Using Common Knowledge to Improve Fixed-Dependency-After-Send*

Islene C. Garcia   Luiz E. Buzato
Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 788 5876
Fax: +55 19 788 5847
{islene,buzato}@dcc.unicamp.br

Abstract

Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of consistent global checkpoints that include a given set of checkpoints. Fixed-Dependency-After-Send (FDAS) is a well-known RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event. In this paper, we explore processes' common knowledge about their behavior to derive a more efficient condition to induce checkpoints under FDAS. We consider that our approach can be used to improve other RDT checkpointing protocols.

Keywords: Distributed systems; Fault-tolerance; Distributed checkpointing; Rollback-dependency trackability

1 Introduction

A checkpoint is a stable memory record of a process' state. A consistent global checkpoint is a set of checkpoints, one per process, that could have been seen by an idealized observer external to the computation [2]. The set of all checkpoints taken by a distributed computation forms a checkpoint pattern. Checkpoint patterns that enforce the rollback-dependency trackability (RDT) property allow efficient solutions to the determination of the maximum and minimum consistent global checkpoints that include a set of checkpoints [8]. Many applications can benefit from these algorithms: rollback recovery, software error recovery, deadlock recovery, mobile computing and distributed debugging [8].

In order to enforce the RDT property, an RDT checkpointing protocol [1, 8] allows processes to take checkpoints asynchronously (basic checkpoints), but they may be induced by the protocol to take additional checkpoints (forced checkpoints). Fixed-Dependency-After-Send (FDAS) is a well-known RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event [8]. Upon the reception of a

*This work has been supported by FAPESP under grant no. 99/01293-2 for Islene C. Garcia and grant no. 96/1532-9 for the Laboratory of Distributed Systems. Islene C. Garcia received financial support from CNPq under grant no. 145563/98-7. We also received financial support from PRONEX/FINEP, process no. 76.97.1022.00 (Advanced Information Systems).

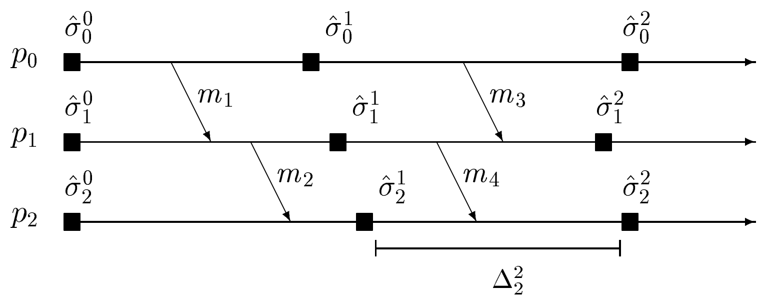


Figure 1: A distributed computation

message, a process must take a forced checkpoint if an entry of its dependency vector is about to change [1, 3, 7, 8].

The usual approach to implement FDAS does not take advantage of the processes' common knowledge about their behavior. Consider a scenario where a process p_i receives a message from p_j sent during an interval already known by p_i. Since p_i knows that p_j cannot increase its dependency vector after a send, it can skip the verification of checkpoint dependencies. This simple observation reduces the complexity of the decision to take a forced checkpoint from $O(n)$ to $O(1)$, where n is the number of processes in the computation.

The paper is structured as follows. Section 2 describes the computational model adopted. Section 3 introduces rollback-dependency trackability. Section 4 describes Fixed-Dependency-After-Send. Section 5 presents and proves the correction of the proposed optimization. Section 6 summarizes the paper.

2 Computational Model

A distributed computation is composed of n sequential processes $(p_0, \ldots, p_{n-1})$ 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 reception 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 [6] augmented with checkpoints (black squares).

Let $\hat{\sigma}_i^\gamma$ denote the γth checkpoint taken by p_i. Checkpoint $\hat{\sigma}_i^{\gamma-1}$, $\gamma > 0$, and its immediate successor $\hat{\sigma}_i^\gamma$ define a checkpoint interval Δ_i^γ. This interval represents the set of events produced by p_i between $\hat{\sigma}_i^{\gamma-1}$ and $\hat{\sigma}_i^\gamma$.

3 Rollback-Dependency Trackability

This Section introduces the concept of rollback-dependency trackability as defined by Wang [8], beginning with the definition of an R-graph, a digraph used to capture dependencies among checkpoints [8].

Definition 3.1 R-graph—*In an R-graph, each node represents a checkpoint and a directed edge is drawn from $\hat{\sigma}_a^\alpha$ to $\hat{\sigma}_b^\beta$ if (i) $a = b$ and $\beta = \alpha + 1$ or (ii) $a \neq b$, and a message m is sent in Δ_a^α and received in Δ_b^β.*

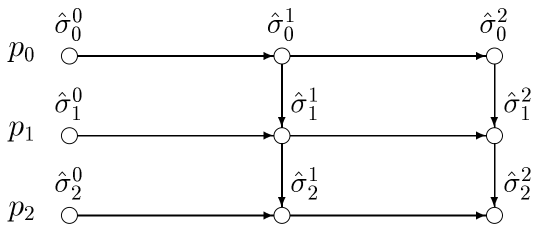


Figure 2: R-graph

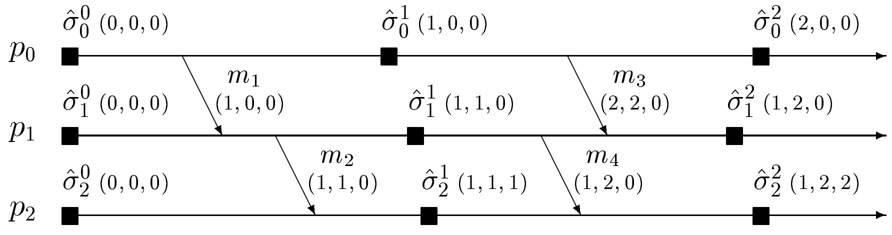

p_0
p_1
p_2
$\hat{\sigma}_0^0$ (0,0,0)
$\hat{\sigma}_0^1$ (1,0,0)
$\hat{\sigma}_0^2$ (2,0,0)
m_1 (1,0,0)
m_3 (2,2,0)
$\hat{\sigma}_1^0$ (0,0,0)
$\hat{\sigma}_1^1$ (1,1,0)
$\hat{\sigma}_1^2$ (1,2,0)
m_2 (1,1,0)
m_4 (1,2,0)
$\hat{\sigma}_2^0$ (0,0,0)
$\hat{\sigma}_2^1$ (1,1,1)
$\hat{\sigma}_2^2$ (1,2,2)

Figure 3: A distributed computation with dependency vectors

Figure 2 shows the R-graph correspondent to the distributed computation depicted in Figure 1. The name R-graph (rollback-dependency graph) comes from the observation that if there is a path in the R-graph from $\hat{\sigma}_a^\alpha$ to $\hat{\sigma}_b^\beta$ and Δ_a^α is rolled back, Δ_b^β must also be rolled back [8].

Dependency Vector A dependency tracking mechanism can be used to capture causal dependencies among checkpoints. Each process p_i maintains and propagates a size-n dependency vector dv_i, that is initially $(0, \ldots, 0)$. The entry $dv_i[i]$ represents the current interval of p_i and it is incremented immediately after a new checkpoint is taken. Every other entry $dv_i[j]$, $j \neq i$, represents the highest interval index of p_j upon which p_i is dependent; it is updated every time a message m with a greater value of $dv_m[j]$ arrives at p_i.

Figure 3 depicts the dependency vectors associated to checkpoints of the distributed computation presented in Figure 1. Note that the dependency vector associated to checkpoint $\hat{\sigma}_2^1$ is (1, 1, 1) and it correctly represents the nodes that can reach this checkpoint in the R-graph (Figure 2). Unfortunately, not all checkpoint dependencies can be tracked on-line. For example, the dependency vector associated to checkpoint $\hat{\sigma}_2^2$ does not capture that $\hat{\sigma}_0^2$ can reach $\hat{\sigma}_2^2$ in the R-graph, since the edge from $\hat{\sigma}_0^2$ to $\hat{\sigma}_1^2$ was established after m_4 was sent.

Rollback-Dependency Trackability Wang established a property in a checkpoint pattern that allows dependency vectors to carry all information needed to perform reachability analysis in its correspondent R-graph [8].

Definition 3.2 Rollback-Dependency Trackability—*A checkpoint pattern satisfies rollback-dependency trackability if the following property holds:*

For any two checkpoints $\hat{\sigma}_a^\alpha (\alpha \neq 0)$ and $\hat{\sigma}_b^\beta$ of the pattern, there is a path from $\hat{\sigma}_a^\alpha$ to $\hat{\sigma}_b^\beta$ in the R-graph if, and only if, $dv(\hat{\sigma}_b^\beta)[a] \geq \alpha$.

4 Fixed-Dependency-After-Send

One way to enforce RDT is to consider Fixed-Dependency-After-Send (FDAS): in any interval, after the first message has been sent, the dependency vector remains unchanged until the next checkpoint [8]. Thus, upon the reception of a message, a forced checkpoint is induced if any entry of the dependency vector is about to be changed. The descriptions of FDAS presented in the literature always compare all entries of the dependency vector to induce a forced checkpoint [1, 3, 7, 8].

An implementation of FDAS is described in class `FDAS` (Class 4.1), using Java[1] [4]. Each process p_i maintains and propagates a dependency vector (Section 3). Process p_i also maintains a flag *afterSend* that captures whether a message has been sent or not during the current interval. The flag is reset after a checkpoint is taken and it is set after a message is sent.

The method `receiveMessage` contains the part of the FDAS that enforces RDT. Upon the reception of a message m, the dependency vector of the message is scanned. If a new dependency is established, say at $dv[k]$, and at least one message was sent in the current interval a forced checkpoint is taken. The dependency vector of the process is updated from $dv[k]$ to $dv[n]$, to register the new dependencies. The complexity of this method is $O(n)$.

```java
public class FDAS {
    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 [ ] DV = new int [N];  // Automatically initialized to (0,...,0)
    protected boolean afterSend;

    public void takeCheckpoint() {
        // Save state to stable memory
        DV[pid]++;
        afterSend = false;
    }

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

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

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

    public void sendMessage(Message m) {
        m.DV = (int [ ]) DV.clone();  // Piggyback DV onto the message
        afterSend = true;
        // Send message
    }

    public void receiveMessage(Message m) {
        int k;
        for (k = 0; k < N && m.DV[k] ≤ this.DV[k]; k++)
            ;  // Stop at the first new dependency
        if (k < N) {  // New dependency
            if (afterSend)
                takeCheckpoint();
            for (; k < N; k++)  // Update DV starting at the first new dependency
               if (m.DV[k] > DV[k]) DV[k] = m.DV[k];
        }
        // Message is processed by the application
    }
}
```

Class 4.1: FDAS.java

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

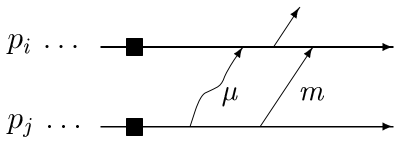
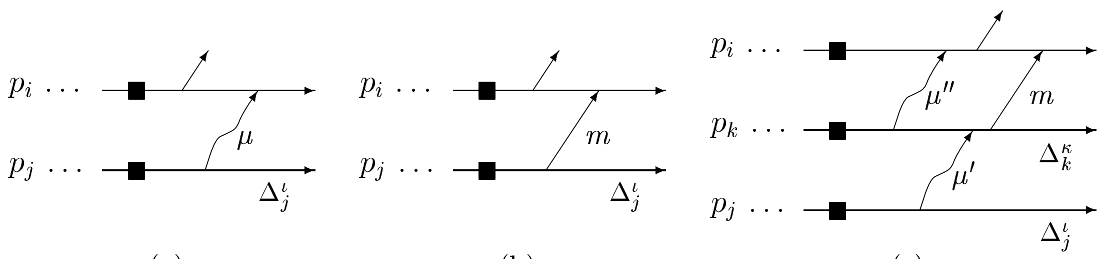


Figure 4: Process p_i already knows the interval in which m was sent



(a) (b) (c)

Figure 5: Contradiction hypothesis (a), base (b), and induction step (c) of Theorem 5.1

5 An Optimization based on Common Knowledge

The approach presented in the previous Section does not take advantage of the processes' common knowledge about their behavior. Consider a scenario where a process p_i receives a message m from p_j which has been sent during an interval already known by p_i due to the causal sequence of messages μ (Figure 4). Since p_j is not allowed to increase its dependency vector after a message-send event, process p_i can verify if a new dependency is established based solely on $dv_m[j]$. This observation takes us to an optimized version of the method `receiveMessage`.

```
public class FDAS {
    /* ... */ // Same as in Class 4.1
    public void receiveMessage(Message m) {
        if (m.DV[m.sender] > DV[m.sender]) { // New dependency
            if (afterSend) takeCheckpoint();
            for (int k=0; k < N; k++) // Update DV
                if (m.DV[k] > DV[k]) DV[k] = m.DV[k];
        }
        // Message is processed by the application
    }
}
```

Class 5.1: FDAS.java (Optimized version of receiveMessage)

Theorem 5.1 *The optimized version of* `receiveMessage` *correctly implements FDAS.*

Proof: Consider the sequence of messages $\mu = (m_1, \dots, m_\ell)$ from Δ_j^ι to p_i, such that each message m_k, $1 \leq k < \ell$, is prime (it is the first message that carries information about Δ_j^ι to the process that receives it). Consider that the last message of μ is received by p_i after a message-send event (Figure 5 (a)). We prove that p_i cannot have changed $dv_i[j]$ regardless the number ℓ of messages in μ.

Base: $\ell = 1$ In this case, μ is formed by a single message m (Figure 5 (b)). If $dv_m[j] > dv_i[j]$, process p_i would have taken a forced checkpoint before processing m.

Step: $\ell > 1$ Consider that no process is allowed to increase an entry of its dependency vector due to a sequence of $\ell - 1$ messages. We prove that this behavior also holds for a sequence of ℓ messages. Let m be the last message of μ, sent by process p_k during Δ_k^κ and let μ' be the first $\ell - 1$ messages of μ from Δ_j^ι to p_k (Figure 5 (c)). Since p_i does not take a checkpoint upon the reception of m, there must exist a sequence of messages μ'' from Δ_k^κ that arrives at p_i before m. Since m is the first message that brings information about Δ_j^ι to p_i, the last message of μ' must arrive at p_k after it has sent the first message of μ''. Thus, p_k increases the jth entry of its dependency vector during Δ_k^κ after a message-send event. $\square$

The observation of the knowledge shared by the processes has allowed us to get a reduction from $O(n)$ to $O(1)$ on the complexity of the decision to take a forced checkpoint. Besides that, the total complexity of the method `receiveMessage` is reduced to $O(1)$ when no new dependency is established. However, the sender of the message must be identified.

6 Conclusion

Fixed-Dependency-After-Send (FDAS) is an RDT protocol that forces the dependency vector of a process to remain unchanged during a checkpoint interval after the first message-send event [8]. In this paper, we have explored processes' common knowledge about their behavior to derive a simpler condition to induce checkpoints under FDAS. We have obtained a reduction from $O(n)$ to $O(1)$, where n is the number of processes in the computation, on the complexity to check if a new dependency is about to be established.

Our improvement can be directed applied to Fixed-Dependency-Interval, a previous version of FDAS that forces the dependency vector of a process to remain unchanged during a checkpoint interval [5, 8]. At the moment, we are investigating whether a similar improvement can be applied to the RDT protocol proposed by Baldoni, Helary, Mostefaoui, and Raynal [1]. A more general result would indicate that this approach to detect new dependencies can be used in all RDT checkpointing protocols.

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

[3] 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, Carnegie Mellon University, 1996.

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

[5] 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.

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

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

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