or deadline) permissible for the server as a whole.

The recovery action itself is i
back-up replica. The required rep
information contained in the chec
updated from the stored values
software component code at whic

6.6.4. Performance Consi

Passive replication has a few *po*
since hosts supporting passive re
necessary for house-keeping by
replication case; replicas that are
the host is also supporting pass
checkpointing is used — thus
without ensuring identical orde
message to the primary replica
sure that the other rep_entities
speed of response to an input me

However, it is important to
be weighed against two separate

- a *permanent* commu
checkpoints for back
- a *temporary* processi
checkpoint, when a f

The permanent checkpoi
components with large internal
decrease the communication co
the processing contribution):

a) As mentioned earl
multicasting is use
primary and all back

b) Data compression a
avoid sending "unu
stacks and dynamic
If a replica consis
processes that have

c) With appropriate su
be possible to identi
have been modified
rep_entity could k
checkpoint, only t
copy.

Despite these possible opt
usually outweigh the potential
Even in the fault-free case, it
performance than that of active

The recovery action itself is initiated by the checkpoint rep_entity of the selected substitute back-up replica. The required replica data and stack areas are allocated and initialized from the information contained in the checkpoint. Similarly, the processor registers and stack pointer are updated from the stored values and the substitute replica starts execution at that point in the software component code at which the checkpoint was taken.

6.6.4. Performance Considerations

Passive replication has a few *potential* performance advantages over active replication. First, since hosts supporting passive replicas do not carry out redundant computation (other than that necessary for house-keeping by rep_entities), the computation load is lower than the active replication case; replicas that are active on a host will not be severely penalized by the fact that the host is also supporting passive replicas. Furthermore, when systematic or transactional checkpointing is used — thus permitting input messages to be multicasted to all replicas without ensuring identical orders of reception — the primary rep_entity can submit an input message to the primary replica as soon as it is received, i.e., without having to wait until it is sure that the other rep_entities have received the same message. In the absence of faults, the speed of response to an input message can thus be faster than in the case of active replication.

However, it is important to bear in mind that these potential performance advantages must be weighed against two separate time overheads:

- a *permanent* communication and processing overhead to provide back-ups with checkpoints for backward error recovery;
- a *temporary* processing overhead due to rollback and re-execution from a previous checkpoint, when a fault occurs.

The permanent checkpointing overhead could be prohibitively high for software components with large internal states. However, several optimizations are possible to at least decrease the communication contribution to this overhead (to be weighed against an increase in the processing contribution):

a) As mentioned earlier, the frequency of checkpointing can be decreased if multicasting is used to ensure that all input messages are delivered to both the primary and all back-up I/O rep_entities.

b) Data compression algorithms can be applied to checkpoints before transmission to avoid sending "unused" parts of a replica's data space. Similarly, unused areas of stacks and dynamically-allocated data spaces can be excluded from the checkpoint. If a replica consists of multiple processes, then only the data spaces of those processes that have been scheduled since the last checkpoint need to be included.

c) With appropriate support from system hardware and/or operating system, it would be possible to identify and include in the checkpoint only those memory "pages" that have been modified since the last checkpoint. Alternatively, the primary checkpoint rep_entity could keep a copy of the last checkpoint and, after taking a new checkpoint, only transmit the changes that are identified by comparison with the copy.

Despite these possible optimizations, the overheads due to checkpointing and rollback will usually outweigh the potential performance advantages outlined at the beginning of this section. Even in the fault-free case, it is to be expected that passive replication will provide a lower performance than that of active replication.

6.7. Semi-Active Replication

The previous section has pointed out that active replication has significant performance advantages over passive replication since (a) the communication overheads due to checkpointing are avoided and (b) the time to ensure recovery is lower since redundant computations are executed in parallel. However, passive replication (using transactional or systematic checkpointing) has the important advantage of not requiring replicas to be deterministic. The technique discussed in this section attempts to make the best of both worlds in order to provide speedy recovery despite potential non-determinism [Barrett et al. 1990].

As its name implies, the *semi-active replication* technique is in effect a hybrid between active and passive replication. One of the replicas of the group is termed "leader" and the others "followers"[8]. In the absence of errors, only the leader replica produces output messages (like the primary replica of a passively-replicated group). Since, in the absence of errors, only one replica produces output messages, this technique is only suitable for fail-restrained replicas[9].

On the contrary to the passive replication technique, the other replicas (the followers) are not completely inactive (whence the term "semi-active"); they receive the same inputs as the leader and autonomously execute all *deterministic* computation and update their local state accordingly. The leader is responsible for taking all *non-deterministic* decisions and informing the followers of these decisions by means of *notification* messages or "mini-checkpoints" (figure 9).

As will be seen later, the semi-active replication technique does not require messages to be delivered to all replicas in identical orders.

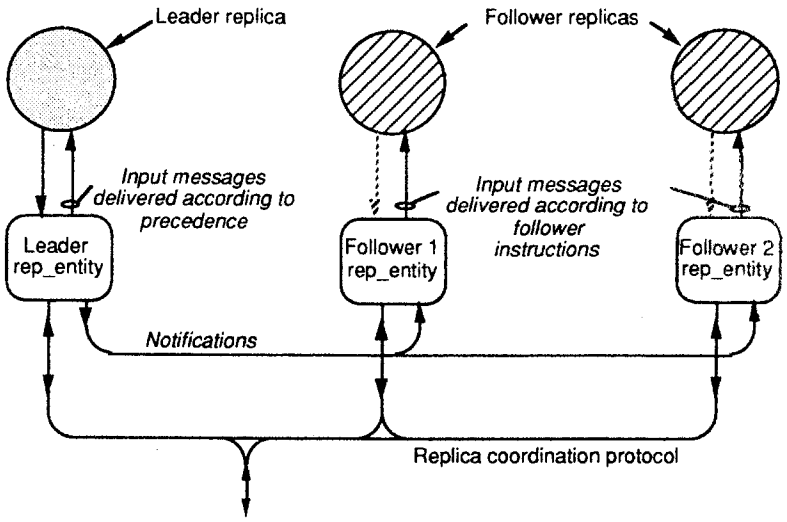


Fig. 9 - Leader-Follower Technique

The non-determinism that the leader-follower technique aims to resolve can stem from:

8 Whence the alternative name for passive replication — the *leader/follower* technique.

9 A combination of active and semi-active replication is considered in section §7.6 that, in certain applications, allows fail-uncontrolled replicas to be accommodated.

a) purposely-introduced

b) a requirement to use active replication in m

In both cases, the rep_entit protocol that forces follower repli

6.7.1. Non-Determinism d

Preemption allows computation lower-precedence computation. P

- in the handling of mes overtake messages of
- in the interruption component may be all

6.7.1.1. Message Preemption. the rep_entities of the leader repl be delivered is of lower precede ensure replica group determina messages in the same order (cf. some replicas could already have would therefore process *HI* afte message *LO* would find message processing *HI* before *LO*. The re order and replica group determin

The leader-follower solution messages as and when they arri rep_entity, the latter informs the figure 9):

- "Present message *M* t

Since the leader rep_entity follower rep_entities, messages multicasted with assurance of enforced on the followers.

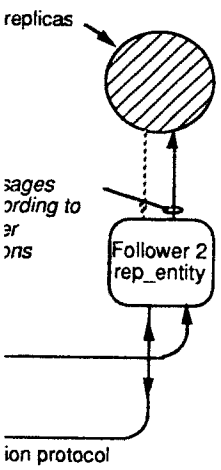
An alternative solution wou rep_entity and for the latter to fo replica consumes them. Howev shorter than input messages, the messages would be lost.

6.7.1.2. Process Preemption. required that a low-precedence message or signal. Clearly, such group determinacy is to be ens 1978]). The leader-follower tech the follower computations to be p

The technique makes use of the computation of a software cc reaches a preemption point, the

has significant performance
unication overheads due to
ery is lower since redundant
ication (using transactional or
not requiring replicas to be
o make the best of both worlds
inism [Barrett et al. 1990].
is in effect a hybrid between
termed "leader" and the others
roduces output messages (like
he absence of errors, only one
for fail-restrained replicas[9].
ier replicas (the followers) are
receive the same inputs as the
n and update their local state
nistic decisions and informing
sages or "mini-checkpoints"

oes not require messages to be



o resolve can stem from:

technique.
in section §7.6 that, in certain

a) purposely-introduced preemption in order to improve performance;

b) a requirement to use off-the-shelf software that was not specifically designed with active replication in mind and so cannot be assumed to be deterministic.

In both cases, the rep_entities associated with the group of replicas must carry out a protocol that forces follower replicas to take the same decisions as the leader replica.

6.7.1. Non-Determinism due to Purposely-Introduced Pre-emption

Preemption allows computations of higher priority or *precedence* (see chapter 5) to displace lower-precedence computation. Preemption may be introduced at two levels:

- in the handling of message queues: messages of high precedence may be allowed to overtake messages of lower precedence;
- in the interruption of processes: a high-precedence event within a software component may be allowed to interrupt a lower-precedence process.

6.7.1.1. Message Preemption. Consider a sequence of two input messages multicasted to the rep_entities of the leader replica and the follower replicas. Suppose that the first message to be delivered is of lower precedence (message "*LO*") than the second one (message "*HI*"). To ensure replica group determinacy, the requirement is that all replicas should process the messages in the same order (cf. §6.5). However, in the absence of inter-replica coordination, some replicas could already have started processing message *LO* before message *HI* arrives and would therefore process *HI* after *LO*. Those replicas that have not already started to process message *LO* would find message *HI* at the head of their input queues and would thus end up by processing *HI* before *LO*. The replica group would therefore not process messages in the same order and replica group determinacy would not be ensured.

The leader-follower solution to this situation is to allow only the leader replica to process messages as and when they arrive. Each time that the leader accepts a message *M* from its rep_entity, the latter informs the follower rep_entities by means of a *notification* of the form (cf. figure 9):

- "Present message *M* to your local replica"

Since the leader rep_entity enforces the order of processing of input messages onto the follower rep_entities, messages sent to a semi-actively replicated component need not be multicasted with assurance of identical order — the order adopted by the leader will be enforced on the followers.

An alternative solution would be for messages to the group to be sent only to the leader rep_entity and for the latter to forward these to the follower rep_entities as and when the leader replica consumes them. However, since the notification messages will generally be much shorter than input messages, the performance advantages of concurrent multicasting of input messages would be lost.

6.7.1.2. Process Preemption. Consider now the case of process interruption. It may be required that a low-precedence process be interrupted by the arrival of a high precedence message or signal. Clearly, such preemption must be synchronized across all replicas if replica group determinacy is to be ensured (see, for example [Frison and Wensley 1982, Sheridan 1978]). The leader-follower technique enables such synchronization to be achieved by forcing the follower computations to be preempted at the same point as the leader's computation.

The technique makes use of the concept of a *preemption point* that is a predefined point in the computation of a software component at which it may be preempted. Each time the leader reaches a preemption point, the leader rep_entity increments a counter. When a message *M*

arrives at the leader rep_entity, a check is made to determine whether *M* requires the leader to be preempted. If so, the preemption point *P* at which this will take place is selected (the current counter value plus 1 represents the next preemption point). The leader rep_entity informs the follower rep_entities by means of a *notification* of the form:

- "Present message *M* to your local replica at preemption point *P*"

Since the preemption point code inserted in the software component must be executed more often than the maximum allowable preemption delay, it is essential that the normal, non-preempting path through this code be efficient.

6.7.2. Non-Determinism in Off-the-Shelf Software

The use of off-the-shelf application software is often a commercial necessity. However, it is not always easy to provide transparent fault-tolerance in the underlying hardware. The only totally transparent way of doing so is to build a tightly-synchronized fault-tolerant machine constructed to provide an interface to the application that is identical to that of the (non fault-tolerant) machine for which the application software was written. This approach to fault-tolerance is diametrically opposite to ours and as such, several advantages of our approach would be lost (cf. section §6.1).

The problem with off-the-shelf software in our approach is that it will probably not have been designed with fault-tolerance in mind — in particular, such software may use non-deterministic language constructs or host-specific information that would violate replica determinism. Two cases need to be considered, the case of "white-box" off-the-shelf software whose source code is easily available and "black-box" software whose internal structure is unknown.

6.7.2.1. White-Box Software. A white-box application is one whose source code is easily available and can thus be inspected to implement mechanisms to intercept non-deterministic decisions. An example would be an Ada application using non-deterministic constructs such as *Interrupt*, *Delay* or the *Clock* function[10].

To implement the required system call interception mechanisms, the rep_entities must be placed "between" each replica and its local executive. When the leader rep_entity intercepts a system call *C*, it informs the follower rep_entities (who will intercept the same call) of the corresponding system reply *R* by means of a notification of the form:

- "When you intercept call *C*, substitute the reply *R*"

A similar mechanism could be used if the leader reads some non-replicated peripheral and also to ensure that preemption occurs at the same point in all replicas.

The number and frequency of such leader-follower notifications therefore depend on the White Box. Exceptionally, it may be unacceptably high and the White Box could then only be supported after some source-code modification to reduce the frequency of non-deterministic actions.

6.7.2.2. Black-Box Software. A black-box application is one whose internal structure is totally unknown, although a description of its external interfaces is normally available. Black boxes can be used if the rep_entities act as "front ends" to interface each replica to the rest of the system.

A typical example is that of a commercial database system such as Oracle.

10 Note that we do not assume that host clocks are necessarily synchronized to some common reference.

Passive replication of an Ora
checkpointing its multiple process

Active replication would be
would process concurrent inputs,
inputs are presented to Oracle re
processed in the same order due t
is particularly important for concu
present inputs to replicas only aft
this sequentialization would lead t

Semi-active replication allow
must be more complex. The lead
and local responses and instruct th

- "Pass *lock(item)* to you
- "Discard *lock(item)* sin
- "Discard *read(item)* sin
- "Takeover the leader r

The important point to gras
particular black box. Another
rep_entities and a different set of

6.7.3. Error-Detection and

The semi-active replication tech
tolerance of hardware faults (of f
detection of silence of any entity
group membership facility are the
replication.

A reduction in failure granul
can also be considered in the sam
since follower replicas are in fa
excessive desynchronization betw

The recovery action is simp
principle — the semi-active repl
replicas is *almost* consistent with
is detected as having failed, a foll
itself up to date by processing th
leader may be carried out either b
between follower replicas.

6.7.4. Performance Consi

The whole purpose of the semi-ac
advantages of the active replicati
deterministic processing like in th

In the presence of faults, the
that is limited to the maximum all

In the absence of faults, th
performance than either the activ
constraint on input message orde

ether *M* requires the leader to be
ce place is selected (the current
e leader rep_entity informs the

on point *P*"
nponent must be executed more
ssential that the normal, non-

ial necessity. However, it is not
ying hardware. The only totally
ılt-tolerant machine constructed
that of the (non fault-tolerant)
approach to fault-tolerance is
of our approach would be lost

s that it will probably not have
such software may use non-
on that would violate replica
ıite-box" off-the-shelf software
are whose internal structure is

is one whose source code is
nechanisms to intercept non-
cation using non-deterministic

nisms, the rep_entities must be
ıe leader rep_entity intercepts a
intercept the same call) of the
orm:

e non-replicated peripheral and
licas.
ations therefore depend on the
White Box could then only be
frequency of non-deterministic

ne whose internal structure is
es is normally available. Black
ce each replica to the rest of the

ıch as Oracle.

d to some common reference.

Passive replication of an Oracle database would be unsuitable because of the difficulty of checkpointing its multiple processes, shared memory and disc data.

Active replication would be more suitable but since we do not know how Oracle replicas would process concurrent inputs, the risk of non-determinism eliminates this choice. Even if inputs are presented to Oracle replicas in the same order, we cannot be sure that they will be processed in the same order due to scheduling decisions carried out within the black box. This is particularly important for concurrent *lock* requests. One solution would be for rep_entities to present inputs to replicas only after having received the reply for the previous input. However, this sequentialization would lead to rather poor performance.

Semi-active replication allows more concurrency but the leader and follower rep_entities must be more complex. The leader rep_entity needs to be able to interpret incoming requests and local responses and instruct the follower rep_entities according to their semantics, e.g.:

- "Pass *lock(item)* to your local replica since leader has granted this lock"
- "Discard *lock(item)* since leader says item is already locked"
- "Discard *read(item)* since leader has already replied"
- "Takeover the leader role since my replica has crashed"

The important point to grasp is that such protocols cannot be generic but specific to a particular black box. Another black box would require different programming of the rep_entities and a different set of *notifications*.

6.7.3. Error-Detection and Recovery

The semi-active replication technique, like passive replication, is primarily intended for the tolerance of hardware faults (of fail-silent hosts) — error detection can thus be reduced to the detection of silence of any entity executing on a given host and the requirements for a node group membership facility are the same as those already discussed in section §6.6.3 for passive replication.

A reduction in failure granularity from that of a complete host down to individual replicas can also be considered in the same way as for the passive replication technique. Alternatively, since follower replicas are in fact active, the set of replicas can be managed so as to detect excessive desynchronization between the leader and the follower replica(s).

The recovery action is simpler than in the passive replication case since — by its very principle — the semi-active replication technique ensures that the internal state of follower replicas is *almost* consistent with that of the leader replica. When the leader replica (or it's host) is detected as having failed, a follower replica is selected to take on the role of leader and brings itself up to date by processing the messages present in its input queue. The selection of a new leader may be carried out either by a dynamic election or be based on a pre-established ordering between follower replicas.

6.7.4. Performance Considerations

The whole purpose of the semi-active replication technique is to be able to reap the performance advantages of the active replication technique and the ability to accommodate potentially non-deterministic processing like in the passive replication technique.

In the presence of faults, the semi-active replication technique will ensure a recovery delay that is limited to the maximum allowable skew between leader and follower replicas.

In the absence of faults, the semi-active replication technique may provide a better performance than either the active or passive replication techniques. First, the relaxation of the constraint on input message order (identical order between replicas is not required) means that,

like in the passive replication case, the leader rep_entity can submit an input message to the leader replica as soon as it is received, i.e., without having to wait until it is sure that the other rep_entities have received the same message. Second, the overheads due to the transmission of notification messages can be expected to be much smaller than those due to checkpoints in the passive replication technique. In fact, since it is expected that notification messages will be much shorter than normal input messages then, due to the aforementioned advantage of immediate processing of input messages by the leader replica, the semi-active replication technique should be of better fault-free performance than the active replication technique.

6.8. Group Management and Fault Treatment

The installation and the management of groups are the responsibility of system administration (see sections 8.2 and 9.5). There are three sorts of groups to be managed:

- the group of fault-free *nodes* or *stations* in the system,
- groups of software component *replicas*,
- groups of *software components* (each of which may or may not be replicated).

Membership of the *node group* can be managed as a function of an underlying multicast protocol (cf. §6.9.2). When a message is multicasted, explicit acknowledgements from the designated destinations enable the presence of nodes to be established. Multiple retransmissions can be considered to resolve the ambiguity between lack of acknowledgement due to node failure or due to a transmission error (see chapter 10). Nodes may leave the node group either in an orderly fashion (by issuing an explicit disconnect command) or suddenly — due to local error detection by fail-silent hardware. Nodes must enter the node group by means of an explicit join procedure that enables their presence to be detected consistently by all other nodes in the group.

Membership of *replica groups* must first be established when a group of replicas in brought into being for the first time. The number of replicas of a particular component is determined by a replication policy that takes into account whether hosts are assumed to be fail-silent or fail-uncontrolled and the cardinality of the specified replication domain of that component. Memberships of replica groups are managed by "replication domain managers" that, like any other software component, may also be replicated. The recursion stops with a "replication superdomain manager" whose member replicas are installed at system (re-) boot time (see sections 8.2 and 9.5).

Logically-distinct *software components* (i.e., *not* replicas) may also be gathered together for some cooperative interaction (e.g., a group of servers providing a "similar" service or a group of transaction managers). Groups such as these provide useful communication abstractions that can simplify distributed application programs. The multipoint associations of the Delta-4 MCS communication system provide such a group abstraction (see section §8.1.3.3). Note that the notions of replica groups and software component groups are orthogonal — each member of a software component group may or may not be replicated.

The replica coordination techniques discussed in sections 6.5 to 6.7 were concerned with the processing of errors so as to hide replica failures from the rest of the system. From the group management viewpoint, the detection of errors may or may not result in the immediate withdrawal of a replica or a node from its corresponding group. Locally-detected errors (i.e., by a fail-silent host or by a NAC) lead to the immediate removal of the node from the set of working nodes (cf. §6.6.3). Similarly, the consequence of a *replica* failure is (usually) the removal of a replica from its group (cf. §6.5.4). Note that failures of multiple replicas (of different software components) residing on the same host would suggest that the fault lies in the

host hardware or in its configu
conclude by a declaration of hos

The set of functions necess
necessary to remove and re-cre
specifically, fault treatment cons

- fault diagnosis,
- fault passivation,
- system reconfiguratio
- system maintenance.

Fault diagnosis is necessary
decide whether the fault is solid
solid fault has occurred, then fau

Fault passivation is necessar
Fault passivation is carried ou
detected errors that result in silen
whose cause is later diagnosed t
be carried out.

System reconfiguration car
entails re-allocation and re-initia
order to restore the level of redu
correctly despite further faul
components may either have to
fault-tolerant operation is degrad
distributed application(s). In the
have to be deferred until a node r

Re-allocation of software co
that creates a new replica on a sp

a) creation of a *templa*
sometimes be done in
application-specific c

b) creation of a copy of
(this is equivalent in
replication, cf. §6.6),

c) activation of the new
automatic managem
between replicated co

Note that in the case of node
can allow sub-operation a) abov
template). Similarly, sub-operati
previously-stored consistent stat
according to a transactional mod

11 Note that in the event of a fault dia
be carried out on the same node as

submit an input message to the
vait until it is sure that the other
neads due to the transmission of
those due to checkpoints in the
.t notification messages will be
aforementioned advantage of
ca, the semi-active replication
ive replication technique.

t

ibility of system administration
be managed:

n,

or may not be replicated).
:tion of an underlying multicast
:it acknowledgements from the
)lished. Multiple retransmissions
acknowledgement due to node
nay leave the node group either
and) or suddenly — due to local
ıe node group by means of an
d consistently by all other nodes

en a group of replicas in brought
ılar component is determined by
assumed to be fail-silent or fail-
ɔn domain of that component.
lomain managers" that, like any
ırsion stops with a "replication
l at system (re-) boot time (see

) may also be gathered together
ɔviding a "similar" service or a
ırovide useful communication
s. The multipoint associations of
group abstraction (see section
ıftware component groups are
ay or may not be replicated.
6.5 to 6.7 were concerned with
he rest of the system. From the
may not result in the immediate
up. Locally-detected errors (i.e.,
oval of the node from the set of
. *replica* failure is (usually) the
failures of multiple replicas (of
d suggest that the fault lies in the

host hardware or in its configuration. In this case, a system-level diagnosis function should conclude by a declaration of host failure.

The set of functions necessary for system-level diagnosis and coordination of the actions necessary to remove and re-create group members is referred to here as *fault treatment*. More specifically, fault treatment consists of:

- fault diagnosis,
- fault passivation,
- system reconfiguration, and
- system maintenance.

Fault diagnosis is necessary to (a) localize the fault (at host level or replica level) and (b) to decide whether the fault is solid or soft (cf. chapter 4). If fault diagnosis should conclude that a solid fault has occurred, then fault passivation must be carried out.

Fault passivation is necessary if it is judged that the faulty entity could cause further errors. Fault passivation is carried out automatically and autonomously in the case of hardware-detected errors that result in silence of a host or a NAC. However, in the case of replica failures whose cause is later diagnosed to be a solid host fault, an explicit fault passivation action must be carried out.

System reconfiguration can be envisaged if there are sufficient redundant resources. It entails re-allocation and re-initialization of the software component replicas that have failed in order to restore the level of redundancy required for the error-processing protocols to function correctly despite further faults[11]. If re-allocation is not possible, then some software components may either have to be abandoned in favour of more critical ones. Alternatively, fault-tolerant operation is degraded to fail-safe operation to ensure safety and/or integrity of the distributed application(s). In the absence of sufficient resources, system reconfiguration will have to be deferred until a node recovers (following maintenance).

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

a) creation of a *template* of the software component at the new location; this can sometimes be done in advance of an actual cloning request in accordance with some application-specific contingency plans,

b) creation of a copy of the component's persistent data or "state" at the new location (this is equivalent in effect to the checkpointing operation necessary for passive replication, cf. §6.6),

c) activation of the new replica whilst ensuring group-consistency; this involves the automatic management of the dynamic, configuration-dependent associations between replicated components.

Note that in the case of nodes possessing stable storage (cf. section 6.2.4), node-recovery can allow sub-operation a) above to be carried out completely locally (from a locally-stored template). Similarly, sub-operation b) above may be replaced by an operation recovering some previously-stored consistent state of the replica if the distributed computation is carried out according to a transactional model (cf. section 6.3.2).

---

11 Note that in the event of a fault diagnosed as a *soft fault*, the re-allocation and re-initialization of replicas can be carried out on the same node as where they resided before failure.

6.9. Communications Support

The previous sections dealt with techniques based on "macroscopic" replication of software components and software-oriented error processing and fault treatment, in a distributed environment. In Delta-4, these techniques rely on basic services such as inter-replica coordination and group membership management, as explained in section 6.4. In the sections that followed, it became apparent that providing these services places some demands on the communications support system, depending on the particular replication technique.

The implementation of distributed fault-tolerance in Delta-4 benefits from the availability of high-quality communication services, such as the ones materialized by protocols designated as *reliable broadcast* or *multicast* [Birman and Joseph 1987, Chang and Maxemchuck 1984, Cristian et al. 1985]. Furthermore, since openness and versatility are desired, the reliable communication service is designed to operate over widely-used local area networks.

This section surveys the properties of a communications system that are desirable to support fault-tolerance based on groups of replicas of components residing in different nodes of the system. The group communication service will be discussed in more detail in chapter 10.

6.9.1. Support for Replication

The replication techniques presented earlier in this chapter have different inter-replica interaction requirements. These requirements are satisfied by appropriate properties of the communication service.

Active replication has a determinism requirement that obliges messages to replicas to be delivered to all of them and in the same order: this is called unanimity and total order (see section 10.1). A service providing these properties is called *atomic*.

Passive and semi-active replication rely on a privileged participant, which is the representative of the replica set. The presence of such a representative allows the unanimity and total order requirements to be relaxed. This can be achieved by using auxiliary protocols at a higher level that take advantage of semantic knowledge. However, in the techniques exploited in Delta-4, at least unanimity may be advantageously preserved, given that: follower replicas have to execute the same commands as the leader replica, checkpointing to standby replicas affects all equally, and changes in the replica set (re-insertion, takeover, etc.) should be perceived consistently. This is especially important when there are more than two replicas in a set. A service providing unanimity alone (and at most ordering messages from individual senders in the order sent, i.e., FIFO) is called *reliable*.

Protocols that order messages according to a cause-effect relationship are called *causal* protocols. For the sake of generality, the communications service should provide causal order, since it is the genuine ordering of events in a distributed system (see section 10.1). However, the cost of causally ordering messages may be avoided in certain settings (see section 9.5.1 for a detailed discussion). In short, this happens when it is shown that there is not a causal relation between senders in different nodes, either because of a restricted concurrency of the computation model, or because the semantics of the application is known not to require such an order. Simpler, non-ordered or FIFO protocols can then be used.

6.9.2. Support for Groups

A modified form of broadcast, where messages only arrive at a subset of the possible system destinations, is called multicast. It is the basis for *group communication*, and is an efficient way of disseminating information.

Multicasting efficiency can
delivery of messages addresse
sender) of the number and locati
mapped into communication-lev

The group paradigm in D
simultaneously, whether they
replicas could be, for example
system, or several groups work
groups of replicated clients, ac
communication services are pro

Inside a group, it is importa
manner consistent with their s
check whether the actual memb
application, to activate recove
consequence, the most general
view to participants, i.e., each
participants in that group. This
management procedures, like
management.

As a rule, any replication
facilities, enhanced with group
simplify implementation of er
observation will be confirmed i

The group communicatio
chapter 10.

6.10. Conclusion

In this chapter, after having dis
elements of a distributed syste
fail-silent network attachment
fail-uncontrolled computers. T
discussed — table 1 summaris

Tabl

Replication technique	
Active	
Passive	
Semi-active	

In the Delta-4 Open Sys
technique is implemented wit

12 An extension of the semi-ac
presently being investigated (s

copic" replication of software
lt treatment, in a distributed
ervices such as inter-replica
in section 6.4. In the sections
s places some demands on the
lication technique.
benefits from the availability of
zed by protocols designated as
hang and Maxemchuck 1984,
tility are desired, the reliable
ocal area networks.
s system that are desirable to
ts residing in different nodes of
in more detail in chapter 10.

lifferent inter-replica interaction
roperties of the communication

iges messages to replicas to be
unanimity and total order (see
mic.
ed participant, which is the
tative allows the unanimity and
y using auxiliary protocols at a
ver, in the techniques exploited
d, given that: follower replicas
ckpointing to standby replicas
ion, takeover, etc.) should be
are more than two replicas in a
ing messages from individual

relationship are called *causal*
ce should provide causal order,
1 (see section 10.1). However,
ain settings (see section 9.5.1
iown that there is not a causal
a restricted concurrency of the
is known not to require such an
l.

a subset of the possible system
nication, and is an efficient way

Multicasting efficiency can be easily increased, if *logical addressing* is used. This allows delivery of messages addressed to "whom-it-may-concern", i.e., with transparency (to the sender) of the number and location of recipients. In Delta-4, system-level logical designation is mapped into communication-level logical addressing.

The group paradigm in Delta-4 allows independent sets of replicas to be supported simultaneously, whether they use the same or different replication techniques. Such sets of replicas could be, for example, several independent fault-tolerant applications in the same system, or several groups working in parallel for the same fault-tolerant application, such as groups of replicated clients, accessing the same group of replicated servers. Several group communication services are provided, to adequately support the different techniques.

Inside a group, it is important for participants to observe the changes in its composition in a manner consistent with their semantics. Group composition must be known, for example, to check whether the actual members gather the necessary functionalities to execute a distributed application, to activate recovery procedures, to reestablish the level of redundancy, etc. In consequence, the most general service of group management, is to provide a *consistent group view* to participants, i.e., each change in group composition is indicated, in a total order, to all participants in that group. This may be used to facilitate the implementation of high-level group management procedures, like group membership, group replication and group cooperation management.

As a rule, any replication technique may take advantage of logical group communication facilities, enhanced with group housekeeping, like the consistent group view property. These simplify implementation of error detection and recovery and error masking protocols. This observation will be confirmed in chapters 8 and 9.

The group communication service of Delta-4, called xAMp, is discussed with detail in chapter 10.

6.10. Conclusion

In this chapter, after having discussed the failure assumptions that can be made for the different elements of a distributed system, we have outlined a simple hardware architecture — based on fail-silent network attachment controllers — that can accommodate the worst-case failures of fail-uncontrolled computers. Three basic software component replication strategies have been discussed — table 1 summarises the relative merits of each.

Table 1 - Comparison of Replication Techniques

Replication technique	Recovery overhead	Non-determinism	Accommodates fail-uncontrolled behaviour
Active	*Lowest*	*Forbidden*	*Yes*
Passive	*Highest*	*Allowed*	*No*
Semi-active	*Low*	*Resolved*	*No*[12]

In the Delta-4 Open System Architecture (OSA, see chapter 8), the active replication technique is implemented within the MCS communication system. The inter-replica protocol is

[12] An extension of the semi-active replication technique to accommodate fail-uncontrolled behaviour is presently being investigated (see section 7.6).

situated within the MCS session-layer and the replication coordination entities of section 6.5 are in effect protocol entities executed on the fail-silent NAC hardware (see section 8.1.4.3). Since it is implemented within the communication system of OSA, active replication is therefore possible in OSA independently of the host computational support environment. The passive replication technique is also available in OSA when the Delta-4 Application Support Environment is used (see chapter 7). The checkpoint rep_entities (cf. figure 8, section 6.6) are implemented as part of the object envelopes of replicated Deltase capsules and use is again made of the MCS session-layer inter-replica protocol whose protocol entities now play the role of the I/O rep_entities of figure 8. Semi-active replication is used in OSA in order to implement a dependable database system (see section 7.6); the leader/follower rep_entity functionality (cf. figure 9, section 6.7) is split between a Deltase *transformer* and the MCS inter-replica protocol.

The Delta-4 Extra-Performance Architecture (XPA, see chapter 9) is intended to accommodate only fail-silent hosts. The semi-active replication technique was pioneered in this architecture since, as explained in section 6.7, two of the primary objectives of semi-active replication are to allow high performance and to resolve non-determinism due to preemption. The leader/follower rep_entities are implemented as part of the XPA group managers (see section 9.6.1.3).

The present implementations of passive and semi-active replication assume a failure granularity equal to that of a complete host — reduction of the failure granularity down to the level of individual replicas (cf. section 6.4.2) is under consideration.

Chapter 7

Delta-4 Applicatio

The Delta-4 system architectu
purpose of Deltase, the *Delta-4*
computing environment for the

This chapter covers the co
actual functions and features in
related to that implementation.
described in the *Delta-4 Implem*

7.1. Purpose and Bacl

The decomposition of an applic
interact with one another by
applications that are implemen
aim of Deltase is to support thi
to distributed fault-tolerant s
applications for execution on a

Deltase provides a unifor
interactions between, *languag*
environment can be mapped on
physical configuration of the di
the hosts.

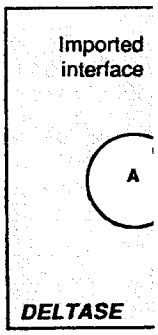


Fig. 1-