Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Reversibility with Holes

This is the final peer-reviewed author’s accepted manuscript (postprint) of the following publication:

Published Version:
Fabbretti, G., Lanese, I., Stefani, J.-B. (2024). Reversibility with Holes. GEWERBESTRASSE 11, CHAM, CH-6330, SWITZERLAND : Springer Science and Business Media Deutschland GmbH [10.1007/978-3-031-62076-8_5].

Availability:
This version is available at: https://hdl.handle.net/11585/997981 since: 2024-12-03

Published:
DOI: http://doi.org/10.1007/978-3-031-62076-8_5

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

06 May 2026

Reversibility with Holes*
(Work in Progress)

Giovanni Fabbretti[1], Ivan Lanese[2], and Jean-Bernard Stefani[1]

[1] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
[2] Olas Team, Univ. of Bologna, INRIA, 40137 Bologna, Italy

Abstract. According to Landauer's principle, any non-reversible system can be made reversible -that is, capable of undoing its actions- by keeping information about the past of the computation. In the area of concurrent and distributed systems, this often takes the form of *memories*. Memories are special devices that keep track of past states of a system execution. Memories can be looked up to restore past states, upon necessity. This paper investigates and lays down ideas on how to achieve reversibility in systems that are subject to events that, as a side effect, erase some memories, creating then holes in the structure of memories. The chosen application area is concurrent and distributed systems, where the events erasing memories are the failure of nodes.

1 Introduction

A system is reversible if it is capable of executing not only in the standard, forward manner, but also backwards. Reversibility in computer science was first studied in the sixties in the seminal paper by Landauer [4] and then over the years flourished and appeared in many diverse application areas, from hardware [4] to formal languages such as CCS [1,8] and programming languages such as Erlang [7].

The vast majority of systems existing today have not been designed with reversibility in mind because they compute irreversible functions. However, one way to make them reversible is by employing *memories*. A memory is an artificial device that keeps track of the past state of a system execution that can be looked up to restore such past state later on, when needed.

In this paper we focus on concurrent and distributed reversible systems, relying on memories, for which *causal consistency*, proposed by Danos and Krivine in [1], is the most used notion of reversibility (alternative notions of reversibility are used, e.g., for biological systems [9]). Causal consistency states that an action can be undone iff all of its consequences, if any, have been already undone.

* The work has been partially supported by French ANR project DCore ANR-18-CE25-0007. The second author has also been partially supported by MSCA-PF project 101106046 — ReGraDe-CS and by INdAM – GNCS 2023 project RISICO, code CUP_E53C22001930001. The authors thank the anonymous reviewers for their useful comments and suggestions.

To check whether an action is a consequence of another one, memories need to keep trace of causal dependencies between past actions. Then, from memories it is possible to build the *causal graph* of the system execution - that is, an acyclic graph capturing causal relations between process states. Hence, before undoing an action, it suffices to check that it has no descendants in the causal graph.

To the best of our knowledge, a common assumption of the works on concurrent and distributed reversible systems relying on memories, such as [1,6,7], is that memories cannot be lost. However, in distributed systems this assumption is extremely challenging to fulfill at best. Indeed, distributed systems are notoriously subject to failures that affect the normal functioning of their nodes. Because of their structure, even relying on persistent storage may not easily solve the problem, because the failed node may not be readily accessible by, e.g., a human operator to retrieve memories.

In this paper, we study *causal-consistent rollback* [5], which allows one to undo an action in a causal consistent way, namely by undoing all and only its consequences. We propose two approaches for rollback in distributed systems subject to failures, which differ on the degree of cooperativeness that their nodes are willing to put in place.

We consider distributed systems where each node, locally, records, on volatile memory, all the actions, local or remote, performed by the processes it hosts. When a node fails all the processes and memories it was hosting are forever lost.

In the *non-cooperative* approach, the loss of memories creates *holes* in the system causal graph. We also assume that failures cannot be undone. Indeed, even if a node could be restarted the lack of information makes it impossible to restore its state prior to the failure. The resulting notion of reversibility, for the rollback algorithm to be causal consistent, requires to approximate causal dependencies.

In the *cooperative* approach, each node, besides locally recording the actions of the processes it hosts, when interacting with another node, copies on it the content of the memories related to the causes of the interaction. As a consequence of this copying mechanism, failures can be undone. When a failure is undone the rollback algorithm restores the last state known by the alive components of the system, available in the copied memories. This mechanism, inspired by the MANETHO rollback algorithm [3], makes so that when a locality fails, the only pieces of information lost are the ones which were only present on the failed node, either because they have never been copied elsewhere or because the nodes on which they were copied already failed. The resulting notion of reversibility never needs to over approximate causal dependencies between actions while undoing them.

Organization of the paper. The paper is structured as follows. Section 2 describes the base framework common to non-cooperative and cooperative systems. Section 3 and Section 4 present non-cooperative systems and cooperative systems, respectively. Section 5 concludes the paper.

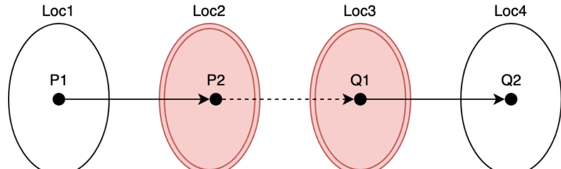

Fig. 1: Causal Graph: Dependencies

2 The Basic Framework

This section presents the basic reversible distributed systems on which we will discuss the two variants of rollback.

Processes are the smallest computational unit and are hosted inside nodes. We assume that nodes are connected among themselves through a reliable network. We do not assume any specific topology but we assume that any node, possibly through routing, is always able to reach any other node. The behavior of a node is defined by the behavior of the processes it hosts. Actions can either be local or remote. We assume that local actions can be either synchronous or asynchronous while remote actions are always asynchronous. Local actions do not interact with other nodes, like, e.g., a sys-call to open a local file, a local message exchange, etc. Remote actions interact with other nodes like, e.g., sending a message to another node, creating a new node, etc.

When doing an action, a node also produces a memory to record the past state of the process doing the step. This memory is kept in the node's local volatile memory. If the action is remote, a memory is also produced in the target node and kept on its local volatile memory. We assume that memories, belonging to two different nodes, corresponding to a same interaction, can be matched. This could be achieved, e.g., by having the two memories sharing the same unique id.

Nodes can fail abruptly. Whenever a node fails, all the processes on it immediately stop working and all the memories on it are lost.

3 Non-Cooperative Rollback

We now discuss how causal-consistent non-cooperative rollback ought to work in the context of the distributed systems described in Section 2.

In absence of failures, the rollback algorithm [5] works as follows. First, given a target action to undo, it identifies its consequences. Then, it undoes them following a reverse causal order.

In presence of a single failure, rollback works as follows. Given a target action, the rollback algorithm computes its consequences. If no consequence is an interaction with a failed node, then it proceeds as in absence of failures. If at least one of the consequences is a remote action, interacting with some action a on a failed node, say N_1, then ideally we should undo every remote interaction node N_1 had which is causally dependent on a, together with its consequences.

However, N_1's causal graph, due to N_1 failure, is lost, hence it is impossible to know which of its interactions are causally dependent on a. Because of this lack of knowledge, the only way to undo all the consequences is to undo every interaction between N_1 and other nodes, together with their consequences. This may include many actions which are actually not consequences of a, hence it is an over approximation.

In presence of several failures, causal-consistent rollback gets even more intricate. Before discussing the general case, we first represent a concrete example of rollback in presence of two failed nodes in Fig. 1. A white, single-circled ellipse denotes a running node, while a red, double-circled ellipse denotes a failed node, dots inside ellipses denote processes and arcs (dashed or not) between dots denote interactions, causally linking states of two processes. Now, suppose we wish to undo the interaction between P1 and P2. Ideally, we should compute the set of consequences, which would be different if the dashed arc between P2 and Q1 existed or not. However, because of the failures of Loc2 and Loc3, it is impossible to determine if this dependency existed or not. Hence, again, the only way to be sure to undo all the consequences is to undo every interaction between the failed nodes, Loc2 and Loc3 here, and the other nodes. This includes in particular the interaction between Q1 and Q2 in Loc4.

In general, when several nodes fail, if at least a consequence of the target action interacts with a failed node, then, the only safe way to be causal consistent is to undo every interaction that every failed node had, together with its consequences. Whereas, if no consequence of the target action to undo is an interaction with a failed node, then rollback proceeds as in absence of failed nodes.

4 Cooperative Rollback

The distributed systems considered for cooperative rollback extend the base framework described in Section 2 as follows. Whenever there is a remote interaction between two nodes, besides recording locally the actions of the processes they host, the interacting process shares with the remote node its antecedent graph [3], i.e., the content of those memories recording the causes of the interaction. The redundancy of memories creates resilience to failures.

Indeed, when a node fails the only pieces of causal information that cannot be retrieved are those that either were never communicated to other nodes or that were communicated only to nodes that already failed.

As a consequence of redundancy the failure of a node can be undone, provided that there exists at least an alive node *remembering* an interaction with it. That is, we assume to have sufficient control over the system to be able to recreate nodes that failed. In undoing a failure, the rollback algorithm restores on the recovering node the processes it was hosting by retrieving their latest state from the corresponding memories on the other alive nodes. Conversely, if there is no alive node holding information about a failed node, then its failure cannot be undone.

We now discuss how causal-consistent rollback ought to work in this setting.

Given a target action, the rollback algorithm computes the set of consequences as usual. Then, it proceeds to undo them in a causal reverse order.

Here, we claim that the rollback algorithm can operate without requiring to approximate the set of consequences even in presence of failures. Indeed, in cooperative systems any alive process knows all the interactions, local or remote, that led to its current state. Hence, all the consequences of an action a can be retrieved by checking which processes descend from it. If a process does not have action a in its causes, then it is guaranteed to not be a consequence of a.

For example, let us see how the cooperative approach eliminates the need to approximate causal dependencies in the system of Fig. 1. Suppose, as before, that we wish to undo the interaction between processes P1 and P2. Here, since process Q2 on Loc4 is alive we can, by looking in Q2's memories, assert with certainty whether or not Q1 is a descendant of P2. In case Q1 descends from P2, and consequently from P1, then the actions to undo would be: i) interaction between Q1 and Q2; ii) failure of Loc3; iii) failure of Loc2; iv) interaction between P2 and Q1; v) interaction between P1 and P2. Whereas, in the case in which Q1 is not a descendant of P2, the actions to undo would be: i) failure of Loc2; ii) interaction between P1 and P2.

The mechanism proposed to propagate histories among localities in this section is similar to the one used in the MANETHO rollback algorithm [3]. Although at first the overhead imposed by duplicating memories may seem significant, this strategy has been already put in place successfully. Indeed, one can exploit various optimization techniques as, e.g., only copying the piece of past not already present in the remote locality, to reduce the overhead. We refer to [2] for further discussion on the topic.

5 Conclusion

In this paper we proposed ideas on how to achieve reversibility in systems that are subject to events that erase memories. The chosen application area is concurrent and distributed systems and the erasing events are failures. In Sections 3 and 4 we identify two different approaches to causal-consistent rollback, which depend on the degree of cooperativeness of the underlying distributed system.

When localities are not cooperative, to ensure causal-consistency in the context of rollback, causal dependencies must be over approximated in case of failures. Without surprise, cooperation among localities enables strategies, like memory redundancy, to avoid the above mentioned over approximation. We remark that cooperation may not be possible since different nodes may obey to different authorities, or since the memory overhead required to replicate information may not be always desirable.

The cooperative and non-cooperative strategies described above are not the only possibilities. Indeed, by playing with the degree of cooperativeness, e.g., one could think to a system in which some nodes cooperate and some do not cooperate and by adding more hypothesis, as e.g., timestamps on interactions, many

more interesting strategies, worth studying, emerge. The possible strategies can then be ordered on a spectrum according to their need of over approximation. As a first step, we chose the cooperative and non-cooperative strategies above as they well represent opposite extremes on the spectrum above.

References

1. V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.
2. E. N. Elmootazbellah. *Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication.* Phd thesis, Rice University, 1993.
3. E. N. Elmootazbellah and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, 1992.
4. R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5(3), 1961.
5. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311. Springer, 2011.
6. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.
7. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.
8. I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007.
9. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *RC*, volume 7581 of *LNCS*, pages 218–232, 2012.