COMPUTING
SCIENCE

Active Replication of Distributed Programs: Problems and Solutions

S.K. Shrivastava and A. Tully

TECHNICAL REPORT SERIES

No. 418 March, 1993
Active Replication of Distributed Programs: Problems and Solutions

S.K. Shrivastava and A. Tully

Abstract

Replicated execution of distributed programs provides a means of masking hardware (processor) failures in a distributed system. Application level entities (processes, objects) are replicated to execute on distinct processors. Non-deterministic program constructs within the replicas could cause messages to be processed in non-identical order, or computations to choose different execution paths producing divergence of states. The replicas could thereafter produce inconsistent responses to identical messages and hence appear to be faulty. We identify possible sources of non-determinism and present general solutions for ensuring that non-faulty replicas process messages in identical order and follow identical execution paths in their computations thereby preventing state divergence. Particular attention is paid to real-time programs which can contain a variety of non-deterministic program constructs.
Bibliographical details

SHRIVASTAVA, Sanjosh Kumar
Active Replication of Distributed Programs: Problems and Solutions.
[By] S.K. Shrivasrava and A. Tully

Added entries

TULLY, Alan
UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Replicated execution of distributed programs provides a means of masking hardware (processor) failures in a distributed system. Application level entities (processes, objects) are replicated to execute on distinct processors. Non-deterministic program constructs within the replicas could cause messages to be processed in non-identical order, or computations to choose different execution paths producing divergence of states. The replicas could thereafter produce inconsistent responses to identical messages and hence appear to be faulty. We identify possible sources of non-determinism and present general solutions for ensuring that non-faulty replicas process messages in identical order and follow identical execution paths in their computations thereby preventing state divergence. Particular attention is paid to real-time programs which can contain a variety of non-deterministic program constructs.

About the author

S.K. Shrivasrava joined the Department of Computing Science in August 1975, where he is a Professor.

Mr. A. Tully is a researcher in the Parallel Applications Centre at the University of Southampton.

Suggested keywords

ACTIVE REPLICATION
BYZANTINE AGREEMENT
DISTRIBUTED SYSTEMS
FAULT-TOLERANCE
ORDER PROTOCOLS
REAL-TIME SYSTEMS

Suggested classmarks (primary classmark underlined)
Dewey (18th): 621.381958 001.6404
U.D.C. 681.322.02 519.687
Active Replication of Distributed Programs: Problems and Solutions

Santosh K. Shrivastava¹ and Alan Tully²

¹Department of Computing Science,
University of Newcastle upon Tyne.
Newcastle upon Tyne, NE1 7RU, UK.

²Parallel Applications Centre
University of Southampton
Southampton, UK.

Abstract

Replicated execution of distributed programs provides a means of masking hardware (processor) failures in a distributed system. Application level entities (processes, objects) are replicated to execute on distinct processors. Non-deterministic program constructs within the replicas could cause messages to be processed in non-identical order, or computations to choose different execution paths producing divergence of states. The replicas could thereafter produce inconsistent responses to identical messages and hence appear to be faulty. We identify possible sources of non-determinism and present general solutions for ensuring that non-faulty replicas process messages in identical order and follow identical execution paths in their computations thereby preventing state divergence. Particular attention is paid to real-time programs which can contain a variety of non-deterministic program constructs.

Keywords:

Active replication, Byzantine agreement, distributed systems, fault-tolerance, order protocols, real-time systems.
1. Introduction

Replicating computational entities (processes, objects) on processors with independent failure modes is a principal means of achieving fault-tolerance against processor failures in distributed systems. There are two basic approaches to replication. In active replication, every correctly functioning replica performs processing. In contrast, in passive replication only one member of the group, the co-ordinator (primary), performs processing and checkpoints its state to the rest of the secondary replicas. However, passive replication is only capable of tolerating a limited class of failures: essentially “crash failures” of processors whereby a processor is assumed to either work correctly or fail by stopping to function. When tolerance to larger classes of failures is required (in particular, a failed processor is assumed to perform arbitrary state transitions), then there is no alternative but to use active replication. Active replication is also the preferred choice for supporting high availability of real-time services where masking of replica failures with minimum time penalty is considered highly desirable. However, in active replication it is necessary to ensure that all correctly functioning replicas behave identically. In distributed systems this can be ensured by the use of the state machine approach [19] which requires that (i) non-faulty replicas process identical messages in identical order; and (ii) the computation performed by each replica on a selected message is deterministic. Practical distributed programs for real-time applications however employ a variety of time-based mechanisms for controlling interactions between processes (e.g. non-deterministic message selection using timeouts, deadline driven computations etc.), which cannot be expressed within the existing state machine model. Unchecked, non-determinism within replicas could lead to divergence of states. The replicas could thereafter produce inconsistent responses to identical messages and hence appear to be faulty.

This paper seeks to identify possible sources of non-determinism in distributed real-time computations, and proposes well-defined solutions based on the state machine approach via which consistency may be maintained between active replicas (a short preliminary version of this paper appeared in [23]). We will assume a computation model based on processes interacting via messages and then show how a large variety of non-deterministic input message selection techniques employed in real-time programs can be expressed as special cases of a generic input function, and propose techniques for solving non-determinacy problems for this function. The advantage of our approach for preventing state divergence is that a wide variety of message based concurrent processing techniques can be supported. Examples taken from a number of languages are presented in the paper to support this observation. In addition to using non-deterministic message selection techniques, real-time programs also frequently make use of time-constrained statements in their computations which are a potential source of state divergence; we will show how such statements can be safely employed using the solutions developed here. We will develop our solutions under two types of failure models: crash failures, where it is assumed that a processor either functions as specified or fails (crashes) by ceasing to function at all, and fail-arbitrary (also known
as Byzantine [10]), where it is assumed that a failed processor can exhibit fail-
uncontrolled or arbitrary behaviour.

2. Understanding active replication

2.1. Principles

We assume that (non-replicated) distributed computations have been composed
of a number of processes that interact only via messages. As an example, the function
of a typical 'server' process is to pick up an input message from one of its input ports,
process it and, if necessary, output one or more messages on its output ports
(throughout, we will use the notation action(m) to indicate some application specific
processing of the received message m). We also assume initially that if a process with
multiple input ports has input messages on those ports then any one of these messages
is chosen non-deterministically for processing. A primitive operation, receiveany() is
assumed for receiving a message from any of the input ports:

```
process S /* a typical server */
cycle
   m := receiveany()
   action(m) /* process the message */
   send (result, msg)
end
end S
```

Fig.1: Unspecified Input

The model presented here is based on the well known state machine model
(where a state machine is a process) for which the precise requirements for supporting
replicated processing are known [19]. As remarked earlier, it is also necessary to
assume that the computation performed by a process on a selected message, action(m),
is deterministic, to ensure that "outputs of a state machine are completely determined by
the sequence of requests it processes, independent of time and any other activity in a
system" [19]. Just to emphasise this point, consider the following 'time-constrained'
non-deterministic program:

```
i := 0; within i do i := i + 1 od
```

As it stands, such a program cannot be replicated on independently functioning
processors without causing divergence of states (the value of i at timeout need not be
identical among the replicas). The non-faulty replicas of this program will produce
identical results provided all of them perform identical number of iterations. Later on, in
section 5, we will relax the determinacy restriction and show how such programs can
be transformed to permit replicated processing.

Given the above state machine model of computation, active replication of a
process, such as S will require the following two conditions to be met:

Agreement: All the non-faulty replicas of a process receive identical input messages;

Order: all the non-faulty replicas process the messages in an identical order.
So, if all the non-faulty replicas of a process have identical initial states then identical output messages will be produced by them. In active replicated processing, the degree of replication employed will depend on the classes of faults the overall system is intended to tolerate. If it is assumed that a failed processor can exhibit arbitrary behaviour, then the degree of replication must be at least $2\pi+1$, where $\pi$ is the maximum number of processor failures that can be tolerated in any replica group (with each process on a distinct processor), and an output from a group would be obtained by majority voting. This also assumes that the messages sent by a non-faulty processor can be authenticated by any non-faulty receiver (digital signatures [18] for example can be used for message authentication). If authentication is not possible then the degree of replication must be at least $3\pi+1$ [10]. If failed processors are assumed to exhibit crash failures, then the degree of replication need only be $\pi+1$, and there is no need for majority voting, as only correctly functioning replicas are assumed to produce outputs.

2.2. Implementation

There are several ways of integrating agreement, order and if necessary, voting protocols in a given implementation of state machines. We briefly discuss a few such implementations, starting with a scheme which is particularly suitable when both clients and servers are replicated and tolerance to Byzantine failures is required (this scheme has been implemented for VOLTAN reliable nodes [20]). We assume message authentication is possible. We therefore assume that each processor has a mechanism to generate a unique unforgeable signature for a given message and further that each processor has an authentication function for verifying the authenticity of a message signature. Thus if a non-faulty processor sends a message with its signature to some other non-faulty processor, any corruption of this message during the transmission can be detected by the receiver by authenticating the signature associated with the message. The client replica group uses an intra-group voting protocol to produce voted request messages which are $\pi+1$ signed by distinct processors. Such a signed and voted request message (referred to as a valid message) is sent to the relevant server replica group. A non-faulty server accepts only valid messages for processing. To ensure processing of identical messages in identical order, each process replica has an order process which executes an agreement protocol (this is the signed message algorithm for atomic broadcast [6]) with the order processes of other members of the replica group to ensure that valid messages get queued in an identical order at all the non-faulty replicas; the order process also filters out replicas of a valid message while enqueuing (see fig 2, which depicts a triplicated server, $\pi=1$).

![Diagram](image.png)

VMP: Valid message pool OP: Order process Q: queue of valid messages RS: a replica

Fig. 2: Ordering of valid messages
A server process now picks up the first message from its input queue for processing:

```
process RS /* a replicated version of S */
cycle
  m := message at the head of Q
  action(m)
  send (result_msg)
end
end RS
```

**Fig. 3: A replicated server**

The DELTA-4 distributed systems architecture intended for tolerating fail-uncontrolled nodes has communications level firmware support for providing atomic broadcast services which meet both agreement and order, permitting active replication to be implemented with relative ease [15]. The scheme discussed with respect to fig. 2 can be adapted easily to the case where tolerance to only crash failures is required. There is no need for the client group to sign and vote messages; all messages produced by clients are assumed to be valid; the atomic broadcast protocol executed by the order process need also be tolerant to only crash failures (see [4] for an example).

In the case of crash failures, a particularly attractive performance optimisation for ordering is possible: this consists of forcing the decision of the faster replica onto the slower ones. Such an approach has been suggested for fault-tolerant concurrent C [5] and the 'leader-follower' model for DELTA-4 architecture intended for tolerating crash failures of nodes [2, 15]. We briefly describe the specific solution implemented in the DELTA-4 distributed system. Assume the availability of a reliable multicast message sending facility satisfying the agreement property, which ensures that every functioning recipient gets the message. In this approach then, whilst all the replicas receive and process all requests, only one of them, the leader, is responsible for determining the processing order by selecting the next message to process and forcing its choice onto the rest (the followers). The leader does this by multicasting a *synchronisation* message to the followers. The client-server interaction takes place as follows. The client multicasts its request to the serve replica group and picks up the first reply coming from the group, ignoring the rest. The leader in the server group picks up a request message for processing, and also multicasts a *synchronisation message* to the followers informing them of the choice. Thus, the processing of a request, and the imposition of order can take place concurrently if desired (an attractive feature for real-time systems where it is desirable that a request be processed as soon as possible). A follower, upon receiving such a synchronisation message, picks up the indicated request message and starts processing (thus a follower slightly lags behind its leader). If the leader fails, then the followers must detect this failure and elect one of them as a new leader (this is the price paid for obtaining ordering quickly in the absence of failures). These details, although important, are not central to the subject matter of this paper, so will not be discussed here.

There are other ways imposing order, for example by exploiting a given property of computations. Thus, if computations are structured as atomic transactions,
then the concurrency control necessary for transaction scheduling can be relied upon to impose consistent ordering. The scheme used in the Arjuna distributed system works as follows [11]: if a replicated object is read locked, then no ordering on incoming object invocations from independent client groups need be imposed; in case of write locks, only a single client group will be permitted access, so the need for ordering requests from independent client groups does not arise.

3. Varieties of non-deterministic message selection

In the model presented in Section 2, a 'server' process selects any input message for processing. However, it is often desirable for a process to exercise some control over the selection of messages. This section examines a range of possible selection criteria in terms of classes of inputs. We will assume that a process has a number of input ports for receiving messages, so control over selection may be exercised by choosing messages from specified ports.

3.1. Blocking Input

The server may specify a subset of ports from which it is prepared to accept a message at any given point, but must accept one before continuing. This could happen when the server is acting as a resource allocator. Here is a simple example: a server has several units of a resource, and has two ports, one for receiving 'acquire' requests from clients for a unit of resource and the other for receiving 'release' requests; initially, the server listens on both the ports, however, if the resource units get exhausted, the server switches to listening on the 'release' port till such time as sufficient units (at least one) have been returned. If the message supplied by the order process (fig. 2) is for acquire, and the server is only accepting release requests, then we have a deadlock. Two forms of blocking input are considered next.

(i) Selective Input

The port P1 from which a message is to be accepted is specified by the server, as shown in the algorithm of fig. 4.

```
process S
  var m: message
  ...
  m := receivefrom (P1)
  action(m)
  ...
end
```

Fig. 4: Selective Input
(ii) Alternative Input

A generalisation of the above scheme: one of a set of actions is performed depending on which available message from a set of ports \( P = \{ P_1, \ldots, P_n \} \) is accepted for processing (see fig. 5).

```plaintext
process S
  var m: message,
  P: setof ports
  ...
  \( P := \{ P_1, \ldots, P_n \} \)
  m := receivefrom(P)
  if m.port = P_1 \rightarrow \text{action}_1(m)
  \( \ldots \)
  \( \ldots \)
  \( \ldots \)
  \( \ldots \)
    if m.port = P_n \rightarrow \text{action}_n(m)
fi
end
```

Fig. 5: Alternative Input

3.2. Non-Blocking Input

The server may specify a range of ports from which it is prepared to accept a message but may continue execution instead of accepting one if no message is available. When we consider replicating such a server we must ensure that either all the replicas accept the same message or all of the replicas continue execution without the message. Since messages can arrive at replicas at different times, just preserving order is not sufficient to cope with non-blocking inputs. Three variants of non-blocking input are discussed.

(i) Conditional Input

The port \( P_i \) from which a message is to be accepted is specified by \( S \), as in the example program of fig. 6. Whereas in fig. 4 the receivefrom(\( P_i \)) function is assumed to block until an appropriate message is received, here a null message is returned by the primitive operation receive_nowaitfrom(\( P_i \)) if no message is present when the operation is executed. Thus in the absence of a message from \( P_i \), the server continues execution with some alternative action, altaction(\( \ldots \)).
process S
  var m: message
...
  m := receiveNowWaitfrom(P1)
  if m = null → action(m)
  □ m = null → altaction(…)
fi
...
end

Fig. 6: Conditional Input

(ii) Timed Input

With timed input, the alternative action is performed only if an input does not arrive before the expiry of a timeout as shown in fig. 7. The process blocks for a maximum duration of \( t \) time units, awaiting the reception of a message from the specified port; a null message is returned by receivefrom \((P_1, t)\) only if no message is available and the timeout expires.

process S
  var m: message,
  t:时间
...
  t := <period>
  m := receivefrom(P1, t)
  if m = null → action(m)
  □ m = null → altaction(…)
fi
...
end

Fig. 7: Timed Input

(iii) Timed Alternative Input

With timed alternative input, one of a set of actions is performed depending on which message from a set of ports \( P = [P_1, P_n] \) is accepted for processing; if an input does not arrive before the expiry of a timeout (a null message is therefore returned) then some alternative action is performed, as shown in fig. 8.
process S
var m: message; t: timeval;
P: setof ports
...
P := [P₁...Pₙ]
t := period
m := receivefrom(P, t)
if m = null → if m.port = P₁ → action₁(m)
... m.port = Pₙ → actionₙ(m)
fi
m = null → action(…)
fi
...
end

Fig. 8: Time Alternative Input

4. Replicated processing

4.1. The Generic Input Function

Two forms of enhancement were proposed: blocking input on specified ports and non-blocking input. For the case of blocking inputs, both unspecified and selective inputs (figs. 1 and 4 respectively) are special cases of alternative input (P = 0, where 0 is the set of all ports of a process and P = P₁ respectively). Conditional input is a special case of timed input (t = 0). Finally, alternative and timed inputs are both special cases of timed alternative input (t = ∞ and P = P₁ respectively). Hence, we will term the timed alternative input function receivefrom(P, t) as the generic input function. Resolving non-determinism in the generic input function will also resolve non-determinism for all the other cases. The complete mapping of input types onto the generic input function is shown in fig. 9 (where t = actual timeout value).

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Notation</th>
<th>Ports</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unspecified Input</td>
<td>receiveany()</td>
<td>P = 0</td>
<td>t = ∞</td>
</tr>
<tr>
<td>Selective Input</td>
<td>receivefrom(P)</td>
<td>P = P₁</td>
<td>t = ∞</td>
</tr>
<tr>
<td>Alternative Input</td>
<td>receivefrom(P)</td>
<td>P = P₁...Pₙ</td>
<td>t = ∞</td>
</tr>
<tr>
<td>Conditional Input</td>
<td>receive_nowaitfrom(P)</td>
<td>P = P₁</td>
<td>t = 0</td>
</tr>
<tr>
<td>Timed Input</td>
<td>receivefrom(P, t)</td>
<td>P = P₁</td>
<td>t = τ</td>
</tr>
<tr>
<td>Timed Alternative Input</td>
<td>receivefrom(P, t)</td>
<td>P = P₁...Pₙ</td>
<td>t = τ</td>
</tr>
</tbody>
</table>

Fig. 9: Input Mapping

In the next two sub-sections we will describe how the generic input function may be implemented for the two representative failure models, namely Byzantine and crash.
4.2. Byzantine Failures

The state machine based implementation discussed in Section 2 (fig. 2) is not sufficient to deal with inputs other than unspecified: it could cause deadlocks for the cases of selective and alternative and state divergence for the non-blocking inputs. Any solution to active replication will normally require constructing identically ordered message queues at the replicas, and replacing the message receive operation of the non-replicated process by the "equivalent" message selection operation from the queue. Deadlocks can be prevented if a process replica is allowed to search its message queue for the first acceptable message. Since messages in the queues of replicas are ordered identically, this solution will work. Thus blocking inputs (fig. 4 and fig. 5) can be handled easily. More complex mechanisms are required to solve the second case of non-blocking inputs.

Consider the algorithm of fig. 10 which describes how the generic input function can be implemented by using blocking operations on (identically ordered) message queues. (For the remainder of the paper, CAPITALised function names will be used to denote queue manipulation operations). The generic input function waits for a maximum duration \( t \) to receive a message. Assuming timeout does not occur, the blocking function \( \text{RECEIVEFROM}(P) \) returns the first message in the queue from a port \( P_i \in P, 1 \leq i \leq n \). Since the order of queues is consistent among replicas, so is the choice of message. Hence the subsequent state and response of the replicas will remain consistent.

```plaintext
procedure receivefrom (P: setof port, t: timeval) returns m: message
begin
  within t do
    m := RECEIVEFROM(P)
  timeout:
    (send(P_self, marker)
     m := RECEIVEFROM([P, P_self])
   od
   if m = marker then m := null
   else m := skip
  fi
end
```

Fig. 10: Implementing The Generic Input Function

If no message is received within the duration \( t \), then the timeout clause will be executed. The function then sends a self-directed marker message to itself, on port \( P_{self} \), and waits for either the self-directed message or a message from a member of \( P \). This marker message will get ordered identically at all the replicas with respect to other messages in the respective queues. Therefore either all replicas will select the same message from a member of \( P \) or all will select the self-directed message. The problem is therefore transformed to that of alternative input from \( P = [P_1, \ldots, P_n, P_{self}] \) for which consistency is maintained. If messages produced by replicas are being voted before
processing (e.g., as discussed in section 2, with respect to fig. 2), a timeout has a
chance of succeeding only if a majority of the replicas timeout, in which case a self-
directed message will be majority voted and will arrive at all the replicas.

The power of our approach derives from the fact that blocking and non-
blocking message input operations (with and without timeouts) have been converted to
message selection operations on identically ordered queues. Since identical ordering can
be maintained in the presence of Byzantine failures, our approach is capable of
tolerating such failures.

So far we have assumed that message selection is not based on any notion of
priority: for example in the case of alternative input (fig. 3), if there are messages
available on several input ports $P_1..P_n$ then it is acceptable to select a message from
any one of these ports. However, real-time programs often require a processing
strategy based on a concept of urgency; so let us assume that messages can be assigned
priorities, and the priority of a message can be derived from some information
contained in the message itself. We therefore extend the functionality of the generic
input function by introducing an additional parameter $pr$, with value true indicating that
the selection is to be based on message priority (conversely, false indicates that no
preferential treatment is required):

```
procedure receivefrom (P: setof port, t: timeval, pr: Boolean) returns m: message
```

If pr is true, then the message selection is performed as follows: the procedure
looks for a message from the set of ports $P$ and returns the highest priority one amongst
those present; if no message is present then the procedure execution blocks awaiting
arrival of messages; it is unblocked either because one or more messages arrive on
those ports, in which case the highest priority one is returned, or the timeout occurs, in
which case the null message is returned.

In Section 4.1, it was stated that the message returned by a call to
receivefrom($P$) is the first in the queue from a port $P_i$ of $P$. To implement selection
based on priority, a replica must search the queue for the highest priority message from
ports $P_1..P_n$. Although the order of the queues is guaranteed to be consistent among
replicas, the time at which the queues are updated relative to the call to
receivefrom($P$) is non-deterministic. So the set of messages present in the queue
at the time of the search need not be identical among replicas.

The problem is similar to that of conditional input where the presence or
absence of a message may affect the future behaviour of a process. We present a
function priorityreceivefrom($P$) which will be used instead of
receivefrom($P$) for handling prioritised messages. The problem may be solved
again by using a self-directed marker message to 'mark' the queue, as shown in Figure
11. The set of messages which arrive before the mark is consistent among replicas. So
a search which is restricted to this set will always produce a consistent result.
procedure PRIORITYRECEIVEFROM
  (P: setof ports) returns m: message
begin
  RECEIVEREADYFROM(P)
  send(P_self, marker)
  RECEIVEREADYFROM(P_self)
  m := the highest priority message
  in queue ahead of the marker message;
  Delete the message and the marker
end

Fig. 11: Priority based selection

A call is made to a function RECEIVEREADYFROM(P) which blocks until a message from at least one \( P \in P \) is available in the queue, then returns without removing it. A self-directed message is then sent to mark the queue. A further call is made to RECEIVEREADYFROM(P_self) which blocks until the marking message arrives in the queue then returns without removing it. At this stage, each replica queue contains at least one message from \( P \in P \) followed by the marker message and the replica queues are identical up to and including the marker message. A search of the restricted queue is then performed to identify the highest priority message. Finally the marker and the highest priority message are both removed from the queue and the latter returned to the caller.

procedure receivefrom (P: setof port, t: timeval, pr: Boolean) returns m: message
begin
  within t do
    if pr = true \&\& \exists \ p \in P \ do
      m := PRIORITYRECEIVEFROM(P)
    od
    pr = false \&\& \exists \ p \in P \ do
      m := RECEIVEFROM(P)
    od
  timeout:
    (send(P_self, marker)
     m := RECEIVEFROM([P, P_self])
   od
    if m = marker \&\& m := null
    m \neq marker \&\& skip
  fi
end

Fig. 12: Generic input function with priority

The generic input function with priority is shown in fig. 12. It uses the PRIORITYRECEIVEFROM(P) function when message priority is to be used for selection. We have assumed that if the timeout occurs, then the first message in the queue from the set \([P_1..P_n, P_{self}]\) is returned.
4.3. Crash failures

Let us see how the generic input function can be implemented for the case of the leader-follower replication. This is given below (see figs. 13 a and b). For a leader, we assume that the function choose selects the highest priority message from P, if priority selection is required, or any available message from P when no priority is specified. Further, the availability of a reliable multicast primitive, mcast, is assumed, which is used by clients (for sending requests to server groups) and by a leader (for sending synchronisation messages to its followers). The code for a follower is straightforward: a follower waits for a synchronisation message from the leader and then uses it for selecting a particular message from a port for processing. Note that a follower does not make use of timeout and priority parameters; those are used only by its leader.

```
procedure receivefrom (P:: setof port, t:: timeval, pr: Boolean) returns m: message
begin
  within t do
    (m := choose(P, pr))
    timeout:
      (m := null)
    od
    if m ≠ null → mcast(followers, m:: port, m:: seq) /* multicast
      synch msg with selection information */
      m := mcast(followers, 1:: 1:: ) /* empty synch msg */
      fi
  fi
end

Fig. 13 (a): Generic Input Function (leader)
```

```
procedure receivefrom (P:: setof port, t:: timeval, pr: Boolean) returns m: message
var µ: synchmsg
begin
  receivesynch(µ:: type) /* wait for synch msg */
  if type = empty → m := null
  if type ≠ empty → m := getmsg(µ:: port, µ:: seq)
  fi
end

Fig. 13 (b): Generic Input Function (follower)
```

5. Non-determinism in real-time programs

In this section we investigate the sources of non-determinism in practical real-time programs and show how they may be controlled for active replication using the generic input function. We will be considering the general case of dynamic (event driven) real-time systems which contain a variety of constructs for specifying the timing constraints on computations. We will assume the existence of a global time base in the
distributed system (means of implementing this abstraction by synchronising clocks of the processors in a fault-tolerant manner have been studied extensively (see [17] for a review). We will consider constructs from a variety of languages such as, Occam [12], Ada [1, 16] and Real-Time Concurrent C [7], RTC++ [8] etc. If the language dictates synchronised communication between a sender and the receiver (e.g. Occam), we will assume that it is implemented using a handshake protocol at the communications level. It is at this lower level that we consider the mapping on to the generic input function. Thus in case of a remote procedure call between replicated clients and servers, the generic input function would be used for receiving call requests (at servers) and replies (at clients).

Non-determinism can be classed into external and internal: the former refers to non-deterministic message selection which we have discussed at length in the previous sections; the latter refers to non-determinism in the computation performed by a process. So far we have assumed that the computation performed after selecting a message, action(m), is deterministic. In the latter half of this section we will relax this assumption to permit internal non-determinacy.

5.1. External non-determinism

5.1.1. Occam

In Occam [12] processes can communicate only through message passing; there is no sharing of data. Messages are input from and output to named channels. An output statement succeeds only when some other process executes a corresponding input statement on the same channel. Thus communication is synchronised. The ALT construct, shown in Figure 14, is composed of a number of input statements, only one of which may succeed. Each input statement is followed by an action statement. An action statement is executed if and when its corresponding input statement succeeds.

```
CHAN chan_i, chan_j:
TIMER clock:
INT n, start:
SEQ
  clock? start
  ALT
    chan_i ? n
      action_i(n)
    ...
    chan_j ? n
      action_j(n)
  clock ? AFTER start PLUS t
      action(\ldots)
```

Fig. 14: The Occam ALT Construct
If more than one alternative is ready when the ALT construct is executed, selection between them is arbitrary. One of the input statements may be replaced by either a SKIP statement, which succeeds immediately if no other input can succeed or, a watchdog timer input statement, which succeeds if no other input statement succeeds before completion of the specified period. The SKIP statement is therefore equivalent to a timer input statement with a zero time value.

It is easy to see that the ALT construct can be transformed into the generic input function by mapping each channel onto a port and the watchdog input time value onto the time parameter t. Occam also permits priority based message selection through the use of PRI ALT construct: priorities are assigned according to the textual order of the various input statements. In the above example, assuming PRI ALT in place of ALT, if say chan_j and chan_k both had messages, then the message from chan_j will be accepted and action_j will be executed. It is relatively easy to implement the priority based message selection of Occam using the generic input function.

5.1.2. Ada

An Ada task exports its services as a collection of entry procedures. Tasks communicate by calling the entry procedures of each other using what is called the rendezvous mechanism: a call to an entry procedure succeeds only when the recipient executes an accept statement for that entry procedure; conversely, an accept statement succeeds only when the specified entry procedure is called by another task.

```ada
  task server is
    entry service(...);
    ...
    entry service(...);
  endserver;
  task body server is begin
    select
      accept service(...) do
        action(i);
      end;
      ...
      or
      accept service(...) do
        action(j);
      end;
      or
      delay <period>;
      altaction();
    end select;
  end server;
```

Fig. 15: Selection in Ada (accept)
The select statement, shown in Fig. 15, may be composed of a number of accept statements, only one of which may succeed. Each accept statement may be followed by an action statement. An action statement is executed if and when its accept statement succeeds. One of the accept statements may be replaced by either an else statement, which succeeds immediately if no other accept can succeed or, a delay statement, which succeeds if no other accept statement succeeds before completion of the specified period. The else statement is therefore equivalent to a delay statement with a zero time value. If more than one entry call is pending when the select statement is executed, selection between the accept statements is arbitrary. The generic input function provides all the functionality required for supporting the Ada select statement. A task may also use a select statement to time its entry calls as shown in Fig. 16:

```
task client is
  ...
end client;

task body client is
  begin
    select
      server.service();
      action1();
    or
      delay <period>;
      action2();
    end select;
  end server;
```

Fig. 16: Selection in Ada (entry)

The mapping onto the generic input function in this case is not so clear, as the client performs no explicit input. However the implementation of a rendezvous must employ some protocol using messages between the client and server as explained earlier for the case of remote procedure calls (see also [24] for protocol related issues concerning distributed execution of timed entry calls in Ada). The server timeout is on the request entry message while the client timeout is on the request accepted message (see fig. 17). The select statement with entry calls may therefore be implemented using the generic input function.

```
  Client ------- > request entry

  request accepted < ---------
  request confirmed/aborted

  results returned < ---------
```

Fig. 17: Rendezvous Protocol

In addition to Occam and Ada, we have also investigated the sources of external non-determinism in other languages, such as Real-Time Concurrent C. The message input-output constructs of this particular language for example are a mixture of those
5.2. Internal non-determinism

We will now relax the determinacy restriction assumed so far on the computations performed after selecting messages. Our approach will be to identify the sources of internal non-determinacy and to transform them into "equivalent" external non-deterministic forms, for which we have the generic input function based solution for the prevention of state divergence (for clarity, in the description to follow, we will make use of the appropriate special cases of the generic input function, fig. 9).

5.2.1. Timeouts, deadlines and pre-emptions

We now return to the non-deterministic program mentioned in section 2, and reproduced below:

\[ i := 0; \text{within i do} \quad i := i + 1 \text{ end} \]

As it stands, such a 'time-constrained' program cannot be replicated on independently functioning processors without causing divergence of states as the value of \( i \) at timeout need not be identical among the replicas. However, it is possible to transform it such that identical number of iterations are performed at each replica. Assume that the timeout signal can be transformed into a message (which will, naturally, be ordered identically with respect to other messages at the replicas). Before each iteration the process sends a marker message to itself. If the marker message gets ordered before the timeout message, one iteration is performed. The loop is exited when the timeout message gets selected (see fig. 18, which also shows that just the alternative input functionality is required from the generic input function).

```plaintext
var m: message,
i := 0;
set-timer(t, Pj) /* send a timeout message on port Pj at time now+t */
send(Pself, marker)
cycle
m := receivefrom([[Pself, Pj]])
if m = marker \rightarrow i := i+1; send(Pself, marker)
\square m = timeout \rightarrow exit cycle
fi
end
```

Fig. 18: Replicating a time-constrained program

This simple example can be used to illustrate several problems which arise when replication of time-constrained programs is considered.

(i) In the above example we have assumed that an iteration once begun cannot be aborted. While this may be acceptable in the above simple example, a more
general solution is required, permitting "immediate" halt to current processing. Practical real-time applications often require the use of asynchronous signals (e.g., an alarm signal) to force a process to stop its current activity and perform some alternative action. However, the process replicas could be stopped at different points in their execution, resulting in a difference in their internal states. If the subsequent behaviour of the alternative action is dependent on this state then divergence could occur. Such signals may be handled effectively by transforming them into 'expected events', that is, allowing the process to decide when a signal is to be accepted. We will shortly present a general solution based on the concept of 'pre-emption points', permitting the program in execution to be pre-empted at identical points at all the non-faulty replicas.

(ii) In active replication, there is the general expectation that the replicas execute their programs 'at about the same time'; however the underlying agreement and order protocols themselves cannot be relied upon to meet this expectation (that is not their function). If replicas are running on machines with differing loads, the replica on a lightly loaded machine can potentially race ahead of the one on the heavily loaded one. Special scheduling measures will be necessary to keep the maximum difference between the fastest and the slowest replica to some known and bounded quantity. This is easily achieved if replicas run on identical machines with identical loads, but hard otherwise (we leave this particular case as a topic for further research).

Pre-emption of a program may be achieved by transforming signals which can force pre-emptions (timeouts, alarms etc.) into messages, and ensuring that a process frequently polls its input port where these signal messages are expected to arrive. If the opportunities for message input are infrequent, it may be desirable to transform the program such that the process does poll its input ports frequently enough. This can be done quite simply by explicitly inserting calls to a 'preemptionpoint' procedure in the text of the program. The algorithm of this procedure is shown in Fig. 19 (here it is assumed that signal messages are expected on the set of ports $\rho$).

```
procedure preemptionpoint($\rho$ ; setof port, event-handler: procedure)
begin
  var m: message
  m := receivefrom($\rho$)
  if m $\neq$ null $\rightarrow$ event-handler(m)
  fi
  m $=$ null $\rightarrow$ skip
end
```

**Fig. 19: Pre-emption point**

Here is an example, expressed using Real-Time Concurrent C, which requires the program 'get-clearance' to accomplish its task within 60 time units; failure to do so results in 'abort-landing' being executed:
within deadline (60) get-clearance;
else abort-landing

Suppose the program 'get-clearance' can be decomposed into program units $S_1;...;S_n$ for the insertion of preemption points. Then by inserting preemption points as depicted below, it can be ensured that a process, while executing 'get-clearance', will also examine its port $\rho_1$ (where the deadline expiry message is expected) for the presence of the a deadline expiry message:

set-timer(60, $\rho_1;...;\rho_n$ preemptionpoint($\rho_1$, abort-landing); $\Sigma$b...)

The granularity of units $S_j$ will naturally depend upon the urgency with which a process is expected to respond (in the above example, a preemption point once every few thousand instructions would probably suffice). The revised version of Ada (Ada 9X) also contains provision for writing pre-emptable code, using asynchronous select [1]:

```
select
  event-alternative1;
or
  event-alternative2;
or
  .......
in -- the abortable part
  sequence of statements
end select
```

The asynchronous select statement allows a task to continue executing a sequence of statements while at the same time being sensitive to certain events, which are specified as alternatives of the `select` statements. See [1] for a detailed explanation, but briefly: if no alternative is selected immediately, the task does not suspend itself, rather the abortable part of the program is executed. If one of the events occurs before the abortable part completes, then the execution of this part is aborted in favour of the event alternative. Let $\rho_1;...;\rho_n$ be the ports on which the event messages are expected; then the above construct can be transformed for replicated processing by inserting preemption points into the abortable part of the code, with the body of the 'event-handler' procedure of the form:

```
case m.port of
  $\rho_1$: event-alternative1
  .......
  $\rho_n$: event-alternative0
end case
```

Consider one more example: making a remote procedure call with a deadline. In the non-replicated case, the client sends a call message with a deadline $d$ to the server (so a server has $d = d$-message round trip time for processing the request). The client
times out if no reply is received within $d$ time units; on the server side, the server’s execution is aborted if its computation does not finish within $d$ time units. In the replicated case, naturally we have to ensure that non-faulty replicas behave identically; so for example, either all the non-faulty replicas of the client receive the reply or all timeout. The framework developed here provides a simple means of achieving this; the clients use the timed alternative version of the generic input function to receive a reply, while the servers make use of preemption points to be sensitive to the deadline.

The preemption point approach is particularly efficient to implement under crash failure assumption (which permits optimised solutions of the type discussed in section 4.3, fig. 13). So, in the case of leader-follower replication, the leader upon encountering a preemption point, will send a synchronisation message to the followers, indicating whether to pre-empt or not. However, the use of non-blocking receive means that in the general approach for tolerating Byzantine failures (figs. 10 and 12), every time a preemption point is executed and the preemption message is not present in the queue, the replica is delayed waiting for the arrival of either the self-directed message or the preemption message. We can largely eliminate this delay by taking advantage of the periodic nature of preemption points by making use of blocking receive, and explicitly sending a self-directed message at the end of every preemption point, as shown in fig. 20. The basic “trick” is to ensure that a replica will always find a message in the queue: either the event message or a self-directed message.

```
procedure preemptionpoint(p: α; σ; τ; φ; port, event-handler: procedure)
begin
  var m: message
  m := receivesfrom([p, PSELF])
  if m = marker → event-handler(m)
    if m = marker → sendslice(PSELF, marker)
fi
end
```

Fig. 19: Optimised Preemptionpoint (Byzantine failures)

Assume that a self-directed marker message has been sent during initialisation, to prevent the first preemption point blocking on the call to receivefrom([p, PSELF]) until the occurrence of an event. If no event occurs before that message is entered into the queue then the next preemption point will receive that message and execution will continue. Only one such self-directed message will be in transit at any time.

5.2.2. Programs with external time values

Real-time computations are often required to perform certain actions at specified times (e.g., "open doors at 6 am"). Languages such as Real-Time Concurrent C, RTC++, MPL [14] and Ada 9X all have a variety of constructs for specifying that a given statement should be executed at (or before) some specified time. Consider the following time-constrained statement which is representative of the type of statements provided by the above languages:
at T do $S_1$ else $S_2$

Let the function 'now' return the current time value; then the meaning of the above statement can be expressed as shown below, where we have assumed that delay(T-now) = skip if T≤ now.

\[
\begin{align*}
\text{if } & \text{now} \leq T \rightarrow \text{delay}(T-\text{now}); \ S_1 \\
\text{if } & \text{now} > T \rightarrow \ S_2 \\
\end{align*}
\]

We show below how the technique of sending self-directed messages to make a choice can be used to transform the above program, making it suitable for replicated execution for the general case of Byzantine failures: the choice will depend upon whether the majority of replicas send "at" (in which case $S_1$ will be executed) or "after" (causing $S_2$ to be executed). We assume that the quantity $\Psi$ is based on an estimation of the time required to send and then receive a self directed message.

\[
\begin{align*}
\text{......} \\
\text{if } T \geq \text{now} + \Psi \rightarrow \text{send}(P_{\text{self}}, \text{"at"}) \\
\text{if } T < \text{now} + \Psi \rightarrow \text{send}(P_{\text{self}}, \text{"after"}) \\
\end{align*}
\]

We show below how the technique of sending self-directed messages to make a choice can be used to transform the above program, making it suitable for replicated execution for the general case of Byzantine failures: the choice will depend upon whether the majority of replicas send "at" (in which case $S_1$ will be executed) or "after" (causing $S_2$ to be executed). We assume that the quantity $\Psi$ is based on an estimation of the time required to send and then receive a self directed message.

\[
\begin{align*}
\text{receive}(P_{\text{self}}) \\
\text{if } m = \text{"at"} \rightarrow \text{delay}(T-\text{now}); \ S_1 \\
\text{if } m = \text{"after"} \rightarrow \ S_2 \\
\end{align*}
\]

The solution for the case of crash failures (leader-follower scheme) is quite straightforward: the leader must send a synchronisation message prior to executing $S_1$ or $S_2$ (a follower does not use its clock to make a choice).

5.2.3. **Real-time Object models**

Modern distributed systems are increasingly being designed and implemented using the concept of objects interacting via method invocations. An object is an instance of some class and it consists of *instance variables*, which define its internal state, and a set of *operations or methods*, which define its externally visible behaviour. A remote procedure call is the natural candidate for invoking an operation on a remote object. In such systems, objects are the unit of replication. If the replication technique chosen is active, then all the sources of non-determinacy within an object must be identified and resolved (this is especially true for dynamic real-time systems, which as we have seen can contain a variety of non-deterministic constructs). It is necessary then to design the underlying object model such that it is amenable to active replication. In this subsection we examine the implications of this observation.

An object is typically implemented by mapping it onto a *multithreaded* process (the threads within a process constitute independent 'threads of control', sharing the common address space of the process). For example, the ARTS real-time object model [13] permits multiple threads within an object; these threads rely upon the use of locks
and critical regions to synchronise their activities when accessing shared data inside the object. However, interacting threads within an object are a potential source of internal non-determinacy. It should be clear that if an object contains several concurrent, interacting threads and that object is replicated on distinct nodes, then, unless the pattern of interactions between threads is made identical at all the non-faulty replicas, the states of these replicas could diverge.

There are two ways of ensuring that threads within the replicas of an object do behave identically. One way is to forbid the use of shared memory based thread interactions (e.g., critical regions) and make all intra-object thread interactions message based which can then be agreed and ordered at replicas using the generic input function based approach. The second approach - which only works for the case of single processor user-level threads - is to make user-level thread scheduling non-preemptive: the task of suspending and resuming threads is left to the threads package which is explicitly invoked only from specific points in a computation, thereby ensuring that threads of an object can be made to interact identically across all the non-faulty replicas. This second approach has been used successfully in the DELTA-4 system referred to earlier where the state machine approach has been used for resolving external non-determinacy and internal non-determinacy introduced by threads has been resolved by careful thread scheduling as described above (in addition, preemption points have been used to ensure that a thread in execution is responsive to events). If however, threads are managed by the underlying kernel - as is the case with modern multi-processor operating system kernels (e.g., Mach, which supports preemptable kernel threads [3]), then explicit scheduling control over threads as performed in the second approach is not possible, in which case recourse must be made to the first approach. Thus object models such as ARTS (and several others) are not suitable for active replication over distinct nodes if objects are mapped onto multithreaded processes which make use of kernel-level threads. Elsewhere [21, 25] we have proposed a real-time object model which, while permitting a limited degree of concurrency within an object, relies entirely on the use of generic input function based approach to exercising control over method selection, permitting in addition timeouts and deadlines etc. to be resolved as indicated previously.

5.3. Latency implications

In this sub-section we will briefly consider the latency implications of active replication (a detailed analytical performance analysis is left as a topic for further research). Let $\Delta_r$ be the time taken for ordering a valid message. In case of Byzantine failures (scheme indicated by fig. 2), $\Delta_r$ will be the time taken for atomically broadcasting the message amongst the replicas. In the case of leader-follower replication (fig. 13), $\Delta_r$ will be the time taken to multicast a synchronisation message to the followers. Thus a received message experiences a delay of at least $\Delta_r$ before it is ready for processing by a replica. If the generic input function detects a timeout, it sends a timeout marker message to itself, which must also experience a delay of at least $\Delta_t$ before the actual timeout can occur. If the intended message arrives before this timeout message, no timeout will occur. Thus the effective timeout period will be at
least \( t + \Delta t \) where \( t \) is the timeout value parameter given in the call to the generic input function. If the actual timeout period must be \( t \), then the value of \( t - \Delta t \) must be passed as a parameter to the generic input function. It follows therefore that timeout periods of less than \( \Delta t \) are not possible.

An external event requiring a pre-emption of the program must be converted into a message, which will experience a delay of at least \( \Delta t \). If external events are handled using preemption points, implemented as in fig. 19, and the execution time between preemption points is \( \Delta t \), then there will be an event latency of between \( \Delta t \) and \( \Delta t + \Delta t \), that is an average of: \( \Delta t + \Delta t / 2 \). Note that the time between pre-emption points, \( \Delta t \) cannot be less than \( \Delta t \). If external events are handled using preemption points, implemented as in fig. 20, the latency will increase to between \( \Delta t \) and \( 2\Delta t \), that is an average value of: \( 3 \Delta t / 2 \).

6. Concluding Remarks

Active replication is often the preferred choice for supporting high availability of real-time services where masking of replica failures with minimum time penalty is considered highly desirable. Practical distributed programs for real-time applications however employ a variety of time based mechanisms for controlling interactions between processes (e.g. non-deterministic message selection using timeouts, deadline driven computations etc.), which are sources of non-determinism. Unchecked, non-determinism within replicas could lead to divergence of states. The replicas could thereafter produce inconsistent responses to identical messages and hence appear to be faulty. This paper has identified possible sources of non-determinism in distributed real-time computations, and proposed well-defined solutions via which consistency may be maintained between active replicas.

The ideas presented in this paper can be used to appreciate the full implications of employing active replication for distributed real-time programs which make use of priorities, deadlines and timeouts and are intended to tolerate either crash or Byzantine failures. If overheads imposed by active replication of such programs are deemed excessive for the general case of Byzantine failures, then the computational model should be made restrictive or (in addition) the underlying fault model should be restricted to crash failures. Of course, there must be sound engineering justifications for making such a restrictive failure assumption: for example, because each processor has self-checking logic which prevents erroneous outputs from being produced. Many system designs have opted for highly restrictive computational models which even forbid non-determinism in message selection. Examples are SIFT [26] and MARS [9]. The MARS system consists of a fixed number of periodic processes with known and bounded cycle times, and every message type has a specific globally known time slot within a system cycle for either reception or production. This automatically achieves order amongst replicas: once a process has been designed to process messages from a given set of time slots in a cycle, that process can be replicated without recourse to any additional protocols. This is of course the great advantage of such static systems, which
in addition can be designed so as not to require the notions of timeouts and deadlines [9]. However, it has been argued that many dependable distributed real-time systems of the future will of necessity be dynamic, incorporating a variety of time-based constructs [22]. Our paper has shown how active replication can be incorporated in such systems.

8. Acknowledgements

This work has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project DELTA-4 (Project No. 2252). Our special thanks go to colleagues: Paul Ezhilchelvan, Neil Speirs, David Drackley and Doug Steaton.

References


