The External Recovery Problem*

Arkadiusz Danilecki, Mateusz Hołenko, Anna Kobusińska, and Piotr Zierhoffer

Institute of Computing Science
Poznań University of Technology, Poland
{adanilecki,akobusinska,mholenko}@cs.put.poznan.pl

Abstract. We consider an external recovery problem, where a system is divided into autonomous subsystems which can be recovered only by the means of logging the messages exchanged between the subsystems. The question follows: what restrictions to the subsystem's autonomy are required to make the external recovery possible? We present example solutions affecting different aspects of system's independence.

Keywords: Message logging, fault tolerance, checkpointing, distributed system.

1 Introduction

The probability of a node crash in a modern, large-scale computing systems, consisting of hundreds of thousands of nodes, comes near certainty. One approach is to divide the system into subsystems, and to isolate the crash effects within a subsystem where the crash occurred. Then, a coordinated checkpointing can be used within a subsystem [13], while to prevent crash effects from spreading, the messages exchanged with processes from different subsystems could be logged in a pessimistic manner. An interesting theoretical question arises: under which conditions a subsystem could be recovered *only* by logging the messages exchanged with other subsystems – by what we call an *external recovery*.

There is an unspoken assumption that all parts of the system are under control of one organization, that they cooperate freely and that they expose all information necessary for the recovery. These assumptions may not hold in the future, when subsystems may be more independent. Future cooperating components involved in distributed computation may be unwilling to restrict their independence by e.g. revealing the information commonly assumed to be available for the message logging protocols. Nevertheless, if the subsystem is to be recovered using external message logging, it can't completely retain its independence.This observation spurred the question: *What must be minimally known about a system and what minimal restrictions must be imposed on a system behavior, in order to make the external recovery possible?*

This paper is a first step in the direction of solving this puzzle, by identifying the problem, the possible trade-offs, and by presenting two example approaches

* This work was supported by the Polish National Science Center under Grant No. DEC-2011/03/D/ST6/01331.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 535–546, 2014.

to restricting system independence. While we find the problem interesting from purely theoretical reasons, we are convinced that it will have practical applications, for example in creation of recovery protocols for federated clusters [16].

Our paper is organized as follows. Section 2 introduces the system model. Sections 3 and 4 present solutions restricting different aspects of a subsystem's independence. Section 5 discusses related work. Section 6 concludes the paper.

2 System Model

We consider a distributed system $\mathcal{S} = \mathcal{P} \cup \mathcal{W} \cup \mathcal{I}$, where $\mathcal{P} = \{P_1, P_2, \dots P_n\}$ is a set of n processes, $\mathcal{W}$ denotes external world, and $\mathcal{I}$ is an *interceptor*. Intuitively, if $\mathcal{S}$ would be a set of clusters, then $\mathcal{P}$ would denote processes in a particular cluster under a consideration and $\mathcal{W}$ would consist from processes from all other clusters (for simplicity treated as singular entity). Interceptor $\mathcal{I}$ is a layer responsible for logging the messages exchanged between $\mathcal{P}$ and $\mathcal{W}$. The processes in $\mathcal{P}$ communicate via message passing using asynchronous reliable FIFO links. In addition, $\mathcal{W}$ may send messages to any $P_i \in \mathcal{P}$ (hereafter denoted as *input*s) and any $P_i \in \mathcal{P}$ may send messages to $\mathcal{W}$ (*output*s). The inputs and outputs always pass through $\mathcal{I}$ where they may be inspected, delayed, discarded, or stored in a stable storage. We stress that the interceptor is just a theoretical construct. $\mathcal{I}$ may be implemented as a single node, many independent nodes, or even as logging mechanism atop processes in $\mathcal{P}$.

Processes have states. At any given time t, a sum of states of all processes in $\mathcal{P}$ forms a global state of $\mathcal{P}$. We understand the consistent state of $\mathcal{P}$ in a usual way [7]. Processes execute programs, generating *events*. Each event changes the process state. Of special interest are *send* and *receive events*, produced when process sends or receives a message. For brevity, a receive event of an input will be called an *input event*. Processes in $\mathcal{P}$ are initially passive; they become active only as an effect of receive event. It follows that absent inputs, processes in $\mathcal{P}$ are continuously passive. Each time a process P_i becomes active, it produces a finite sequence of events. Each event changes P_i's state and may involve sending messages to some $P_j \in \mathcal{P}$ or to $\mathcal{W}$.

History $H(t)$ at a global time t is a set of events produced by all processes in $\mathcal{P}$ until t, partially ordered by Lamport's happened-before $\mapsto$ relation [12]. It follows that the global state of $\mathcal{P}$ at t is a result of $H(t)$. In addition, there is a relation of *true dependency* $\xrightarrow{true}$ between the events [24][1], and we say that if $e^j \xrightarrow{true} e^k$, then e^k was *truly caused* by e^j. A *session* S of an input M is a set of all events truly caused by the receipt of M, ordered by $\mapsto$ relation. Each event in a history H belongs to some session. We assume sessions do not overlap and are of finite size. A *reaction* to an input M is a set of outputs forwarded by the $\mathcal{I}$ to $\mathcal{W}$, whose send events belong to session of M.

[1] Lamport's relation reflects system point of view; that is, if $e \mapsto e'$, system must assume that indeed e' was caused by e, and without e, e' would not happen. True dependency reflects real, logical dependencies resulting from application logic. See also always-happened-before relation in [21].

For any $P_i \in \mathcal{P}$, given the P_i state, only some events are possible. If only one event is possible, then this event is deterministic. Otherwise, it is non-deterministic. In a piecewise-deterministic (PWD) model, all the non-deterministic events (usually, only the receive events) can be identified, and their *determinants* (the information needed to replay the event) can be stored [23]. Given some initial state of P_i, always the same state of P_i is produced when a sequence of events is generated from a given set of determinants. In a send-deterministic [6] model, informally, the order of receive events does not influence the send events.

The processes in $\mathcal{P}$ may crash, losing their state. In that case we assume all processes within $\mathcal{P}$ restart from an initial, consistent state, but the algorithms may be extended to $\mathcal{P}$ periodically taking consistent checkpoints (e.g. by finding out which messages do not have to be replayed during the recovery). A perfect failure detector is available, notifying $\mathcal{I}$ when all processes in $\mathcal{P}$ are restarted. We assume $\mathcal{I}$ and $\mathcal{W}$ do not crash, crashes are rare and no crash happens during a recovery; this is a reasonable assumption, if the size of a $\mathcal{P}$ is small enough (e.g. within a range of hundreds, rather than thousands of processes). The recovery is correct if the reaction to any input in a history with the crash and recovery events would be possible in some history without crash events.

There exists a mechanism discarding messages sent before, but received after the crash, including inputs sent to $\mathcal{P}$ by interceptor, e.g. by using epoch numbers. Let all processes in $\mathcal{P}$ and $\mathcal{I}$ maintain epoch number e incremented after each restart (note $\mathcal{I}$ and $\mathcal{P}$ will have always identical epoch numbers). Each message (including inputs) is assigned a current epoch number. Processes ignore all messages with epoch smaller than e.

3 Restricting the Behavior Only

The following constraints may be imposed on $\mathcal{P}$'s autonomy: its behavior (set of possible histories) may be restricted a priori; the information about the behavior (the history of the computation) and about the $\mathcal{P}$'s structure (e.g. how many processes are in $\mathcal{P}$) may be exposed to $\mathcal{I}$; finally, processes in $\mathcal{P}$ may have to cooperate with $\mathcal{I}$ during the recovery (e.g. exchanging control messages with $\mathcal{I}$).

Intuitively, the external recovery is possible if $\mathcal{P}$ can be treated (from the point of view of $\mathcal{I}$) as a single, piecewise-deterministic (PWD) process. If processes in $\mathcal{P}$ are piecewise-deterministic, then from the definition of PWD we conclude that unless the determinants of non-deterministic events are exposed to $\mathcal{I}$, and unless $\mathcal{P}$ cooperates with $\mathcal{I}$ during the recovery, $\mathcal{P}$ cannot be externally recovered. From this we can see that either the system behavior must be restricted (e.g. it have to follow more restrictions than piecewise-determinism), and/or it must expose more information about its behavior and the structure. Algorithms which do not follow either of those methods, must be incorrect.

Remark 1. If $\mathcal{P}$ does not expose internal information and does not cooperate with $\mathcal{I}$, and if processes in $\mathcal{P}$ work under a PWD model, then external recovery of $\mathcal{P}$ is impossible.

Variables used in a description:
queue<messages> Q :: *Queue of input messages*
queue<messages> L :: *Queue of messages sent to* $\mathcal{P}$
queue<messages> Out :: *Queue of output messages*
enum<**ready**,**busy**> $state \leftarrow$ **ready** :: *State of the interceptor*
integer $count$ :: *Number of outputs for last input*

Upon receiving message M **from** $\mathcal{W}$ **at** $\mathcal{I}$
1: $Q \leftarrow Q \cup M$ :: *M is appended at the end of Q*

When $Q \neq \emptyset \wedge state =$ **ready**
2: $state \leftarrow$ **busy**
3: $M \leftarrow Q.front$
4: $count \leftarrow 0$
5: **send** M **to** $\mathcal{P}$

Upon detecting a restart in $\mathcal{P}$ **at** $\mathcal{I}$
:: *Event is fired when restart of* $\mathcal{P}$ *is finished*
6: $Q \leftarrow L \cup Q$ :: *The* $\cup$ *operator prepends L to Q*
7: $state \leftarrow$ **ready**

Upon receiving message M^o **from** $\mathcal{P}$ **at** $\mathcal{I}$
8: **if** $M \notin Out$ **then**
9: $Out \leftarrow Out \cup M^o$
10: **forward** M^o **to** $\mathcal{W}$
11: **else**
12: **discard** M^o :: *discarding duplicates*
13: **end if**
14: **increment** $count$
15: **if** $count = k_M$ **then**
16: $M \leftarrow Q.front$
17: $Q \leftarrow Q \setminus M$
18: $state \leftarrow$ **ready**
19: **if** $M \notin L$ **then**
20: $L \leftarrow L \cup M$
21: **end if**
22: **end if**

Fig. 1. Algorithm 1, requiring no cooperation with $\mathcal{P}$, restricting $\mathcal{P}$'s behavior only

We will now demonstrate the possibility of external recovery with restricting only one aspect of $\mathcal{P}$'s independence, with no cooperation between $\mathcal{I}$ and $\mathcal{P}$ during the recovery and without requiring $\mathcal{P}$ to expose any information to $\mathcal{I}$. As a reminder, we require that after a single crash in $\mathcal{P}$, all processes restart and $\mathcal{I}$ is notified by failure detector restart is completed.

We impose the following restrictions on the $\mathcal{P}$'s behavior: (R1) Within each session, processes in $\mathcal{P}$ are send-deterministic. Note (R1) applies only to receive events within each session, not to input events. (R2) After k_M-th message in a reaction to input M is sent to $\mathcal{I}$, no events truly caused by M may occur in $\mathcal{P}$, where $k_M > 0$ is known a priori and may differ for every M.

The interceptor $\mathcal{I}$ maintains three FIFO message queues Q, L and Out, and a variable *state*, set initially to *ready* (fig. 1). Arriving inputs are appended at the end of Q. While in *ready* state, $\mathcal{I}$ continuously checks a condition $Q \neq \emptyset$. If $Q \neq \emptyset$, *state* is set to *busy* and $\mathcal{I}$ sends the message M in front of Q to $\mathcal{P}$. When an output M^o arrives at $\mathcal{I}$, it is discarded if it is already in an output queue Out, otherwise $\mathcal{I}$ stores M^o in Out before forwarding it to $\mathcal{W}$. If this is k_M-th output in reaction to M, M is removed from Q and appended to L queue. Finally, $\mathcal{I}$ sets *state* to *ready*. When a crash and restart of $\mathcal{P}$ is detected, interceptor prepends

all messages in L at the beginning of Q and sets *state* to *ready*. Processes in $\mathcal{P}$ do not distinguish between messages sent during normal operation or recovery.

Theorem 1. *For any history H with a crash, the recovery of $\mathcal{P}$ is correct*

Proof. From R1), the send events are determined only by the ordering of the input events in H and the initial state. The algorithm ensures that after receiving an input M, new input may arrive to $\mathcal{P}$ only when $\mathcal{I}$ receives k_M outputs. From R2), at that point the session of M has ended and the processes in $\mathcal{P}$ have already sent all messages forming a reaction to M, so new inputs cannot impact a reaction to M. It follows that a reaction to M is determined solely by the order in which $\mathcal{I}$ sends inputs. In case of crash all processes restart from the initial state and $\mathcal{I}$ resends inputs in an exactly the same order as before the crash, so after the recovery $\mathcal{P}$ produces the reactions possible if the crash would not occur in H. Some of the messages forming reaction to M produced by $P_i \in \mathcal{P}$ (outputs) may be duplicates of the messages sent before the crash – due to our assumptions, those are the only duplicates which our algorithm must handle. Those duplicated outputs are discarded by $\mathcal{I}$. We conclude that the reactions observed by $\mathcal{W}$ would be possible if a crash would not occur in H.

Constraints on the $\mathcal{P}$'s behavior could be understood both as expressing which histories are possible, and which possible histories with different event ordering lead to the same $\mathcal{P}$'s global state (i.e. which histories are equivalent with respect to $\mathcal{P}$'s state). For example, ordering of receive events matters in PWD, but not in the send-deterministic model. If a constraint C allows for all histories allowed by C' and in addition at least one history not allowed by C', then C' is more strict than C. Similarly, if under C histories H and H' are equivalent (despite having different ordering of the events), while under C' they are not, C is more strict than C' with respect to event ordering. Full determinism is more strict (in both senses) than PWD, PWD is more strict than send-determinism and so on.

We will now prove that the constraints on $\mathcal{P}$'s behavior are minimal in the sense that a deterministic external recovery (producing the same outputs in the same order) is possible without requiring cooperation and/or more information from $\mathcal{P}$, only when a constraint w.r.t. event ordering is at least as strict as R1)[2], and R1) is not enough unless a constraint at least as strict as R2) is imposed.

Theorem 2. *The restrictions imposed on a $\mathcal{P}$'s behavior are minimal.*

Proof. Minimality of R1). Assume that send events may be impacted not just by the ordering of the input events, but in addition there is a non-deterministic event e and depending on whether a history H contains e, or depending on ordering of e with respect to other events, it is possible that H will produce different outputs, say either M^o or $M^{o\prime}$ (not necessarily a reaction to the same M). Assume that e occurred before a crash, producing M^o. After the recovery e may not happen or may be ordered differently, producing $M^{o\prime}$. Since M^o

[2] Channel-determinism[21] would produce the same outputs, but possibly in different order, which may or might not matter from the point of $\mathcal{W}$.

and $M^{o\prime}$ are different, $\mathcal{I}$ would not discard one of them as a duplicate, causing $\mathcal{W}$ to observe both – impossible in a history without a crash. That could be prevented only by replaying e, which would require preserving its determinants. Since processes have no stable storage, determinants could be preserved only by sending them to $\mathcal{I}$ (exposing information) and during the recovery the processes would have to know how to use that information (requiring cooperation).

Minimality of R2). Assume that R1) holds and while there still may occur the events truly caused by M, a new input M' arrives. Depending on the ordering of the event of receiving M' with respect to the events truly caused by M, a history may contain an event of sending either M^o or $M^{o\prime}$, but not both. By reasoning analogous as with the discussion of R1) we conclude that the correct recovery requires exposing information to $\mathcal{I}$ and a cooperation with $\mathcal{I}$. Therefore, new input must arrive into $\mathcal{P}$ only when no events truly caused by the previous input will occur. Interceptor cannot determine this if $\mathcal{I}$ does not not know a priori the number of outputs, unless processes in $\mathcal{P}$ expose information to $\mathcal{I}$ (e.g. by attaching tags to outputs, to notify $\mathcal{I}$ whether the output is the last one).

4 Restricting the Behavior, Exposing the Information

The solution analysed in previous section is mostly of theoretical value. Obviously it would severely limit the performance of most applications, as it forces $\mathcal{P}$ to process all inputs serially. If $\mathcal{P}$ could process n inputs in parallel, both recovery and normal processing would be at least n times slower. This would be acceptable for applications where processing is done in request-reply manner, requiring cooperation of all processes in $\mathcal{P}$, or with applications where size of $\mathcal{P}$ is limited. Allowing inputs to be processing in parallel would both increase the performance, the potential size of $\mathcal{P}$, and the number of cases where external recovery would be applicable. The approach presented below, based on our previous work[5], demonstrates trade-offs inherent in the external recovery: to achieve higher performance, we must put more constraints on $\mathcal{P}$'s independence.

Let $\mathcal{S}$ be a set of all sessions in $H(t)$, and let $\forall e \in H(t), \exists S \in \mathcal{S} : e \in S$. Set of sessions $\mathcal{S} \subset H(t)$ is serializable if there exists a total order relation $\xrightarrow{s}$ in $\mathcal{S}$, preserving $\mapsto$ relation ($\forall e \in S, \forall e' \in S' : e \mapsto e' \Rightarrow S \xrightarrow{s} S'$). Each session $S \in \mathcal{S}$ has a unique session identifier $S.id$. Slightly abusing the notation, we will say that a message m belongs to a session S if the event of sending m belongs to S.

We restrict the $\mathcal{P}$'s behavior as follows: (R1) processes in $\mathcal{P}$ are send-deterministic within each session. (R3) The sessions are serializable. (R4) For each input M, number of outputs $k_M > 0$ in a reaction to M is known a priori (note R4 is less strict version of R2). (R5) $\forall P_i \in \mathcal{P}$, when P_i receives a message from session S, it must eventually send at least one message (possibly, an output) within S.

$\mathcal{P}$ expose the information about how the sessions were serialized: each message m (including each output) has the session identifier $m.sId$ and the ordered set of all the preceding sessions $m.prec$. We assume sessions are serializable.

The interceptor $\mathcal{I}$ maintains sets $\mathcal{M}^{in}$ and $\mathcal{F}$ (fig. 2). Each element $x \in \mathcal{M}^{in}$ represents a session and has five fields: an input msg, an input's session identifier

sId, a set of preceding session identifiers $prec$, a set of outputs out (a reaction to msg), and a boolean fwd. When an input M arrives at $\mathcal{I}$, $\mathcal{I}$ creates an element x with $x.msg \leftarrow M$, sets $M.sId$ and $x.sId$ to a new unique session identifier sId, $x.prec \leftarrow \mathcal{F}$, $x.out \leftarrow \emptyset$ and $x.fwd \leftarrow false$. The x is then added to $\mathcal{M}^{in}$, and M is send to $\mathcal{P}$.

When an output M^o arrives at $\mathcal{I}$, $\mathcal{I}$ finds an element x in $\mathcal{M}^{in}$ with $x.sId = M^o.sId$. M^o is discarded if $M^o \in x.out$. Otherwise $\mathcal{I}$ appends $M^o.prec$ to a $x.prec$ field, M^o is stripped from $prec$ and sId fields and added to $x.out$. If cardinality of $x.out$ is k_M, then $x.sId$ is added to $\mathcal{F}$. For each $x \in \mathcal{M}^{in}$, the interceptor $\mathcal{I}$ periodically checks a condition $x.sId \in \mathcal{F} \wedge \forall y \in \mathcal{M}^{in} : y.sId \notin \mathcal{F} \Rightarrow x.sId \in y.prec \vee y.sId \in x.prec$. If the condition is true, $\mathcal{I}$ sets $x.fwd \leftarrow true$ and forwards messages in $x.out$ to $\mathcal{W}$.

When $\mathcal{P}$ crashes, $\mathcal{I}$ stops passing messages into $\mathcal{P}$ and creates two sets: $\mathcal{M}'$ contains a copy of all elements $x \in \mathcal{M}^{in}$ such that $x.fwd = true$, while $\mathcal{M}^{oth} = \mathcal{M}^{in} \setminus \mathcal{M}'$ (fig. 3). Then, $\forall x \in \mathcal{M}^{in} : \neg x.fwd \Rightarrow x.out \leftarrow \emptyset$. Next, an element $x \in \mathcal{M}' : x.prec = \emptyset$ is chosen and a message $x.msg$ is sent to $\mathcal{P}$. Interceptor $\mathcal{I}$ then waits until it receives k_M outputs, discarding each of them. When this happens, x is removed from $\mathcal{M}'$ and $\forall y \in \mathcal{M}', y \neq x$, $x.sId$ is removed from $y.prec$. Finally, another element with an empty $prec$ set is chosen. If $\mathcal{M}'$ is empty, $\mathcal{I}$ sends all messages from $\mathcal{M}^{oth}$ set to $\mathcal{P}$ (in any order, sequentially or in parallel). Normal execution then resumes.

The information about session serialization is gathered as follows: each process P_i maintains set of session identifiers $prec_i$ and current session identifier sId_i. When P_i receives m such that $m.sId \neq sId_i$, $prec_i \leftarrow prec_i \cup m.prec \cup sId_i$ and $sId_i \leftarrow m.sId$. When P_i sends a message m, the $prec_i$ is added to m as $m.prec$. The set $prec_i$ can be prevented from growing indefinitely with the use of garbage collecting, or by putting stricter constraints on $\mathcal{P}$'s behavior.

Lemma 1. *For every session $S' \neq S$ such that S' has started before S has ended, and for any event $e \in S$ at P_i, e can be replayed during recovery if and only if either no event from S' occurs at P_i, or information of precedence between S' and S with respect to $\mapsto_i$ is preserved at $\mathcal{I}$.*

Proof. From R3), we know that if a process P_i receives message first from a session S and then from a session $S' \neq S$, P_i will receive no further messages from S (P_i sees that locally S precedes S'). Let $\mapsto_i$ denote the session ordering as seen by P_i. Obviously, $S \mapsto_i S' \Rightarrow S \xrightarrow{s} S'$. From R1), a send event e at P_i from S can be impacted only by the previous receive events at P_i, belonging to different sessions, i.e. by the sessions preceding S w.r.t. $\mapsto_i$ relation. To replicate e, we must 1) replay all sessions preceding S before S and 2) make sure that all the sessions preceded by S will be executed after S. If S has ended before S' has started, S' cannot precede S w.r.t. $\mapsto_i$. So, to be sure we can replay e, for every session $S' \neq S$ such that S' has started before S has ended, we must be able to tell that either no event from S' occurs at P_i, or whether $S' \mapsto_i S$ or $S \mapsto_i S'$. This information must be then preserved at $\mathcal{I}$.

Message type Packet $\langle\rangle$ **is:**
  integer sId
  set<integer> prec
  message data

Structure Session is:
  message msg :: *Input starting the session*
  integer sId :: *Session identifier*
  set<integer> prec :: *preceding sessions*
  set<message> out :: *Outputs to the session*
  boolean fwd :: *Were outputs forwarded to* $\mathcal{W}$

Variables used in a description:
  set<**Session**> $\mathcal{M}^{in}$ :: *Sessions*
  set<integer> $\mathcal{F}$ :: *Finished sessions*
  set<**Session**> $\mathcal{M}\prime$ :: *Part of* $\mathcal{M}^{in}$
  set<**Session**> $\mathcal{M}^{oth}$ :: *Part of* $\mathcal{M}^{in}$
  Session x :: *Single session element*
  Packet *pkt* :: *Packets used within* $\mathcal{P}$
  enum<**ready**,**busy**> *mode* $\leftarrow$ **ready**

Upon receiving message M **from** $\mathcal{W}$ **at** $\mathcal{I}$
1: $x.msg \leftarrow M$
2: $x.sId \leftarrow$ **new unique session identifier**
3: $x.prec \leftarrow \emptyset$, $x.out \leftarrow \emptyset$
4: $x.fwd \leftarrow$ **false**
5: $\mathcal{M}^{in} \leftarrow \mathcal{M}^{in} \cup x$
6: $pkt.data \leftarrow M$
7: $pkt.sId \leftarrow x.sId$
8: $pkt.prec \leftarrow \emptyset$
9: **wait until** $mode =$ **ready**
10: **send** *pkt* **to** $\mathcal{P}$

Upon receiving message M^o **of type Packet from** $\mathcal{P}$ **at** $\mathcal{I}$
11: $x \leftarrow \{x : x \in \mathcal{M}^{in} \wedge x.sId = M^o.sId\}$
12: $\mathcal{M}^{in} \leftarrow \mathcal{M}^{in} \setminus x$
13: **if** $M^o.data \notin x.out$ **then**
14: $x.prec \leftarrow x.prec \cup M^o.prec$
15: $x.out \leftarrow x.out \cup (M^o.data)$
16: $\mathcal{M}^{in} \leftarrow \mathcal{M}^{in} \cup x$
17: **if** $|x.out| = k_M$ **then**
18: $\mathcal{F} \leftarrow \mathcal{F} \cup x.sId$
19: **end if**
20: **else**
21: **discard** M^o
22: **end if**

When $\exists x \in \mathcal{M}^{in} : x.sId \in \mathcal{F} \wedge x.fwd =$ **false** $\wedge \forall y \in \mathcal{M}^{in}, y.sId \notin \mathcal{F} \Rightarrow x.sId \in y.prec \vee y.sId \in x.prec$

23: $x.fwd \leftarrow$ **true**
24: **foreach** $msg \in x.out$ **do**
25: **send** *msg* **to** $\mathcal{W}$
26: **end for**

Fig. 2. Algorithm 2, part 1: failure-free execution

Upon detecting a restart in $\mathcal{P}$ at $\mathcal{I}$
:: *Event is fired when restart of $\mathcal{P}$ is finished*

```
1: mode ← busy
2: M′ ← {x ∈ M^in : x.fwd = true}
        :: Copy of elements in M^in
3: M^oth ← M^in \ M′
4: foreach x ∈ M^in do
5:     if x.fwd = true then
6:         x.out ← ∅
7:     end if
8: end for
9: while M′ ≠ ∅ do
10:    foreach x ∈ M′ do
11:        if x.prec = ∅ then
       :: At least one element with
       :: empty prec field must be in
       :: M′ due to (R4)
       :: restriction on P's behavior
12:            break
13:        end if
14:    end for
15:    pkt.msg ← x.msg
16:    pkt.prec ← ∅
17:    pkt.sId ← x.sId
18:    send pkt to P
19:    for i ∈ {1..k_M} do
20:        receive pkt from P
21:        discard pkt
22:    end for
23:    M′ ← M′ \ x
24:    foreach y ∈ M′ do
25:        y.prec ← y.prec \ x.sId
26:    end for
27: end while
28: foreach x ∈ M^oth do
29:    pkt.msg ← x.msg
30:    pkt.prec ← ∅
31:    pkt.sId ← x.sId
32:    send pkt to P
33: end for
34: mode ← ready
```

Fig. 3. Algorithm 2, part 2: recovery

Lemma 2. *For some $x \in \mathcal{M}^{in}$ and $S \in \mathcal{S}$, let S.prec be x.prec where $x.sId = S.id$. When a k_M-th output M^o from S arrives to $\mathcal{I}$, for each $S' \in \mathcal{S}$ such that S' has not started after S has ended, $S'.id \in S.prec \iff \exists P_i \in \mathcal{P} : S' \mapsto_i S$.*

Proof. When P_i receives a message from a session S', $sId_i \leftarrow S'.id$. Later, when P_i receives a message from a session S, S' is added to the $prec_i$ ($S' \mapsto_i S \wedge sId_i = S.id \Rightarrow S'.id \in prec_i$). Before sending a message m within S, P_i sets $m.prec \leftarrow prec_i$, so $S' \mapsto_i S \wedge m.sId = S.id \Rightarrow S'.id \in m.prec$. When P_j receives a message m from P_i it adds $m.prec$ to $prec_j$. By recursion, it follows that for every m (including every output) $m.sId = S.id \wedge S'.id \in m.prec \Rightarrow \exists P_i \in \mathcal{P} : S' \mapsto_i S$. Since $S.prec$ is a sum of all outputs' $prec$ fields, this concludes the if part. From R5), at least one message is sent by P_i within a session S, containing $prec_i$, either to $\mathcal{W}$ or to some P_j. Since our system models assumes all sessions are finite, and by R4) number of outputs is finite, eventually there must be an output (at most k_M-th) including ordering of S' and S w.r.t. $\mapsto_i$ (only if part).

Theorem 3. *For any history H with a crash, the recovery of $\mathcal{P}$ is correct*

Proof. Let $x \in \mathcal{M}'$, and let $S.prec$ be $x.prec$ where $x.id = S.id$. Let $\xrightarrow{p}$ will be a relation on $\mathcal{S}$, such that if $S' \in S.prec \Rightarrow S' \xrightarrow{p} S$. Algorithm ensures that no output M^o from session S (reaction to S) is send to $\mathcal{W}$ until for all other session

S', $S' \in \mathcal{F} \vee S' \xrightarrow{p} S \vee S \xrightarrow{p} S'$. Assume $S' \in \mathcal{F}$ and $\neg(S' \xrightarrow{p} S \vee S \xrightarrow{p} S')$. From R4) and R5), no new events within S' will happen. If any event in S' would occur at P_i, then by R5) and lemma 2), $S' \xrightarrow{p} S \vee S \xrightarrow{p} S'$, so we conclude that no event from S' occurred at P_i.

Assume $S' \xrightarrow{p} S \vee S \xrightarrow{p} S'$. From lemma 2) we conclude that $S' \xrightarrow{p} S \iff \exists P_i \in \mathcal{P} : S' \mapsto_i S$ (resp. $S \xrightarrow{p} S' \iff \exists P_i \in \mathcal{P} : S \mapsto_i S'$), or S started after S' has ended. From that and from lemma 1) we conclude that to replicate reaction to S, it is enough to replay all the sessions preceding S with respect to $\xrightarrow{p}$. It is easy to see that algorithm does exactly that during the recovery. After that, inputs for which no output was forwarded to $\mathcal{W}$ may be send to $\mathcal{P}$ in any order; from lemma 1) it is clear that any such input couldn't impact any output forwarded to $\mathcal{W}$ before the crash, and reaction to any such input could be impacted by sessions from $\mathcal{M}'$ in a history without a crash.

After the recovery all sessions which produced outputs forwarded to $\mathcal{W}$ before a crash will be replayed while outputs which could not be replicated and duplicates will be discarded. Any additional output forwarded to $\mathcal{W}$ could also appear if a crash would not occur in H. We conclude that a reaction to any input M in H could occur in a history without a crash.

5 Related Work

In the context of message-passing systems, there are two general techniques for system recovery: checkpointing and message logging (See [15] for survey). Checkpointing may be done in a coordinated manner [13] or independently. In the latter case a domino effect may appear [19], leading researchers to propose many communication-induced protocols: index-based [25] or model-based [22,17].

Message logging may be pessimistic, optimistic or causal. With pessimistic logging event determinants are saved immediately into the stable storage [3]. The performance penalty of this approach is avoided by optimistic logging where determinants may be stored in volatile memory before being written to the stable storage [11]. Optimistic approach complicates recovery. Causal logging tries to combine advantages of both approaches [1]. Send-determinism is a promising model of system behavior, allowing new, efficient protocols [8,9]. Message logging was also studied in the context of the SOA systems [2] and Distributed Shared Memory [18].

The independent recovery of large system's sub-components was analyzed in the context of cluster federations, in which processes/nodes could be statically assigned to the independent, non-overlapping groups [16,10], may be divided into groups using either the code analysis or based on their behavior [20,14], or by residing on single, multi-core machine [4]. Usually coordinated checkpoint is used within a group, while only messages exchanged between the groups are logged [9].

6 Conclusions

This paper is a first step in an exploration of the external recovery problem. We presented two algorithms allowing the system to be recovered using the messages logged by an intercepting layer. The algorithms differ in the kind of restrictions imposed on a system, serving as an illustration of the tradeoffs involved when a system is to be recovered by external message logging: one must either expose more system information and agree to cooperate more with intercepting layer, or impose a stricter constraints on possible system behaviors.

Our work could be extended in several ways. First, the externally recoverable systems offer a possibility of building large-scale systems where each subsystem could be dynamically modified or replaced, with different internal fault-handling logic. Second, it would be interesting to investigate other ways in which system's behavior could be restricted, especially if there exists other set minimal restrictions allowing system's external recovery. Third, a similar work could be carried on the questions of minimal information needed by external recovery mechanism given particular restriction on system's behavior. Finally, important part of future work is experimental analysis of performance of presented solutions, and of their scaling characteristics.

References

1. Alvisi, L., Marzullo, K.: Message logging: Pessimistic, optimistic, causal, and optimal. Software Engineering 24(2), 149–159 (1998)
2. Barga, R.S., Lomet, D.B., Shegalov, G., Weikum, G.: Recovery guarantees for internet applications. ACM Trans. Internet Techn. 4(3), 289–328 (2004)
3. Bouteiller, A., Cappello, F., Hérault, T., Krawezik, G., Lemarinier, P., Magniette, F.: MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In: SC, p. 25. ACM (2003)
4. Bouteiller, A., Hérault, T., Bosilca, G., Dongarra, J.J.: Correlated set coordination in fault tolerant message logging protocols for many-core clusters. Concurrency and Computation: Practice and Experience 25(4), 572–585 (2013)
5. Brzeziński, J., Danilecki, A., Hołenko, M., Kobusińska, A., Kobusiński, J., Zierhoffer, P.: D-reServE: Distributed reliable service environment. In: Morzy, T., Härder, T., Wrembel, R. (eds.) ADBIS 2012. LNCS, vol. 7503, pp. 71–84. Springer, Heidelberg (2012)
6. Cappello, F., Guermouche, A., Snir, M.: On communication determinism in parallel HPC applications. In: 2010 Proceedings of 19th International Conference on Computer Communications and Networks (ICCCN), pp. 1–8 (2010)
7. Elnozahy, E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (2002)
8. Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F.: Uncoordinated checkpointing without domino effect for send-deterministic message passing applications. In: Accepted to the 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS (May 2011)

9. Guermouche, A., Ropars, T., Snir, M., Cappello, F.: HydEE: Failure containment without event logging for large scale send-deterministic mpi applications. In: 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS), pp. 1216–1227 (2012)
10. Gupta, B., Rahimi, S., Allam, V., Jupally, V.: Domino-effect free crash recovery for concurrent failures in cluster federation. In: Wu, S., Yang, L.T., Xu, T.L. (eds.) GPC 2008. LNCS, vol. 5036, pp. 4–17. Springer, Heidelberg (2008)
11. Johnson, D., Zwaenepoel, W.: Recovery in distributed systems using optimistic message logging and checkpointing. J Algorithms 11, 462–491 (1990)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
13. Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., Cappello, F.: Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In: CLUSTER 2004: Proceedings of the 2004 IEEE International Conference on Cluster Computing, Washington, DC, USA, pp. 115–124 (2004)
14. Luo, Y., Manivannan, D.: Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems. Future Generation Comp. Syst. 28(8), 1217–1235 (2012)
15. Maloney, A., Goscinski, A.: A survey and review of the current state of rollback-recovery for cluster systems. Concurrency and Computation: Practice and Experience 21(12), 1632–1666 (2009)
16. Monnet, S., Morin, C., Badrinath, R.: A hierarchical checkpointing protocol for parallel applications in cluster federations. In: IPDPS (2004)
17. Netzer, R.H.B., Xu, J.: Necessary and sufficient conditions for consistent global snapshots. IEEE Transactions on Parallel and Distributed Systems 6(2), 165–169 (1995)
18. Park, T., Lee, I., Yeom, H.Y.: An efficient causal logging scheme for recoverable distributed shared memory systems. Parallel Computing 28(11), 1549–1572 (2002)
19. Randell, B.: System structure for software fault tolerance. IEEE Transactions on Software Engineering 1(2), 221–232 (1975)
20. Ropars, T., Guermouche, A., Uçar, B., Meneses, E., Kalé, L.V., Cappello, F.: On the use of cluster-based partial message logging to improve fault tolerance for MPI HPC applications. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 567–578. Springer, Heidelberg (2011)
21. Ropars, T., Martsinkevich, T.V., Guermouche, A., Schiper, A., Cappello, F.: Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2013, pp. 8:1–8:12. ACM, New York (2013)
22. Russell, D.L.: State restoration in systems of communicating processes. IEEE Trans. Software Eng. 6(2), 183–194 (1980)
23. Storm, R., Yemini, S.: Optimistic recovery in distributed systems. ACM Trans. Comput. Syst. 3(3), 204–226 (1985)
24. Tarafdar, A., Garg, V.K.: Addressing false causality while detecting predicates in distributed programs. In: Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS 1998), pp. 94–101 (1998)
25. Tsai, J.: An efficient index-based checkpointing protocol with constant-size control information on messages. IEEE Trans. Dependable Sec. Comput. 2(4), 287–296 (2005)