Message delivery semantics for the rollback-recovery of undeterministic processes[*]

Denis Conan[†]

Université d'Évry Val d'Essonne, LaMI, F 91025 ÉVRY Cedex
e-mail: conan@lami.univ-evry.fr

Abstract

A well-known method to provide fault tolerance in distributed systems composed of networks of workstations is rollback-recovery. One of the main problems of rollback-recovery is the handling of processes which execute two contradictory types of actions : undeterministic actions and frequent communications between the processes of the application and processes not belonging to it. In this paper, we propose a new solution inspired from the concept of message delivery semantics. There are four types of message deliveries : *exactly once*, *at least once*, *at most once* and *no constraint*. In the solution, the programmer gathers undeterministic actions into a few processes and isolates these processes from the rest of the application by using message delivery semantics. Being isolated, the processes that execute undeterministic actions can send messages to the outside world without incurring a global checkpointing.

Keywords

Fault-tolerance, distributed computing, rollback-recovery, undeterminism, message delivery semantics.

1 Introduction

Our computation model of a distributed application consists in several processes running in parallel on different machines, with communication links between them. Several methods can be used to provide fault tolerance in distributed systems for this model. A well-known one is rollback-recovery [Chan85, Stro85, Koo87, Lai87, Sist89, John90, Elno92, Sens94, Cona96]. This method is becoming increasingly popular because it is economic and transparent to the user. In this solution, when a machine failure is detected, a replacement machine is found in the network. Practically, a checkpointing algorithm registers from time to time the state of the distributed application, a message logging algorithm saves inter-process messages in stable memory during the failure-free execution and a recovery algorithm restarts the distributed application from a previous state in case of a machine failure.

Rollback recovery is often used for long-lived and calculus-intensive distributed applications. But, it doesn't already support distributed applications that simultaneously interact with their environment - e.g., by printing intermediary results to or loading new data from external entities[1] - and execute undeterministic actions - e.g, getting the current time or setting a timer[2]. In fact, undeterminism prevents processes from using message logging. Hence, to recover the

[*]Accepted to the 2nd European Research Seminar in Advanced Distributed Systems, Zinal (Valais, Switzerland), March 17-21, 1997.
[†]This work was supported by the Institut National des Télécommunications, France Télécom, when the author was a PhD candidate.
[1]In the rest of the paper, these external entities are called the outside world of the distributed application.
[2]For the sake of clarity, we exclude from the set of undeterministic actions the receipts of a message from *any* process. Distributed applications that do not contain processes that execute undeterministic actions are said to be *piece-wise deterministic* [Stro85]. Distributed applications that do contain processes that execute undeterministic actions are said to be *undeterministic*.

distributed application from a consistent global state, processes must take coordinated checkpoints. But, each time a process sends a message to the outside world, the global state of the distributed application must be consistent. Therefore, undeterministic distributed applications must take a consistent global checkpoint each time some of their undeterministic processes send a message to their outside world.

In this paper, we propose a new solution inspired from the concept of message delivery semantics [Brze95] to support the fault tolerance of long-lived calculus-intensive undeterministic distributed applications that frequently interact with their outside world. There are four different types of deliveries : *exactly once*, *at least once*, *at most once* and *no constraint* . In the solution, the programmer gathers undeterministic actions into some processes and isolates these processes from the rest of the application by using message delivery semantics. Clearly, we ask the programmer to know in advance which actions are undeterministic and to operate minor modifications of his (her) source code.

This paper is organized as follows. Section 2 presents a short overview of the initial algorithms. Section 3 details the treatment of inter-process message delivery semantics and the benefits obtained. Section 4 shows how we take advantage of these benefits to support undeterministic actions. Section 5 compares the proposed solution with related works and concludes.

2 Short overview of the initial algorithms

First, we give a short overview of our initial checkpointing, message logging and rollback-recovery algorithms. Before describing the algorithms, we explain the organization of processes executing on nodes of the same network in distributed recovery units. The reader can find an exhaustive description of the algorithms in [Cona96].

Distributed applications may involve several networks of workstations. Processes executing on nodes belonging to the same network are gathered into an entity called a *distributed recovery unit* (DRU) [Vaid93]. Each DRU is managed by a dedicated process named the Fault Tolerance Manager (FTM). The algorithms are designed so that no inter-DRU protocol is needed : each DRU is completely isolated from the others - i.e., the FTM acts as a pessimistic gateway [Lowr91].

Checkpointing can be either consistent or independent. The choice is left to the user or the system administrator. If checkpointing must be consistent, the algorithm is similar to the ones described in [Chan85] and [Lai87] : a coordinator (the FTM) reliably broadcasts a checkpoint order and processes being checkpointing mark the messages they send. The third phase of the algorithm transforms the tentative checkpoints into permanent ones [Koo87]. If checkpointing mustn't be consistent, the algorithm is similar to the one described in [Stro88]. The differences are :

- Processes being checkpointing still mark the messages they send.
- When the processes receive marked messages they don't checkpoint their state.
- After their state checkpointing, the processes wait a given time period in order to discover if other processes of the same DRU are concurrently checkpointing. This knowledge is used in the computation of *needed* messages - i.e., messages that must be saved in stable memory for recovery [Stro88].

Message logging can be either pessimistic or optimistic. The choice is left to the user or the system administrator. The algorithm is similar to the one described in [John90] : regularly, each process P_i transmits to the FTM its history and its current dependency vector. The FTM keeps in volatile memory these two sets. Regularly, the FTM saves in stable memory the histories and the dependency vectors, one vector per process (the last one received). The differences from Johnson's and Zwaenepoel's algorithm are :

- The execution machine of the coordinator (the FTM) doesn't need to be reliable (replicated).
- The dependency vectors help in speeding up the computation of the maximum recoverable global state and the commit of the messages sent to the outside world. Moreover, the global state computation is only done when necessary - i.e., when rollbacking or sending a message to the outside world.

Rollback-recovery is managed by the FTM. When a process detects a failure (by means of a connection failure), it alerts the FTM. The FTM thus executes four phases : the detection of faulty processes (crashed or orphan), the reconfiguration (creation of the new incarnations of the faulty processes), the retrieving of information for the re-execution, and finally, the starting of the re-execution. During the re-execution, every receive action is re-played, but messages are not re-sent if receiving processes weren't faulty.

3 Treatment of inter-process message delivery semantics

We now show how message delivery semantics influences the initial algorithms.

Each message is tagged by the application with a delivery semantics. Delivery semantics relates to the definitions of orphan and missing messages as follows [Brze95]. A message is *orphan* if the *receive* action occurs during the execution, but not the corresponding *send* action. A message is *missing* if the *send* action occurs during the execution, but not the corresponding *receive* action. A recovery line R is consistent with respect to a message m if and only if :

- m is neither orphan nor missing and tagged *exactly once*, or,
- m is orphan and tagged *at least once*, or,
- m is missing and tagged *at most once*, or,
- m is either orphan or missing and tagged *no constraint*.

From the last property, delivery semantics relates to rollback-recovery as follows :

- We have to make sure that *exactly once* messages are neither orphan nor missing.
- We have to make sure that *at least once* messages are not missing. In other words, *at least once* messages *can be* orphan.
- We have to make sure that *at most once* messages are not orphan. In other words, *at most once* messages *can be* missing.
- *no constraint* messages *can be* orphan and missing.

These rules modify the definition of *needed* messages. Because *at most once* and *no constraint* messages *can be* missing, they are no longer *needed* for the recovery. Their content is not saved in stable memory when their sender checkpoints.

When a process P_i sends or receives a message, P_i increments the integer counter of its logical clock. The goal is to detect orphan and missing messages. Since *no constraint* messages *can be* orphan or missing, P_i doesn't increment any more its logical clock when sending them.

When a process P_i sends a message m, P_i keeps a copy of m in its volatile memory. *At most once* and *no constraint* messages can be missing, thus their content isn't saved any more in the sender's volatile memory.

When a process P_i receives a message, P_i keeps this new dependency in its dependency vector. This recording doesn't exist any more for *at least once* and *no constraint* messages because P_i isn't orphan when receiving them.

At least once and *no constraint* messages can be received twice or more. Therefore, such messages don't need to be committed any more before being sent to the outside world.

Finally, during the re-execution, processes do not need to re-play sending and receiving actions of *at most once* and *no constraint* messages.

4 Support of undeterministic processes

In this section, we first explain how we support undeterministic actions and then show which benefits can be obtained by using message delivery semantics.

When they start, processes register the FTM and indicate whether they will execute undeterministic actions or not. By definition, undeterministic actions can't be re-played. Therefore, before an undeterministic process sends a message to the outside world, the process must checkpoint and the message must be committed. Moreover, if a process causally depends on undeterministic processes by means of messages sent by these processes since their last checkpoint, it must checkpoint too. This is because such messages might not be re-played. The consequence is that undeterministic processes must cause a checkpoint that must be global and consistent each time they send messages to the outside world. Thus, it is very expensive to support undeterministic processes that communicate with the environment of the application. Moreover, it is not necessary to log messages sent or received by undeterministic processes and the rollback-recovery algorithm doesn't require any modification.

Now, we add the treatment of message delivery semantics to limit the message propagation of the undeterminism. The idea is to gather undeterministic actions into a few processes and isolates these processes from the rest of the distributed application by using message delivery semantics. Being isolated, the processes that execute undeterministic actions can send messages to the outside world without incurring a global checkpointing.

To be more practical, we take an example aimed at being piece-wise realistic : a distributed application collects exchange rates all over the world and does numerous calculations in order to anticipate the next exchange rates. Most of the processes do only calculations, they are called "calculators". The messages exchanged between the calculators are tagged *exactly once*. The calculators execute a calculus-intensive distributed application, and because exchange rates continuously change, it is a long-lived application. The figure 1 represents the architecture of the distributed application.

Some of the calculators are dedicated to the request and the receipt of exchange rates. These processes request and receive the new values of exchange rates from one process that is dedicated to the collection of exchange rates from Stock Exchanges. This process is named the "investigator".

Each value is tagged with a date by the Stock Exchange that gives the value and the distributed application is designed so that calculators are able to request and to receive the same value several times. Thus, the messages exchanged by the calculators and the investigator can be lost due to the failures of either the investigator's machine or the calculators' machines. In other words, we don't have to make sure that these messages are neither orphan nor missing. They *can be* orphan or missing. Therefore, the messages exchanged by the calculators and the investigator are tagged *no constraint*.

The investigator uses timers for periodically requesting new values of some exchange rates to Stock Exchanges. It is able to request the new values of the last n minutes and to request the same values several times. Moreover, the external entity is assumed to be able to treat the same request several times. Thus, requests (*resp.* responses) can be lost due to the failures of either the investigator's machine or the external entity's machine. Thus, requests (*resp.* responses) are tagged *no constraint* and they don't need to be committed (resp. saved) by the FTM before being sent (*resp.* received).

Some of the calculators are dedicated to the display of the values that are newly calculated. These processes send the values to one process that is dedicated to the display of exchange rates to users. This process is named the "displayer".

As in [Gold90], we allow stuttering : outputs may be re-played during re-executions. In fact, we can design the displayer so that *(1)* it can make a difference between an execution and a re-execution, and, *(2)* during the re-execution, it only selects some values to display and avoids the (too fast) display of numerous different values. Thus, the messages sent by the calculators to the displayer *can be* orphan but *can't be* missing. So, they are tagged *at least once*. At the beginning of the re-execution, the displayer asks for one value per exchange rate. Those messages only appear when rollbacking. Since the history of the execution must be equivalent to the one of the re-execution, the logical clock of the calculators mustn't change when receiving them. Therefore, those messages are tagged *no constraint*.

We assume that the displayer sends messages to an entity that doesn't belong to the application and that controls a screen. Since the set of messages sent to the external entity is not the same when re-executing, the messages exchanged between the calculators and the displayer are tagged *no constraint*.

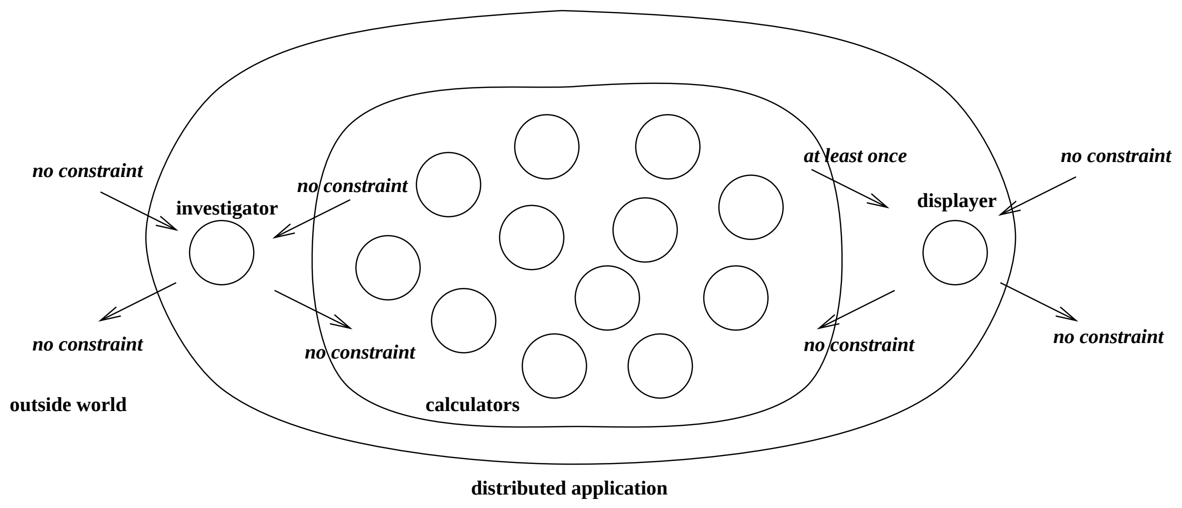

Figure 1: *The distributed application of the example.*

In consequence, the investigator and the displayer are completely isolated from the calculators and the distributed application is capable of dealing with its environment without incurring a higher overhead :

- the checkpointing of the investigator or the displayer doesn't cause the checkpointing of the calculators : the calculators don't causally depend on messages sent by the investigator or the displayer because these messages are tagged *no constraint*.
- the consistent checkpointing of the calculators does cause the checkpointing of the displayer but not the one of the investigator : the displayer causally depends on messages sent by the calculators because these messages are tagged *at least once*.
- the failure of the investigator or the displayer doesn't cause the failure of the calculators : the messages sent by the investigator or the displayer to the calculators are tagged *no constraint*.
- the failure of the calculators doesn't cause the failure of the investigator or the displayer : since the messages sent by the calculators to the investigator (*resp.* the displayer) are tagged *no constraint* (*resp. at least once*), these messages can be received twice or more.
- the sending of messages by the investigator or the displayer to the outside world doesn't cause the checkpointing of the investigator or the displayer, respectively : these messages are tagged *no constraint* and thus don't need to be committed.

The message delivery semantics has permitted the design of a distributed application capable of dealing with its environment without incurring a higher overhead. We only ask the programmer to make a minor intervention aimed at being natural according to the design of distributed applications.

5 Discussion and conclusion

In [Leon94], the semantics used to reduce the amount of rollback required after a failure is the message treatment semantics [Leon94]. The authors detect *insignificant* messages in a *post mortem* analysis. The state changes of processes are of two types : an implicit state change is a change in the content of the process, an explicit state change is the obligation to send out response messages, as a result of the processed message. For example, messages that don't cause either implicit or explicit state changes are insignificant. Our work differs mainly from theirs in three ways. Firstly, in their solution, the programmer doesn't modify his (her) source code while we ask the programmer for a slight intervention. Secondly, their analysis is *post mortem* while ours also happens during the execution. Thirdly, the treatment semantics of a message isn't known when sending the message. On the contrary, in our case the delivery semantics of a message is known when sending the message because the programmer gives it. Finally, it should be possible to use both treatment and delivery semantics.

In [Shri87] and in [Manc89], the authors claim that the transaction model (using objects and actions) and the conversation model (using communicating processes) are duals of each other. *"A pleasing consequence has been the recognition that the techniques which have been developed for one model, turn out to have interesting and hitherto unexplored duals in the other model"* [Shri87]. We have drawn our inspiration from this approach. In this paper, we develop an extension to the conversation model that permits a nice support of undeterministic processes that communicate with their environment. The solution is based on the concept of message delivery semantics. By analogy, the message delivery semantics is similar to the invocation treatment semantics - with *read only* and *modify* actions - that is often used in the objects / actions model.

The example of distributed application we have studied is typically naturally designed using the transaction model. We think that using the conversation model would be as easy as using the transaction model. We think that gathering all undeterministic actions and actions that manage the environment in a few processes is as natural as writing transactions.

Acknowledgements

We would like to thank Guy Bernard for his valuable comments on various versions of this paper. The referees are also acknowledged for their remarks, which helped to improve the presentation.

References

[Brze95] J. Brzeziński, J.-M. Hélary, and M. Raynal. Semantics of recovery lines for backward recovery in distributed systems. *Annales des Tél'ecommunications*, 50(11-12), 1995.

[Chan85] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), February 1985.

[Cona96] D. Conan. *Tolérance aux fautes par recouvrement arrière dans les systèmes informatiques répartis*. PhD thesis, Université Paris VI (France), Septembre 1996.

[Elno92] E.N. Elnozahy and W. Zwaenepoel. Manetho : Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41(5), May 1992.

[Gold90] A.P. Goldberg, A. Gopal, K. Li, R. Strom, and D.F. Bacon. Transparent Recovery of Mach Applications. In *Proc. 1st USENIX Mach Symposium*, 1990.

[John90] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11, September 1990.

[Koo87] R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1), January 1987.

[Lai87] T.H. Lai and T.H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25, May 1987.

[Leon94] H.V. Leong and D. Agrawal. Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes. In *Proc. 14th International Conference on Distributed Computing Systems*, Poznan (Poland), May 1994.

[Lowr91] A. Lowry, J.R. Russell, and A.P. Goldberg. Optimistic Failure Recovery for Very Large Networks. In *Proc. 10th IEEE Symposium on Reliable Distributed Systems*, 1991.

[Manc89] L.V. Mancini and S.K. Shrivastava. Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality. In *Proc. 19th IEEE Symposium on Fault Tolerant Computing*, June 1989.

[Sens94] P. Sens. *Conception et mise en œuvre d'une plate-forme logicielle de tolérance aux fautes pour le support d'applications réparties*. PhD thesis, Université Paris VI (France), Décembre 1994.

[Shri87] S.K. Shrivastava, L.V. Mancini, and B. Randell. On the Duality of Fault Tolerant System Structures. In J. Nehmer, editor, *Experiences with Distributed Systems*. Springer-Verlag, 1987.

[Sist89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Stro85] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[Stro88] R.E. Strom, D.F. Bacon, and S.A. Yemini. Volatile Logging in N-Fault-Tolerant Distributed Systems. In *Proc. 18th IEEE Symposium on Fault Tolerant Computing*, Los Alamitos (USA), June 1988.

[Vaid93] N. Vaidya. Distributed recovery units: An approach for hybrid and adaptative distributed recovery. Technical Report 93-052, Texas A&M University, Texas (USA), November 1993.