HAL
open science

A Fault Tolerance protocol for ASP calculus: Design and Proof

Françoise Baude, Denis Caromel, Christian Delbé, Ludovic Henrio

To cite this version:

Françoise Baude, Denis Caromel, Christian Delbé, Ludovic Henrio. A Fault Tolerance protocol for ASP calculus: Design and Proof. [Research Report] RR-5246, INRIA. 2004, pp.37. inria-00070752

HAL Id: inria-00070752

https://inria.hal.science/inria-00070752v1

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Fault Tolerance protocol for ASP calculus: Design and Proof

Françoise Baude — Denis Caromel — Christian Delbé — Ludovic Henrio

N° 5246

June 2004

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5246--FR+ENG

A Fault Tolerance protocol for ASP calculus: Design and Proof

Françoise Baude , Denis Caromel, Christian Delbé , Ludovic Henrio

Thème COM — Systèmes communicants
Projet Oasis

Rapport de recherche n° 5246 — June 2004 — 37 pages

Abstract: This research report first details a communication induced checkpointing fault tolerance protocol adapted to ProActive, a Java library that implements the ASP model. This model is based n a request/reply mechanism.

In order to prove the correctness of this protocol, we introduce a *local partial order* between events occurring on a given process. This order is extended into a global order by the Lamport's happened-before relation. Finally, we prove that from a cut that is "consistent enough", a second execution is constrained to go equivalently to the first one until a consistent global state of the first one (the history closure).

Thus, the protocol described in this report ensures that, even from a inconsistent recovery line, a reexecution cannot lead to an inconsistent state that could not exist in a normal execution.

Key-words: fault tolerance, checkpointing, message logging, causality relation

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Un protocole de tolérance aux pannes pour ASP : conception et preuve

Résumé : Ce rapport de recherche fait tout d'abord une présentation détaillée d'un protocole de tolérance aux pannes par points de reprise induits par message adapté à ProActive, une librairie Java implémentant le modèle ASP. Ce modèle est basé sur un mécanisme de requêtes/réponses.

Dans le but de prouver la correction de ce protocole, nous introduisons un *ordre partiel local* entre les évènements qui ont lieu sur un processus donné. Cet ordre est étendu en un ordre global grâce à la relation "happened-before" de Lamport. Nous prouvons finalement qu'à partir d'une coupe "suffisamment cohérente", une seconde exécution est contrainte de se dérouler de façon équivalente à la première jusqu'à un état global cohérent de la première (la cloture de l'historique).

Ainsi, le protocole présenté garantit que, même à partir d'une ligne de recouvrement non cohérente, une réexécution ne peut pas mener à un état inconsistent qui ne pourrait pas exister dans une exécution normale.

Mots-clés : tolérance aux pannes, point de reprise, journalisation de messages, relation de causalité

Contents

I A fault Tolerance protocol **5**

1 An Active Object Model **6**
1.1 Communication . . . 6
1.2 Rendez-vous . . . 7
1.3 Properties and Assumptions . . . 7
1.4 Stable States . . . 8

2 Basic Elements of the Protocol **8**
2.1 Checkpoints . . . 9
2.2 Promised Requests . . . 9
2.3 Avoid Orphan Messages . . . 10
2.3.1 Request $Q_{i,j}$ . . . 10
2.3.2 Reply $R_{i,j}$ . . . 10
2.4 Avoid In-transit Messages . . . 11
2.5 Consecutive and Impossible Checkpoints . . . 11

3 Request Reception History **12**
3.1 Problems . . . 12
3.1.1 Duplicated Replies . . . 12
3.1.2 Causal Dependencies . . . 13
3.2 Solution: Request Reception History . . . 14
3.3 Closing Request Reception History . . . 15

4 Recovery **16**
4.1 Incarnation Numbers . . . 16

5 Algorithmic Description **17**
5.1 Principles . . . 17
5.2 Algorithms . . . 17

II Correctness Proof **20**

6 Causality relation for asynchronous request services **20**
6.1 Elements of event-based analysis . . . 21
6.1.1 Characterizing Executions . . . 21
6.1.2 Partial Local Order . . . 21
6.1.3 Cuts . . . 21
6.1.4 $\oplus$ operator . . . 22
6.2 Causality in ASP . . . 23

6.2.1 Local causality relation for ASP 24
6.2.2 Piecewise Determinism 26
6.2.3 Consistent Enough Cuts 26

7 Proving correctness of the protocol 26
7.1 Promised Receptions 27
7.2 $\mathcal{P}$-cut 27
7.3 Consistent $\mathcal{P}$-cuts and constrained execution 29
7.4 Recovery and Correctness 30
7.4.1 Recovery Cut 30
7.4.2 Logged Messages 31
7.4.3 Correctness of the protocol 31

8 Relation between ASP, ProActive and this work 33

Part I
A fault Tolerance protocol

This first part presents a fault tolerance protocol for distributed object middleware. This protocol has been implemented within ProActive [6], an open source Java middleware for asynchronous and distributed objects implementing the ASP (Asynchronous Sequential Processes) model [7].

Many fault tolerance protocols for distributed systems have already been proposed. Three main approaches can be identified for rollback-recovery fault tolerance [14]: coordinated checkpointing [9, 13], communication-induced checkpointing (CIC) [5, 21, 17, 22, 29, 16], and message logging [28, 1, 4]. In this work, we are interested in applications running on heterogeneous systems with low failure rate; thus we first came to a CIC approach because it is usually characterized by a rather low overhead for fault free executions.

CIC protocols usually make the assumption that every process of the system can be checkpointed at any time, to ensure consistency of recovery lines. But this assumption can fail for complex or particular systems where the processes' state is not always available. In the context of Java middlewares like ProActive, persistence can be obtained in a convenient and portable way by standard Java serialization. However, as threads cannot be serialized, an important part of the activities[1], the stacks, cannot be checkpointed.

Threads persistence can be achieved by modifying the execution environment at the OS level [24] or at the virtual machine level [18, 3]. Another solution is to use a specific compiler which adds code to capture enough informations to characterize the state of a process [25], or use compile-time reflection to provide persistence functions [26]. But those tools usually involve a loss of portability and/or efficiency. In the context of Java, it is rather unfortunate to lose portability.

A portable and convenient solution for taking advantage of CIC protocols when checkpoints can only be taken at some program points is presented in this paper. In our approach, we counterbalance the unserialisability of threads by the fact that, at some program points, the state of the activity is fully characterized without any knowledge about the state of its thread. As will be shown, our model allows us to identify such program points, called *stable states* in the following. As these stable states occur in a restricted and unpredictable way, we designed an *hybrid CIC-message logging* protocol that allows recovery from global states made of restrictively placed checkpoints. More generally, our protocol provides CIC-based fault tolerance in middlewares where the checkpointable program points are specified either explicitely by the executed program or implicitely due to the middleware model.

Starting with a CIC protocol, message logging mechanisms have been added to deal with unpredictable checkpoint locations. Overall, the contribution is a new mixed approach between CIC and selective message logging. The proposed protocol delays checkpoints during the execution, mainly thanks to a *request reception history*. *Promised requests* allow to log only request senders identifiers and to perform fully asynchronous recovery. An

[1] We prefer the term activity rather than process to identify the runtime entity.

implementation has been achieved within ProActive. It allows us to present first benchmarks so as to evaluate overhead induced by our protocol.

This part is organized as follows. Section 1 presents the active object model being used. Section 2 summarizes the main principles of the protocol. Section 3 introduces and resolves the problem of equivalence between different executions. Section 4 describes the recovery protocol. Section 5 gives the algorithms of the protocol.

1 An Active Object Model

Our protocol has been developed in the context of the ASP object calculus and more specifically for the ProActive library that implements this calculus. ASP object calculus is based on concurrent activities. Communications are asynchronous method calls with transparent futures, based on a message passing mechanism with requests and replies. Each activity consists of a process, a set of objects (which we call its *applicative state*), and a request queue. There is a master object among the applicative state that is called the *active object*; more precisely every request sent to an activity is actually an asynchronous method call on this active object.

1.1 Communication

When an activity calls a method on an active object, a new request is added to the request queue of the receiver. When the signature of the called method has a return value, a future is created on the sender side: this future represents the result of the request that is not known yet. Futures are generalized references that can be manipulated as classical objects. However, some operations (e.g. field access) need a real object value to be performed. Performing such operations on future objects leads to a blocked state called *wait-by-necessity*. When the receiver ends the service of a request, the associated future can be updated: it sends a reply that will take the place of the future.

We use the following notations:

- $Q_{i,j}$ a request from i to j,
- $R_{i,j}$ a reply from i to j,
- $M_{i,j}$ a message from i to j, ($M_{i,j}$ is $Q_{i,j}$ or $R_{i,j}$),
- $\mathbb{Q}_i^{rcv}$ the request queue of i,
- $X(Q_{i,j})$ the service of the request $Q_{i,j}$.

The activity identities i and j can be omitted when there is no ambiguity, and a request or a reply can be identified with an index (Q^n or R^n).

Figure 1 shows two activities i and j. j calls a method on i: a request is sent (Q). Eventually, this request is served on i ($X(Q)$) and a reply (R), result of the service, is sent from i to j. Once received, this result transparently updates the future object.

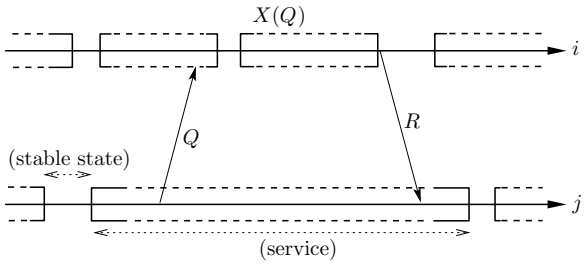

Figure 1: Communicating activities

The impact of a message reception on an activity is different depending on the kind of the message. On one hand, a request reception modifies only the request queue of the receiver until it serves the request: the applicative state is not altered while this request is pending. On the other hand, a reply reception can change the applicative state of the receiver (modification which is moreover irreversible). This difference regarding the impact of the reception of a message is shown on Figure 1 through the endpoints of the arrows. A request does not come through dotted lines rectangles, but a reply does.

1.2 Rendez-vous

In our model, communications are asynchronous, but there is a short *rendez-vous* at the beginning of the communication. When an activity sends a message to another, it stops its execution until the message is in the context of the receiver. The rendez-vous implies the two following properties:

- communications are FIFO point-to-point,
- communications are acknowledged.

1.3 Properties and Assumptions

In [7], the ASP calculus allowed to prove the two following main properties:

Property 1.1 *The order of reception of replies during a distributed execution has no consequence on the behavior of the program, assuming that no deadlock is caused by wait-by-necessity.*

Property 1.2 *An execution can be characterized only by the ordered lists (one for each activity) of request sender identifiers.*

We also make the following assumptions regarding the system and the activities:

- activities are fail-stop [27],
- failures are detected in an arbitrary but finite time [8],
- an available host always exists, in order to restart a failed activity,
- a stable storage, known by each activity, exists in order to save checkpoints,
- there is a process, called the *recovery process*, which has access at any time to the stable storage and knows the number of activities of the system.

1.4 Stable States

In ASP, an activity is in a *stable state* when it does not serve any request. Indeed, between two request services, the applicative state and the pending request queue are sufficient to fully characterize the state of the activity. Consequently, the stable state of an activity can be recorded through standard Java serialization of the applicative state (starting at the active object) and of the pending request queue; there is no need to serialize the thread, and a checkpoint can thus be taken.

On Figure 1, a rectangle drawn using dotted lines schematizes the period during which the activity is serving a request, and thus is not in a stable state. Conversely, a period during which the activity is in a stable state is represented by a simple line.

2 Basic Elements of the Protocol

The proposed protocol is based on [5] and [21]. On an activity, each checkpoint is identified by a sequence number, which increases monotonously. We denote $N_i^{current}$ the latest checkpoint number of the activity i. The receiver of a message $M_{i,j}$ *should be forced to take a checkpoint* if the sender's latest checkpoint number piggybacked by $M_{i,j}$, denoted $N_{\overrightarrow{M_{i,j}}}$, is greater than that of the receiver. If forced checkpoints could be taken at any time, then a global state composed of checkpoints with the same number is guaranteed to be consistent. Since checkpoints can only occur when the activity is in a stable state and since the occurrences of stable states are unpredictable, an additional mechanism is necessary.

Each activity has a checkpointing time counter, denoted TTC_i, which allows the activity to periodically take a checkpoint, in order to ensure the regular formation of recovery lines. This time counter is initialized *each time a checkpoint is taken* with a value denoted by TTC_INIT.

In this paper, a message logging mechanism is added to tolerate restrictively placed checkpoints. Moreover, the proposed protocol differs from [5] and [21] by the lost of *process autonomy*, as defined in [2]: an activity never decides itself to take a checkpoint. When a stable state is reached, the protocol can trigger a checkpoint due to a previous message reception, or due to elapsed checkpointing timer counter. We will see that this sacrifice of a certain degree of process autonomy induces a consequent reduction of the total number of checkpoints.

2.1 Checkpoints

Figure 2 shows a checkpoint C_i^n on an activity i. The graphical representation includes its sequence number (n), the current request queue $\mathbb{Q}_i^{rcv}$ ($[Q^1, Q^3, Q^4]$) , and an additional queue called the *resend queue* ($]Q^2, R^5[$) (see Section 2.4). If a queue is empty, it is denoted by $[\emptyset]$ or $]\emptyset[$. Note that incoming messages are blocked during a checkpoint; their reception is postponed *after* the checkpoint.

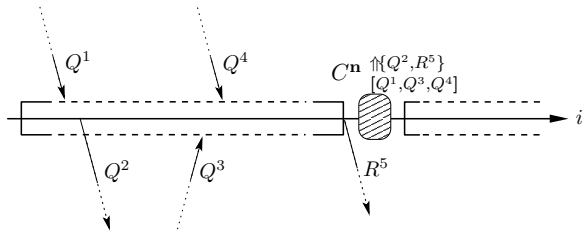

Figure 2: Checkpoint on an activity

We now call *first execution (from C_i^n)* the execution of an activity i after the checkpoint C_i^n is taken, and *reexecution (from C_i^n)* the execution of i after a recovery from C_i^n.

2.2 Promised Requests

A promised request is a local[2] substitute for a request that is not yet received in the reexecution; it only contains the identity of the activity from which a request is awaited. It allows the protocol to perform additional lazy synchronization in case of recovery. A promised request awaited from i in the request queue of j is denoted by $Q_{i,j}^{promised}$. The service of a promised request is subject to synchronization through a wait-by-necessity mechanism: when an activity j tries to serve a promised request $Q_{i,j}^{promised}$, it is blocked until this promised request is updated. So, when an activity j receives a request $Q_{i,j}$ from i:

- if j is blocked on a promised request (with the same sender i) $Q_{i,j}^{promised}$, then the wait-by-necessity on j is released and j serves $Q_{i,j}$,
- if j is not blocked, but there is one or more promised requests (with the same sender i) $Q_{i,j}^{promised}$ in its request queue, then the oldest $Q_{i,j}^{promised}$ is updated with $Q_{i,j}$,
- else $Q_{i,j}$ is enqueued in the request queue of j.

To summarize, a promised request is a place holder for a request that will be received after a recovery and has already been received in the first execution. Such requests either correspond to logged requests or to requests sent during the reexecution.

[2] A promised request is *never* sent between activities.

2.3 Avoid Orphan Messages

An activity *cannot be forced* to take a checkpoint; it must wait for the next stable state, then can only react *a posteriori*. Thus, we virtually shift receptions of messages that should force a checkpoint, to the moment when the receiver actually takes a checkpoint. Consequently, the reception of such message, more precisely a message indicating that $N_{\overrightarrow{M_{i,j}}} > N_j^{current}$, triggers an action depending on its kind, request or reply.

2.3.1 Request $Q_{i,j}$

If the received message is a request, this request will be replaced with a promised request $Q_{i,j}^{promised}$ in the request queue of the next checkpoint taken by j. Doing this, we virtually postpone the reception of $Q_{i,j}$ in case of recovery, like Q in case of recovery from $n+1$ in Figure 3. There is no promised request in the request queue during the first execution, since the request is replaced with a promised request *only* in the checkpoint.

Thanks to promised requests, the relative order of request receptions is preserved in case of recovery.

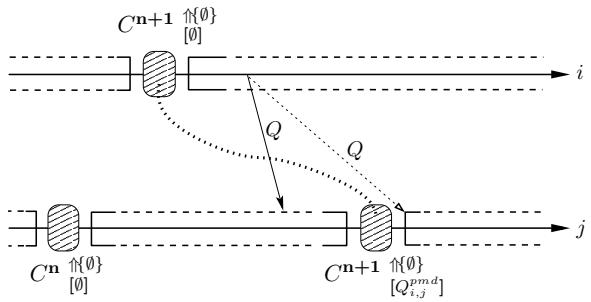

Figure 3: The request Q is replaced by a promised request in C_j^{n+1}

2.3.2 Reply $R_{i,j}$

If the received message is a reply, it is impossible to postpone this reception. As written in Section 1.1, the reception of a reply may cause an alteration of the whole activity state that is irreversible. Thus, a reply that should force a checkpoint on the receiver is definitively an orphan message. Consequently, after a recovery, the receiver may receive a reply it had already received. Hopefully, the mechanism of futures ensures that a such a "duplicated reply" is *automatically* ignored. There is nevertheless one constraint: *ensure that duplicated replies are identical to the original replies.* We will further discuss about this necessity of equivalence of executions in Section 3.

Provided that duplicated replies are identical to the original ones, those two mechanisms avoid inconsistency due to orphan messages in case of recovery.

2.4 Avoid In-transit Messages

For each message $M_{i,j}$, an acknowledgment message piggybacks $N_j^{current}$, the latest checkpoint number of the receiver j; this piggybacked value is denoted denoted $N_{\overline{M_{i,j}}}$. Therefore, when $N_{\overline{M_{i,j}}} > N_i^{current}$, an activity is forced to checkpoint as if it had not yet sent the applicative message $M_{i,j}$ (request or reply). The sending activity is inevitably serving a request while sending the message, thus it is not in stable state. This forced checkpoint must then be postponed to the next stable state. Consequently, the sent message is an in-transit message: it will be lost in case of recovery. Our solution is to log this message so as to resend it in case of recovery. This message can be logged in volatile memory while the next checkpoint is not taken: logging on a stable storage occurs during the checkpoint, allowing only one access to the stable storage. We will note $\mathbb{M}_i^{resend}(n)$ the *resend queue of i for n* i.e. the ordered list of messages that have to be sequentially sent from i in case of recovery from the checkpoint n.

Figure 4 shows the logging of Q (noted $]Q[$) in the checkpoint n.

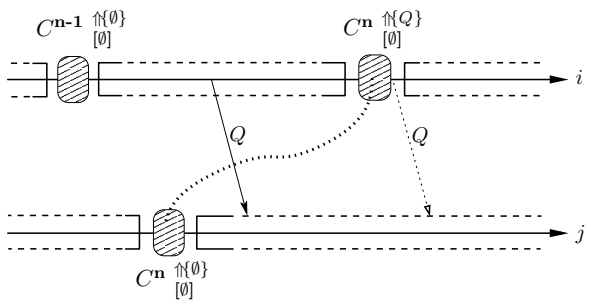

Figure 4: The request Q is logged for resent in the checkpoint n of i.

Thanks to this logging mechanism, no message is lost after a recovery from any recovery line.

2.5 Consecutive and Impossible Checkpoints

As an activity catches up to the latest checkpoint number of other activities, it might be behind by more than one checkpoint. In this case, it must take several consecutive checkpoints, *without serving any request between these checkpoints.* These checkpoints differ only in their associated queues, since the applicative state does not change. In practice, an activity that is more than one checkpoint behind takes *one checkpoint with several sequence numbers*, and several different resend and request queues. For example, on Figure 5, j must take two checkpoints differing only by the resend queue (here $]Q^1, Q^2[$ and $]Q^2[$).

Under the default service policy of our model, i.e. FIFO serving, an activity is in a stable state before each request service. It can thus take a checkpoint before each request service.

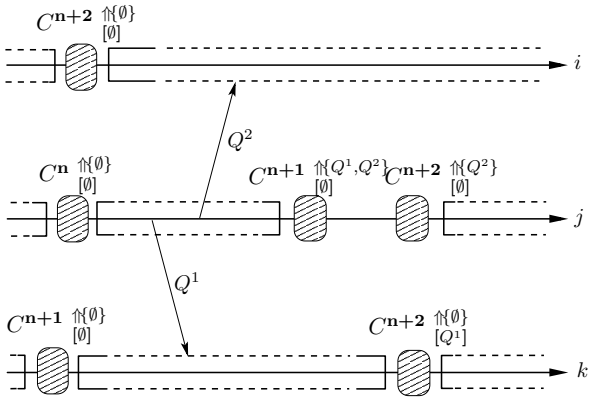

Figure 5: The activity j must take two consecutive checkpoints

However, it is possible to progam a different service policy and consequently prevent an activity to checkpoint before *every* service. If an activity serves a request $Q_{i,j}$ that should force a checkpoint, but cannot actually take this checkpoint before the service, then this checkpoint become impossible. It will not be taken anymore because $Q_{i,j}$ is removed from the request queue (indeed it has been served) and thus cannot be replaced with a promised request.

In order to model consecutive and impossible checkpoints, we introduce two values for each activity: N_i^{min} and N_i^{max}. The next time i is in stable state, it must then take all checkpoints from $C_i^{N_i^{min}}$ to $C_i^{N_i^{max}}$. If all checkpoints are possible, $N_i^{min} = N_i^{current} + 1$.

3 Request Reception History

This Section shows first that the usage of message logging mechanisms can lead to potential inconsistency after a recovery because of duplicated replies and causal dependencies between messages. We then propose our solution to avoid this inconsistency.

3.1 Problems

3.1.1 Duplicated Replies

As a duplicated reply is ignored by the receiver, this reply must be identical to the original one. The first consequence is that the activities must be *piecewise deterministic* [28]: as two replies must be identical in a given context, they cannot be the result of an indeterministic operation.

Provided that activities are piecewise deterministic and thanks to the Property 1.1, the content of a reply, and in particular of a duplicated reply, is the consequence of the history of request services of the sender of this reply. Thus, in order to ensure that duplicated replies are identical to the original one, we must ensure that the histories of request services are identical from one execution to another. So, we must ensure the following requirement for each checkpoint C_i^n:

$\mathcal{R}1$: The histories of request receptions are identical during the first execution and the reexecution, as long as there might exist in the first execution replies that are duplicated in the reexecution.

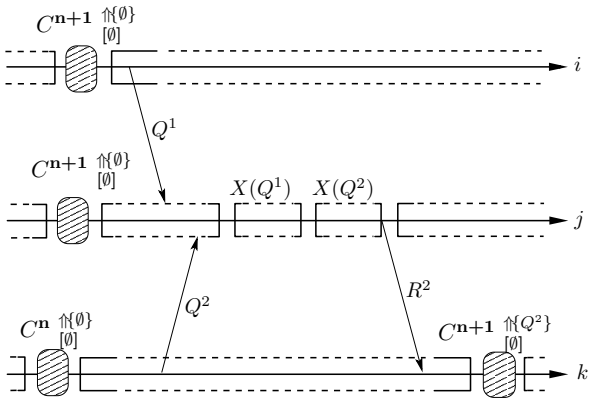

Figure 6: An alteration of the reception order of Q^1 and Q^2 may modify R^2.

Let us consider Figure 6: in case of recovery from the global state $n+1$, Q^2, which is logged for resending in C_k^{n+1}, may be received by j *before* Q^1. As a consequence, the duplicated reply R^2 may be different after the recovery from the original R^2. Thus we need to record the order of reception of requests Q^1 and Q^2 on j.

3.1.2 Causal Dependencies

The mechanism of resending messages of the resend queue during the recovery could lose causal relations between messages. Since these message sending are triggered by the protocol, the potential causal dependencies with messages sent by the application are consequently lost if there is no mean of scheduling. As logged for resending messages are in fact in-transit messages, we must ensure the following requirement for each checkpoint C_i^n:

$\mathcal{R}2$: The histories of request receptions are identical during the first execution and the reexecution, as long as there might exist in-transit messages in the currently built global state n in the first execution.

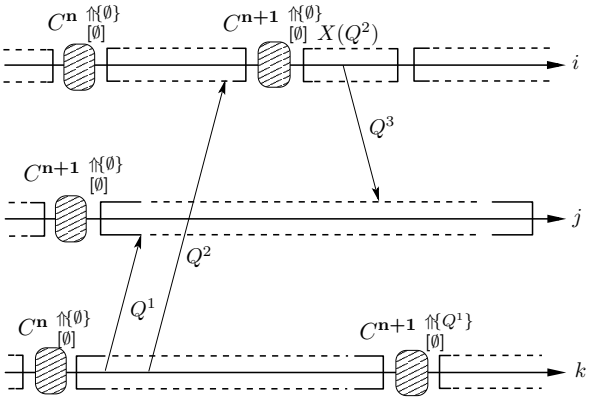

Figure 7: The reception of Q^3 on j must follow reception of Q^1

Such a situation is shown in Figure 7: as the sending of Q^3 is a consequence of the reception of Q^2 on i, and as the sending of Q^1 precedes the sending of Q^2, Q^1 *must be received by* j *before* Q^3.

3.2 Solution: Request Reception History

Consider a given checkpoint C_i^n. The two requirements $\mathcal{R}1$ and $\mathcal{R}2$ are verified for i and C_i^n when the global state n is completed, that is to say when every activity has taken the checkpoint n. Thus, we introduce a mechanism that logs sufficient information during the first execution from C_i^n to the completion of the global state n, in order to ensure the reexecution equivalence. We call this mechanism the *request reception history*. Thanks to the Property 1.2, the request reception history[3] just needs to record the ordered list of the identity of activities that have sent requests; this information is sufficient to ensure execution equivalence.

Each checkpoint is associated with a request reception history. We denote $\mathbb{H}_i^n$ the request reception history associated with the checkpoint C_i^n: when C_i^n is taken, $\mathbb{H}_i^n$ is *opened*. This history is in fact an ordered list of *promised requests*: when an activity i receives a request from an activity j, a promised request $Q_{j,i}^{promised}$ is added to the histories of i that are currently open. This promised request allows i to wait for this request in case of recovery, and thus ensures the same request reception order during the reexecution. Note that there can be more than one open history at a given time, since one history is opened for each checkpoint.

[3]We sometimes use the term *history* for request reception history.

3.3 Closing Request Reception History

$\mathcal{R}1$ and $\mathcal{R}2$ imply that reexecution must be equivalent to the first execution only until the completion of the currently built global state. Moreover, a checkpoint C_i^n is usable for recovery only if its associated request reception history $\mathbb{H}_i^n$ is closed and stored. Consequently, request reception histories must be finally *closed* and that is possible as soon as every activity has taken the checkpoint n. When a history is closed, it is stored with its associated checkpoint. As a consequence, a recovery line is not only a global state n, but also the set of all histories $\mathbb{H}^n$ closed on every activity and stored. Thus, the activities must be informed of the completion of the global state n to close their history $\mathbb{H}^n$.

We introduce a new kind[4] of message, denoted $\mathcal{M}_n^{globalState}$, which informs the receiver that the global state n is completed, thus that it can close all request reception histories $\mathbb{H}_i^m$, where m is less or equal than n. The recovery process is responsible for informing activities that a global state has completed, as it is assumed that this process has access to the stable storage where checkpoints are recorded. Each message sent between activities and the stable storage or between two activities can be used to spread the information that some histories can be closed. Whichever strategy is chosen, the recovery process first needs to inform a set of activities that a global state is formed. We distinguish two methods for triggering this information spreading:

- *a broadcast-based method*: the recovery process broadcasts a message $\mathcal{M}_n^{globalState}$ when the global state n is completed. This method involves a broadcast communication for each global state completion, but ensures the earliest closure of histories, then reduces history sizes and allows faster creation of recovery lines.
- *a pulling-based method*: activities are informed of the latest complete global state when they take a checkpoint and then communicate to send the checkpoint on the stable storage: a message $\mathcal{M}_n^{globalState}$ is sent as an acknowledgment. Broadcast messages are thus avoided, but history sizes are bigger, and recovery lines are created slower.

As asynchronous methods are used to close histories, consistency problems can occur. Let us consider Figure 8: each activity has closed its request reception histories $\mathbb{H}_i^n$ (closure is denoted by $\diamondsuit$). But the request Q^6 has been sent after the closure on k, and has been received before the closure on j. After a recovery from the global state n, the reexecution of k could be different *after the request reception history closure* since the request reception order is allowed to be different; thus the request Q^6 might not be sent. As a consequence, Q^6 must not be awaited in j (else j might wait indefinitely for the reception of this request). Consequently j should have closed the request reception history $\mathbb{H}_j^n$ *before* the reception of Q^6.

We must then avoid orphan requests between history closures. Hence, we use a protocol similar to [5] to coordinate this closures: each message $M_{i,j}$ piggybacks the number n of the latest closed history $\mathbb{H}_i^n$ on i; this piggybacked value is denoted $H_{\overrightarrow{M_{i,j}}}$. When j receives the

[4]This kind of message is added *at the middleware level*. At the user level, communications are only requests and replies.

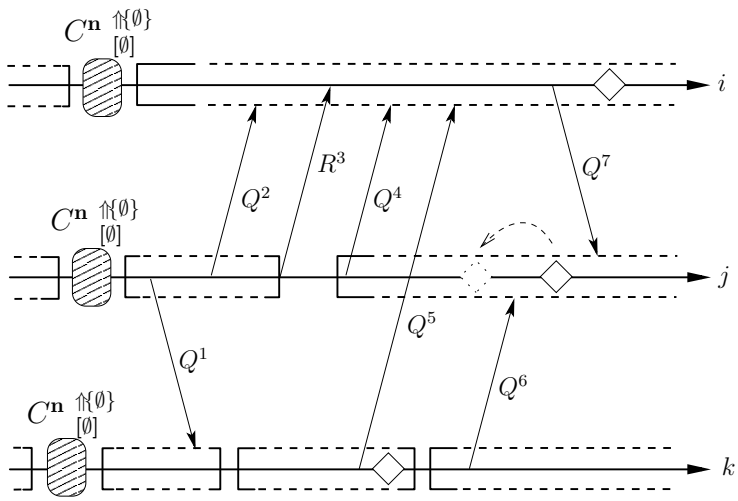

Figure 8: Request reception history closures ($\diamondsuit$) on j and k must be coordinated

message $M_{i,j}$, it must close all the currently open histories $\mathbb{H}_j^m$, where m is less or equal than $H_{\overrightarrow{M_{i,j}}}$, before processing $M_{i,j}$.

4 Recovery

When an activity i recovers from a checkpoint C_i^n, before restarting its thread, it must:

- append the history $\mathbb{H}_i^n$ to its request queue $\mathbb{Q}_i^{rcv}$: the reception order is forced, in a lazy manner, to be the same as the reception order of the first execution.
- send sequentially all messages in $\mathbb{M}_i^{resend}(n)$. The relative order of the request receptions is guaranteed by a FIFO point to point order and by the fact that each of these messages is awaited by a promised request. In other words, the relative order of reception from different activities has been recorded through the awaited requests.

This lazy synchronization allows the recovery procedure to be uncoordinated: as soon as one activity recovers from a checkpoint, each activity will recover in an independent manner.

4.1 Incarnation Numbers

Since the recovery is fully asynchronous, we use an incarnation number mechanism, as in [28] and [21]: the recovery process knows at any time the number of recoveries triggered since the beginning of the execution, denoted I^{global}. Each activity i has its own value of incarnation,

denoted I_i, and the number of the checkpoint used for the latest recovery, denoted LR_i. As every activity must recover from the same global state, a message of recovery is broadcasted by the recovery process. We denote $\mathcal{M}_{n,k}^{recovery}$ the message indicating that the kth recovery has to be performed from the global state n.

The incarnation number allows to avoid communication between activities running in different incarnations: each message $M_{i,j}$ piggybacks the incarnation number of the sender i and the number of the checkpoint used for the last recovery. These piggybacked values are respectively denoted by $I_{\overrightarrow{M_{i,j}}}$ and $LR_{\overrightarrow{M_{i,j}}}$.When j receives $M_{i,j}$,

- if $I_{\overrightarrow{M_{i,j}}} < I_j$, then $M_{i,j}$ is ignored by j and j sends $\mathcal{M}_{LR_j,I_j}^{recovery}$ to i,
- if $I_{\overrightarrow{M_{i,j}}} > I_j$, then j must recover from $LR_{\overrightarrow{M_{i,j}}}$. i is blocked on the sending of $M_{i,j}$ until j recovers,
- if $I_{\overrightarrow{M_{i,j}}} = I_j$, then $M_{i,j}$ is delivered to j.

5 Algorithmic Description

5.1 Principles

Table 1 summarizes the principles of the protocol described in the preceding sections. We distinguish actions that have to be performed during the first execution and during the reexecution (see Section 2.1).

		First Execution	Reexecution
Avoid orphan message	Received request	Promised request in next checkpoint(s)	Wait-by-necessity on promised request
	Received reply	No action	Duplicated reply is ignored
Avoid in-transit message	Sent request	Logged in next checkpoint(s)	Resent during recovery
	Sent reply	Logged in next checkpoint(s)	Resent during recovery
Ensure execution equivalence	Received request	Promised request in the open histories	Wait-by-necessity on promised request
	Received reply	No action	Equivalence ensured by the model

Table 1: Protocol principles

5.2 Algorithms

Figure 9 formally describes the checkpointing protocol. We denote $\oplus$ the append operator and $\rightarrow$ the substitution operator. One must note that:

- the **Init** procedure is automatically called at the creation of an activity,
- the **Checkpoint attempt** procedure is automatically called when *stableState* is true,
- TTC_i is re-initialized to TTC_INIT each time a checkpoint is taken, even if this checkpoint has been triggered by a message.

- **Init**
  $I_i = 0$
  $LR_i = 0$
  $N_i^{current} = 0$
  $N_i^{min} = 0$
  $N_i^{max} = 1$
  $TTC_i = TTC_INIT$
  Checkpoint attempt
- **Send** $M_{i,j}$ **(from** i**)**
  <u>if</u> $N_{\overrightarrow{M_{i,j}}} > N_i^{current}$ <u>then</u> $N_i^{max} = max(N_i^{max}, N_{\overrightarrow{M_{i,j}}})$ <u>and</u> $\forall m$ <u>st</u> $(m \leq N_{\overrightarrow{M_{i,j}}})$ <u>do</u> $M_{i,j} \oplus \mathbb{M}_i^{resend}(m)$
- **Receive** $M_{i,j}$ **(on** j**)**
  <u>if</u> $I_{\overrightarrow{M_{i,j}}} == I_j$ <u>then</u>
  $\forall k$ <u>st</u> $(\mathbb{H}_i^k$ open <u>and</u> $k < H_{\overrightarrow{M_{i,j}}})$ <u>do</u> close $\mathbb{H}_i^k$
  <u>if</u> $M_{i,j}$ <u>is</u> $Q_{i,j}$ <u>then</u> $\forall k$ <u>st</u> $(\mathbb{H}_i^k$ open)<u>do</u> $Q_{i,j}^{promised} \oplus \mathbb{H}_i^k$
  <u>if</u> $N_{\overrightarrow{M_{i,j}}} > N_j^{current}$ <u>then</u> $N_j^{max} = max(N_j^{max}, N_{\overrightarrow{M_{i,j}}})$
  <u>else</u> <u>if</u> $I_{\overrightarrow{M_{i,j}}} > I_j$ <u>then</u>
  block i on the send of $M_{i,j}$
  Recovery $I_{\overrightarrow{M_{i,j}}}$ **of** j **from** $LR_{\overrightarrow{M_{i,j}}}$ (Figure 10)
  <u>else</u>
  send $\mathcal{M}_{LR_j,I_j}^{recovery}$ to i
  ignore $M_{i,j}$
- **Receive** $\mathcal{M}_n^{globalState}$
  $\forall k$ <u>st</u> $(\mathbb{H}_i^k$ open <u>and</u> $k < n)$ <u>do</u> close $\mathbb{H}_i^k$
- **Checkpoint attempt (called if *stableState* is true)**
  $N_i^{current} = max(N_i^{min} - 1, N_i^{current})$
  <u>if</u> $N_i^{max} > N_i^{current}$ <u>then</u>
  <u>while</u> $N_i^{max} > N_i^{current}$ <u>do</u> **Checkpoint** $C_i^{N_i^{current}+1}$
  <u>else</u>
  <u>if</u> $TTC_i == 0$ <u>then</u> **Checkpoint** $C_i^{N_i^{current}+1}$
- **Checkpoint** C_i^n
  $\forall Q_{j,i} \in \mathbb{Q}_i^{rcv}$, <u>if</u> $N_{\overline{Q_{j,i}}} \geq n$ <u>then</u> $Q_{i,j} \rightarrow Q_{i,j}^{promised}$
  add $\mathbb{M}_i^{resend}(n)$ to C_i^n
  delete $\mathbb{M}_i^{resend}(n)$
  $N_i^{current} = n$
  open $\mathbb{H}_i^n$
  $TTC_i = TTC_INIT$
- **Serve request** $Q_{i,j}$
  <u>if</u> $N_{\overrightarrow{Q_{i,j}}} > N_i^{current}$ <u>then</u>
  <u>if</u> *stableState* <u>then</u>
  $N_i^{max} = N_{\overrightarrow{Q_{i,j}}}$ <u>and</u> **Checkpoint attempt**
  <u>else</u> $N_i^{min} = max(N_i^{min}, N_{\overrightarrow{Q_{i,j}}} + 1)$

Figure 9: The checkpointing protocol

Figure 10 describes the recovery protocol.

- **Failure of the activity** i
  mapping of i
  $I^{global} = I^{global} + 1$
  Recovery I^{global} **of** i **from** n
  broadcast $\mathcal{M}^{recovery}_{n,I^{global}}$
- **Reception of** $\mathcal{M}^{recovery}_{n,k}$ **on** i
  <u>if</u> $I_i < k$ <u>then</u>
  Recovery k **of** i **from** n
  <u>else</u>
  ignore $\mathcal{M}^{recovery}_{n,k}$
- **Recovery** k **of** i **from** n
  stop activity if any
  download C^n_i and $\mathbb{H}^n_i$
  recover activity state from C^n_i
  $\mathbb{H}^n_i \oplus \mathbb{Q}^{rcv}_i$
  $I_i = k$
  $LR_i = n$
  send all messages in $\mathbb{M}^{resend}_i(n)$
  restart activity

Figure 10: The recovery protocol

Part II
Correctness Proof

This part presents the formal proof of the correctness of the proposed protocol. After introducing some relations and properties on ASP distributed computations, we prove that the global state characterized by the history closures is also an existing global state in the reexecution from the corresponding recovery line. As history closures are consistent global states and exist in both execution and reexecution, reexecution is then equivalent to the first execution until this state and thus the reexecution cannot lead to inconsistent states that should not exist in a normal execution.

6 Causality relation for asynchronous request services

Lamport introduces in [19] the concept of one event happening before another in a distributed system, and defines a partial ordering between events. His seminal paper is the first to emphasize the "causal domain", in opposition to the "time domain": notion of time and simultaneity are avoided. Mattern presents in [23] a formal model for characterizing a distributed execution. It consists in a set of events partially ordered by the Lamport's happened-before relation. This characterization is an important foundation for specifying and proving distributed programs, and particularly for designing and proving correctness of fault-tolerance protocols [14]. Indeed, proving rollback-recovery protocols needs a way to characterize distributed executions and states.

The Lamport's happened-before relation supposes that events that occur on the same process are *totally ordered*. Based on such local events ordering, Lamport defines a partial global order that also takes into account synchronization due to message passing between processes. In the context of asynchronous request/reply communication patterns, such total ordering of local events is too much restrictive: the asynchronous service of requests allows to safely exchange request receptions with some other events.

We take into account the kind of events in the causality relation. A first distinction between message reception events is drawn in [20]: the authors propose an algorithm for identifying messages that are not influential in a computation, and then provide a message logging recovery scheme that takes into account message semantic. The same distinction is used in [15] to define the *significant precedence order*, and to provide a group protocol that supports the significantly ordered delivery of messages.

We add a concept similar to significant precedence into ASP. We introduce *consistent-enough* cuts and formalized promised requests presented in 2.2. These concepts are at the root of the correctness proof of the proposed protocol.

6.1 Elements of event-based analysis

Characterizing executions and cuts of an execution are both based on the definition of the causality relation between events. We then first introduce in this section general notations and properties on distributed executions without specifying this relation. We will see in the next section that this relation can be defined specifically for the ASP model.

In the following, i, j, k range over the activities of the system; e_i is an event that occurs on the activity i.

6.1.1 Characterizing Executions

Definition 1 (execution and causally correct order) *Let $\mathcal{S}_i^k$ be a state of the activity i. Let us consider a distributed execution characterized by a set of events partially ordered $(E, \prec)$ (called below an* execution*). This execution allows the system to change from the global state $S = \mathcal{S}_0^{ini}|...|\mathcal{S}_n^{ini}$ to the global state $S' = \mathcal{S}_0^{final}|...|\mathcal{S}_n^{final}$:*

$$S \xrightarrow{(E,\prec)} S'$$

All the linear extensions $[e^1 \dots e^n]$ of an execution $(E, \prec)$ correspond to all possible real execution leading from state S to the same state S':
$\forall [e^1 \dots e^n]$, if $e^x \prec e^y \Rightarrow x < y$, then

$$S \xrightarrow{[e^1 \dots e^n]} S'$$

We call an order $\prec$ verifying this property a causally correct order.

Note that S and S' are not necessarily the initial state and the final state of the total execution. Usually, for distributed systems communicating with asynchronous message passing, $\prec$ is the Lamport happened-before relation.

6.1.2 Partial Local Order

We introduce a local causality relation $\prec_i$ characterized as follows:

Definition 2 (Local causality relation) *A local ordering of events $\prec_i$ is a projection on a process of $\prec$:*

$$e \prec_i e' \quad \Leftrightarrow \quad e \prec e' \ \wedge \ (e \text{ and } e' \text{ occur in the process } i)$$

6.1.3 Cuts

As defined by Mattern in [23], a cut of an execution is a partially ordered set defined as follows:

Definition 3 (Cut) *A cut $\mathcal{C}$ of an execution $(E, \prec)$ is a finite subset $\mathcal{C} \subseteq E$ such that*

$$\forall e, e' \in E, e \in \mathcal{C} \wedge e' \prec_i e \Rightarrow e' \in \mathcal{C}$$

Definition 4 (Equality of cuts $\mathcal{C}^1 = \mathcal{C}^2$) *A cut $\mathcal{C}^2$ of an execution $(E^2, \prec^2)$ is equal to a cut $\mathcal{C}^1$ of an execution $(E^1, \prec^1)$ ($\mathcal{C}^1 = \mathcal{C}^2$) if and only if*

$$\begin{cases} e \in \mathcal{C}^1 \Leftrightarrow e \in \mathcal{C}^2 \\ \forall e, e' \in \mathcal{C}^1 e \prec^1 e' \Leftrightarrow e \prec^2 e' \end{cases}$$

Definition 5 (Ordering cuts $\mathcal{C}^1 \subseteq \mathcal{C}^2$) *A cut $\mathcal{C}^2$ of an execution $(E^2, \prec^2)$ is later than a cut $\mathcal{C}^1$ of an execution $(E^1, \prec^1)$ ($\mathcal{C}^1 \subseteq \mathcal{C}^2$) if and only if*

$$\begin{cases} e \in \mathcal{C}^1 \Rightarrow e \in \mathcal{C}^2 \\ \forall e, e' \in \mathcal{C}^1 e \prec^1 e' \Leftrightarrow e \prec^2 e' \end{cases}$$

Note that this definition does not imply that the two cuts are cuts of the same execution. But the considered executions E^1 and E^2 must consist of the same events ordered in the same manner *until* $\mathcal{C}^1$. Execution E^2 between $\mathcal{C}^1$ and $\mathcal{C}^2$ could be different from E^1. In the same way, two cuts can be said to be equal even if they are cuts of two different executions.

If the executions are the same, ordering and equality is reduced to a set comparison.

A cut is said to be *consistent* if it verifies:

Definition 6 (Consistent cut) *A cut $\mathcal{C}$ is consistent if and only if*

$$e \in \mathcal{C} \wedge e' \prec e \Rightarrow e' \in \mathcal{C}$$

A consistent cut correspond to a real global state of the system, that is to say a possible state.

Mattern provides in [23] general properties on cuts. We give here two of them that could be useful for our proof :

- Cuts form a lattice $(\mathcal{C}, \subseteq)$.
- Consistent cuts form a sub-lattice of cuts $(\mathcal{C}, \subseteq)$.

6.1.4 $\oplus$ operator

We define $\oplus$ an operator that goes along an execution $(E, \prec)$ from a consistent cut to a successor (for $\subseteq$) consistent cut. This operator allows to cover a set of consistent cuts of an execution.

Notation 6.1 ($\oplus$) *Let $\mathcal{C}$ be a consistent cut of an execution $(E, \prec)$. Let $e \in E \backslash \mathcal{C}$ such that e is minimal in $E \backslash \mathcal{C}$ that is to say:*

$$\nexists e' \in E \backslash \mathcal{C}, e' \prec e$$

then, we define $\mathcal{C} \oplus e$ as the ordered set of events $\mathcal{C} \cup \{e\}$

Property 6.1 ($\oplus$ **maintains consistency**) $\mathcal{C} \oplus e$ *is a* consistent cut *of* $(E, \prec)$.

Proof: As e is minimal in $E \backslash \mathcal{C}$, any event e' such that $e' \prec e$ belong to $\mathcal{C}$. $\mathcal{C} \oplus e$ is then consistent. A similar concept can be found in [11]. □

Property 6.2 below states that, as $\prec$ is a causality relation, then events that are not causally dependent can be safely exchanged when covering execution.

Property 6.2 ($\oplus$ **and** $\prec$)

$$e \not\prec e' \wedge e' \not\prec e \mathcal{C} \oplus e \oplus e' = \mathcal{C} \oplus e' \oplus e$$

Property 6.3

$$\mathcal{C}^1 \subseteq \mathcal{C}^2 \wedge e \in \mathcal{C}^2 \wedge e \textit{ is minimal in } \mathcal{C}^2 \backslash \mathcal{C}^1 \Rightarrow \mathcal{C}^1 \oplus e \subseteq \mathcal{C}^2$$

6.2 Causality in ASP

In a first time, we focus only on the request service mechanism because it has been proved in [7] that, for the ASP model, the reply mechanism has an important property stating that replies can be sent in any order without any consequence on the execution. We will then consider four kinds of events: request sending, request reception, request service and internal operation. An internal operation is an event that manipulates only the internal state of an activity (it does not alter the pending request queue), and a request service event is the beginning of the service of a request.

The asynchronous service of requests introduces particular relations between events on a single process: a request reception event can be safely exchanged with an internal event, a request or reply sending, or a service of another request. The consequence is that a computation characterized using the Lamport's happened-before relation is too restrictive: the relative order of some events that occur on the same process is not relevant to characterize a distributed computation.

The fact that the happened-before relation defines a total order on the events inside each process makes it inadequate to an asynchronous request service model. So as to characterize computations in ASP, we consider a causality relation inside an activity (called *local causality relation*) that uniquely defines a *partial order* on local events.

This section applies the preceding results to the ASP calculus, and particularly to the specification of consistent global states. Let M denote a request message, e_i is an event that occurs on the process i that can be one of the following:

$$e_i ::= send(M)|rcv(M)|serve(M)|int$$

Let Γ denote the corresponding pairs of communication events, Γ defines a bijection between request sending and request receptions:

$$\Gamma = \{(e_i^s, e_j^r)|i \neq j \wedge \exists M, e_i^s = send(M) \wedge e_j^r = rcv(M)\}$$

Let Σ denote the pairs associating request receptions with their services, Σ defines a bijection between request receptions and request services:

$$\Sigma = \{(e_i^r, e_i^x) | e_i^r \rightarrow_i e_i^x \wedge \exists M, e_i^r = rcv(M) \wedge e_i^x = serve(M)\}$$

The precedence order is a *total* local order given by the projection of a *real* execution on a given process i: $E_i = [e_i^1, ..., e_i^m]$, that is to say a linear extension of the execution $(E, \prec)$.

Notation 6.2 (Precedence order) *$e_i^x \rightarrow_i e_i^y$ for a real execution $E_i = [e_i^1, ..., e_i^m]$ if and only if $x < y$.*

6.2.1 Local causality relation for ASP

A causally correct local order for ASP is:

Definition 7 *$\prec_i$ is a transitive* partial order *such that*

$$e_i^x \prec_i e_i^y \Leftrightarrow \begin{cases} (e_i^x \neq rcv \wedge e_i^y \neq rcv \wedge e_i^x \rightarrow_i e_i^y) \vee & (a) \\ (e_i^x = rcv \wedge e_i^y = rcv \wedge e_i^x \rightarrow_i e_i^y) \vee & (b) \\ (e_i^x, e_i^y) \in \Sigma i \vee & (c) \\ \exists e_i,\ e_i^x \prec_i e_i \prec_i e_i^y & (d) \end{cases}$$

This local causality relation expresses three kinds of causes:

(a) *Evaluation order*, that is the order directly given by the local computation. It is composed of internal, request sending and service events. Provided that activities are piecewise deterministic [28], such order does not need to be remembered to characterize execution because it is a consequence of the evaluation mechanism. This is mainly due to the fact that each activity is made of a single thread.

(b) *External event order*, that is the order of message reception. This order must be remembered to characterize execution.

(c) *Service order*, that is the order relating the reception and the service of a request.

Since we consider an asynchronous message-passing system, the definition of the global partial order, as formalized by Lamport, is unchanged. But since it is based on a partial local order $\prec_i$, its signification is altered. This global relation is still defined as follow:

Definition 8 (partial global order - Happened-before relation) *$\prec$ is a quasi ordering such that $e_i^x \prec e_j^y$ if and only if*

$$\begin{cases} e_i^x \prec_i e_i^y \vee \\ (e_i^x, e_j^y) \in \Gamma \vee \\ \exists e_i, e_i^x \prec e_i \prec e_i^y \end{cases}$$

We call such a partial global order, the *happened-before extension* of the local order $\prec_i$. This defines a causally correct global order $\prec$ adapted to ASP.

Property 6.4 ($\oplus$ maintains equality of cuts)
If $\mathcal{C}^1$ is a consistent cut of $(E^1, \prec^1)$ and e is a minimal event of $E \backslash \mathcal{C}^1$ and $\mathcal{C}^2$ is a consistent cut of $(E^2, \prec^2)$ and e' is a minimal event of $E \backslash \mathcal{C}^2$ then

$$\mathcal{C}^1 = \mathcal{C}^2 \wedge e = e' \quad \Rightarrow \quad \mathcal{C}^1 \oplus e = \mathcal{C}^2 \oplus e'$$

Proof: To prove that the two consistent cuts $\mathcal{C}^1 \oplus e$ and $\mathcal{C}^2 \oplus e'$ are equal, we must prove that they have the same elements ordered identically :

$$\begin{cases} e^1 \in \mathcal{C}^1 \oplus e \Leftrightarrow e^1 \in \mathcal{C}^2 \oplus e' \\ \forall e^1, e^2 \in \mathcal{C}^1 \oplus e,\ e^1 \prec^1 e^2 \Leftrightarrow e^1 \prec^2 e^2 \end{cases}$$

The first assertion is trivial from the definition of $\oplus$. The second depends on the definition of the causality relation $\prec$. As the property is symmetric, we will prove that $e^1 \prec^1 e^2 \Rightarrow e^1 \prec^2 e^2$.

Suppose $e^1 \prec^1 e^2$. If $e^2 \neq e'$ then $\mathcal{C}^1 = \mathcal{C}^2$ ensures $e^1 \prec^2 e^2$. From Definition 8, $e^1 \prec^1 e'$ if and only if one of the following occurs:

- $(e^1, e) \in \Gamma$ in the execution $(E^1, \prec^1)$ then necessarily $(e^1, e) \in \Gamma$ in the execution $(E^2, \prec^2)$ and thus $e^1 \prec^2 e$
- $e^1 \prec_i{}^1 e'$ and then, from Definition 7 several cases are possible (we denote $\rightarrow_i{}^1$ the precedence relation associated to the execution $(E^1, \prec^1)$):
  - $e^1 \neq rcv \wedge e \neq rcv \wedge e^1 \rightarrow_i{}^1 e$ then as $e^1 \in \mathcal{C}^1 = \mathcal{C}^2$, and $\mathcal{C}^2$ is a cut we cannot have $e \prec^2 e^1$.

    Moreover, e^1 ans e are on the same process thus we have:
    * either $e \rightarrow_i{}^2 e^1$, but $e^1 \neq rcv \wedge e \neq rcv$ would imply $e \prec^2 e^1$ which is impossible,
    * or $e^1 \rightarrow_i{}^2 e$ and $e^1 \neq rcv \wedge e \neq rcv$, and we have and in that case $e^1 \prec_i{}^2 e$. Finally, $e^1 \prec^2 e$.
  - $e^1 = rcv \wedge e = rcv \wedge e^1 \rightarrow_i{}^1 e$ the same arguments than the case $e^1 \neq rcv \wedge e \neq rcv \wedge e^1 \rightarrow_i{}^1 e$ are used to prove $e^1 \prec^2 e$.
  - $(e^1, e) \in \Sigma i$ in the execution $(E^1, \prec^1)$ then necessarily $(e^1, e) \in \Sigma$ in the execution $(E^2, \prec^2)$ and thus $e^1 \prec_i{}^2 e$. Finally, $e^1 \prec^2 e$

The remaining transitivity cases are proven by a trivial induction on the number of events needed to transitively prove $e^1 \prec^1 e$. □

6.2.2 Piecewise Determinism

In ASP, activities are supposed to be piecewise determinitic. Piecewise determinism states that an event (that is not a request reception) that occurs on the activity i is the consequence of local history of i (remember that e_i is an event that occurs on the activity i):

Definition 9 (Piecewise Determinism) *Let $\mathcal{C}$ be a cut of an execution $(E, \prec)$, and let $\mathcal{C}^2$ be a cut of an execution $(E^2, \prec^2)$.*
If $e_i \neq rcv$ is a minimal event of $E \backslash \mathcal{C}$ and ${e_i}' \neq rcv$ is a minimal event of $E^2 \backslash \mathcal{C}^2$,
Then $\mathcal{C} = \mathcal{C}^2 \Rightarrow e_i = {e_i}'$

Property 6.4 and preceding definition imply the following property:

Property 6.5 (Piecewise deterministic execution) *If $\mathcal{C}^1$ and $\mathcal{C}^2$ are consistent cuts then:*

$$\begin{cases} \mathcal{C}^1 = \mathcal{C}^2 \\ e_i \neq rcv \\ {e_i}' \neq rcv \end{cases} \Rightarrow \quad \mathcal{C}^1 \oplus e_i = \mathcal{C}^2 \oplus {e_i}' \wedge e_i = {e_i}'$$

This is due to the fact that, if $\mathcal{C}^1$ is a consistent cut of an execution $(E, \prec)$ writing $\mathcal{C}^1 \oplus e_i$ implies that e_i is a minimal event of $(E, \prec)$.

6.2.3 Consistent Enough Cuts

The partial local order given in 6.2.1 defines a more permissive notion of consistent cuts. Moreover, an interesting property of ASP is that requests can be safely added or removed from the pending request queue. Thus making a cut consistent can be achieved by adding or removing messages in the pending request queue but modifying the internal state of an activity is not possible. Consequently, a cut will be said to be *consistent enough* [12] if it can be transformed into a consistent cut. That means that a cut is consistent enough if there is no served orphan request. Indeed, serving a message has a direct influence on the internal state of an activity.

Definition 10 (consistent enough) *A cut $\mathcal{C}$ is consistent enough if and only if*

$$(e_i^x, e_i^y) \in \Sigma \wedge (e_j^z, e_i^x) \in \Gamma \wedge e_j^z \notin \mathcal{C} \Rightarrow e_i^y \notin \mathcal{C}$$

Such a consistency property allows to define new kinds of consistent global states from which a recovery could be performed. This is particularly useful because it allows much more flexibility in the placement of checkpoints: checkpoints synchronizations (i.e. checkpoints forced by a message reception) could then be delayed while the communication that should trigger the checkpoint *has no consequence on the internal state of the activity.*

7 Proving correctness of the protocol

In this section, we first introduce concepts that will allow us to formalize the proposed protocol such as promised receptions. We then present the formal proof.

7.1 Promised Receptions

We introduce a new kind of events: the promised receptions, corresponding in ASP to a promised request. For any reception event $e = rcv(M)$, $\mathcal{P}(e)$ is a place-holder that will be automatically replaced by e when the message M will be received. Of course, promised receptions can only be added in an existing execution: when e is replace by $\mathcal{P}(e)$, $\mathcal{P}(e)$ is ordered in the same way that e was. Actually, promised receptions are added into global states made by our protocol during the first execution. These particular events are only used in case of reexecution from theses global states, that is to say after a recovery.

Since some events can then be automatically re-ordered, the local causality realtion of Definition 7 is slightly modified:

Definition 11 $\prec_i$ *is a transitive* partial order *such that*

$$e_i^x \prec_i e_i^y \Leftrightarrow \begin{cases} (e_i^x \neq rcv \wedge e_i^y \neq rcv \wedge e_i^x \rightarrow_i e_i^y) \vee & (a) \\ (e_i^x = rcv \wedge e_i^y = rcv \wedge e_i^x \rightarrow_i e_i^y \wedge \nexists \mathcal{P}(e_i^x) \wedge \nexists \mathcal{P}(e_i^y)) \vee & (b) \\ (e_i^x, e_i^y) \in \Sigma i \vee & (c) \\ \exists e_i,\ e_i^x \prec_i e_i \prec_i e_i^y & (d) \end{cases}$$

and

$$\mathcal{P}(e_i^x) \prec_i e_i^y \Leftrightarrow e_i^y = rcv \wedge \nexists \mathcal{P}(e_i^y)$$

Note that the preceding definition is still valid for a normal execution where there is no promised reception. This relation mainly consists in not ordering message receptions if a corresponding promised reception exists and ordering promised receptions before receptions. This order is justified by the following facts:

- Promised receptions are place-holders that replace receptions that occurred in the first execution, and thus are ordered like in the first execution.
- Promised events will occur another time in the second execution, and thus it is not necessary to order them in the reexecution as they will take the place of their place-holder.
- receptions are ordered after promised reception because they did not occur in the first execution.
- This order is not valid and will never be used to order events that occurred before the recovery. In other words events belonging to a recovery state are only ordered by the first execurion.

7.2 $\mathcal{P}$-cut

A cut which can contain promised receptions is called a $\mathcal{P}$-cut and is denoted by $\mathcal{C}_{\mathcal{P}}$. Every cut is a $\mathcal{P}$-cut.

We denote by $\mathcal{C}_{\mathcal{P}}\{\!\{e \leftarrow \mathcal{P}(e)\}\!\}$ the replacement of the reception event e belonging to $\mathcal{C}_{\mathcal{P}}$ by a place-holder for this reception. Considering the order of events, $\mathcal{P}(e)$ has the same place as e in $\mathcal{C}_{\mathcal{P}}$. Such a replacement will not belong to a normal execution but is a necessary manipulation in order to be able to recover from a consistent enough cut and as such the causality relation on $\mathcal{C}_{\mathcal{P}}\{\!\{e \leftarrow \mathcal{P}(e)\}\!\}$ may relate promised and normal events even if Definition 11 is sufficient and does not order promised events (except for normal receptions): for example $e' \prec \mathcal{P}(e)$ in $\mathcal{C}_{\mathcal{P}}\{\!\{e \leftarrow \mathcal{P}(e)\}\!\}$ if and only if $e' \prec e$ in $\mathcal{C}_{\mathcal{P}}$.

In other words, $\mathcal{C}_{\mathcal{P}}\{\!\{e \leftarrow \mathcal{P}(e)\}\!\}$ is defined in order to ensure.

Property 7.1

$$\mathcal{C}_{\mathcal{P}}\{\!\{e \leftarrow \mathcal{P}(e)\}\!\} \oplus e = \mathcal{C}_{\mathcal{P}}$$

We denote by $\mathcal{C}_{\mathcal{P}} + \mathcal{P}(e)$ the $\mathcal{P}$-cut formed by adding a promised reception at the end of the $\mathcal{P}$-cut (x is an event or a promised reception):

$$\begin{cases} x \in \mathcal{C} + \mathcal{P}(e_i^z) \Leftrightarrow x \in \mathcal{C} \vee x = \mathcal{P}(e_i^z) \\ e_i^y \prec_i \mathcal{P}(e_i^z) \Leftrightarrow e_i^y = rcv \vee e_i^y = \mathcal{P}(e) \end{cases}$$

$e \in \mathcal{C}_{\mathcal{P}}$ means e belong to $\mathcal{C}_{\mathcal{P}}$ and *is not a promised reception.*

We extend the equality on $\mathcal{P}$-cuts by ignoring the promised events and denoting it $\equiv$ ($=$ being the strict equality):

Notation 7.1 (equivalence of $\mathcal{P}$-cuts) *A $\mathcal{P}$-cut $\mathcal{C}_{\mathcal{P}}^2$ of an execution $(E^2, \prec^2)$ is equivalent to a $\mathcal{P}$-cut $\mathcal{C}_{\mathcal{P}}^1$ of an execution $(E^1, \prec^1)$ ($\mathcal{C}_{\mathcal{P}}^1 \equiv \mathcal{C}_{\mathcal{P}}^2$) if and only if*

$$\left(\begin{cases} e \in \mathcal{C}_{\mathcal{P}}^1 \Leftrightarrow e \in \mathcal{C}_{\mathcal{P}}^1 \\ \forall e, e' \in \mathcal{C}^1 e \prec^1 e' \Leftrightarrow e \prec^2 e' \end{cases} \right)$$

Ordering $\mathcal{P}$-cuts is the following:

Definition 12 (Ordering $\mathcal{P}$-cuts $\mathcal{C}_{\mathcal{P}}^1 \subseteq \mathcal{C}_{\mathcal{P}}^2$)

$$\mathcal{C}_{\mathcal{P}}^1 \subseteq \mathcal{C}_{\mathcal{P}}^2 \;\Leftrightarrow\; \begin{cases} e \in \mathcal{C}_{\mathcal{P}}^1 \;\Rightarrow\; e \in \mathcal{C}_{\mathcal{P}}^2 \\ \mathcal{P}(e) \in \mathcal{C}_{\mathcal{P}}^1 \;\Rightarrow\; \mathcal{P}(e) \in \mathcal{C}_{\mathcal{P}}^2 \vee e \in \mathcal{C}_{\mathcal{P}}^2 \\ \forall x, y \in \mathcal{C}_{\mathcal{P}}^1 x \prec^1 y \Leftrightarrow \big(x' \prec^2 y' \wedge (x = \mathcal{P}(x') \vee x' = x) \wedge (y = \mathcal{P}(y') \vee y' = y)\big) \end{cases}$$

where x, y, x' and y' can be events or promised events.

Definition 13 (Least cut $Cut(\mathcal{C}_{\mathcal{P}})$) *Let $Cut(\mathcal{C}_{\mathcal{P}})$ be the greatest cut earlier than $\mathcal{C}_{\mathcal{P}}$, it is obtained by removing the promised receptions and **some** normal receptions:*

$$Cut(\mathcal{C}_{\mathcal{P}}) = \bigcup \{\mathcal{C} | \mathcal{C} \subseteq \mathcal{C}_{\mathcal{P}}\}$$

The following property is a consequence of the preceding definitions (mainly the causality relation of Definition 11):

Property 7.2

$$\mathcal{P}(rcv(M)) \in \mathcal{C}_{\mathcal{P}} \wedge e' \neq snd(M) \quad \Rightarrow \quad \mathcal{C}_{\mathcal{P}} \oplus rcv(M) \oplus e' = \mathcal{C}_{\mathcal{P}} \oplus e' \oplus rcv(M)$$

This property ensures that promised receptions can occur at any time, their only cause being the sending of the corresponding message.

7.3 Consistent $\mathcal{P}$-cuts and constrained execution

We must adapt the concept of consistent cuts to the cuts containing promised receptions.

Definition 14 ($\mathcal{P}$-consistent cuts) *A $\mathcal{P}$-cut $\mathcal{C}_\mathcal{P}$ is $\mathcal{P}$-consistent if and only if:*

$$e \in \mathcal{C}_\mathcal{P} \wedge e' \prec e \Rightarrow e' \in \mathcal{C}_\mathcal{P} \vee \mathcal{P}(e') \in \mathcal{C}_\mathcal{P}$$

We extend the operator $\oplus$ to $\mathcal{P}$-consistent cuts. Note that we cannot add a promised reception as promised receptions only appear during manipulation of cuts and not in normal execution.

Notation 7.2 ($\oplus$ for $\mathcal{P}$-consistent cuts)
Let $\mathcal{C}_\mathcal{P}$ be a $\mathcal{P}$-consistent cut of an execution $(E, \prec)$.
Let $e \in E \backslash \mathcal{C}_\mathcal{P}$ such that e is minimal in $E \backslash \mathcal{C}_\mathcal{P}$ that is to say:

$$\nexists e' \in E \backslash \mathcal{C}_\mathcal{P}, e' \prec e$$

then, we define $\mathcal{C}_\mathcal{P} \oplus e$ as the set of events $\mathcal{C}_\mathcal{P} \cup \{e\}$ if $\mathcal{P}(e) \notin \mathcal{C}_\mathcal{P}$ and the replacement of $\mathcal{P}(e)$ by e $\mathcal{C}_\mathcal{P}\{\!\{\mathcal{P}(e) \leftarrow e\}\!\}$ if $\mathcal{P}(e) \in \mathcal{C}_\mathcal{P}$

Property 7.3 ($\oplus$ maintains $\mathcal{P}$-consistency) *$\mathcal{C}_\mathcal{P} \oplus e$ is a $\mathcal{P}$-*consistent cut *of $(E, \prec)$.*

Proof: As e is minimal in $E \backslash \mathcal{C}_\mathcal{P}$ and is not a promised reception, any event e' such that $e' \prec e$ belong to $\mathcal{C}_\mathcal{P}$. $\mathcal{C}_\mathcal{P} \oplus e$ is then $\mathcal{P}$-consistent . □

Property 7.4 ($\oplus$ maintains equality of $\mathcal{P}$-cuts)
If $\mathcal{C}^1_\mathcal{P}$ is a $\mathcal{P}$-consistentcut of $(E^1, \prec^1)$ and e is a minimal event of $E^1 \backslash \mathcal{C}^1_\mathcal{P}$
And $\mathcal{C}^2_\mathcal{P}$ is a $\mathcal{P}$-consistentcut of $(E^2, \prec^2)$ and e' is a minimal event of $E^2 \backslash \mathcal{C}^2_\mathcal{P}$ then

$$\mathcal{C}^1_\mathcal{P} = \mathcal{C}^2_\mathcal{P} \wedge e = e' \quad \Rightarrow \quad \mathcal{C}^1_\mathcal{P} \oplus e = \mathcal{C}^2_\mathcal{P} \oplus e'$$

Proof: This proof is very similar to the proof of Property 6.4. □

Property 7.5 ($\oplus$ maintains inclusion of $\mathcal{P}$-cuts)
If $\mathcal{C}^1_\mathcal{P}$ is a consistent cut of $(E^1, \prec^1)$ and e is a minimal event of $E^1 \backslash \mathcal{C}^1_\mathcal{P}$ and $\mathcal{C}^2_\mathcal{P}$ is a consistent cut of $(E^2, \prec^2)$ and e' is a minimal event of $E^2 \backslash \mathcal{C}^2_\mathcal{P}$ then

$$\mathcal{C}^1_\mathcal{P} \subseteq \mathcal{C}^2_\mathcal{P} \wedge e = e' \quad \Rightarrow \quad \mathcal{C}^1_\mathcal{P} \oplus e \subseteq \mathcal{C}^2_\mathcal{P} \oplus e'$$

Proof: Consequence of Property 6.4. □

Property 7.6 ($\oplus$ and equivalence of $\mathcal{P}$-cuts) *] If $\mathcal{C}^1_\mathcal{P}$ is a $\mathcal{P}$-consistent cut of $(E^1, \prec^1)$ and e is a minimal event of $E \backslash \mathcal{C}^1_\mathcal{P}$*
And $\mathcal{C}^2_\mathcal{P}$ is a $\mathcal{P}$-consistent cut of $(E^2, \prec^2)$ and e' is a minimal event of $E \backslash \mathcal{C}^2_\mathcal{P}$ then

$$\mathcal{C}^1_\mathcal{P} \equiv \mathcal{C}^2_\mathcal{P} \wedge e = e' \wedge e \neq rcv \quad \Rightarrow \quad \mathcal{C}^1_\mathcal{P} \oplus e \equiv \mathcal{C}^2_\mathcal{P} \oplus e'$$

Proof: This proof is a direct adaptation of the proof of Property 6.4. □

Property 7.7 (Piecewise determinism) *If $\mathcal{C}^1_{\mathcal{P}}$ and $\mathcal{C}^2_{\mathcal{P}}$ are $\mathcal{P}$-consistent cuts then:*

$$\begin{cases} \mathcal{C}^1_{\mathcal{P}} \equiv \mathcal{C}^2_{\mathcal{P}} \\ e_i \neq rcv \\ {e_i}' \neq rcv \end{cases} \Rightarrow \quad \mathcal{C}^1_{\mathcal{P}} \oplus e_i \equiv \mathcal{C}^2_{\mathcal{P}} \oplus {e_i}'$$

Proof: This proof is a direct consequence of Property 6.5. □

Property 7.8 (Generalised piecewise determinism) *If $\mathcal{C}^1_{\mathcal{P}}$ and $\mathcal{C}^2_{\mathcal{P}}$ are $\mathcal{P}$-consistent cuts then:*

$$\begin{cases} \mathcal{C}^1_{\mathcal{P}} \subseteq \mathcal{C}^2_{\mathcal{P}} \ \wedge \ e_i \notin \mathcal{C}^2_{\mathcal{P}} \\ e_i \neq rcv \ \wedge \ {e_i}' \neq rcv \end{cases} \Rightarrow \quad e_i = {e_i}' \ \wedge \ \mathcal{C}^1_{\mathcal{P}} \oplus e_i \subseteq \mathcal{C}^2_{\mathcal{P}} \oplus {e_i}'$$

Proof: This proof is a direct consequence of Property 7.7. □

7.4 Recovery and Correctness

This section supposes that we have a consistent enough cut $\mathcal{C}$ of a (first) execution $(E^1, \prec^1)$. This first execution encountered a fault enough later than $\mathcal{C}$, so that there is a consistent cut $\mathcal{C}^H$ later than the cut $\mathcal{C}$ such that all messages sent in $\mathcal{C}$ have been received in $\mathcal{C}^H$:

$$snd(M) \in \mathcal{C} \Rightarrow rcv(M) \in \mathcal{C}^H$$

This consistent cut will be used to constrain the second execution. The objective is to prove that the second execution can reach the global state corresponding to $\mathcal{C}^H$, or more precisely that $\mathcal{C}^H$ *will necessarily be a cut of the second execution.*

7.4.1 Recovery Cut

We introduce the notion of *recovery cut* associated to the consistent enough cut $\mathcal{C}$, and noted $\mathcal{C}^*$.

Definition 15 (Recovery cut of $\mathcal{C}$)

$$\mathcal{C}^* = \mathcal{C}\{\!\{e \leftarrow \mathcal{P}(e) | \exists e' \, (e', e) \in \Gamma \wedge e \in \mathcal{C} \wedge e' \notin \mathcal{C}\}\!\} + (\{\mathcal{P}(r)cv(M_k) | rcv(M_k) \in \mathcal{C}^H$$

Property 7.9 ($\mathcal{P}$-consistent recovery cut) *A recovery cut is a $\mathcal{P}$-consistent cut.*

7.4.2 Logged Messages

Some messages (in transit) have not been received in the first execution and would be lost if they were not logged somewhere. Let $\mathcal{L}$ be the set of logged messages:

$$\mathcal{L} = \{M_k | snd(M_k) \in \mathcal{C} \wedge rcv(M_k) \notin \mathcal{C}\}$$

By definition of $\mathcal{C}^H$ every logged message has been received in $\mathcal{C}^H$ and thus, every logged messages correspond to a promised reception:

$$M \in \mathcal{L} \Rightarrow \mathcal{P}(rcv(M))\mathcal{C}^*$$

Consequently, we can consider that, as those messages are logged and their receptions is promised, their receptions can be added without constrain to the recovery cut:

$$M \in \mathcal{L} \Rightarrow rcv(M) \text{is minimal in } E^2$$

7.4.3 Correctness of the protocol

The objective of this section is to prove the following theorem.

Theorem 7.1 (A consistent enough cut is sufficient for recovery)
If $\mathcal{C}$ *is a consistent enough cut of a (first) execution* $(E^1, \prec^1)$,
and $\mathcal{C}^H$ *is a consistent cut of* $(E^1, \prec^1)$ *later than the cut* $\mathcal{C}$,
and *a second execution* $(E^2, \prec^2)$ *starts from the state* $\mathcal{C}^* \oplus \{rcv(M_k) | M_k \in \mathcal{L}\}$,
then $\mathcal{C}^H$ *is a consistent cut of the second execution* $(E^2, \prec^2)$.

This theorem supposes that the second execution does not encounter a fault before reaching the cut $\mathcal{C}^H$.

Proof: This theorem will be proven by a recurrence proof. Let us first introduce notations for the proof. First, we define $\mathcal{C}^0 = Cut(\mathcal{C}^*)$.

Lemma 7.1 $\mathcal{C}^0$ *is a consistent cut of* $(E^1, \prec^1)$.

We introduce a set of consistent cuts $\mathcal{C}^k$ going from $\mathcal{C}^0$ to $\mathcal{C}^H$ defined as follows:

- $\mathcal{C}^0 = Cut(\mathcal{C}^*)$
- $\mathcal{C}^{k+1} = \mathcal{C}^k \oplus e$ such that e is minimal in $\mathcal{C}^H \backslash \mathcal{C}^k$

Notation 6.1 ensures:

$$\mathcal{C}^k \subseteq \mathcal{C}^{k+1} \subseteq \mathcal{C}^H$$

$\mathcal{C}^k$ is a consistent cut of $(E^1, \prec^1)$.
As execution between $\mathcal{C}^0$ and $\mathcal{C}^H$ is finite, $\mathcal{C}^k$ finally reaches $\mathcal{C}^H$:

$$\exists N, \mathcal{C}^N = \mathcal{C}^H$$

The objective is to build a set of $\mathcal{P}$-cuts belonging to the second execution.
The recurrence hypothesis (RH_n) is the following:

$$\exists \mathcal{C}^n_{\mathcal{P}}, \mathcal{C}^n \subseteq \mathcal{C}^n_{\mathcal{P}}$$

with

- $\mathcal{C}^0_{\mathcal{P}} = \mathcal{C}^*$
- $\mathcal{C}^n_{\mathcal{P}} \subseteq \mathcal{C}^{n+1}_{\mathcal{P}}$ and $\mathcal{C}^n_{\mathcal{P}}$ is a $\mathcal{P}$-consistent cut of E^2.

RH_0 is verified because $\mathcal{C}^0 \subseteq \mathcal{C}^* \subseteq \mathcal{C}^0_{\mathcal{P}}$.
$RH_n \Rightarrow RH_{n+1}$ let e be the event that is added to $\mathcal{C}^n$ in order to obtain $\mathcal{C}^{n+1} = \mathcal{C}^n \oplus e$.
First, if $e \in \mathcal{C}^n_{\mathcal{P}}$ then $e \in \mathcal{C}^* \subseteq \mathcal{C}$. Indeed, this could be added as a recurrence hypothesis: when an event is added to $\mathcal{C}^n_{\mathcal{P}}$, it is the same event that has been added to $\mathcal{C}^n$.
In that case, e does not need to be added to $\mathcal{C}^n_{\mathcal{P}}$:

$$\mathcal{C}^{n+1}_{\mathcal{P}} = \mathcal{C}^n_{\mathcal{P}}$$

Moreover:

$$e \in \mathcal{C}^* \subseteq \mathcal{C} \subseteq \mathcal{C}^H$$

Thus the position of e in $\mathcal{C}^n$ and in $\mathcal{C}^n_{\mathcal{P}}$ is necessarily the same as in $\mathcal{C}$: Property 6.3 ensures

$$\mathcal{C}^{n+1} \subseteq \mathcal{C}^{n+1}_{\mathcal{P}}$$

Else, $e \notin \mathcal{C}^n_{\mathcal{P}}$, then recurrence is performed by a case analysis on the kind of the event e.

- $e = e_i \neq rcv$, then Property 7.8 ensures that $\mathcal{C}^{n+1}_{\mathcal{P}} = \mathcal{C}^n_{\mathcal{P}} \oplus e_i$ and $\mathcal{C}^{n+1} \subseteq \mathcal{C}^{n+1}_{\mathcal{P}}$.
  Note that this means that, if a request is served, then the corresponding reception event has already occurred:

  $$e = srv(M) \Rightarrow rcv(M) \in \mathcal{C}^n \subseteq \mathcal{C}^n_{\mathcal{P}}$$

- $e = e_i = rcv(M)$, we must first ensure that e occurs in the second execution and is minimal, this will justify the definition $\mathcal{C}^{n+1}_{\mathcal{P}} = \mathcal{C}^n_{\mathcal{P}} \oplus e$.
  - e occurs in E^2, two cases are possible:
    * Either $send(M)$ has occurred in the second execution:

      $$\exists m,\ \mathcal{C}^{m+1}_{\mathcal{P}} = \mathcal{C}^m_{\mathcal{P}} \oplus send(M)$$

    * Or $send(M) \in \mathcal{C}^0_{\mathcal{P}}$, $send(M) \in \mathcal{C}^n$, and thus $send(M) \in \mathcal{C}^*$, by hypothesis, $rcv(M) \notin \mathcal{C}^* \subseteq \mathcal{C}^n$ and by definition of $\mathcal{L}$, $M \in \mathcal{L}$: the message has been logged and thus is replayed in the second execution.

– e is minimal in $E^2 \backslash \mathcal{C}^n_{\mathcal{P}}$: $e \notin \mathcal{C}^n_{\mathcal{P}}$ implies $e \notin \mathcal{C}^*$, and as $e \in \mathcal{C}^H$, we have $\mathcal{P}(e) \in \mathcal{C}^n_{\mathcal{P}}$, and thus e is minimal.

Property 7.5 ensures:

$$\mathcal{C}^{n+1}_{\mathcal{P}} = \mathcal{C}^n_{\mathcal{P}} \oplus e \subseteq \mathcal{C}^n \oplus e = \mathcal{C}^{n+1}$$

Finally,

$$\mathcal{C}^N = \mathcal{C}^H \subseteq \mathcal{C}^N_{\mathcal{P}}$$

Thus $\mathcal{C}^H$ is a consistent cut of $\mathcal{C}^N_{\mathcal{P}}$ and therefore, $\mathcal{C}^H$ is a consistent cut of the second execution $(E^2, \prec^2)$.

To be precise, we have moreover:

$$\mathcal{C}^H = \mathcal{C}^N_{\mathcal{P}}$$

and every logged message has been received because $M \in \mathcal{L}$ implies $rcv(M) \in \mathcal{C}^H$ by definition of $\mathcal{C}^H$. □

This proof also verifies an important correctness property: the message corresponding to a promised reception cannot be served. In fact, it is sufficient to guarantee that a promised reception will always be filled:

If $\mathcal{P}(e) \in \mathcal{C}_{\mathcal{P}}$ and $\mathcal{C}_{\mathcal{P}}$ is a $\mathcal{P}$-cut of the execution $(E, \prec)$ then $e \in E$.

Indeed if this is the case, then we have $(e = rcv(M), e' = serve(M)) \in \Sigma$ and necessarily e occurs before e' because of the minimal requirement.

8 Relation between ASP, ProActive and this work

This part highlights the properties of ASP and ProActive authorizing to perform the hypothesis that are done in this work.

Manipulating Messages First of all, ASP and ProActive authorize to manipulate a message that has been received but not served because such message has no effect on the applicative state of the activity.

Content of Promised Receptions Secondly, ASP execution is only characterized by the order of request senders. It is then sufficient to set that $\mathcal{P}(e)$ is the identifier of the sender of e [7].

This property is a consequence of the FIFO ordering of message in ASP and ProActive. In any case, FIFO ordering must be maintained by the protocol.

Constrained placement of Checkpoints Using Java implies that threads are non serializable and cannot be interrupted at any time. This is the reason why protocol cannot always recover the application from a consistent state. Consistent enough states have been introduced to caracterize recovery lines created by the protocol. Moreover, it can be useful to start from a non consistent state for other reasons (e.g. smaller internal state,...).

Existence of a Consistent Request Reception History It is not necessary to stop and serialize a thread to store (part of) the state of a pending request queue. Indeed, the pending request queue is not directly manipulated by the application, thus manipulation of the request queue can be interrupted. Consequently, the history closures on activities make a consistent cut of the execution.

Replies As shown in [7], reply messages can occur at any time, thus adding them to the protocol does not raise any technical difficulty except that in-transit reply messages must be logged.

Moreover, the protocol must guarantee that while the request reception history is not closed, the reply messages must have the same content (but they can occur in any order). This condition is a direct consequence of the Theorem 7.1 because it ensures that the internal state of activities will be the same in the first and second execution (until the history closure).

Causally Ordered Messages In ProActive, a rendez-vous phase make communications synchronous. However, request sending are asynchronous as they have a delayed consequence on the internal state of the activity. In [10], such executions are called *causally ordered.*

As shown in [10], causally ordered execution are particular cases of asynchronous ones. The correctness of the protocol is still valid for a causally ordered communication timing. Indeed, such kind of communications imply additional relations between events in the causality relation:

$$send(M) \prec_i e \Rightarrow rcv(M) \prec e$$

As the first execution verifies this relation and the second one is constrained to be equivalent until $\mathcal{C}^H$, the second execution will be causally correct. In the future, we would like to prove that the definition of the cut $\mathcal{C}^H$ is necessary to maintain causally ordered communication timing in the second execution.

This section justifies the way the extended cut is built: manipulating the request queue of a consistent enough cut.

References

[1] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[2] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.

[3] Sara Bouchenak. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, 1999.

[4] A. Bouteiller, F. Cappello, T. Herault, G.Krawezik, P. Lemarinier, and F. Magniette. A fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In *ACM/IEEE International Conference on Supercomputing*, 2003.

[5] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, pages 207–215, 1984.

[6] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd., November 1998.

[7] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*. ACM Press, 2004.

[8] T. Deepak Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[9] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 63–75, 1985.

[10] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communications, 1995.

[11] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. In ACM Press, editor, *ACM Transactions on Programming Languages and Systems*, pages 157–179.

[12] Christian Delbé. Causal ordering of asynchronous request services. In *Dependable Systems and Networks - Student Forum*, 2004.

[13] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[14] M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[15] Tomoya Enokido, Hiroaki Higaki, and Makoto Takizawa. Significant message precedence in object-based systems. In *ICPADS*, pages 284–291, 1998.

[16] J.M. Hélary, A. Mostefaoui, R.H.B. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. In *16th IEEE Symposium on Reliable Distributed Systems*, pages 183–190, 1997.

[17] J.M. Hélary, A Mostefaoui, and M. Raynal. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):865–877, 1999.

[18] Mick Jordan and Malcolm Atkinson. Orthogonal persistence for java - a mid-term report. In *Advances in Persistent Object Systems: Proceedings of POS-8 and PJAVA-3*, pages 335–352. Morgan Kaufmann, 1998.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565, July 1978.

[20] Hong Va Leong and Divyakant Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *International Conference on Distributed Computing Systems*, pages 227–234, 1994.

[21] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Proceedings of the 16th ICDCS*, pages 100–107, 1996.

[22] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. In *IEEE Transactions on Parallel and Distributed Systems*, volume 10, pages 703–713, 1999.

[23] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel And Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[24] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[25] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogenous architectures. In *Fault-Tolerant Parallel and Distributed Systems*, pages 73–92, 1998.

[26] J.C. Ruiz-Garcia, M.O. Killijian, J.C. Fabre, and S. Chiba. Optimized object state checkpointing using compile-time reflection. In *Workshop on Embedded Fault-Tolerant Systems (EFTS'98)*, pages 46–48, 1998.

[27] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. In *ACM Transactions on Computer Systems*, volume 1, pages 222–238, 1983.

[28] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems*, volume 3, pages 204–226, 1985.

[29] Yi-Min Wang and W. Kent Fuchs. Lazy checkpointing coordination for bounding rollback propagation. In *Symposium on Reliable Distributed Systems*, pages 78–85, 1993.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399