RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability*

Islene C. Garcia Gustavo M. D. Vieira Luiz E. Buzato

Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 3788 5876
Fax: +55 19 3788 5847
{islene, gdvieira, buzato}@ic.unicamp.br

Abstract

Checkpoint patterns that enforce rollback-dependency trackability (RDT) have only on-line trackable checkpoint dependencies and allow efficient solutions to the determination of consistent global checkpoints. The design of RDT checkpointing protocols that are efficient both in terms of the number of forced checkpoints and in terms of the data structures propagated by the processes is a very interesting research topic. Fixed-Dependency-After-Send (FDAS) is an RDT protocol based only on vector clocks, but that takes a high number of forced checkpoints. The protocol proposed by Baldoni, Helary, Mostefaoui and Raynal (BHMR) takes less forced checkpoints than FDAS, but requires the propagation of an $O(n^2)$ matrix of booleans.

In this paper, we introduce a new RDT protocol, called RDT-Partner, in which a process can save forced checkpoints in comparison to FDAS during checkpoint intervals in which the communication is bound to a pair of processes; a very interesting optimization in the context of client-server applications. Although the data structures required by the proposed protocol maintain the $O(n)$ complexity of FDAS, theoretical and simulation studies show that it takes virtually the same number of forced checkpoints than BHMR.

Keywords: distributed algorithms, fault-tolerance, checkpointing, rollback recovery, rollback-dependency trackability

*Islene C. Garcia receives financial support from FAPESP under grant 99/01293-2; she also received financial support from CNPq under grant 145563/98-7. Gustavo M. D. Vieira receives financial support from CAPES under grant 01P-15081/1997. This work is partially supported by FAPESP under grant 96/1532-9 for the Distributed Systems Laboratory, and grant 97/10982-0 for the High Performance Computing Laboratory. We also received financial support from PRONEX/FINEP, process 76.97.1022.00 (Advanced Information Systems).

Resumo

Padrões de *checkpoint* que garantem a propriedade *rollback-dependency trackability* (RDT) apresentam apenas dependências entre *checkpoints* que podem ser detectadas em tempo de execução e permitem a utilização de soluções eficientes para a determinação de *checkpoints* globais consistentes. O desenvolvimento de protocolos para *checkpointing* que garantem RDT de maneira eficiente tanto em termos do número de *checkpoints* tirados quanto em termos das estruturas de dados propagadas é um tópico de pesquisa muito interessante. *Fixed-Dependency-After-Send* (FDAS) é um protocolo RDT baseado apenas em vetores de relógios, mas que tira um número alto de *checkpoints* forçados. O protocolo proposto por Baldoni, Helary, Mostefaoui e Raynal (BHMR) tira menos *checkpoints* forçados que o FDAS, mas requer a propagação de uma matriz de booleanos $O(n^2)$.

Neste artigo, nós propomos um novo protocolo RDT, chamado RDT-Partner, no qual os processos podem economizar *checkpoints* forçados em relação ao FDAS durante intervalos de *checkpoint* nos quais a comunicação está restrita a um par de processos. Esta otimização pode ser muito interessante no contexto de aplicações cliente-servidor. Embora as estruturas de dados requeridas pelo protocolo proposto mantenham a complexidade $O(n)$ do protocolo FDAS, estudos teóricos e de simulação mostram que o RDT-Partner tira praticamente o mesmo número de *checkpoints* forçados que o BHMR.

Palavras chave: algoritmos distribuídos, tolerância a falhas, *checkpointing*, recuperação por retrocesso, *rollback-dependency trackability*

1 Introduction

A checkpoint is a recording in stable memory of a process' state. The set of all checkpoints taken by a distributed computation and the dependencies established among these checkpoints due to message exchanges form a checkpoint pattern. Checkpoint patterns that enforce rollback-dependency trackability (RDT) present only checkpoint dependencies that can be online trackable using vector clocks or dependency vectors, and allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [16]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, and distributed debugging [16].

In order to enforce the RDT property, an RDT checkpointing protocol [1, 16] allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints). Some RDT checkpointing protocols presented in the literature are based only on checkpoints, message-send, and message-receive events: No-Receive-After-Send, Checkpoint-After-Send, Checkpoint-Before Receive, and Checkpoint-After-Send-Before-Receive [16]. These protocols are instantiations of the Mark-Receive-Send model [13] and are prone to induce a large number of forced checkpoints.

An effort to reduce the number of forced checkpoints can be done through the collaboration of the processes to maintain and propagate control structures. The decision to take a forced checkpoint must be based on information piggybacked on application messages; there are no control messages and no knowledge about the future of the computation. An important goal is to develop an efficient protocol both in terms of the number of forced checkpoints and in terms of the complexity of the required data structures.

Fixed-Dependency-Interval (FDI) [9, 16] and Fixed-Dependency-After-Send (FDAS) [16] maintain and propagate vector clocks. They force the vector clock of a process to remain unchanged during an entire checkpoint interval (FDI) or after the first message-send event of an interval (FDAS). Trying to reduce even more the number of forced checkpoints, Baldoni, Helary, Mostefaoui, and Raynal have explored the RDT property at the message level [1, 2, 3]. A protocol proposed by them, called BHMR, never takes more forced checkpoints than FDAS [14]. However, the more elaborated condition used by BHMR requires the propagation of an additional $O(n^2)$ matrix of booleans [1].

In this paper, we introduce a new RDT protocol, called RDT-Partner, in which a process can save forced checkpoints in comparison to FDAS during checkpoint intervals in which the communication is bound to a pair of processes; a very interesting optimization in the context of client-server applications. This protocol is based on a recent result that characterized the strongest condition that can be used on-line by an RDT checkpointing protocol [6]. Although the data structures required by the proposed protocol maintain the $O(n)$ complexity of FDAS, theoretical and simulation studies show that it takes virtually the same number of forced checkpoint than BHMR.

The paper is structured as follows. Section 2 introduces rollback-dependency trackability. Section 3 describes the RDT-Partner protocol. Section 4 compares the proposed protocol with FDAS and BHMR. Section 5 summarizes the paper.

2 Rollback-Dependency Trackability

The literature presents two approaches to define rollback-dependency trackability. The first one is based on the study of on-line trackable dependencies, implemented through the use of vector clocks or dependency vectors [16]. The other approach is based on the study of sequence of messages [1, 2, 3, 6].

2.1 Computational model

A distributed computation is composed of n sequential processes $(p_1, \ldots, p_n)$ that communicate only by exchanging messages. Messages cannot be corrupted, but can be delivered out of order or lost. The activity of a process is modeled as a sequence of events that can be divided into internal events and communication events realized through the sending and the receiving of messages. Checkpoints are internal events; each process takes an initial checkpoint (immediately after execution begins) and a final checkpoint (immediately before execution ends). Figure 1 illustrates a space-time diagram [10] augmented with checkpoints (black squares).

Let c_i^γ denote the γth checkpoint taken by p_i. A non-final checkpoint c_i^γ, $\gamma \geq 1$, and its immediate successor $c_i^{\gamma+1}$ define a checkpoint interval I_i^γ. This interval represents the set of events produced by p_i between c_i^γ and $c_i^{\gamma+1}$, including c_i^γ and excluding $c_i^{\gamma+1}$.

2.2 Checkpoint dependencies

A consistent global checkpoint is a set of checkpoints, one per process, that could have been seen by an idealized external observer [4]. Checkpoints that are part of the same consistent

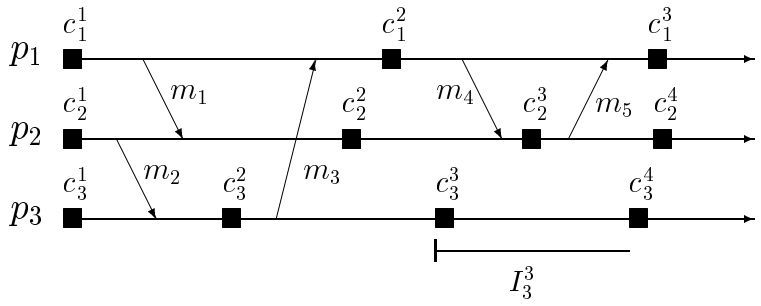


Figure 1: A distributed computation

global checkpoint cannot be related by checkpoint dependencies. Netzer and Xu have determined that checkpoint dependencies are created by sequences of messages called *zigzag paths* [12].

Definition 2.1 Zigzag path—*A sequence of messages $\mu = [m_1, \ldots, m_k]$ is a zigzag path that connects c_a^α to c_b^β if (i) p_a sends m_1 after c_a^α; (ii) if m_i, $1 \leq i < k$, is received by p_c, then m_{i+1} is sent by p_c in the same or a later checkpoint interval; (iii) m_k is received by p_b before c_b^β.*

Two types of zigzag paths can be identified: (i) causal paths and (ii) non-causal paths. A zigzag path is causal if the reception of each message but the last one causally precedes the send event of the next one in the sequence. In Figure 1, $[m_2, m_3]$ is a causal path from c_2^1 to c_1^2 and $[m_1, m_2]$ is a non-causal zigzag path from c_1^1 to c_3^2.

A zigzag path that connects a checkpoint to itself is called a Z-cycle and identifies a useless checkpoint, that is, a checkpoint that cannot be part of any consistent global checkpoint [12]. In Figure 1, $[m_3, m_1, m_2]$ and $[m_5, m_4]$ are examples of Z-cycles; c_3^2 and c_2^3 are useless checkpoints.

Let $\zeta = [m_1]$ and $\zeta' = [m_2, m_3]$ be two zigzag paths; the concatenation of ζ and ζ' will be denoted by $\zeta \cdot \zeta'$, or $\zeta \cdot [m_2, m_3]$, or $[m_1] \cdot \zeta'$, or $[m_1, m_2, m_3]$ [3].

2.3 Vector clocks

A transitive dependency tracking mechanism can be used to capture causal dependencies among checkpoints. Each process p_i maintains and propagates a size-n vector clock vc_i, such that $vc_i[i]$ is initialized to 1 and all other entries to 0. The entry $vc_i[i]$ represents the current interval of p_i and it is incremented immediately before a new checkpoint is taken. Every other entry $vc_i[j]$, $j \neq i$, represents the highest checkpoint index of p_j that p_i causally depends and it is updated every time a message m with a greater value of $vc_m[j]$ arrives to p_i.

Figure 2 depicts the vector clocks established during a distributed computation. Note that the vector clock associated to checkpoint c_3^2 is $(1,\ 1,\ 2)$ and it correctly represents the dependencies of this checkpoint. Unfortunately, not all dependencies can be tracked on-line. For example, the vector clock associated to checkpoint c_3^3 does not capture the existence of a zigzag path that connects c_1^2 to c_3^3.

Definition 2.2 Rollback-dependency trackability (vector clock characterization)
A checkpoint pattern enforces RDT if all checkpoint dependencies can be on-line trackable through the use of vector clocks.

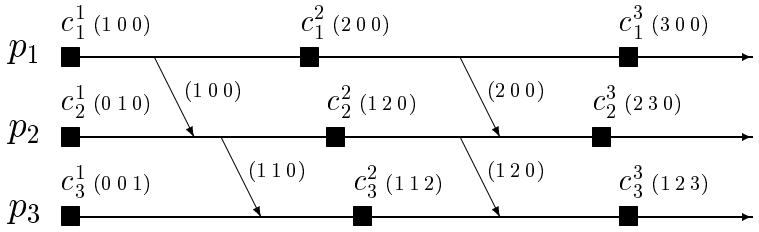


Figure 2: A distributed computation with vector clocks

RDT is a desirable property because when all dependencies are causal dependencies, efficient algorithms can be used to construct consistent global checkpoints. Also, an RDT checkpoint pattern does not admit useless checkpoints [16].

When a communication-induced checkpointing protocol is used to enforce RDT, processes take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints) [5, 11]. Forced checkpoints can be taken upon the arrival of a message, but before this message is processed by the computation.

In the FDAS protocol, a forced checkpoint is taken upon the reception of a message if (i) at least one message has been sent during the current interval, and (ii) at least one entry of the vector clock is about to be changed [16]. Figure 3 shows the same scenario of Figure 2 under FDAS; the resulted checkpoint pattern enforces RDT.

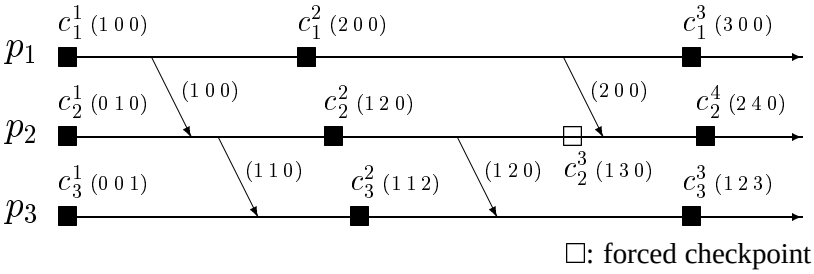


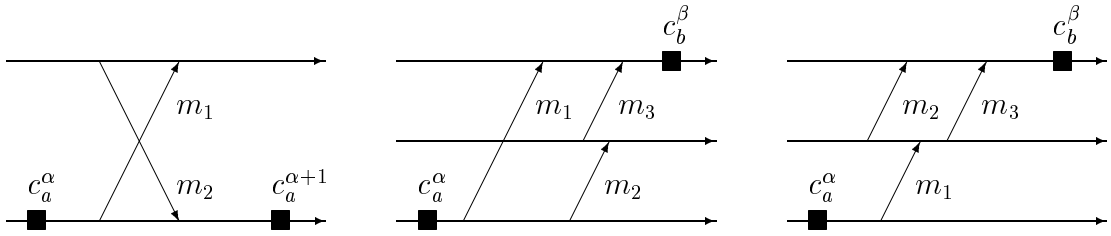
Figure 3: A distributed computation under FDAS

2.4 A message-based characterization of RDT

Trying to design more efficient protocols, Baldoni, Helary, Mostefaoui, and Raynal have explored the RDT property in the message level [1, 2, 3].

2.4.1 Causal doubling

A non-causal zigzag path is doubled by a causal one if the pair of checkpoints related by that zigzag path is also related by a causal dependency [3]. In Figure 4 (a) $[m_1, m_2]$ connects c_a^α to $c_a^{\alpha+1}$ and it is trivially doubled by the execution flow of p_a. In Figure 4 (b), the non-causal zigzag path $[m_2, m_3]$ that connects c_a^α to c_b^β is causally doubled by m_1. In Figure 4 (c), $[m_1, m_2]$ is causally doubled by $[m_1, m_3]$.



(a) $[m_1, m_2]$ is trivially doubled

(b) $[m_2, m_3]$ is doubled by m_1

(c) $[m_1, m_2]$ is doubled by $[m_1, m_3]$

Figure 4: Causal doubling

Definition 2.3 Trivial doubling—*A non-causal zigzag path from c_a^α to c_b^β is trivially doubled if $a = b$ and $\alpha < \beta$.*

Definition 2.4 Causal doubling—*A non-causal zigzag path from c_a^α to c_b^β is* causally doubled *if it is trivially doubled or there exists a causal path μ from c_a^α to c_b^β.*

A Z-cycle cannot be causally doubled. In Figure 5, the zigzag path $[m_1, m_2]$ cannot be doubled by the process execution because a causal dependency from c_b^β to c_b^β cannot exist.

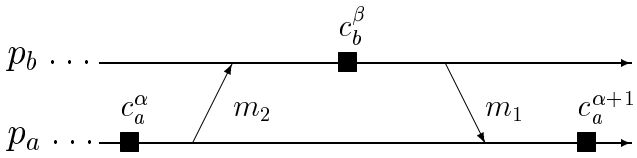


Figure 5: Z-cycle: $[m_1, m_2]$ cannot be causally doubled

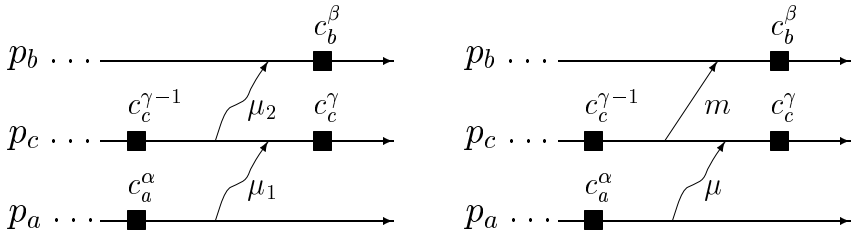
Definition 2.5 Rollback-Dependency Trackability (zigzag path characterization)
A checkpoint pattern enforces the RDT property if all zigzag paths are causally doubled.

Since it appears to be very hard to track non-causally doubled zigzag paths, Baldoni, Helary and Raynal have suggested an approach that tries to minimize the number of non-causal zigzag paths that must be causally doubled to enforce RDT [3].

2.4.2 PCM-paths

A non-causal zigzag path can be seen as a concatenation of causal paths $\mu_1.\mu_2.\cdots.\mu_k$. The number of causal paths in a non-causal zigzag path is the *order* of this path. A non-causal zigzag path of order 2 (CC-path) is composed of exactly two causal paths μ_1 and μ_2 (Figure 6 (a)). A CM-path is a non-causal zigzag path of order 2 composed of a causal path μ and a single message m (Figure 6 (b)).

In order to further reduce the set of zigzag paths that must be doubled, let us consider an additional constraint on the causal path μ of a CM-path $\mu.[m]$. Let μ be a *prime* path from c_a^α to c_c^γ, that is, the first path that brings to p_c the knowledge about c_a^α. In Figure 7 (a), μ is not prime due to existence of μ'; in Figure 7 (b), μ is a prime path.



(a) CC-path (b) CM-path

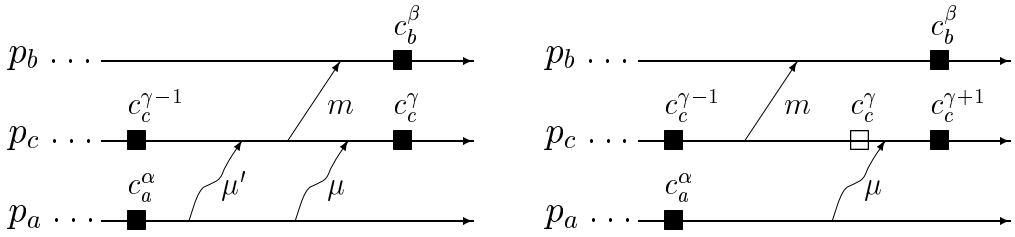
Figure 6: Non-causal zigzag paths of order 2

Definition 2.6 Prime path—*A causal path μ from c_a^α to c_c^γ is prime if the last message of μ is the first message that brings to p_c the knowledge about c_a^α.*

Definition 2.7 PCM-path—*A PCM-path is a non-causal zigzag path composed of a **p**rime **c**ausal path μ and a single **m**essage m.*

Definition 2.8 Rollback-Dependency Trackability (PCM-path characterization)
A checkpoint pattern enforces the RDT property if all PCM-paths are causally doubled.

This characterization leads to the development of a simple RDT protocol that *breaks* all PCM-paths, that is, induces a process to take a forced checkpoint upon the establishment of a PCM-path. This behavior is illustrated in Figure 7 and can be seen as a reinterpretation of the FDAS protocol [2, 3].



(a) $\mu.[m]$ is not a PCM-path (b) p_c must *break* $\mu.[m]$

Figure 7: FDAS can be seen as a protocol that breaks PCM-paths

2.4.3 Visibly doubling

A PCM-path need not to be broken by a process p_c if, upon the establishment of this path, p_c is able to detect that it is already causally doubled [1, 2, 3]. Figure 8 shows a PCM-path $\mu \cdot [m]$ that is causally doubled by a causal path ν; process p_c is able to detect this doubling due to the causal path ν'. In this case, $\mu \cdot [m]$ is *visibly doubled* by ν.

Definition 2.9 Visibly Doubled PCM-path—*A PCM-path $\mu \cdot [m]$ is visibly doubled if (i) is causally doubled by a causal path ν and (ii) the reception of the last message of ν causally precedes the sending of the last message of μ.*

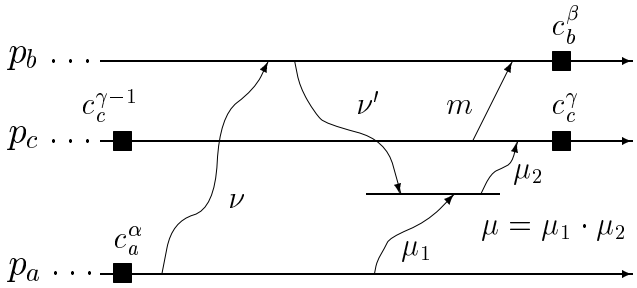


Figure 8: A visibly doubled PCM-path

Definition 2.10 Rollback-Dependency Trackability (Visibly doubled characterization)
A checkpoint pattern enforces the RDT property if all PCM-paths are visibly doubled.

BHMR is a protocol that breaks all non-visibly doubled PCM-paths. Unfortunately, this more efficient strategy in terms of forced checkpoints is less efficient in terms of data structures [2]. Each process must maintain and propagate information regarding other processes' knowledge about causal relantionships, requiring an $O(n^2)$ data structure. Indeed, in the implementation of BHMR [1] each process maintains and propagates an $O(n)$ vector clock, an $O(n)$ vector of booleans, and an $O(n^2)$ matrix of booleans.

2.5 The minimal characterization of RDT

Recently, we have determined the minimal (strongest) condition that can be used to enforce RDT [6]. A PMM-path is a non-causal zigzag path composed of two single messages m_1 and m_2, such that m_1 is prime (Figure 9). We have proved that a protocol that breaks all non-visibly doubled PMM-paths must enforce RDT [6]. In the next Section, we are going to explore this characterization to propose a protocol that is efficient both in terms of forced checkpoints and in terms of data structures.

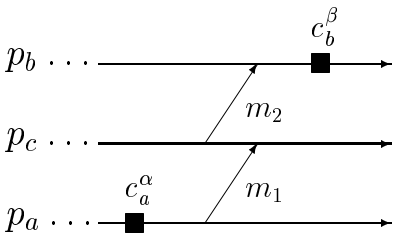


Figure 9: A PMM-path

Definition 2.11 PMM-path—*A PMM-path is a non-causal zigzag-path composed of a **p**rime single **m**essage m_1 and a single **m**essage m_2.*

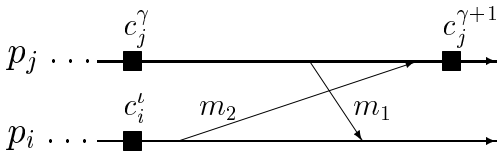
Definition 2.12 Rollback-Dependency Trackability (minimal characterization)
A checkpoint pattern enforces the RDT property if all PMM-paths are visibly doubled.

3 RDT-Partner protocol

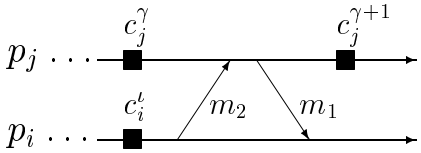
The implementation of a protocol that keeps track of all visibly-doubled paths seems to require $O(n^2)$ information [2]. In this Section, we introduce an $O(n)$ protocol, called RDT-Partner, that keeps track of only trivially-doubled PMM-paths.

3.1 PMM-cycles

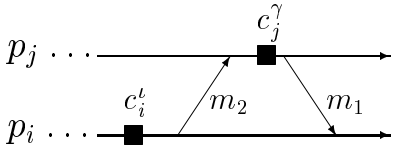
A PMM-path $[m_1, m_2]$ that starts and finishes in the same process is a PMM-cycle. Some PMM-cycles are trivially doubled by the process execution; others cannot be causally doubled because they form Z-cycles. Figure 10 illustrates the possibilities of PMM-cycles from the perspective of a process p_i. In Figure 10 (a) and (b), $[m_1, m_2]$ is trivially doubled due to the p_j's execution flow from c_j^γ to $c_j^{\gamma+1}$. In Figure 10 (c), $[m_1, m_2]$ is a Z-cycle and cannot be causally doubled (a causal dependency from c_j^γ to c_j^γ cannot exist).



(a) When m_1 is sent, process p_j has no knowledge about c_i^ι
($[m_1, m_2]$ is trivially doubled)



(b) When m_1 is sent, process p_j has knowledge about c_i^ι
($[m_1, m_2]$ is trivially doubled)



(c) When m_1 is sent, process p_j has knowledge about c_i^ι
($[m_1, m_2]$ cannot be causally doubled)

Figure 10: PMM-cycles

Let us assume that a process p_i maintains and propagates a vector clock vc_i and let us analyze the complexity required to distinguish trivially doubled PMM-cycles from Z-cycles. First, let us consider the scenario depicted in Figure 10 (a), in which upon the sending of m_1, p_j has no knowledge about c_i^ι. Process p_i can deduce, upon the reception of m_1, that m_2 will be received during I_j^γ or a later checkpoint interval and that $[m_1, m_2]$ will be trivially doubled by the p_j's execution flow. To identify this scenario, p_i must evaluate the following condition: $vc_{m_1}[i] < vc_i[i]$.

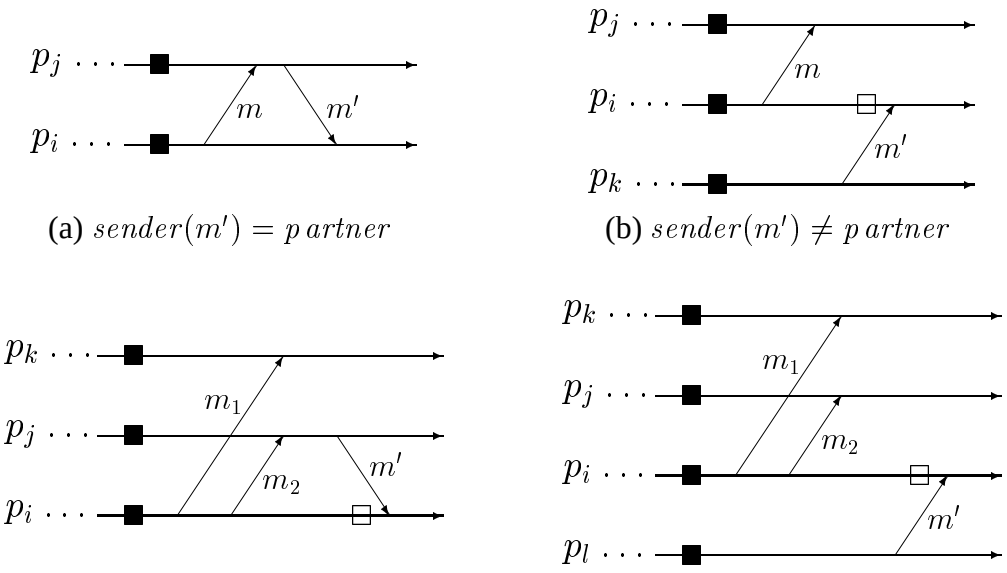
Let us consider the other scenarios, in which upon the sending of m_1, p_j has knowledge about c_i^t. In both cases, upon the reception of m_1, p_i would detect that $vc_{m_1}[i] = vc_i[i]$. Thus, p_i needs additional information to distinguish the scenario depicted in Figure 10 (b) from the scenario depicted in Figure 10 (c). Message m_1 can carry a boolean value to indicate whether p_j has taken a forced checkpoint after the last time it received knowledge about a new checkpoint index in p_i.

3.2 Partner relationships

In the previous section, we have shown that a process can efficiently detect trivially doubled PMM-cycles. The detection of visibly doubled PMM-paths is similar to the detection of visibly doubled PCM-paths and seems to require $O(n^2)$ information [2]. This means that a process p_i can efficiently save a forced checkpoint only in a constrained situation, in which p_i sends a message to p_j and receives another message from p_j.

Definition 3.1 Partner—*Process p_j is a partner of process p_i if p_i has sent a message to p_j during the current interval.*

Let us consider that p_i has just one partner, say p_j, in the current interval. If p_i receives a message m' from p_j (Figure 11 (a)) it can save a forced checkpoint if a Z-cycle is not established, as described in Section 3.1. If p_i receives a message m' from another process, say p_k (Figure 11 (b)), that forms a PMM-path, it must take a forced checkpoint before processing m'. Process p_i will take this forced checkpoint even if this PMM-path is visibly doubled, because it will not be able to efficiently detect this doubling.



(a) $sender(m') = partner$

(b) $sender(m') \neq partner$

(c) $sender(m')$ is one of the partners

(d) $sender(m')$ is not one of the partners

Figure 11: The behavior of the RDT-Partner protocol

Let us consider that p_i has more than one partner, say p_k and p_j, and it receives a message m' from p_j (Figure 11 (c)). Process p_i can efficiently detect that $[m', m_2]$ is not a Z-cycle, but it cannot efficiently detect whether the PMM-path $[m', m_1]$ is visibly doubled. Thus, process p_i must take a forced checkpoint before processing m'. A similar situation occurs when p_i receives a message from another process, say p_l (Figure 11 (d)). In this case, two PMM-paths $[m', m_1]$ and $[m', m_2]$ are formed, and p_i must take a forced checkpoint before processing m'.

Finally, Figure 12 illustrates that forced checkpoints can be saved in a nested sequence of partner interactions.

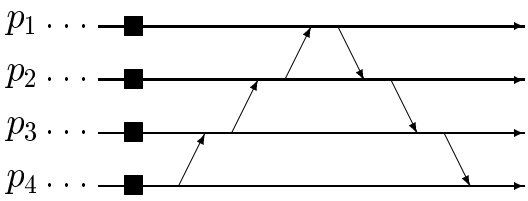


Figure 12: A nested sequence of partner interactions

3.3 RDT-Partner implementation

An implementation of RDT-Partner is described in class `Partner` (Class 3.1), using Java[1] [7]. Every process, say p_i, maintains and propagates a vector clock vc_i in order to characterize casual precedence among checkpoints. When p_i sends a message, vc_i is piggybacked onto it. Before consuming a message m, process p_i takes a component-wise maximum of vc_i and vc_m. When p_i takes a checkpoint, it increments $vc_i[i]$.

Process p_i also maintains a variable $partner\_pid$ that keeps track of partner relationships during checkpoint intervals:

- $partner\_pid =$ `NO_PARTNER` indicates that p_i has not sent any message;
- $partner\_pid = j$ indicates that p_i has sent message(s) only to p_j;
- $partner\_pid =$ `MORE_THAN_ONE_PARTNER` indicates that p_i has sent messages for more than one process.

In order to distinguish trivially doubled PMM-cycles from Z-cycles, each process p_i maintains a vector of booleans $simple$, such that $simple[j]$, $j \neq i$, indicates that a checkpoint has not been taken after p_i has received knowledge about $c_j^{vc_i[j]}$. When p_i sends a message to p_j it piggybacks $simple[j]$ onto the message. When process p_i takes a checkpoint it sets all entries in $simple$ to false, except its ith entry.

A checkpoint is induced by p_i before delivering a message m if a PMM-path is detected and one of the following condition holds: (i) it is not a PMM-cycle or (ii) it is a PMM-cycle, but forms a Z-cycle.

[1]We have chosen Java because it is easy to read and has a precise description. Java is a trademark of Sun Microsystems, Inc.

Class 3.1 RDT_Partner.java

```
public class RDT_Partner {
    public static final int N = 100; // Number of processes in the computation
    public int pid; // A process unique identifier in the range 0..N-1
    protected int [ ] vc = new int [N]; // Vector clock, automatically initialized to (0 ... 0)
    protected boolean [ ] simple = new boolean [N]; // Keeps track of simple PMM-cycles

    protected int partner_pid;
    public static final int NO_PARTNER = - 1;
    public static final int MORE_THAN_ONE_PARTNER = N + 1;

    public class Message {
        public int sender, receiver;
        public int [ ] vc;
        public boolean simple;
        // Message body
    }

    public void takeCheckpoint() {
        vc[pid]++; // Increment checkpoint index immediately before the checkpoint
        // Save state to stable memory
        partner_pid = NO_PARTNER;
        for (int i = 0; i < N; i++) simple[i] = i == pid;
    }

    public RDT_Partner(int pid) { this.pid = pid; }   // Constructor

    public void run() { takeCheckpoint(); }   // Initiate execution

    public void finalize() { takeCheckpoint(); }   // Finish execution

    public void sendMessage(Message m) {
        m.vc = (int [ ]) vc.clone(); // Piggyback vc onto the message
        m.simple = simple[m.receiver];
        if (partner_pid == NO_PARTNER) partner_pid = m.receiver;
        else if (partner_pid ≠ m.receiver) partner_pid = MORE_THAN_ONE_PARTNER;
        // Send message
    }

    public void receiveMessage(Message m) {
        if (m.vc[m.sender] > vc[m.sender] && partner_pid ≠ NO_PARTNER) { // PMM-path
            if (partner_pid ≠ m.sender || // Not a PMM-cycle
                    (partner_pid == m.sender && m.vc[pid] == vc[pid] && !m.simple))) // Z-cycle
                takeCheckpoint(); // Forced checkpoint
            simple[m.sender] = true;
        }
        for (int i = 0; i < N; i++) // Update the vector clock
            if (m.vc[i] > vc[i]) vc[i] = m.vc[i];
        // Message is processed by the application
    }
}
```

4 A comparison with FDAS and BHMR

4.1 Simulation results

Our experimental data was obtained using the simulation toolkit for quasi-synchronous algorithms Metapromela [15]. This toolkit was built atop Spin [8], a tool to simulate and perform consistency analysis of distributed protocols and algorithms. In Metapromela, the processes are asynchronous and the simulated execution of each process is a succession of atomic events of three types: internal, message-send and message-receive. The only type of internal event that is relevant for checkpointing is the occurrence of a basic checkpoint. The environment is controlled by adjusting the distribution of these events and the communication network.

The experiment was performed considering a complete network, i.e., each pair of processes is connected by a bidirectional communication channel. The channels do not lose, corrupt or change the order of messages. For each experimental point it was considered the average of 10 measurements. Each measurement was taken by the execution of each of the studied protocols under the same pattern of messages and basic checkpoints, that is, under exactly the same history of events. We counted the ratio of forced checkpoints per basic checkpoint over a period of 300 basic checkpoints for each process. Figure 13 shows the results obtained for $2 \leq n \leq 20$, where n is the number of processes in the computation.

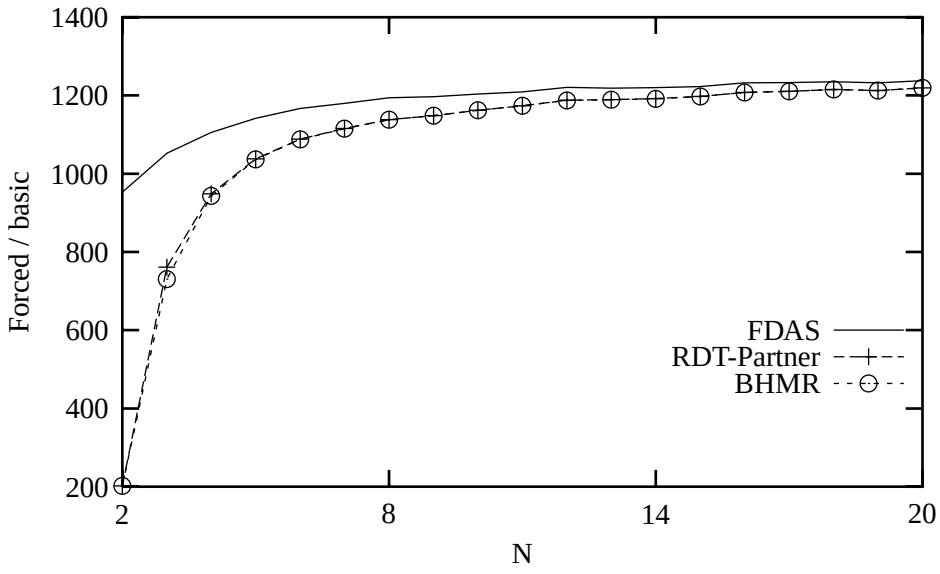


Figure 13: Simulation results

FDAS takes consistently more checkpoints than BHMR and RDT-Partner. For small values of n, say 2 or 3, the difference is large, but the difference becomes less significant as the values of n grow. BHMR and RDT-Partner take almost the same number of forced checkpoints with a very small difference for $3 \leq n \leq 5$. In the following section, we give a theoretical explanation for this behavior.

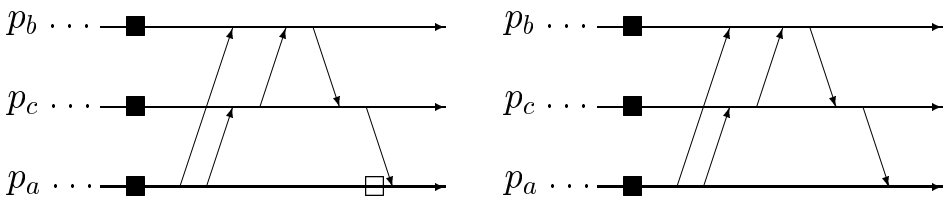
4.2 Theoretical argumentation

The theoretical analysis performed in [14] proved that any protocol that uses a stronger condition than FDAS to take forced checkpoints will outperform FDAS. Thus, FDAS cannot take less checkpoints than BHMR and RDT-Partner, since both protocols break doubled PMM-cycles, and FDAS is not able to track such dependencies. Figure 14 shows two-message scenarios in which BHMR and RDT-Partner can save a checkpoint in comparison to FDAS. Since in our simulation the communication is uniform, the probability of these scenarios is higher for small values of n, explaining the behavior of the curves.

(a) FDAS (b) BHMR and RDT-partner

Figure 14: BHMR and RDT-Partner can save forced checkpoints in comparison to FDAS

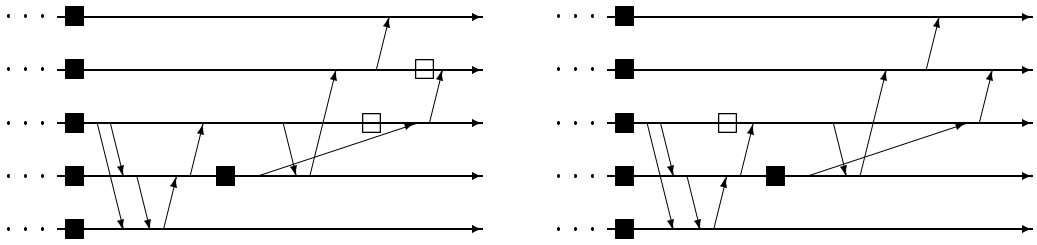
In comparison to RDT-Partner, BHMR can also detect visibly doubled PCM-paths, potentially saving more forced checkpoints. However, the minimum scenario in which BHMR can save a checkpoint in comparison to RDT-Partner requires five messages (Figure 15) and is likely to occur less frequently during a distributed computation. In a system composed of three processes, this situation could be more likely, explaining the slightly difference in the curves around this point ($n = 3$).



(a) RDT-partner (b) BHMR

Figure 15: BHMR can save a forced checkpoint in comparison to RDT-Partner

Although this situation cannot occur with FDAS, it is possible for a protocol based on a weaker condition to outperform a protocol based on a stronger condition [14]. BHMR uses a condition stronger than RDT-Partner and Figure 16 illustrates a scenario in which RDT-Partner can save a forced checkpoint in comparison to BHMR. This scenario involves five processes and ten messages. We can consider such scenarios less probable in distributed computations.



(a) BHMR (b) RDT-Partner

Figure 16: RDT-Partner can save a forced checkpoint in comparison to BHMR

5 Conclusion

Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of consistent global checkpoints [16]. In this paper, we have introduced a new RDT protocol, called RDT-Partner, that is efficient both in terms of the number of forced checkpoints and in terms of the complexity of the required data structures.

We have presented theoretical and simulation studies to show that RDT-Partner presents a very good compromise between the stronger condition of BHMR [1] and the smaller control structure of FDAS [16]. In conclusion, RDT-Partner is so far the best protocol to adopt in practical implementations.

References

[1] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A communication-induced checkpoint protocol that ensures rollback dependency trackability. In *IEEE Symposium on Fault Tolerant Computing (FTCS'97)*, pages 68–77, 1997.

[2] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-dependency trackability: A minimal characterization and its protocol. Technical Report 1173, IRISA, Mar. 1998.

[3] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-dependency trackability: Visible characterizations. In *18th ACM Symposium on the Principles of Distributed Computing (PODC'99)*, Atlanta (USA), May 1999.

[4] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computing Systems*, 3(1):63–75, Feb. 1985.

[5] E. N. Elnozahy, D. Johnson, and Y.M.Yang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[6] I. C. Garcia and L. E. Buzato. On the minimal characterization of rollback-dependency trackability property. In *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, Phoenix, Arizona, EUA, Apr. 2001. To appear.

[7] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Java Series. Addison–Wesley, Sept. 1996.

[8] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), May 1997.

[9] T. R. K. Venkatesh and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–303, 1987.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[11] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Trans. on Parallel and Distributed Systems*, 10(7), July 1999.

[12] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[13] D. L. Russell. State restoration in systems of communicating processes. *IEEE Trans. on Software Engineering*, 6(2):183–194, Mar. 1980.

[14] J. Tsai, S. Y. Kuo, and Y. M. Wang. Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability. *IEEE Trans. on Parallel and Distributed Systems*, Oct. 1998.

[15] G. M. D. Vieira. Metapromela: A toolkit for simulation of checkpointing algorithms. In *Students Forum of the IX Brazilian Symposium on Fault-Tolerant Computers*, Mar. 2001.

[16] Y. M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Trans. on Computers*, 46(4):456–468, Apr. 1997.