الجمهورية الجزائرية الديمقراطية الشعبية
People's Democratic Republic of Algeria
وزارة التعليم العالي والبحث العلمي
Ministry of Higher Education and Scientific Research
جامعة عمار ثليجي الأغواط
AMAR TELIDJI UNIVERSITY

كلية العلوم
FACULTY OF SCIENCES
قسم الرياضيات و الاعلام الالي
Department of Mathematics and Computer Science

Study of Failures in a Distributed Environment: New Coordinated and Communication-induced Checkpointing Protocols

---

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Thesis defended on January 14, 2017

ZOHRA ABDELHAFIDI

JURY

Y. OUINTEN	MCA	UATL	President
M.B YAGOUBI	Pr	UATL	Supervisor
M. BENMOHAMMED	Pr	U.Constantine	Examiner
A. BILAMI	Pr	U.Batna	Examiner
H. CHERROUN	Pr	UATL	Examiner
N. LAGRAA	Pr	UATL	Invited
M. DJOUDI	MAA	UATL	Invited

AMAR TELIDJI UNIVERSITY
FACULTY OF SCIENCES
Department of Mathematics and Computer Science

Doctor of Sciences

by ZOHRA ABDELHAFIDI

Abstract

Distributed systems, nowadays, have gained a wide popularity due to their features like reliability, scalability and the diversity of uses in many application areas. However, such systems are susceptible to failures, ensuring their dependability is a big challenge. For this purpose, fault tolerance by checkpointing is used to cope with this problem, which enables the systems, in the event of a failure, to resume its execution from a previously saved state.

Three main approaches have been proposed in the literature, namely, uncoordinated, coordinated, and communication-induced checkpointing (CIC). In this thesis, we are interested in the two latter approach (coordinated and CIC) that share the common property of domino-effect freedom and differ in the way of synchronizing and taking checkpoints.

In this thesis, we propose, as a first contribution, an extension of Index based CIC protocols by a rollback recovery capability. The chosen protocols are based on either classical or lazy indexing strategies. The proposed extension allows us to provide a complete checkpoint/ restart solution. Additionally, it helps us to study the impact of indexing strategies, skipping techniques, failure rate and timestamping functions on the advancement or retrogression of the recovery line in case of failures. Besides, such a study permit us to calculate the overhead incurred and the bounds of the rollback propagation that, in turns, facilitates garbage collection invocation during normal execution rather than during the recovery.

The second contribution contains two new RDT (Rollback Dependency Trackability) protocols called CSFDAS (Constant Size FDAS) and CSRDTParner (Constant Size RDT-Partner), which aim at reducing the control information overhead. The proposed protocols piggyback a constant size of control information on messages while maintaining

timestamp vector at each process. A simulation study shows that the new protocols achieve a good performance compared to the RDT protocols proposed in the literature.

In the last contribution, we propose a Fast Non-Blocking coordinated checkpointing protocol for distributed systems with the aim of minimizing the number of requests and mutable checkpoints while reducing the checkpointing latency. Our protocol relies on two mechanisms: piggybacking dependency information on computation and reply message, and popular processes. We also present a simulation study that compares our protocol to the CSNB protocol (Cao and Singhal Non-Blocking) and CSB protocol (Cao and Singhal Blocking).

Keywords: Distributed Systems, Fault Tolerance, Coordinated Checkpointing, Transitive Dependency, Popular Process, RDT, NZC, Communication induced checkpointing (CIC)

جامعة عمار ثليجي الأغواط
كلية العلوم
قسم الرياضيات و الاعلام الالي

دكتوراء العلوم

ملخص

الأنظمة الموزعة، في الوقت الحاضر، اكتسبت شعبية واسعة بسبب خصائصها مثل الموثوقية والتدرجية وتنوع الاستخدامات في العديد من مجالات التطبيق. ومع ذلك، فإن هذه الأنظمة عرضة للفشل، وضمان الاعتمادية بها يشكل تحديا كبيرا. لهذا الغرض، يستخدم التسامح مع الخطأ للتعامل مع هذه المشكلة. لتحقيق ذلك، هناك طريقة مشتركة، ألا وهي أساليب وضع نقاط المراقبة، التي تمكن الأنظمة، في حال الخطأ، من إستئناف التنفيذ من الحالة الجيدة التي تمّ حفظها مسبقا.

ثلاث مناهج رئيسية اقترحت في المؤلفات الغير منسق، والمنسق الناجم عن التواصل ،في هذه الأطروحة، نهتم بالمنهجين الأخيرين (المنسق و الناجم عن التواصل)، هذان المنهجان يشتركان في خاصية عدم التعرض لتأثير الدومينو، ولكنهما يختلفان في طريقة التزامن واتخاذ نقاط المراقبة

في هذه الأطروحة وكإسهام أول، نقترح تمديدا للبروتوكولات الناجمة عن التواصل بالقدرة على التراجع. تستند البروتوكولات المختارة إما على استراتيجيات الفهرسة الكلاسيكية اواستراتيجيات الفهرسة الكسولة. التمديد المقترح يسمح بتوفير حل كامل للمراقبة / إعادة التشغيل. بالإضافة إلى ذلك، يساعدنا على دراسة أثر استراتيجيات الفهرسة والتخطي، ومعدل الفشل ودوال الفهرسة على تقديم أوتراجع خط الإسترجاع في حالة الفشل. الى جانب ذلك، سمحت هذه الدراسة لنا بحساب التكاليف المتكبدة وحدود انتشار التراجع الذي بدوره يسهل جمع المخلفات أثناء التنفيذ العادي بدلا من خلال الإسترجاع.

الإسهام الثاني يجمع بروتوكولين جديدين ضامنين لخاصية اتباع الارتباطات التراجعية يهدفان إلى الحد من تكلفة معلومات التحكم. البروتوكولات المقترحة الجديدة تضيف حجما ثابتا من معلومات

التحكم على الرسائل الغير مرتبط بعدد الإجراءات مع الحفاظ على أشعة الطابع الزمني في كل إجراء. تظهرهذه الأطروحة أيضا عن طريق دراسة المحاكاة أنّ أداء البروتوكولات الجديدة جيد مقارنة البروتوكولات المقترحة مسبقا في المؤلفات

تقترح أيضا كإسهام أخير بروتوكولا منسقا سريعا غير معلق. هذا البروتوكول يهدف إلى التقليل من عدد الطلبات ونقاط المراقبة المتغيّرة مع التقليل من زمن اتخاذ نقاط المراقبة. يعتمد هذا البروتوكول الجديد على آليتين، أول آلية هي حمل معلومات التبعية على رسائل التطبيق و الجواب، والإجراء الشعبي. نقدم أيضا دراسة محاكاة التي تقارن بين بروتوكولنا الجديد وبروتوكولي سياو وسينغال المعلق و الغير معلق

الكلمات المفتاحية

الأنظمة الموزعة، التسامح مع الخطأ، نقاط المرافبة المنسقة التبعية المتعدية، الإجراءالشعبي، اتباع الارتبطات التراجعبة، نفاط المراقبة الناجمة عن التواصل

Doctorat en Sciences

par ZOHRA ABDELHAFIDI

Résumé

De nos jours, les systèmes distribués ont acquis une grande popularité en raison de leurs caractéristiques telles que la fiabilité, la sclabilité et la diversité des usages dans nombreux domaines d'application. Cependant, ces systèmes sont sujets à des défaillances et assurer leur fiabilité est un grand défi. A cet effet, la tolérance aux fautes permet de faire face à ce problème en utilisant les techniques de points de reprise.

Ces techniques permettent d'assurer la tolérance aux fautes en offrant aux systèmes la possibilité de reprendre l'exécution depuis un état précédemment enregistré en cas de défaillance.

Trois approches principales de points de reprise ont été proposes dans la littérature, à savoir, non-coordonnée, coordonnée, et celle induite par les communications (CIC: communication-induced checkpointing). Nous nous sommes intéressés aux deux derniéres approches (coodonnée et CIC) qui partagent la propriété commune de l'absence de l'effet domino et diffèrent dans la manière de synchronisation et la prise des points de reprise.

Dans cette thèse, comme première contribution, nous proposons une extension des protocoles CIC basés sur les index. Les protocoles choisis sont fondés sur des stratégies d'indexation classiques ou paresseuses. Cette extension va nous permettre de fournir une solution compléte de sauvegarde/reprise. En outre, elle nous aide à étudier l'impact des stratégies d'indexation, de la technique du saut (skipping technique), du taux de défaillance et des fonctions d'estampillage sur l'avancement ou la régression de la ligne de recouvrement en cas de défaillance. D'ailleurs, une telle étude nous a permis de calculer les surcoûts induits et les limites de la propagation de retour en arrière qui, à son tour, facilite l'appel des algorithmes de garbage collection lors de l'excution normale plutôt que lors de la reprise.

La seconde contribution regroupe deux nouveaux protocoles proposés RDT (Rollback DepednencyTrackability) appelés CSFDAS (Constant size FDAS) et CSRDTParner (Constant Size RDTPartner), qui visent à réduire le surcoût induit par les informations de contrôle. Ces protocoles superposent une taille constante des informations de contrôle sur les messages. Via la simulation, les deux protocoles proposés montrent une bonne performance par rapport aux protocoles RDT proposés dans la littérature.

Dans la dernière contribution, nous présentons un protocole de points de reprise coordonné, non-bloquant et rapide pour les systèmes distribués dans le but de minimiser le nombre de requêtes et des points de reprise mutables tout en réduisant le temps de latence. Notre protocole repose sur deux mécanismes: la superposition des informations de dépendance sur les messages d'application et les réponses, et les processus populaires; Nous dressons également une étude de simulation qui compare notre protocole aux protocoles CSNB (Cao et Singhal non bloquant) et CSB (Cao et Singhal bloquant).

Mots-clés: Systémes Distribués, Tolerance aux fautes, Points de reprise coordonnés, Dépendence transitive, Processus populaire, RDT, NZC, Points de reprise induits par les communications (CIC).

Acknowledgements

First and foremost, I would like to thank Allah for the guidance and for all the blessings that I have received during all these years. The truth is that without Allah's support, this couldn't have been possible.

I would like to express my sincere gratitude and appreciation to my supervisor Pr. M.B.Yagoubi for their continuous support to my Ph.D. study, for their patience, motivation, enthusiasm, and immense knowledge, and for their valuable guidance in all the time of this research as well as my thesis preparation.

Besides my supervisors, I am grateful to the rest of my thesis committee: Dr. Y.Ouinten, Pr. M.Benmohammed, Pr A.Bilami, Pr. H.Cherroun for their insightful comments and suggestions.

Also, I am indebted to my family especially my mother, my sisters and brothers for their constant understanding as well as encouragements throughout my life.

My special thanks also go to Pr N.Lagraa and M.Djoudi for all their suggestions and their valuable guidance.

I would like to thank Dr. B.Bentria, A.Belabbaci, T.Bendouma, L.Bensaad, T.Allaoui for their help.

I must also acknowledge all the people in LIM Laboratory and Mathematics and Computer science especially B.Kerrouche.

I would like to thank my best friend F.Terbah for her continuous support and encouragements.

Last but not the least, I would like to thank all my friends in who has directly or indirectly helped me, by any means, to make this thesis possible.

Contents

List of Figures

List of Tables

Introduction

Context

Distributed Systems, nowadays, witness great development and growth and pervade more and more our daily life expressing our increasing need for services provided by such systems. Distributed systems practically penetrate all domains of computing science. It consists of spatially distributed components that perform work of interest and share information by the mean of the communication. Prominent examples of distributed systems include aircraft systems, health-care monitoring systems, smart power grid systems and intelligent transportation systems (ITS). Such systems provide critical services, which means that failures are not acceptable. Thus, ensuring the dependability is an essential task not only for critical systems but also for our society as a whole.

Dependability defined by Avizienis et al. [1] as the ability of a system to deliver service that can justifiably be trusted. Four means are used to attain the dependability: Fault removal, fault prevention, fault forecasting and fault tolerance. Fault prevention aims to prevent the occurrences or introduction of faults. Fault removal aims to reduce the number or the severity of faults which are present in the system. Fault forecasting aims to estimate how many faults are present, possible future occurrences of faults, and the impact of the faults on the system. Fault tolerance is defined as the ability of the systems to continue operating despite failures [1].

Problem statement

Fault tolerance can be provided through checkpointing techniques. Checkpointing is the operation of saving regularly the entire system state (global checkpoint) on the stable storage, so that when a process fails, the system can resume its execution from the last checkpoint. To ensure that the system successfully rolls back from the most recent state, the global checkpoint must be consistent. A global checkpoint is a set of

local checkpoints one per process; a local checkpoint is the saved state of a process. A global checkpoint is consistent if, for each message whose receive event is recorded; its corresponding send event is also recorded.

Determination of consistent global checkpoint is the object of three proposed classes [2], namely: uncoordinated, communication-induced checkpointing and coordinated protocols. Uncoordinated class is characterized by the autonomy of processes in taking local checkpoints. In an event of failure, processes try to find a consistent global checkpoint. This consistent global checkpoint may not exist leading to the known problem of domino-effect [3]. For that, uncoordinated checkpointing are not used alone, and it is often combined with message logging techniques. Message logging mechanisms rely on the piecewise deterministic (PWD) assumption [4], which states that all non-deterministic events can be replayed, and all the relative information for each event can be logged. Under this assumption, processes can re-execute all logged events and will eventually arrive at a state just before a failure. Based on message logging types, we can distinguish three classes: pessimistic message logging protocols [5] [6][7], optimistic message logging protocols [8][9][10] and causal message logging protocols [11][12].

In CIC class, processes are also allowed to take local (called basic) checkpoints independently similarly to uncoordinated class. Due to the interaction between processes, dependencies are established between checkpoints, which may lead some local checkpoints to be useless, i.e. it cannot be part of a consistent global checkpoint. To solve this problem, processes coordinate only by piggybacking control information on computation messages. Upon receiving such messages, processes evaluate a condition (which we called checkpoint induction condition) to check if there is new information brought by messages and take additional checkpoints (called forced checkpoints) accordingly. Thereby, ensuring that every local checkpoint is useful and belongs at least to a consistent global checkpoint.

Based on the property ensured, CIC protocols can be classified into two classes: i) protocols ensuring the No-Z-Cycle property (NZC) [13–17], and ii) protocols ensuring the Rollback-Dependency Trackability property (RDT) [18–21]. NZC property stipulates that there is no checkpoint involved in a Z-Cycle, in other words, every local checkpoint is useful and belongs at least to a consistent global checkpoint. NZC property can be ensured by using indexing strategies that can be done by either classical or lazy way. However, RDT property introduced by Wang [18] postulates that there are no hidden dependencies between local checkpoints, that is, all dependencies are causal.

CIC class shares with uncoordinated class the following disadvantages: i) Storage overhead because each process maintains several checkpoints on stable storage, ii) A complex

recovery, iii) A complex garbage collection. Besides, CIC class has the problem of large overhead due to piggybacking control information, which is about $O(N)$ or $O(N^2)$ in case of protocols ensuring RDT property. Additionally, CIC protocols, especially those ensuring NZC property suffer from a large amount of useful work to redo during the recovery in case of failure.

In coordinated checkpointing class, processes, in contrast to the previous class, synchronize their activities explicitly in order to ensure the consistency of the global checkpoint. Most of the protocols in this class follow the two-phase scheme. In the first phase, an initiator of checkpointing operation takes its tentative checkpoints and sends requests to other processes asking them to take checkpoints. A process receiving such a request takes its tentative checkpoint and sends a reply back to the initiator. At receiving all the replies, the initiator detects the first phase end and starts the second phase by sending commit messages asking other processes to turn their tentative checkpoints into permanents. If the underlying computation is blocked until the checkpointing process termination, the class is called blocking protocols [22–24]. Otherwise, it is called non-blocking protocols [22, 25, 26]. Besides ensuring the domino-effect freedom property, this class is characterized by a simpler recovery in an event of failure and a simpler garbage collection.

The key feature of coordinated and CIC approaches is ensuring the consistency, which means a recovery from failure without the risk of domino-effect. Meanwhile, coordinated approach faces the major problem of the large overhead whose origins is different from CIC approach.

In coordinated class, blocking time is the main source of overhead especially when the application is blocked during the checkpointing process. For that, many works head to non-blocking alternative solution. However, the latter choice suffers from another problem, which is the large number of synchronization messages used during the checkpointing operation.

Objectives

We focus our efforts on CIC and coordinated class by studying and designing new checkpointing solutions at the aim of lowering as possible the overhead while ensuring the efficiency. Our objectives are summarized as follows:

1. Extending Index based NZC protocols by recovery capability in order to provide a complete checkpoint/restart solution and studying the factors impacting the recovery.
2. Proposing new RDT protocols that piggyback a constant size of control information on computation messages.
3. Proposing a new coordinated protocol that has a low overhead in terms of synchronization messages and checkpoints and reduces the checkpointing latency.

Contributions

In the present thesis, we address the problem of consistent global checkpoint determination. Our efforts concentrate principally on coordinated and CIC classes because of their salient features at the aim of providing efficient solutions in terms of overhead. We target our research primarily on the investigation of the different literature proposals. We broaden the survey of Elnozahy et al. 2002 [2] by new references that includes recent works, we also enriched such a survey by discussions, analysis and comparisons (based on several criteria such as complexity, system model, overhead in terms of piggybacked information, propagation upon a failure ...). Such a survey allows us: i) to identify the weakness points of solutions proposed so far as well as their limitations, and ii) to propose enhancements that we summarized as follows:

1. During the recovery, processes search for a recovery line from which they can restart correctly. Such research raises the problem of rollback propagation, which in turn results in large overhead due the important amount of previous work to redo by processes. Thus, we extend a set of Index based NZC protocols by a recovery capability in order to present a complete checkpoint/ restart solution for failures problem. Such an extension allows us, on one hand, to study the overhead of incurred by CIC protocols; on the other hand, to study the impact of some techniques such as timestamping function, skipping technique, induction condition strength and lazy indexing on the development or curtailment of rollback propagation effect. For this purpose, we choose MS (Manivannan and Singhal) [14], BCS (Briatico, Ciuffoletti ,Simoncini) [27] and BCS-aftersend [28] as CIC protocol based on the classical indexing, and LazyBCS [28], LazyBCS-aftersend [28] and LazyMS as CIC protocols based on lazy indexing. Then, we will show how to extend chosen protocols with BRA (Basic Recovery Algorithms) [14] recovery protocol, and we simulate such protocols in a failure-prone environment (To submit this work for publication in the journal of Acta Informatica).

2. In order to reduce the overhead associated with RDT protocols, we propose two new RDT protocols called CSFDAS (Constant Size FDAS) and CSRDTPartner (Constant Size RDTPartner). In contrast to other RDT protocols [18, 19, 21], the proposed protocols do not rely on transitive dependency vector to ensure the RDT property, it hence uses a different mechanism that allows processes to piggyback only a constant size of control information. We discuss, via a simulation study, the performance evaluation of CSFDAS and CSRDTPartner under various scenarios for complete and ring network topologies (To submit this work for publication in the journal of Acta Informatica).

3. We propose a new coordinated checkpointing protocol called FNB (Fast Non-Blocking) [29, 30] that reduces both the number of requests messages and the number of checkpoints. In this protocol, we use two main techniques: piggybacking dependency information on computation and reply messages, and popular process technique. Piggybacking dependency information on computation messages makes the information about the processes involved in the checkpointing procedure available at hand when a process decides to initiate the checkpointing process. As the initiator has only a partial knowledge about the list of processes involved in the checkpointing procedure, piggybacking dependency information on reply messages has the role of completing such a list. Popular process technique is based on timers and the dependency information collected beforehand to select the most solicited process by communications among the others to play the role of the initiator. It is hence used to aggregate the maximum knowledge about the processes involved in the checkpointing procedure.

Thesis Outlines

The remaining chapters of this thesis are structured as follows:

Chapter 1: Fault tolerance in distributed systems This chapter gives the basic definition of distributed systems, their characteristics, design challenges, fault tolerance and the related techniques. In addition, this chapter presents the abstract model of distributed systems and the basic concepts of checkpointing (consistency, Z-theory and a brief description of properties ensured by the communication patterns).

Chapter 2: Checkpointing protocols:State of the art This chapter aims at providing a global review of checkpointing protocols and their classification based on the way of taking checkpoints and the properties ensured by the communication patterns.

Chapter 3: Extending Index based NZC protocols by Recovery Capability This chapter is devoted to show how Index based CIC (communication Induced Checkpointing) protocols ensuring NZC (No-Z-Cycle) property can be extended by a recovery mechanism (BRA protocol). Then, it describes the main simulation scenarios as well as results analysis under failure-free and failure-prone environments.

Chapter 4: New Rollback Dependency Trackability protocols with constant size messages In this chapter, we present two new RDT protocols called CSFDAS (Constant Size FDAS) and CSRDTParner (Constant Size RDTPartner) and we present a comparison of these protocols to proposed RDT (Rollback Dependency Trackability) protocols in the literature. This chapter gives the basic results and the performance analysis.

Chapter 5: Fast Non-blocking coordinated checkpointing protocol (FNB) This chapter describes in details our proposed coordinated checkpointing protocol called FNB (Fast Non-Blocking) and the basic techniques used to improve its performance. In addition, it presents its correctness proof and the performance comparison carried out via simulation study.

A conclusion draws the main results of this thesis and presents suggestion for further works and future lines of research

Finally, an appendix and list of references complete this document.

Chapter 1

Fault Tolerance in Distributed Systems

Contents

1.1 INTRODUCTION

The main goal of this chapter is to present the main concepts related to distributed systems as well as the fault tolerance. It first presents the basic definition of the distributed systems. Then, it presents examples of distributed systems followed by challenges encountered.

The second part of this chapter is devoted to the fault tolerance by checkpointing, which covers the aspects and the theoretical basis of checkpointing protocols such as the notion of local and consistent global checkpoints. It also presents the concept of Z-path, Z-dependency and states the fundamental consistency theorem. Then, This chapter gives a brief description of No-Z-Cycle and Rollback-Dependency trackability properties. Finally, this chapter is concluded by explaining the domino-effect and garbage collection problems.

1.2 DISTRIBUTED SYSTEMS

1.2.1 Definitions

Various definitions of distributed systems have been given in the literature, from which we choose the following definitions.

–**Tanenbaum's definition** *"A distributed system is a collection of independent computers that appears to its users as a single coherent system" [31].*

–**Coulouris et al definition** *"A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages" [32].*

–**Singhal et al. definition** *"A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved" [33].*

From these definitions, the following key features are inherited :

•**"No common physical clock** This is an important assumption because it introduces the element of *distribution* in the system and gives rise to the inherent asynchrony amongst the processors.

•**No shared memory** This is a key feature that requires message-passing for communication.

- **Geographical separation** The geographically wider apart that the processors are, the more representative is the system of a distributed system. However, it is not necessary for the processors to be on a wide-area network
- **Autonomy and heterogeneity** The processors are *loosely coupled* in that they have different frequency speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly" [33].

1.2.2 Examples of Distributed Systems

This section aims at providing examples of contemporary distributed systems illustrating their complexity, their roles and the diversity of the associated applications.

- *Internet* is the best example of a distributed system that encompasses a diversity of computers, servers,... etc. Many applications and services are built on the top of Internet like E-commerce, E-health, social media, gaming. Such applications give us an idea about what distributed systems can provide.
- *Environmental management system* is a special purpose system. It can be used for natural environment control, ecological monitoring, biological and chemical attack detection. In such systems, sensing units are equipped with processors able to collect real-time data and send it to servers for processing and analysis.
- *Intelligent Transport System* is a modern and a complex form of distributed systems, where modern vehicles can be used as processing and storage units. Its goal is to provide drivers and passengers with a comfortable and safety environment.
- *Grid computing* can be seen as a large distributed system. Its goal is gathering geographically distributed resources (computational power, Data, storage) into large powerful systems. In general, it can be used for scientific applications.
- *Cloud computing* is known as utility based computing. Clouds are new form of distributed systems, which are dynamically scalable and provide virtualized resources, computation services and storage services like those offered by Google, Amazon, Microsoft Asur ...

1.2.3 Challenges

Some challenges, for distributed systems designer, need to be overcome in order to get a good distributed system. As following, the description of these challenges.

Heterogeneity

Heterogeneity is defined as the diversity in resources. Distributed systems encompasses different operating systems (Linux, Windows, iOS ...), networks (local network, mobile networks, satellite links ...), hardware (PCs, supercomputers, laptops, Phones, PDAs) and programming language (Java, C#, C++...). To remediate to the heterogeneity problem, some solutions can be implemented: make agreed standards for programming, protocols, integrating middleware layer.

Transparency

Transparency is defined as hiding from users and application programs the fact that distributed system components are heterogeneous and separated. In other words, transparency is the ability to present the distributed system as a single computer system. Table 1.1 summaries transparency forms.

Table 1.1: Forms of transparency [32]

Transparency	Description
Access	enables local and remote resources to be accessed using identical operations.
Location	enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).
Concurrency	enables several processes to operate concurrently using shared resources without interference between them
Replication	enables multiple instances of resources to be used to increase reliability and performance without knowledge of the replicas by users or application programmers.
Mobility	allows the movement of resources and clients within a system without affecting the operation of users or programs.
Performance	allows the system to be reconfigured to improve performance as loads vary.
Scaling	allows the system and applications to expand in scale without change to the system structure or the application algorithms.

Openness

Openness is the ability of the distributed systems to be extensible, it is also the ability of sharing resources or make it available to use by client programs.

Scalability

A system is said to be scalable if the increase in the number of users or resources does not result performance degradation of the system. However, it is difficult to achieve scalability without making changes in the system, especially, when centralized approaches are used (centralized services, centralized data and centralized algorithms).

Security

Security is important in distributed systems. Since the information are made available, it is noteworthy to set up security mechanisms to ensure confidentiality, integrity and availability of information. Ensuring the security is still a problem that has gain the attention of researcher to develop new solutions.

Fault handling

Distributed systems may be the subject of hardware or software failures. As a result, services may be stopped or produce incorrect results. Fault handling is important as failures can affect the whole system. It can be achieved by implementing fault tolerance techniques.

1.3 Fault tolerance

Before introducing the definition of fault tolerance, we start with what faults, errors, and failures are, and the relation between these concepts.

1.3.1 Fault, error and failure

In everyday language, the terms fault, failure, and error are used interchangeably. In fault-tolerant computing parlance, however, they have distinctive meanings.

- **Failure**. A failure is defined as the inability to provide the specified services.
- **Error**. An error is an abnormal state of the system that may lead to a failure.
- **Fault**. The cause of an error is called a **fault**, which may be a defect, an event or a misbehavior of an actor

For example, a crashed program is clearly a failure, which may have happened because the program entered a branch of code containing a programming bug (i.e., a programming error). The cause of that bug is typically a programmer. In other words, the programmer is the fault of the error (programming bug), in turn leading to a failure (a crashed program).

As another example, disconnecting the network card may be an action constituting a fault. This fault results in the error: the file server is no longer accessible. If a client tries to access to the file server via the network, it will be impossible. The file server is no longer conforms to its specification, so it is a failure.

1.3.2 Failure modes

A system does not always fail in the same way. The ways in which a system can fail are its failure modes [1].

- **Fail-stop failure** The system (or a component) continue delivering its service until it crachs (stops) definitively.
- **Omission failure** The system temporally stops delivering its services, and then resumes its execution.
- **Byzantine failure** The system delivers unexpected services or does not behave as assumed.

1.3.3 Definition of Fault Tolerance

Fault tolerance is the ability of a system to continue performing or delivering the expected service properly despite one or more failures. In other words, the system can tolerate faults and continue to operate normally.

Fault tolerance can be achieved using several techniques, which are summarized in the following section.

1.3.4 Fault tolerance techniques

Fault tolerance techniques can be divided into four categories: migration, fault masking, redundancy and recovery(Figure 1.1).

1.3.4.1 Migration

Migration is a preventive action taken to avoid application failure, it consists of moving a part of application from a node that is likely to fail to a safe node. It can be done in two levels: process level [34] or virtual machine [1] level [35]. This technique must use accurate methods to predict the location, the time and the nature of the failure that are likely to occur.

1.3.4.2 Redundancy

There are two types of redundancy (software or hardware)

- **Physical redundancy** relies on replicating critical hardware component [36]. In event of failure, a failed component is replaced by a good one.
- **Software redundancy** can be achieved by one of the techniques: pairs technique or Triple Modular Redundancy technique (TMR)[37]. In pairs technique, each active process (primary) has a backup process (passive), when a primary process fails, it is replaced by a backup process. In TMR technique, three identical copies of a module are created. When one of the three modules fails, the two others correct the fault.

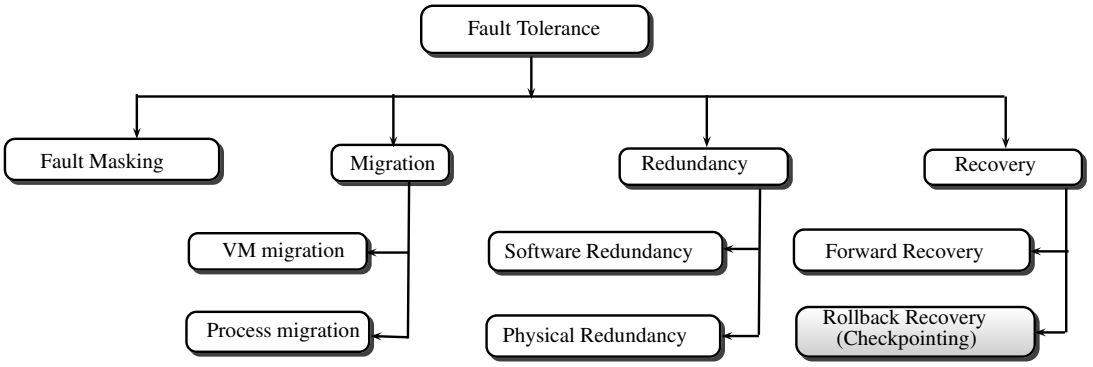


Figure 1.1: Fault Tolerance techniques

1.3.4.3 Fault Masking

Fault masking technique is based on the group of redundant and physically independent processes which may be hierarchical or flat. In flat group [38], processes use voting

[1] "A virtual machine (VM) is an emulation of a particular computer system. Virtual machines operate based on the computer architecture and functions of a real or hypothetical computer, and their implementations may involve specialized hardware, software, or a combination of both." from Wikipedia www.wikepedia.com

technique to decide which process replace the failed one. While, in hierarchical group [39], a coordinator choose which of group members can replace the failed one.

1.3.4.4 Recovery

Recovery is defined as the substitution of the erroneous state by an error-free state, which may be done by one of the following techniques

- **Forward Recovery** By using this technique, the system is brought to a newly reconstructed state (which may be partial) without repeating previous action or computations.
- **Rollback Recovery** (called also fault tolerance by checkpointing). It is based on Checkpointing techniques where a system state is periodically saved in the stable storage. In the event of a failure, the system is restarted from a failure-free state previously saved.

1.4 Fault tolerance by Checkpointing

As we are interested in checkpointing, the following section describes the abstraction of distributed execution with checkpoints.

1.4.1 Distributed Execution: a partial order of events

A distributed execution is composed of a finite set of N sequential processes $\{P_1, P_2 \dots P_N$ that communicate by exchanging messages. Communication channels are assumed reliable and asynchronous. Transmission delays are finite but unbounded. Each process runs on a different processor. Processors do not share either a common memory or global clock, and no assumption is made about their speeds.

The processes are sequential, that is to say, each process P_i produces a sequence of events $\{e_{i,1}, e_{i,2}, \dots e_{i,x} \dots\}$. $e_{i,x}$ denotes the x_{th} event of process P_i (Figure 1.2). There are three types of events: messages emission (denoted send (m)), messages reception (denoted receive (m)) and internal events correspond to the execution of instructions which do not involve communication.

Let H be the set of all events produced by the distributed execution, the execution is modeled by the partial order $\hat{H} = (H, \xrightarrow{hb})$ or $\xrightarrow{hb}$ is defined by

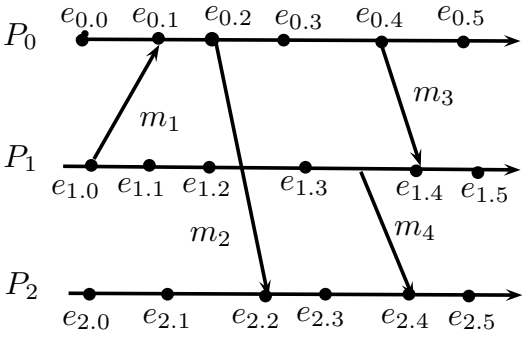


Figure 1.2: Time-space diagram of a distributed execution

$$(a \xrightarrow{\text{hb}} b) = \begin{cases} a \text{ and } b \text{ are events of the same process} \\ \exists \text{ a message } m \mid a = send(m), b = receive(m) \\ \exists \text{an event } c \mid a \xrightarrow{\text{hb}} c \text{ and } c \xrightarrow{\text{hb}} b \end{cases}$$

Each process has an initial state $\sigma_{i.0}$. The state $\sigma_{i.x}$ $x \geq 0$ results from $\sigma_{i.0}$ of the execution of $\{e_{i,1}, e_{i,2}, \ldots e_{i,x} \ldots\}$. By definition, it is said that "$e_{i,x}$ $(x \geq 0)$ belongs to $\sigma_{j,y}$ " $(e_{i,x} \in \sigma_{j.y})$ if i = j and $x \leq y$ or in other words $e_{i,x}$ was produced by P_i before this process reaches the local state $\sigma_{j.y}$.

1.4.2 Distributed Execution: local and global checkpoints

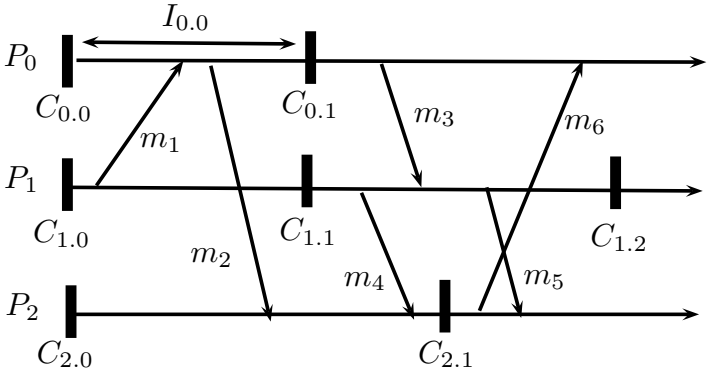


Figure 1.3: Distributed system with local checkpoints

1.4.2.1 *Local checkpoint*

A local checkpoint is the recorded state of process in stable storage. $C_{i,x}$ represents the x^{th} local checkpoint of process P_i (in Fig 1.3, $C_{i,x}$ is indicated by black rectangular box). The sequence of events between $C_{i,x}$ and $C_{i,x+1}$ represents a checkpoint interval (or simply an interval) denoted by $I_{i,x}$.

Note that the set of local checkpoints is a subset local a states as not local states are recorded.

Definition 1.1

A checkpoint and communication pattern of a distributed computation is a pair ($\hat{H}$,C) where $\hat{H}$ is a distributed computation and C is a set of local checkpoints.

1.4.2.2 Causal precedence relation between checkpoints

Before defining the causal precedence relation, we first define the local precedence and the precedence relation due to the communication

Definition 1.2 (*Local precedence relation*)

A checkpoint $C_{i,x}$ locally precedes $C_{j,y}$ (noted by $C_{i,x} \rightarrow_p C_{j,y}$) if i=j and $x < y$

As an example, $C_{1,0} \rightarrow_p C_{1,1}$

Definition 1.3 (*Message order*)

A checkpoint $C_{i,x}$ of process P_i precedes a checkpoint $C_{j,y}$ of process P_j (denoted by $C_{i,x} \rightarrow_M C_{j,y}$) if there exists a message m *such that :*
$(send(m) \in I_{i,x}) \wedge (receive(m) \in I_{j,y-1})$.

Finally, the causal precedence relation is the transitive closure of the union of $\rightarrow_p$ and $\rightarrow_M$

Definition 1.4

A checkpoint $C_{i,x}$ causally precedes $C_{j,y}$ (noted by $C_{i,x} \rightarrow_C C_{j,y}$) if $C_{i,x} \rightarrow_p C_{j,y} \vee C_{i,x} \rightarrow_M C_{j,y} \vee \exists$ a checkpoint $C' \mid C_{i,x} \rightarrow_C C' \wedge C' \rightarrow_C C_{j,y}$

1.4.2.3 Orphan and In-transit messages

Orphan message

A message is said to be orphan with respect to the pair ($C_{i,x}$,$C_{j,y}$) if its receive event is recorded in $C_{j,y}$ but the corresponding send event is not recorded in $C_{i,x}$. This type of messages creates the ordering relation between checkpoints previously noted by "$\rightarrow_M$".

For example, in Figure 1.3 m_2 is orphan with the respect to the pair ($C_{0,0}C_{2,1}$) because $C_{0,0}$ precedes send (m_2) and receive(m_2) precedes $C_{2,1}$).

In-transit message

An message is said to be *in-transit* with the respect to the pair $(C_{i,x}, C_{j,y})$ if its send event is recorded in $C_{i,x}$ but the corresponding receive event is not recorded in $C_{j,y}$

As an example, the message m_2 is *in-transit* message with the respect to the pair $(C_{0,1}, C_{2,0})$ because m_2' send event precedes $C_{0,1}$ while $C_{2,0}$ precedes m_2' receive event.

1.4.2.4 Consistent Global Checkpoint

Definition 1.5

A global checkpoint G is a set of checkpoints, one per a process $\{C_{1,x_1}, \ldots, C_{n,x_n}\}$*.G is consistent if and only if for every pair of checkpoints* $\{C_{i,x_i}, C_{j,x_j}\} \in G$*, the following relation is satisfied.*

$$(\neg(C_{i,x_i} \rightarrow_C C_{j,x_j}) \wedge (\neg(C_{j,x_j} \rightarrow_C C_{i,x_i})).$$

This relation means that neither C_{i,x_i} causally precedes C_{j,x_j} nor C_{j,x_j} causally precedes C_{i,x_i}. Intuitively, from the definition 1.5, a global checkpoint is consistent if all checkpoints are mutually unordered.

In Figure 1.3, the global checkpoint$\{C_{0,1}, C_{1,1}, C_{2,0}\}$ is a consistent global checkpoint. However, the global checkpoint $\{C_{0,1}, C_{1,1}, C_{2,1}\}$ is inconsistent due to the orphan message m_4 ($C_{2,1}$ reflects the reception of the message m_4 while $C_{1,1}$ does not reflect the corresponding send event of m_4).

1.4.3 Z-path, Z-Dependency and Z-Cycle

Netzer and Xu [40] introduced a notion called ZigZag dependency "$\xrightarrow{Z}$" (Z-Dependency), which is stronger than the causality relation. Before stating the Z-dependency relation, we state the notion of Z-path.

1.4.3.1 Z-Path

Definition 1.6

A Z-path from $C_{i,x}$ *to* $C_{j,y}$ *is said to exist if there is a sequence of messages* $\{m_1, m_2, \ldots, m_q\}$ $(q \geq 1)$ *such that:*

1. m_1 *is sent by* P_i *after taking checkpoint* $C_{i,x}$, *and*
2. *for each* m_i, $1 \leq i < q$: $receive(m_i) \in I_{k,s} \wedge send(m_{i+1}) \in I_{k,t} \wedge s \leq t$, *and*
3. m_q *is received by* P_j *before taking its checkpoint* $C_{j,y}$.

- A Z-path is causal (noted by C-path) if the receiving event of each message (except the last one) precedes the sending event of the next message in the sequence. i.e. $\forall i\ 1 \leq i < q, receive(m_i) \xrightarrow{\text{hb}} send(m_{i+1})$.
- A Z-path is non-causal if there is a message m_i whose receive event precedes the send event of the next message . i.e. $\exists i\ 1 \leq i < q\ send(m_{i+1}) \xrightarrow{\text{hb}} receive(m_i)$.
- A Z-path with only one message is causal.

As an example, in Figure 1.3 shows an example of causal path $\{m_1, m_2\}$. In contrary, the Z-path $\{m_3, m_4\}$is a non-causal.

Note μ is a Z-path $\{m_1, m_2, \ldots, m_q\}$ $(q \geq 1)$ The first message m_1 in the path is called $\mu.first$ and last message m_q is called $\mu.last$

Definition 1.7 (*Z-pattern*)

In a Z-path $\{m_1, m_2, \ldots, m_q\}$ $(q \geq 1)$, *two consecutive messages* m_p *and* m_{p+1} *form a Z-pattern if*

$$send(m_p) \xrightarrow{hb} receive(m_{p+1})$$

In Figure 1.3, $\{m_3, m_4\}$ is a Z-pattern.

1.4.3.2 *Z-dependency*

Definition 1.8

A local checkpoint $C_{j;y}$) *Z-depends on a local checkpoint* $C_{i;x}$ *(denoted* $C_{i;x} \xrightarrow{Z} C_{j;y}$) *if:*

1. $j = i$ *and* $y > x$, *or*
2. *There is a zigzag path from* $C_{i;x}$ *to* $C_{j;y}$, *or*
3. $\exists$ *a checkpoint* $C' \mid C_{i;x} \xrightarrow{Z} C') \wedge C' \xrightarrow{Z} C_{j;y})$

1.4.3.3 *Z-Cycle*

The existence of non-causal Z-paths allows via a sequence of messages to establish a relationship between a checkpoint $C_{i,x}$ to itself (Z-cycle). As an example, in Figure 1.3,

the Z-path$\{m_6, m_3, m_4\}$forms a Z-Cycle from $C_{2.1}$ to its self. A Z-Cycle is formally defined as follows:

Definition 1.9

A Z-Cycle exists from a local checkpoint $C_{i,x}$ to its self if $C_{i,x} \xrightarrow{Z} C_{i,x}$

1.4.3.4 Consistency based on Z-path

What is the difficulty ?

Let us consider again Figure 1.3

- Consider the set S_1={ $C_{0.0}, C_{1.1}$}.Is it possible to extend this set to form a consistent global checkpoint? From Figure 1.3, the answer is "yes". By adding $C_{2.0}$, the set S forms a consistent global checkpoint.
- Consider the set S_2={ $C_{0.1}, C_{2.1}$}.Is it possible to extend this set to form a consistent global checkpoint? In this case, the answer is "no", we cannot add any checkpoint from P_1 to complete this set even that $C_{0.1}, C_{2.1}$ are not causally related.
- Consider finally the set S_3={ $C_{2.1}$}. This set can never be extended to form a consistent global checkpoint because $C_{2.1}$ is neither consistent with $C_{0.i}$ $\forall i$ nor with $C_{1.j}$ $\forall j$.

Previous questions are regrouped in one fundamental question which is:

Q(S): S is a set that contains at least one checkpoint, and at most a checkpoint per process, what is the necessary and sufficient condition for this set to be extended to form a consistent global checkpoint ?

Theorem 1.1(Due to Netzer and Xu) provides the answer to the question Q(S) previously laid, Then, a set S can be extended to form a consistent global checkpoint if there is no Z-dependencies between each pair of local checkpoints in S.

Note that, we cannot provide an answer for the previous question based on the causality relation

Theorem 1.1

Let S be a set of local checkpoints, S can be completed to form a consistent global checkpoint if and only if $\forall a, b \in S : \neg(a \xrightarrow{Z} b)$ [40].

Proof:

If part: First of all, we show how the set *S*can be extended to form a global checkpoint G, then, we prove the consistency of G.

The global checkpoint G contains the set S plus one checkpoint $C_{i,x}$per process P_i in $\overline{S}$ as follows:

1. P_i has no checkpoint in S and P_i has a checkpoint $C_{i,y}$ such that $\exists\ C_s \in S \mid C_{i,y} \xrightarrow{Z} C_s$. Then, $C_{i,x}$ must be the first checkpoint that does not Z-depend on any checkpoint in S.
2. P_i has no checkpoint in S and P_i has no checkpoint Z-depending any checkpoint in S. As no checkpoint in *S*can precedes the initial checkpoint of any process. Then, P_i' initial checkpoint is added to S.

Recall that the checkpoint G is consistent means that all checkpoints in G are mutually unordered. To prove the theorem, we assume that $\exists\ a\ and\ b \in G \mid a < b$ and we show that such an assumption leads to a contradiction. Let $\overline{S} = G - S$. There are four cases:

Case a: $a, b \in S$, a precedes b means that there is a Z-path from a to b which contradicts the assumption that $a \xrightarrow{Z} b$.

Case b: $a \in \overline{S}$, $b \in S$ *a*precedes *b*implies that $a \xrightarrow{Z} b$, which contradicts the way of constructing $\overline{S}$

Case c: $a \in S$, $b \in \overline{S}$. As a precedes b means that b cannot be the initial checkpoint. Hence, b is the first checkpoint that does not Z-depend any checkpoint in S and $\exists\ a\ checkpoint\ c \mid c \xrightarrow{Z} d \in S$. Then, the causal path from a to b plus the Z-path from c to d form a Z-path from a to d which contradicts that there no Z-path between *S*'members.

Case d: $a, b \in \overline{S}$. According to the argument in the previous case, it exists a Z-path from a to a checkpoint d in S. But this contradicts the fact that a is the first checkpoints that does not Z-depend any member in S

Only if part: we show that if $a \xrightarrow{Z} b$, then *a*,*b* cannot be part of a consistent global G, Starting by assuming that there exists a Z-path from *a*to *b*, we show that the global checkpoint G containing both a and b by using induction in the number k of Z-path messages.

- Base case ($k = 1$). As a Z-path constituting of only one message is a causal path, then, a precedes b, which implies that G is inconsistent.
- Induction case ($k > 1$): Let us assume that if it exists a Z-path of at most k from a to **b**, a,$b$cannot be part of the same consistent global checkpoint G
- To show that the Z-path of $k+1$ joining a and b implies that a,b belong to the same consistent global checkpoint G, we use the proof by contradiction.

  As the Z-path of $k+1$ is the concatenation of a Z-path of k messages (from a to a checkpoint c) and a message m from (c' to b), on one hand, neither (a,c) nor (a, c") can be part of the same global checkpoint G(induction case) such that c precedes c" (c and c" are checkpoints of the same process). On the other hand, neither (c', b) nor (c"', b) (where c"' precedes b) can be part of the same global checkpoint (base case). Therefore, no checkpoint can be combined with a and b to form a consistent global checkpoint.

■

An important special case when S is reduced to a single checkpoint S = a. In this case, the local checkpoint a is part of a consistent global checkpoint if and only if the checkpoint 'a' does not Z-depend on itself : $\neg(a \xrightarrow{Z} a)$. For example, the checkpoint $C_{2.1}$ cannot be part of any consistent global checkpoint because it involves in a Z-cycle (Figure 1.3), and thus it is said a *useless* checkpoint.

Corollary 1.1

A checkpoint a belongs to at least a consistent global checkpoint if and only if a does not belong to a Z-cycle [40].

1.4.4 Properties ensured by communication patterns

After the introduction of the concept of Z-path and Z-cycle, two fundamental properties of the distributed execution with checkpoints are derived.

1.4.4.1 No-Z-Cycle (NZC) Property

Given a checkpoint $C_{i,x}$ in the distributed execution with checkpoints ($\hat{H}$, C), the checkpoint will be part of at least one consistent global checkpoint if and only if it does not belong to any Z-cycle. If this property of non-membership to a Z-cycle is ensured

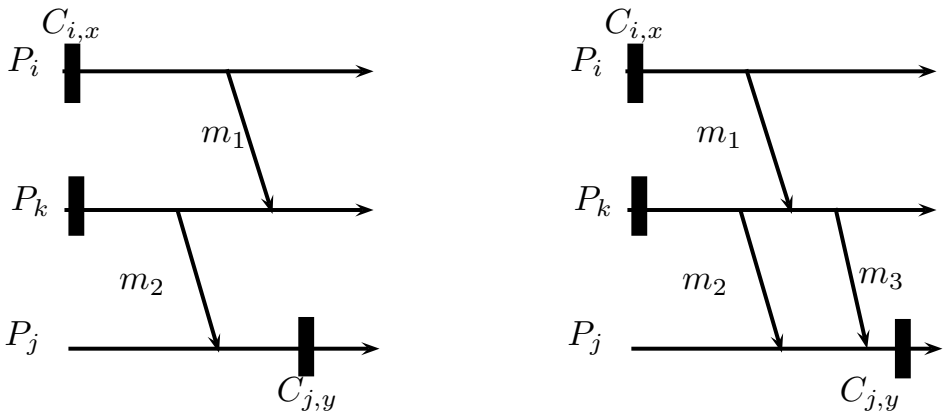
by each checkpoint in ($\hat{H}$, C). Then, the distributed execution with checkpoints ($\hat{H}$, C) satisfies the No-Z-Cycle property.

Property 1.1. *A distributed execution with checkpoints ($\hat{H}$, C) satisfies the property No-Z-cycle (NZC) if and only if there is no Z-cycle in ($\hat{H}$, C).*

NZC is a highly desirable property in the rollback context. Ensuring NZC property implies rollback without the risk of the domino effect.

1.4.4.2 Rollback Dependency Trackability (RDT) Property

Application based checkpoints sometimes introduce other problems. In a particular context, the rollback introduces problems such as recovery line identification, garbage collection etc... Identify a recovery line involves the determination of a newest consistent global checkpoint (the execution is restarted from this recovery line). All checkpoints before the recovery line should be removed (garbage collection).



(a) Non-causal Z-path (b) Causally doubled Z-path

FIGURE 1.4: Non-causal Z-path and Causally doubled Z-path

Thus, effective solutions to the above problems can be found in a simple manner if the processes are able to determine the minimum and maximum global checkpoint that contains a given local checkpoint.

Wang has shown that if the dependencies between checkpoints due to Z-paths are detectable online [18], then the maximum and minimum global checkpoints are easily determined.

Due to non-causal Z-paths, hidden dependencies can be established between checkpoints which is not desirable in a distributed execution (it cannot be tracked online). In Figure 1.4(a), there is a non-causal path $\{m_1, m_2\}$ between $C_{i,x}$ and $C_{j,y}$ which can not

be detected. If this dependence is established by another causal Z-path $\{m_1, m_3\}$, this dependence can be captured. Thus the original Z-path$\{m_1, m_2\}$ is said causally doubled by $\{m_1, m_3\}$(Figure 1.4(b)).

Definition 1.10

A Z-path from $I_{i,x}$ to $I_{j,y}$ is causally doubled if

- $i = j \wedge x \leq y$ *or*
- *there exists a causal path m from $I_{i,x'}$ to $I_{j,y'}$ where $x \leq x'$ and $y' \leq y$*

Hence, the RDT property can be defined as follows

Property 1.2. *A checkpoint and communication pattern satisfies Rollback-Dependency Trackability if and only if all non-causal Z-paths are causally doubled [41].*

1.4.4.3 Relation between RDT and NZC properties

As a Z-cycle is a special case of a non-causal Z-path from $C_{i,x}$ to itself, a Z-cycle can never be causally doubled by a causal Z-path. This observation directly implies the following result: if any Z-path is causally doubled, so no Z-cycle can be formed and we obtain RDT⇒ NZC

In other words, a distributed execution with checkpoints ($\hat{H}$, C) that satisfies the RDT property satisfies also the NZC property.

1.4.5 Checkpoints and Domino Effect

Upon a fault occurrence, the system must be restarted from a consistent state, this state can be built from the checkpoints recorded by the processes

The domino effect problem arises when looking for a consistent global checkpoint, at which no orphan messages are created. This problem is characterized by the fact that the system, in an event of failure, will restart its execution from the initial state (start of execution) despite there are checkpoints.

Figure 1.5 shows an execution in which each process takes checkpoints (black rectangle) without coordination, each process starts running with an initial checkpoint. Suppose that at P_3 level, a failure strikes the system. Then, P_3 rollbacks to point C. However, because it sent out a message m_6 after C was taken, P_2 has to roll back to before it received this message. Consequently, P_2 must roll back to B. But this will trigger a rollback

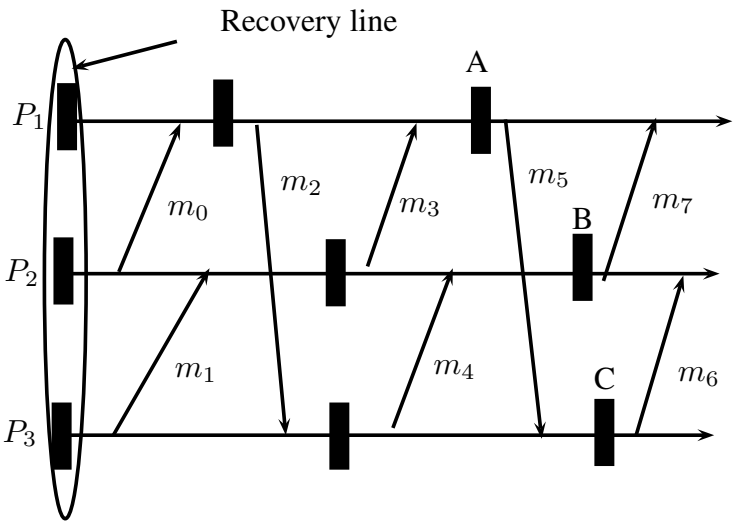


FIGURE 1.5: Domino effect problem

of P_1 to A because P_2 sent a message, m_7 to P_1 and P_1 has to move back to a state in which it never received this message. In turn, this entails P_1 to rollback to a checkpoint before receiving m_5. This operation is continuously repeated until that all processes have rollback to the initial checkpoints. This rollback cascading is an example of domino effect problem.

The previous example shows how a single fault can cause the domino effect problem, and how processes waste their effort. However, to avoid this problem, processes must coordinate their work to build a consistent system state.

1.4.6 Checkpoints and Garbage Collection

The operation of removes all outdated checkpoints that will be no longer needed (useless) in case of recovery is called ”garbage collection”. As storage space is limit, such an operation is important because it frees places for future taken checkpoints

The common approach is the identification of the recovery line, namely, a consistent global checkpoint, and removing all information related to the events occurred before this line. For example, processes that coordinate their checkpoints to form consistent states can always restart from the most recent checkpoint in each process. So, the old checkpoint is deleted whenever a new checkpoint is created. However, this operation is difficult if processes take their checkpoints independently because, we do not know which checkpoint is useless. As a result, garbage collection algorithm is usually invoked after the recovery.

1.5 Conclusion

Through this chapter, we have given the basic definitions of distributed systems and challenges such as heterogeneity, transparency, scalability, fault handling, etc. We have also shown the importance of fault tolerance to ensure systems dependability and the related techniques.

As we are interested in fault tolerance by checkpointing, we have introduced, at the second part of this chapter, the concept of local and global checkpoints. A global checkpoint is the set of local checkpoint, it is said to be consistent if each pair of local checkpoints is unordered, which have led us to discuss and redefine the consistency based on Z-theory (Z-path, Z-dependency, and Z-cycle) introduced by Netzer and Xu [40].

Then, we have briefly described two important properties: No-Z-Cycle and Rollback-Dependency trackability. No-Z-Cycle ensures that any local checkpoint, which has been taken by a process, belongs to a consistent global checkpoint. The rollback-dependency trackability ensures the absence of hidden dependencies. Finally, we have illustrated the domino-effect and garbage collection problems.

Chapter 2

Checkpointing protocols: State of the Art

Contents

2.1 Introduction

Checkpointing is the act of saving system state (called global checkpoint) on a stable storage. Such a state is used to recover the system from failures. To ensure that the system recovers successfully, the global checkpoint must be consistent. this is the principal focus of thee main checkpointing classes: coordinated, communication-induced and uncoordinated.

Over this chapter, we will present the related works of checkpointing protocols according to the classification of Elnoozahy et al. [2] enriched by new references, discussions and comparisons. First, we will start by presenting coordinated checkpointing class and its subclasses including blocking, nonblocking, time-based and coordinated protocols with concurrent initiators. Second, we will present communication-induced checkpointing and its subclasses NZC and RDT protocols. Third, we describe uncoordinated class followed by message logging techniques and its subcategories (pessimistic, optimistic and causal). During this chapter, we will discuss and highlight some differences between protocols (principals and design). Besides, we will give some hints on studies related to checkpoint interval models. Finally, we will terminate this chapter by presenting the techniques used to reduce the stable storage contention such as incremental and diskless checkpointing.

2.2 Checkpointing Protocols

Checkpointing protocols are divided into three main classes: coordinated, uncoordinated and communication-induced [2]. These classes are eventually combined with message logging techniques, especially uncoordinated one resulting the fourth class, namely, checkpointing with message logging.

The following sections describes these main classes and their subclasses as depicted in Figure 2.1.

2.2.1 Coordinated Protocols

Coordinated checkpointing is an attractive approach to achieve fault tolerance. It has many characteristics that make it different from other classes, such that: i) at most one checkpoint per process kept in stable storage, ii) very simple recovery, iii) domino-effect freedom, iv) An automatic garbage collection.

2.2.1.1 Blocking protocols

During the checkpointing process, the application may be blocked until the processes take their checkpoints, hence its name of blocking protocols.

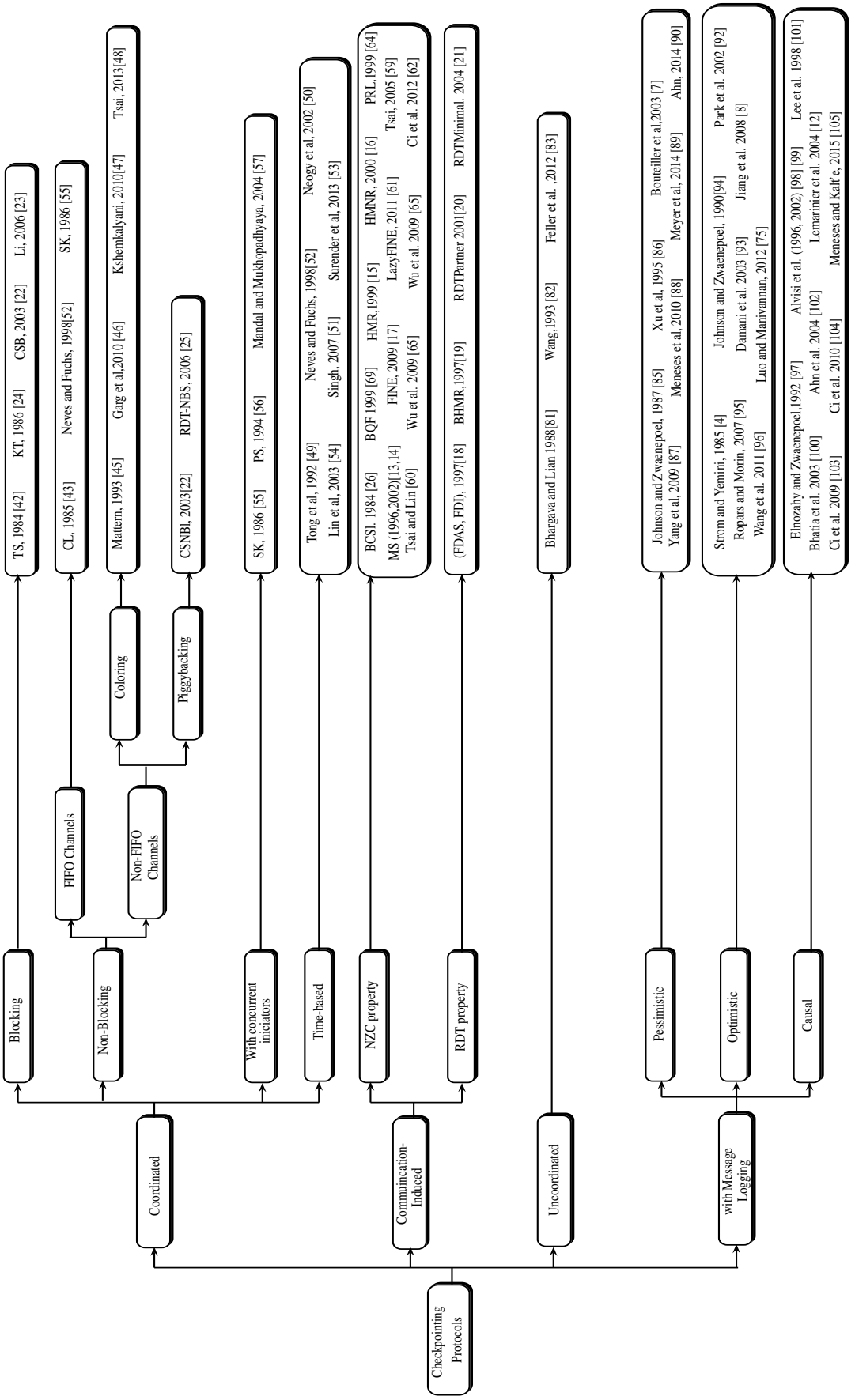


Figure 2.1: Checkpointing protocols classification

Tamir and Sequin's protocol

As an example of blocking protocol, we state that of **Tamir and Sequin** [42], which is a two-phase protocol (called TS protocol)

Phase 1

- An initiator takes its checkpoint and sends requests to other processes.
- Upon receiving such a request, a process stops its execution, takes its checkpoint and sends a reply message back to the initiator.

Phase 2

- On receiving all relevant replies, the initiator starts the second phase and sends commit messages to the involved processes asking them to turn their tentative checkpoints into permanent.
- After receiving the commit message, a process makes its tentative checkpoint permanent and removes the old one.

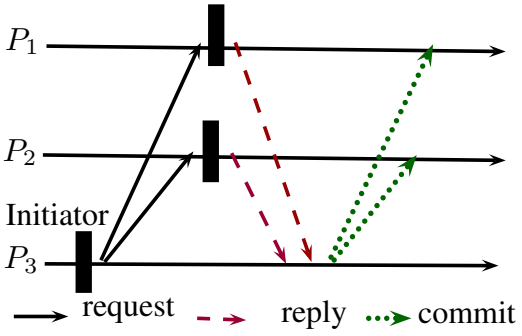


FIGURE 2.2: Coordinated checkpointing procedure

Figure 2.2 depicts an example of TS protocol. When P_3 initiates the checkpointing procedure, it takes a tentative checkpoint and sends requests to P_1 and P_2. Upon receiving these requests, P_1 and P_2, in turn, take tentative checkpoints and send reply messages back to P_3. P_3 hence detects the first phase end and sends commit messages asking P_1 and P_2 to make their tentative checkpoints permanent.

Koo and Toueg's protocol

Tamir and Sequin's strategy has also been used by Koo and Toueg [24]. In Koo and Toueg's protocol (called KT protocol):

1. An imitator takes a tentative checkpoint and sends requests to only a subset of processes called cohort, *i.e.* a set of processes, such that $Q \in cohort_P$ means that: Q has sent a message to P since P's last checkpoint.

2. At receiving such a request, each process inherits the first request and acts as the initiator. If a process has already received a request, it just sends a reply to the process from which it has received a request.
3. If the initiator has received all replies from its cohort, it terminates the first phase by sending commit messages.

Cao and Singhal's protocol

KT and KS protocols suffer from blocking overhead. Hence, **Cao and Singhal** [22] proposed a non-blocking protocol (called CSB) that tries to solve this problem by exploiting the dependency tracking technique. In that, each process maintains an R vector, i.e. $R_i[j] = 1$ means that a process P_i has received a computation message from a process P_j in its current checkpoint interval. CSB works as follow:

1. When an initiator starts the checkpointing procedure, it first blocks its execution and broadcasts R_request messages to all processes to retrieve dependency information.
2. Then, each process at receiving R_request blocks its execution and sends its R vector.
3. At receiving all relevant vectors, the initiator calculates the set of process on which it depends(via a matrix constructed from the received R vectors).
4. After that, the initiator
   (a) sends requests to this set of processes asking them to take tentative checkpoints, or
   (b) sends release messages to processes out of this set.
5. Upon receiving a request, each process takes a tentative checkpoint and sends a reply back to the initiator
6. If a process receives a release message, it resumes its execution without taking a checkpoints
7. Having received all replies, the initiator starts the second phase by broadcasting commit messages.
8. A process receiving a commit turns its tentative checkpoint into permanent, and thus the checkpointing procedure terminates.

Li and Shu's protocol

Li and Shu [23] proposed an enhancement (called LS) of KT protocol. It also requires a subset of processes to participate in global checkpoint determination by exploiting

dependency information. LS protocol retrieves such dependencies during normal execution. Thereby, making such information available for use when initiating the checkpointing procedure. Hence, this strategy lowers the checkpointing latency.

2.2.1.2 *Non-blocking protocols*

In this class, processes do not need to block their executions, but the problem is how to prevent them from receiving orphan messages which can result in checkpoint inconsistency. To explain the problem, suppose that P_3 initiate checkpointing procedure by sending requests to P_2 and P_1. After taking a checkpoint, P_2 sends a message m to P_1. As shown in Figure 2.3(a), this message may be received by P_1 before taking its checkpoint. As a result send (m) is not recorded in $P_2's$ checkpoint while receive (m) is recorded in $P_1's$ checkpoint.

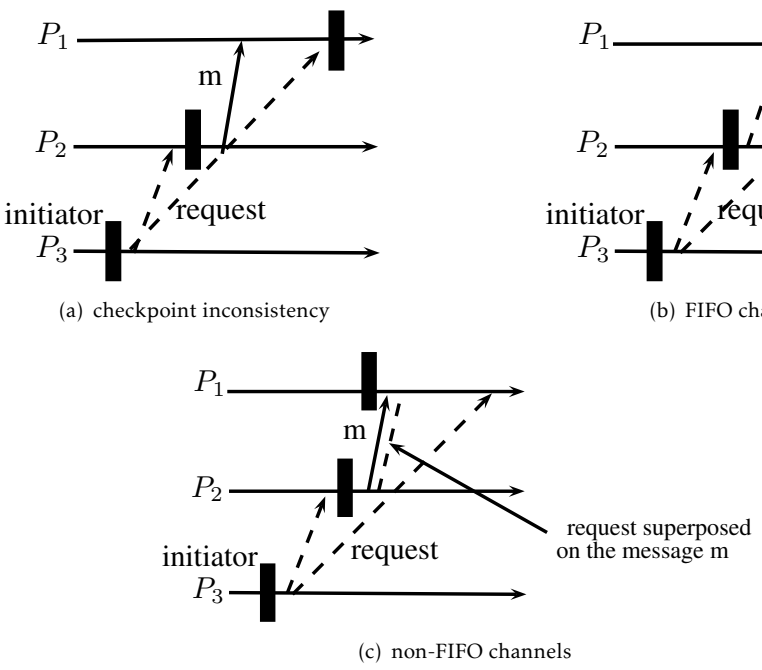


FIGURE 2.3: Non-blocking coordinated checkpointing: (a) checkpoint inconsistency; (b) with FIFO channels; (c) non-FIFO channels (short dashed line represents piggybacked checkpoint request). process

Such a problem can be solved by

1. assuming that channels are FIFO. This assumption ensures that no message sent after $P_i's$ checkpoint will be received before $P_j's$ during checkpointing procedure (Figure 2.3(b)). Or,

2. piggybacking post-request on computation messages. For example, in Figure 2.3(c). P_2 has sent message m currying a post-request to P_1. At receiving such a message, P_1 takes a checkpoint before processing it.

a) Case of FIFO channels

Chandy and Lamport's protocol

Chandy and Lamport [43] proposed a protocol for FIFO channels case called CL protocol based on markers. CL protocol works as follows:

1. First, an initiator takes its checkpoints and broadcasts markers to all processes on outgoing channels.
2. Then, every process, on receiving a marker, takes a checkpoint and propagates it before sending any computation message.
3. If a process P_i receives a message m from a process P_j after taking its checkpoint and has not yet received a marker from P_j, it records the message m (m, in this case, is a transit message).
4. Checkpointing procedure terminates when every process has received a marker on all its channels.

In CL protocol, the marker plays three key roles:

i To inform processes about checkpointing execution.
ii To distinguish between messages sent before the checkpointing procedure (prerecorded messages) from those sent after the checkpointing procedure (postrecorded messages).
iii To mark the end of in-transit messages.

b) Case of non-FIFO channels

In systems with non-FIFO channels, two main techniques are used:

i coloring technique where colors play the role of post-request
ii piggybacking technique where sequence numbers (timestamps) play the role of post-request.

b.1) Checkpointing protocols using coloring technique

b.1.1) General approach

The general approach proposed by [44] is as follows:

Phase 1 In this phase, an initiator takes its checkpoint, change its color to red performs a broadcast of INIT message (INIT messages play the role of requests) to all processes informing them about the checkpointing execution.

(a) A process is initially white and immediately becomes red after it takes a local checkpoint.

(b) A white process (or red) process sends white (or red) colored messages.

(c) Upon receiving a red message or INIT message, a white process takes a local checkpoint.

Phase 2 This phase is dedicated to accumulating in-transit messages numbers that must be recorded and informing all the processes by such a number

Phase 3 In this phase, in-transit messages are recorded as a part of channels state. Moreover, if the second phase terminates successfully and all messages are recorded, processes detect the end of checkpointing procedure and turn their colors white.

b.1.2) Related protocols

All protocols using coloring technique have practically the same first and third phase, but they differ in the *second phase.* Hereafter a brief description of some protocols on how to accumulate in-transit message number.

- **Mattern's protocols**
  Mattern [45] proposed checkpointing protocols for non-FIFO channels, namely, deficiency counter and vector counter protocols.
  1. **Deficiency counting protocol**works as follows: each process P_i maintains a counter ctr_i being part of process checkpoint, which counts the number of messages sent minus the number of messages received, By collecting and accumulating all the counters, the initiator knows how many in-transit messages that must be received. Each red process forwards every white message received to the initiator until that the number of received in-transit messages equals $\sum_i ctr_i$ at the initiator side
  2. **Vector counter protocol**. In this protocol, each process P_i maintains an integer vector ($V_i[j]$ tracks the number of white messages it sent to P_j. When P_i

receives a message, it decrements $V_i[i]$.

A control message with a vector C circulates along a ring to accumulate V vectors. In the first round, a white process visited by the control message is turned red. In the second round, the control message waits until that each process receives all in-transit messages. During this round all relevant in-transit message are collected.

- **Garg et *al.*'s protocols**

  Garg et *al.* [46] proposed two protocols based on Mattern's works.

  1. The first algorithm is **grid-based** and works like vector counter protocol except that the accumulation of in-transit messages number is done along the grid diagonal. For that, it uses $N^{3/2}$ messages to inform non-diagonal processes.
  2. The second algorithm is **tree-based** and works as deficiency counter. It uses a pre-established spanning tree to broadcast checkpoint requests. The accumulation is done from leaves to root in convergecast tree. When the root accumulates in-transit messages number; it distributes it as tokens among processes to inform them about the in-transit messages that should be received. Then, a process consumes a token at every reception until there are no in-transit message to receive.

- **Kshemkalyani' protocols**

  Kshemkalyani [47] also proposed two protocols, namely, tree and hypercube

  1. The first algorithm is *tree-based* that works as Garg et al.'s tree-based algorithm except that it does not use tokens. After taking checkpoint, a convergecast tree accumulates the vector $white_sent$ at the root. The i^{th} component contains the number of messages sent to the process P_i. Then, the root distributed the accumulated among system processes to inform them of the number of in-transit messages. Each process P_i, hence, waits until that it receives all in-transit message as its components indicates before terminating.
  2. The second algorithm is a variant of vector counter technique. It is based on hypercube overlay to accumulate the local vectors, which requires $O(NlogN)$ messages while keeping processes symmetrical, *i.e.* processes execute the same accumulation steps.

- A recent work proposed by **Tsai**[48] uses an r-dimensional grid to accumulates the local vectors in a such way that all processes behave symmetrically.

b.2) Checkpointing protocols using piggybacking technique

b.2.1) General approach

In checkpointing protocols using piggybacking techniques, each process P_i maintains a monotonically increasing number Csn (checkpoint sequence number) which is incremented at each checkpoint and is also used to tag(timestamp) messages. The general approach used by such protocols works as follows:

Phase 1

(a) An initiator starts the checkpointing procedure, it takes a tentative checkpoint and broadcast request accompanying with Csn.

(b) At receiving such a request, every process takes a tentative checkpoint and sends a reply to the initiator.

(c) A process may also receive a computation message before receiving a request. In this case, it is directed to take a tentative checkpoint if such a message indicates that a checkpointing procedure is running, *i.e.* message's Csn is greater than that of the process.

Phase 2

(a) After having received all replies, the initiator detects the end of the first phase and starts the second phase by sending commit messages asking all processes to turn their tentative checkpoint permanent.

(b) Upon receiving a commit message, a process turns its tentative checkpoint into permanent and deletes the old one.

b.2.1) Related protocols

Cao and singhal's protocol

All previous non-blocking protocols require the participation of all processes in the global checkpoint determination. Therefore, **Cao and Singhal** have proposed an algorithm (called CSNB) that direct a subset of processes to take checkpoints. As CSB protocol, CSNB protocol uses the R vectors to record dependencies. In Addition, Cao and Singhal have introduced the concept of mutable checkpoints, which is neither a tentative nor a permanent checkpoint. Such checkpoints are saved in processes local memories to avoid the transferring overhead.

Phase 1 In this phase

–• **Checkpointing initiation** When a process P_i initiates a checkpointing process, it takes a local checkpoint. Then, it sends a checkpoint request to each process P_j such that $R_i[j] = 1$ and resumes its execution.

–• **Reception of a checkpoint request** Upon receiving a request, P_i checks whether it needs to propagate the request. If it is not the case, it just sends a reply message back to the initiator, otherwise, there are three cases:
  (a) P_i has already received a computation message containing information about the current checkpointing process. Consequently, it turns its mutable checkpoint (if it exists) into tentative. Then, it propagates the request message to the processes in which it depends (P_j doesn't) and sends a reply message to the initiator.
  (b) P_i has already taken a tentative checkpoint associated to this initiator. So, it sends a reply message to the initiator (duplicate request).
  (c) P_i doesn't receive any request. In this case, it takes a tentative checkpoint, propagates the request and sends a reply message to the initiator.

–• **Computation messages received during checkpointing**Upon receiving a computation message by P_i from P_j, P_i checks for new information carried by this message. Then, P_i is directed to take a mutable checkpoint if it has already sent a message. Otherwise, it consumes only the received message.

Phase 2 This phase is similar to the second phase of the general approach described above.

Sakata and Garcia's protocol

Sakata and Garcia [25] have proposed a non-blocking protocol called RDT-NBS. The idea behind this protocol is ensuring RDT property during normal execution. To do so, it breaks all non-causal Z-path by taking mutable checkpoints. The main advantage of such a technique is keeping all dependencies apparent. As a result, when an initiator starts checkpointing procedure, it knows all involved process and sends them requests directly following the two-phase approach like CSNB protocol. The main drawback of RDT-NBS protocol is the large overhead introduced due to mutable checkpoints used to ensure RDT property.

2.2.1.3 Time based checkpointing

Time-based protocols are another form of coordinated checkpointing. It alleviates the need for explicit coordination messages during checkpoints creation. In time-based synchronization, processes take checkpoints at previously agreed times. It can be achieved either by using synchronized clocks or timers. Based on synchronized clocks [49, 50], processes take their checkpoints whenever local clocks reach a multiple of checkpoints

interval while processes using timers take their checkpoints at timers expiration [51] [52] [53]. As there is no global time, clocks or timer need to be synchronized periodically.

Unfortunately, Timers and clocks cannot be perfectly synchronized, the consistency becomes a real concern in such protocols. To solve this problem, protocols in [49, 50] disallow messages sending during a period until the checkpoints saving, hence, this scheme is blocking. However, the solution given by Neves and Fuchs [52] is based on piggybacking information on messages while getting processes not blocked. To enhance more this type of protocols, proposed protocols [54][51][53] maintain a minimum number of checkpoints by exploiting dependencies between processes.

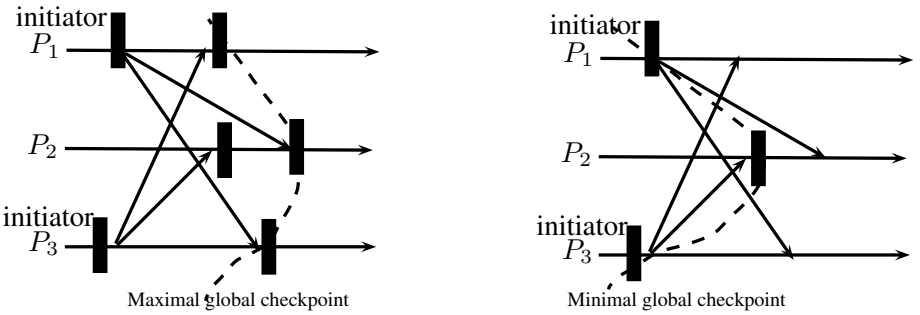
2.2.1.4 Coordinated checkpointing with concurrent initiators

Most the proposed Coordinated checkpointing protocols assume that only one process can initiate checkpointing procedure at a given instant. However, proposed protocols In [55] [56] [57] permit concurrent initiators following one of the two schemes: Intersection approach or Union approach.

As an example of intersection approach, the protocol proposed by Spezialetti and Kearns[55] where, a process takes its local checkpoint in response to only the first checkpoint request in a similar manner as Chandy and Lamport's protocol [43]. In the second phase, initiators communicate with each other to calculate the global checkpoints.Thus, the global checkpoint is minimal.

As an example of intersection approach, consider the system composed by three processes P_1, P_2 and P_3 (Figure 2.4(b)), where P_1 and P_3 are concurrent initiators. At the reception of the first request either from P_1 or P_3, each process takes only one checkpoint. When P_2 receives the second, it ignore it and does not take a checkpoint. Similarly, P_1 and P_3 acts as P_2. Therefore, the global checkpoint is the set of first checkpoints taken by each process.

Mandal and Mukhopadhyaya [57] have proposed an algorithm following Spezialetti and Kearns's approach, where, processes are arranged in a unidirectional ring network. In this protocol, when a process initiates the checkpointing procedure, it takes a checkpoint and sends a request to its successor. Each non-initiator process, at the reception of the first request, takes a checkpoint and forwards such request. Otherwise, the other requests are either forwarded or discarded. For the bidirectional ring, the protocol is quite similar except that the initiator sends requests to its two neighbors.



(a) Union approach (b) Intersection approach

Figure 2.4: Coordinated checkpointing with concurrent initiators). process

Unlike Spezialetti and Kearns protocol, Prakash and Singhal use the union approach [56]. In this protocol, every process takes its checkpoints whenever it receives a request. The generated global checkpoint is maximal, resulting in a large overhead in terms of checkpoints number.

As illustrated in Figure 2.4(a), each process takes checkpoints in response to P_1 and P_3 requests. Hence, each process has two checkpoints, the maximal global checkpoint is the set of second checkpoints.

Sakarta et al. shows that RDT-NBS protocol support concurrent initiators following the Union approach. During the first phase, processes keep several tentative checkpoints, In second phase, only last tentative checkpoints that form the maximal global checkpoint are turned into permanent and the other are discarded.

It is worthy to note that Time-based protocols can be seen as a form of checkpointing with concurrent initiators because each process takes its checkpoint independently whenever the timer expires or the clock reaches a multiple of checkpoint interval.

2.2.1.5 Discussion

A myriad of protocols has been proposed in coordinated class. Table 2.1summarizes these protocols according to which sub-class belong, system model, assumption on message ordering, complexity of the protocol in terms of request number, allowing or no the concurrent initiators, recording in-transit messages as channel state and exploiting or the dependency information.

Protocols categorized as blocking have the best complexity in terms of request message, but on the other hand, it suffers from the blocking time overhead even there are

Table 2.1: Related works of coordinated checkpointing protocols

Reference	Sub-class	System Model	Message ordering	Number of Requests	Concurrent Initiators	Channel state	Using Dependency
Tamir and Sequin, 1984 [42]	Blocking	Point-to-point	FIFO	$O(N)$	Yes	Yes	No
Koo and Tueg, 1986 [24]	Blocking	Point-to-point	FIFO	$O(N^2)$	Yes	No	No
Cao and Singhal, 2003 [22]	Blocking	Point-to-point	None	$O(N)$	No	No	Yes
Li and Shu, 2006 [23]	Blocking	Point-to-point	None	$O(N)$	Yes	No	Yes
Chandy and Lamport, 1985 [43]	Non-blocking	Point-to-point	FIFO	$O(N^2)$	Yes	Yes	No
Mattern, 1993 [45]	Non-blocking	Point-to-point	None	$O(N(W+1))$	No	Yes	No
Garg et *al*,2010 [46]	Non-blocking	Tree	None	$O(NLog(N)Log(W/N))$ [a]	No	Yes	No
Garg et *al*,2010 [46]	Non-blocking	Grid	None	$O(N^{3/2})$ [a]	No	Yes	No
Kshemkalyani, 2010[47]	Non-blocking	Tree	None	$O(N)$ [a]	No	Yes	No
Kshemkalyani, 2010[47]	Non-blocking	Hypercube	None	$O(NLog(N))$ [a]	No	Yes	No
Tsai, 2013[48]	Non-blocking	Tree	None	$O(NLog(N))$ [a]	No	Yes	No
Cao and Singhal, 2003[22]	Non-blocking	Point-to-point	None	$O(N^2)$	No	No	Yes
Sakata and Garcia, 2006 [25]	Non-blocking	Point-to-point	None	$O(N)$	Yes	No	Yes

[a] The complexity is calculated by the protocol's authors

Table 2.1 Related works of coordinated checkpointing protocols (Continued)

Reference	Sub-class	System Model	Message ordering	Number of Requests	Concurrent Initiators	Channel state	Using Dependency
Tong et al, 1992 [49]	Time blocking	Point-to-point	None	$O(N)$	Yes	No	No
Neves and Fuchs, 1998[52]	Time Non-Blocking	Point-to-point	FIFO	$O(N)$	Yes	No	No
Neogy et *al*, 2002 [50]	Time Blocking	Point-to-point	None	$O(N)$	Yes	No	No
Lin et *al*, 2003 [54]	Time Non-Blocking	Point-to-point	None	$O(N)$	Yes	No	Yes
Singh, 2007 [51]	Time Non-blocking	Point-to-point	None	$O(N)$	Yes	Yes	Yes
Surender et *al*, 2013 [53]	Time Non-blocking	Point-to-point	None	$O(N)$	Yes	No	Yes
Spezialetti and Kearns, 1986 [55]	Non-blocking	Point-to-point	FIFO	$O(M^2)$	Yes	No	No
Prakash and Singhal, 1994 [56]	Non-Blocking	Point-to-point	FIFO	$O(MN^2)$	Yes	No	No
Mandal and Mukhopadhyaya, 2004 [57]	Non-blocking	Ring	None	$O(N^2))$	Yes	Yes	No

W is the total number of messages in transit.
M is the number of concurrent initiators

some protocols minimize the number of processes participating in the determination of the global checkpoint.

Assuming communication channels FIFO has the advantages of making the application non-blocked while ensuring the consistency. Besides, solutions based on such assumption are characterized by the simplicity. On the other side, FIFO ordering is a major drawback and some cases may lead to a higher complexity.

Likewise, protocols that are based on coloring techniques do not block the application during checkpointing operation. have also a good complexity but they bear the disadvantage of long checkpointing latency due to the management of in-transit messages in-spite of using network overlay such as Trees, Hypercube, Grid, etc.

Concerning time-based coordinated checkpointing protocols, the major drawback is the consistency insurance, which justifies the few works in this area. However, coordinated protocols that allow multiple initiators have the highest complexity in terms of request message either using union or intersection approach. In addition, union approach adds extra overhead due the large number of checkpoints taken on response to initiator demand.

Majority of the studies require the participation of all processes in the determination of the global checkpoint. This way results in extra overhead even though some processes are not involved in the communication since their last checkpoints. As a result, these protocols are not suitable.

Hence, exploiting the dependency between process is important as it is the only way to lower the overhead. In this regard, Cao and Singhal have proposed their CSNB protocol, but as mentioned before this protocol suffers from extra overhead due to the problem of duplicate requests. In the same way, Sakata and Garcia have proposed their protocol RDT-NBS, but it also suffers from extra overhead due to the extra number of mutable checkpoints.

As a conclusion, this discussion leads us to identify the source of the overhead such us blocking time, synchronization messages, and number of checkpoints. The best solution to the checkpointing problem, in our point of view, must in one hand minimize these costs, and on the other hand, ensures the efficiency. Thus, such solutions must be endowed with mechanisms that lower as possible the overhead.

2.2.2 Communication Induced Protocols

Communication-induced checkpointing (CIC) class (or quasi-synchronous [58]) is a compromise between coordinated and uncoordinated approaches. It allows the autonomy of processes in taking their local (called basic) checkpoints while enforcing the consistency by directing processes to take extra (called forced) checkpoints. To synchronize checkpointing activities, processes piggyback control information on computation messages; such information help processes to decide whether it takes forced checkpoints. CIC protocols do not induce checkpointing latency while keeping checkpoints useful but at the expense of piggybacking extra information.

The general approach of CIC protocols is the following

```
Procedure Take basic checkpoint ()
        save local state as checkpoint;
Procedure Send message m ()
        begin
                append control information to m;
                send (m)
        end

Procedure Receive message m ()
        begin
                Check for new control information;
                Evaluate the induction condition C;
                If {C} then
                        Take forced checkpoint ();
                        Update information;
                Process m;
```

It is worthy to note that each CIC protocol is characterized by its induction condition. Such a condition is evaluated based on control information transported by messages and local knowledge available at processes, which helps make decision about taking forced checkpoint.

Elnozahy et al. classify CIC protocols into Model-based and Index based [2]. Model-based class is based on detecting harm communication patterns that could form Z-cycles. Thus, processes take forced checkpoint to prevent such patterns formation. Index based class uses timestamping functions that assign timestamps (logical clock or sequence

numbers) to basic checkpoints as well as to forced checkpoints so that local checkpoints with the same timestamps form a consistent global checkpoint.

CIC protocols can also be classified based on property ensured into RDT(Rollback Dependency Trackability) protocols and NZC (No-Z-Cycle) protocols

2.2.2.1 *Protocols Ensuring the No-Z-Cycle Property*

NZC property stipulates that there is no checkpoint involved in a Z-cycle, in other words, every local checkpoint is useful and belongs at least to a consistent global checkpoint. NZC protocols (known also as Z-Cycle Free ZCF protocols [58]) are either model based or index based.

Briatico et al. [27] was the first to introduce an index based NZC protocol (called BCS) before introducing the Z-theory by Netzer and Xu [40]. Their protocol is the simplest index protocol where, every process tags its checkpoint intervals with its logical clock (lc_i). When a process sends a message, it piggybacks its lc_i(denote by m.lc). At the receiver side, the process evaluates the following condition ($m.lc < lc$) to decide whether it takes a forced checkpoint.

Another simple protocol (called MS) was proposed by Manivannan and Singhal [14]. In MS protocol, every process maintains a local counter. When it takes a local checkpoint, it tags it by its sequence number which is picked from its local counter. This sequence number is piggybacked on every message. At receiving such a message, a process checks if the piggybacked sequence number is greater than its one. If it is the case, it takes a forced checkpoint assigned with the received sequence number. In addition, MS selectively logs in-transit messages in order to avoid future inconsistencies during the recovery, this strategy has also been used by Hélary et al. [15]

Further enhancements of BCS protocol have been introduced in [16, 17, 28]. The simple enhancement is known as BCS-aftersend [28] which has the following inducing condition ($Aftersend = true \wedge m.lc < lc_i$). Since the Z-pattern occurs only when send and receive event occur in the same checkpoint interval, this condition allows to a process to take a forced checkpoint when it has already sent a message and the logical clock accompanying the message is greater than the logical clock of the current checkpoint of the process receiving the message.

Hélary et al. have proposed a protocol called HMNR1 [16] that behaves in the same manner as BCS-aftersend. Other enhancements have been proposed: HMNR2 [15] and

FINE [17]. These protocols induce less forced checkpoint than HMNR1 but at the expense of a large amount of control information piggybacked on computation messages. Later, Tsai has shown that HMNR2 enhancement rarely takes effect [59]. Thus, he has introduced a constant size message protocol called NMMP(No Message Message Pattern) protocol which has a good performance regardless to its weaker condition.

All the above protocols are based on classical indexing strategy which means that logical clock or sequence number is increased by one whenever a basic checkpoint is taken. Vieira et al. [28] have introduced a new indexing strategy called lazy indexing. They show that processes with different frequencies in taking basic checkpoint may result in gaps on sequences number. Thus, it is unnecessary to increment the sequence number if there is no new information carried by messages received in the current interval. Later, this strategy has been used by Tsai et al. [60] and applied to HMNR2 protocol. Luo and Manivannan [61] has also applied the lazy indexing to FINE protocol.

In addition to this set of protocols, Ci et al. [62] have proposed a time based quasi-synchronous protocol. In this protocol, a process takes a basic checkpoint every checkpoint period. When it sends a message, it appends its local time on the message which helps processes to synchronize their clocks and enforce consistency by taking forced checkpoints.

In contrary to index based NZC protocols, few model based NZC protocols are proposed in the literature. The First one has been proposed by Baldoni, Quaglia, and Ciciani (BQC) [63]. It is the first protocol to use a vector clock and to avoid the domino effect without enforcing RDT. BQC tries to detect harm patterns known as Suspect Z-cycle which may result in Z-cycle. For this purpose, it maintains and propagates $O(N^2)$ control information. Garcia et al. [64] have proposed a protocol called PRL that enforces the same properties and lowers the complexity of the required control information from $O(N^2)$ to O(N). Therefore, Wu et al. [65] has shown that BQC and PRL protocols falsely detect harm patterns and hence induce extra checkpoints. Then, they have proposed a new protocol but with $O(N^3)$ complexity.

2.2.2.2 Protocols Ensuring the Rollback Dependency Trackability Property

Rollback Dependency Trackability property has been introduced by Wang [18] which is very stronger than NZC property. It postulates that there are no hidden dependencies between local checkpoints, that is, all dependencies are causal.

Wang has proposed protocols ensuring RDT property based on dependency vector DV i.e. DV is an integer vector used to track transitive dependency relation. $DV_i[i]$ represents the number of checkpoints of process P_i, $DV_i[j]$ ($| \ i \neq j$) is the number of checkpoints taken by P_j known by P_i. The first protocol is FDI (Fixed Dependency Interval). When a process sends a message m, it tags the message m with its dependency vector DV (denoted by m.DV). At receiving such a message, the receiver first checks if there is new information carried by the message ($\exists i \ m.DV[i] > DV[i]$). If it is the case, it takes a forced checkpoint and updates its DV.

Another protocol proposed by Wang called FDAS (Fixed Dependency After Send) that has the following induction condition ($\exists i \ m.DV[i] > DV[i] \wedge Aftersend$). It means that a process takes a forced checkpoint if it has already sent a message and the received message carries new information, thereby the number of forced checkpoints is reduced.

Baldoni et al. [19] have shown via their study that the number of forced checkpoints induced by FDAS protocol can still be reduced. They have introduced a new characterization based on causal doubling called PCM (Prime Causal Message) characterization, and hence they have designed a new protocol called BHMR (Baldoni Helary Mostfaoui Raynal). BHMR protocol reduces the number of forced checkpoints but at the price of piggybacking $O(N^2)$ control information.

Baldoni et al. [66] have also proposed a minimal characterization called EPSCM (Elementary and Simple PCM) and have proposed a protocol quite similar to BHMR [67] (based in such a characterization) from which they have derived a family of protocols called EPSCM family.

Garcia et al. [20] have proposed a refinement of PCM characterization called PMM (Prime Message Message) characterization. Moreover, they have proposed two protocols. the first one is called RDTPartner [20] and the second one is RDTMinimal [21], which is more efficient. Later, Tsai [68] has derived a set of protocols called PMM family likewise to EPSCM family.

Note that all RDT protocols are known as model based RDT protocols.

We will describe, in more details, BCS, BCS-aftersend and MS protocols and their analogs using lazy indexing in Chapter 3. Besides, we will describe FDAS, BHMR and RDTPartner protocols in Chapter 4.

Table 2.2: Related works of CIC checkpointing protocols

Reference	Sub-class	Property ensured	System Model	Overhead	Indexing	Condition strength	Rollback propagation
Briatico et *al.* 1984 [27]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	<C(HMNR1) [a]	K-rollback
Baldoni et *al.* 1999 [69]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	≃C(HMNR1)	K-rollback
Garcia et al. 1999 [64]	Model Based	NZC	Point-to-Point	$O(N)$	Classic	-	K-rollback
Hélary et *al* 1999 [15]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	<C(HMNR1)	1-rollback
Hélary et al. 2000 [16]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	-	K-rollback
Hélary et al. 2000 (HMNR2) [16]	Index Based	NZC	Point-to-Point	$O(N)$	Classic	>C(HMNR1)	K-rollback
Quaglia et *al.* 2000 [63]	Model Based	NZC	Point-to-Point	$O(N^2)$	Classic	-	K-rollback
Baldoni et al. 2001 [41]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	-	K-rollback
Vieira et al. 2001[28]	Index Based	NZC	Point-to-Point	$O(1)$	Lazy	-	K-rollback
Manivannan and Singhal, (1996, 2002) [13] [14]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	<C(HMNR1)	1-rollback [b]
Tsai, 2005 [59]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	<C(HMNR1)	K-rollback
Tsai and Lin [60]	Index Based	NZC	Point-to-Point	$O(N)$	Lazy	>C(LazyHMNR1)	K-rollback
	Index Based	NZC	Tree	$O(N)$	Classic		

[a] <C(HMNR1) means that it has a weaker condition than that of HMNR1 protocol. While >C(HMNR1) means that it a stronger condition than that of HMNR1 protocol.

[b] claimed by protocol's authors

Table 2.2Related works of CIC checkpointing protocols (Continued)

Reference	Sub-class	Property ensured	System Model	Overhead	Indexing	Condition strength	Rollback propagation
Wu et al. 2009 [65]	Model based	NZC	Point-to-Point	$O(N^3)$	Classic	-	K-rollback
Luo and Manivanan, 2009 [17]	Index Based	NZC	Point-to-Point	$O(N)$	Classic	>C(LazyHMNR1)	K-rollback
Luo and Manivanan, 2011 [61]	Index Based	NZC	Point-to-Point	$O(N)$	Lazy	≃C(LazyHMNR1)	K-rollback
Ci et *al*. 2012 [62]	Index Based	NZC	Point-to-Point	$O(1)$	Classic	<C(HMNR1)	1-rollback [b]
Wang, 1997 (FDAS) [18]	Model based	RDT	Point-to-Point	$O(N)$	Classic	-	1-rollback
Wang, 1997 (FDI) [18]	Model based	RDT	Point-to-Point	$O(N)$	Classic	<C(FDAS)	1-rollback
Baldoni et *al*. 1997 [19]	Model based	RDT	Point-to-Point	$O(n^2)$	Classic	>C(FDAS)	1-rollback
Garcia et al2001[20]	Model Based	RDT	Point-to-Point	$O(N)$	Classic	>C(FDAS)	1-rollback
Baldoni et *al*. 2001 [67]	Model based	RDT	Point-to-Point	$O(n^2)$	Classic	>C(FDAS)	1-rollback
Garcia et al. 2004 [21]	Model based	RDT	Point-to-Point	$O(N)$	Classic	>C(FDAS)	1-rollback

2.2.2.3 Discussion

Several studies have been proposed in CIC class. Table 2.2 summarizes such works according to which sub-class belong, which property ensured, system model, complexity in terms of control information piggybacked and which indexing strategy is used.

Tsai et al. [70] has compared Index based CIC protocols based on induction conditions. Given a protocols *Pr* whose condition is noted by C(*Pr*), if C(*Pr*) is stronger than C(HMNR1) (i.e. C(*Pr*) $\Rightarrow$ C(HMNR1) , Pr will induce at most as many forced checkpoints as HMNR1. In other words, *Pr* outperforms HMNR1. HMNR2 and FINE protocols have stronger conditions than HMNR1, while MS protocol has a weaker condition than HMNR1. Many works later have confirmed such a result theoretically and experimentally through the simulation and extended it to compare any two protocols provided that they belong to the same family (Lazy or Classical) [28, 61, 71–74].

In a similar way, Tsai et al. have compared RDT protocols theoretically based on the strength of induction condition in respect to FDAS 's induction condition [75]. From Table 2.2, all RDT protocols have stronger conditions than FDAS protocol, except FDI protocol. Later, Tsai et al.'s results have been confirmed experimentally via [68, 74, 76, 77].

Conversely, if we compare CIC protocols based on complexities, the order of the protocol obtained by applying induction condition strength will be reversed. Having a stronger condition implies having complex data structures to record processes causal past. Hence, we classify CIC protocols based only on complexity criteria Table 2.3.

From Table 2.3, we can see that Model-based CIC protocols (either ensuring RDT property or NZC property) have the highest complexity because such protocols are based on discovering harm communication patterns that may form a Z-Cycles. It tries to learn the causal past execution of processes and track important information in complex data structures (generally matrix or vectors). Then, processes share their causal pasts by piggybacking such data structures, which justifies the highest complexity.

In contrary, index based CIC protocols have the lowest complexity in terms of information piggybacked size. It does not require a complex data structure to record causal past of processes. In some cases, protocols use only scalars that will be appended to messages.

Such works are carried in a failure-free environment, which neglects the behavior of such protocols under failure assumption. Agbaria et al. [78] classified checkpointing protocols based on on the rollback propagation upon a failure into 1- and K-rollback:

coordinated and RDT protocols into 1-rollback class, and NZC protocols into K-rollback. i.e. K-rollback is the maximal number of checkpoints a process may need to rollback during recovery in order to restart from a state consistent with that of the failed process. Some protocols are in NZC class but are classified as 1-rollback such as MS protocol[13], Ci et al. 's protocol [62] and HMR protocol [79].

Agbaria and Friedman have proposed an analytical model [79] based on their classification [78] that calculates the total overhead incurred due to failures occurrence. Such a model, in our point of view, lacks accuracy because it neglects the specific behavior of each protocol.

In our knowledge, no RDT protocol having a constant size control message has been proposed (as can be seen in Table 2.3). On the other hand, few are the works destined to study CIC protocols in failure-prone environment.

TABLE 2.3: Classification of CIC protocols based on the complexity

		Variable size ()	Constant size
NZC	Index-based	[16, 17, 60, 61]	[14–16, 27, 28, 41, 59, 62, 69]
	Model-based	[63–65]	
RDT	Model-based	[18–21]	

2.2.3 Uncoordinated Protocols

In contrary to coordinated and CIC approaches, the uncoordinated approach does not require either implicit or explicit synchronization between processes. Processes have the full autonomy in taking their checkpoints whenever it is convenient. Thus, uncoordinated protocols eliminate synchronization overhead. This advantage comes at the expense of the following disadvantages: i) Processes may take useless checkpoints, which can never be part of a consistent global checkpoint and hence adding an extra overhead to the application. ii) Due to the useless checkpoints created by processes, there is a possibility of domino effect to occur in the event of failure. iii) In order to construct consistent global checkpoints, processes must maintain more than one checkpoint in the stable storage, which may lead to the stable storage contention,

Protocols in this class rely on one of the following technique: Checkpoint Graph [80] or Rollback Dependency Graph [81][82], where dependencies are saved during failure-free execution. Upon a failure, a process initiates recovery procedure by broadcasting

requests. At receiving such a request. a process sends its dependencies to the initiator, Having received all dependency information, the initiator calculates the recovery line. As said before, such a line may not exist leading to the domino-effect problem.

To cope with this problem, uncoordinated protocols are often combined with message logging techniques.

2.2.4 Message Logging Protocols

Message logging protocols is a particular class of event logging. It is based on PWD (Piece-Wise Deterministic) assumption, which postulates that the state of a process is determined by its initial state and by the sequence of messages it delivers, that is, the execution of a process can be seen as a sequence of state intervals bounded by non-deterministic message reception events. Logging protocols are usually combined with checkpointing techniques to enable the system to recover from the most recent consistent global state.

Message logging techniques fall into one of three broad categories: pessimistic, optimistic, and causal.

2.2.4.1 Pessimistic message logging protocols

In pessimistic message logging, processes synchronously log their messages before sending (or delivering) at sender (or receiver) side. It guarantees that no orphan processes will be created during the execution. An orphan processes is a process whose state is inconsistent with the recovered state of the failed process. This type of protocols has the following advantages: 1) Simple recovery: due to the fact that the effect of the failure is limited to the failed process; 2) Simple garbage collection: older checkpoints and logs can be easily removed; 3) Simple output commit; 4) Survive multiple failures. These advantages come at the expense of incurring performance penalty due to the synchronous logging during failure-free operation.

Special techniques are used to facilitate logging and lower the overhead like using special hardware [83]. Other techniques are based on software solutions like SBML (Sender Based Message Logging) approach.

The first SBML protocol is due to Johnson and Zwaenepoel [84]where message determinants are kept in sender's volatile log. Message determinant, in general, consists of the

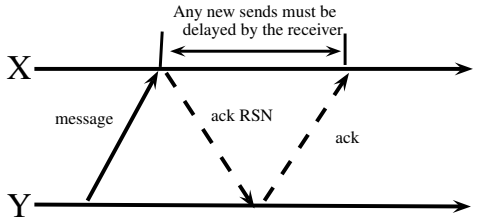


FIGURE 2.5: pessimistic message logging

message payload, sender ID (SID), receiver ID (RID), sender sequence number (SSN) and receiver sequence number (RSN). This determinant is logged according to three handshake approach in three steps. First, when a process sends a message, it logs message payload, SID, RID and SSN. Then, when the receiver receives the message, its sends an acknowledgment containing the RSN. Upon receiving such an acknowledgment (ack RSN), the sender adds the RSN to determinants of the message and sends an acknowledgment to the message receiver (Figure 2.5). SBML protocol has the advantages of low overhead but tolerates only one failure.

TABLE 2.4: Related works of pessimistic logging protocols

Reference	Combined with	System Model	Message ordering	Number of concurrent failures	Event logger	Asynchronous recovery
Johnson and Zwaenepoel, 1987 [84]	Uncoordinated	Point-to-Point	None	1	No	Yes
Xu et *al*, 1995 [85]	Uncoordinated	Point-to-Point	FIFO	1	No	Yes
Bouteiller et *al*,2003 [7]	Uncoordinated	Point-to-Point	None	-	Yes	-
Yang et *al*, 2009 [86]	Coordinated	Group based	FIFO	f	No	No
Meneses et *al*, 2010 [87]	uncoordinated	Group based	None	f	No	No
Meyer et *al*, 2014 [88]	Uncoordinated	point-to-point	None	f	No	No
Ahn, 2014 [89]	Uncoordinated	Group based	FIFO	f	No	Yes

f is the number of faults

Xu et al. [85] use the same SBML scheme of Johnson and Zwaenepoel [84] but they reduce the message logging overhead by exploiting the Z-theory. In this protocol, the sender is informed which intervals have received its messages. Boutillier et al. [7] have proposed an implementation of SBML protocol called MPICH-V2. In addition, they have introduced a special component called event logger to store the dependency information associated with each message.

To achieve scalability, Yang et al. [86] divide the system into clusters where time-based non-blocking coordinated checkpointing is used within the group, whereas pessimistic message logging is used for inter-cluster communication. This combination of techniques has the advantages of tolerating more than one failure and significant reduction of logging overhead. A similar protocol has been proposed by Meneses et al. [87].

In contrary to [86], SBML scheme was combined with RBML(Receiver Based Message Logging) and called HML [88]. HML works as follows: When a process P_1 send a message m to process P_2, it first saves the message in its volatile log, then it sends it. P_2, upon receiving such a message, first logs m in a logging unit situated in a different process called protector (P_2, in this case, is called protected) and then consume this message. Thus, HML protocol has the advantages of avoiding message losses, allowing failed processes to reach the same before-failure state and tolerating more than one failure providing that protectors and protected processes do not fail in the same time.

To lift SBML's limitation, Ahn has also proposed another protocol for clustered system [89] by exploiting the beneficial feature of FIFO broadcast and replicating the log information of each message in the same group members. Therefore, if only one process survives in each group, the system will continue progressing.

Table 2.4 summaries the related works of pessimistic logging protocols.

2.2.4.2 Optimistic message logging protocols

In contrary to pessimistic protocols, optimistic logging protocols are based on optimistic assumption that the failure may occur after a complete logging [4]. In Figure 2.6, process X, in contrary to pessimistic message logging, sends the message m_2 to process Z and do not wait until the message m_1 is completely logged. In optimistic protocols, processes are asynchronous, that is, message determinants are kept in a volatile log, which is spooled later to the stable storage in a non-blocking manner. Therefore, it incurs a little overhead during failure-free operation. However, the cost of these advantages is a complex recovery and more complicated garbage collection than pessimistic protocols.

Optimistic protocols do not avoid orphan processes creation during the execution. Meanwhile, these processes must rollback during the recovery to ensure that the system is restarted from a consistent global state.

Strom and Yemini [4] were the first to introduce an optimistic protocol. Their algorithm is based on dependency vectors to track causality information which will be used

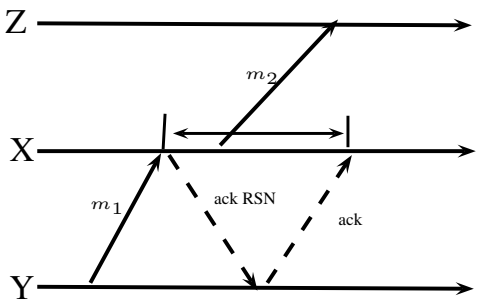


FIGURE 2.6: Optimistic message logging

in the recovery. However, their algorithm suffers from Exponential rollback due to one failure and it is proposed for only systems with FIFO channels.

Sistla and Welch [9] has also used the transitive dependency tracking technique. Their proposed protocol is designed for clustered system and in contrast to Strom and Yemini's protocol, it does not suffer from Exponential rollback.

Peterson and Kearns have proposed another causality technique [90] based on vector clocks. Their algorithm was proposed for FIFO channels case. Park et al. have used the same technique [91] but their algorithm does not require channels to be FIFO.

Damanai et al .[92] have also designed a protocol based on transitive dependency vectors [1]. Their protocol bounds the rollback in an event of failure. However, it does not tolerate more than one failure.

Tolerating multiple failures is the subject of the works done by Johnson and Zwaenepoel [93], and Ropars and Morin [94]. Johnson and Zwaenepoel's protocol has the lowest message overhead; however, it is based on a centralized recovery mechanism. While, Ropars and Morin's protocol keeps a list of determinants on messages headers, which may have a bigger size than dependency vectors.

To enhance more the performance, optimistic logging protocols are combined with other checkpointing techniques like coordinated checkpointing protocols [95] or CIC protocols [8] [96].

Table 2.5 summaries the related works of optimistic logging protocols

[1] A dependency vector has n entries, where n is the number of processes in the system. Each entry contains an incarnation number(is equal to the number of times P_i has failed or rolled back) and a state sequence number (or simply sequence number

Table 2.5: Related works of optimistic logging protocols

Reference	Combined with	System Model	Message ordering	Number of integers piggybacked	Number of concurrent failures	Number of rollbacks per failure	Asynchronous recovery
Strom and Yemini, 1985 [4][a]	Uncoordinated	Point-to-Point	FIFO	$O(N)$	N	$O(2^N)$	Mostly
Johnson and Zwaenepoel, 1990[93][a]	Uncoordinated	Point-to-Point	None	$O(N)$	N	1	No
Peterson and Kearns, 1993 [90][a]	Uncoordinated	Ring	FIFO	$O(N)$	1	1	No
Sistla and Welch, 1996 [9][a]	Uncoordinated	Group based	FIFO	$O(N)$	1	1	No
Park et *al*. 2002 [91]	Uncoordinated	Point-to-Point	None	$O(N)$	1	1	Yes
Damani et *al*. 2003 [92]	Uncoordinated	Point-to-Point	None	$O(N)$	1	1	Yes
Ropars and Morin, 2007 [94]	Uncoordinated	Point-to-Point	FIFO	$O(L)$	f	1	Yes
Jiang et *al*. 2008 [8]	CIC	Point-to-Point	None	$O(N)$	1	1	No
Wang et *al*. 2011 [95]	Coordinated	Point-to-Point	None	$O(N)$	1	1	No
Luo and Manivannan, 2012 [96]	CIC	Group based	None	$O(N)$	1	1	Yes

L: list length

[a] This row is taken from [92]

2.2.4.3 *Causal message logging protocols*

Causal message logging combines the advantages of optimistic and pessimistic message logging. It ensures asynchronous logging like optimistic logging and avoids orphans like pessimistic approach by ensuring that every message determinant is stable (logged in the stable storage) or available in local logs of processes. These features come at the expense of more complex recovery and large message overhead.

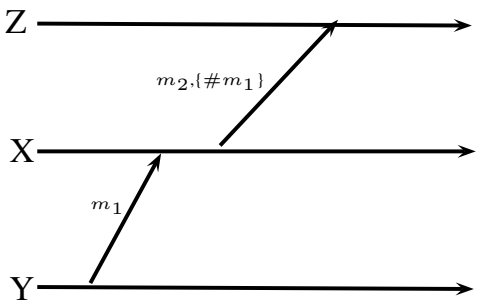


FIGURE 2.7: Causal message logging

Causal message logging requires that volatile log to be piggyback on each message. On receiving such a message, the receiver adds the piggybacked logs to its local log. As shown in Figure 2.7, when process X receives the message $m1$, its adds the determinant of m_1 to its local log. Then, when X send the message m_2, it appends the determinant of the message m_2. Hence its name of causal logging.

Elnoosahy et al. [97] started the area of causal message logging by introducing Manetho protocol. In this protocol, each process piggybacks its causal information in an antecedence graph[2] that provides a complete history of non-deterministic events. However, carrying the entire graph is not practical and may lead to unacceptable overhead.

Further reductions are the subject of FBL (Family Based Logging) protocols proposed by Alvisi et al. [98, 99]. FBL protocols are based on the following assumption: tolerating f failures by FBL protocols require that every message determinant is stored in $f+1$ different processes logs. To trim the antecedence graph, FBL protocols need to piggyback other extra information other than the antecedence graph like dependency matrix[3], stability matrix [4] and stability vector [5].

[2] an antecedence graph consists of nodes representing non-deterministic receiving events and edges corresponding to the happen before relation between such events

[3] Dependency matrix is N×N matrix of integers constructed from dependency vectors

[4] Stability Matrix: $SMat_p$ is a (f + 1)× N matrix of integers. For all processes q, $SMat_p$[i, q] is the highest receive sequence number of any message m delivered by q for which $|\, Log(m) \,|_p = i$.

[5] Stability vector is the row (f+1) of the stability matrix

In contrary to FBL protocols, Bhatia et al. [100] have proposed a Hierarchical Causal Message Logging (HCML) approach for a wide-area environment that does not restrict dependency information propagation.

Ahn [102] has proposed a protocol for mobile systems. These works focus principally on garbage collection to remove redundancy information.

Lemarnier et al. [12] have proposed an implementation of Manetho protocol [97] and LogOn protocol [101] called MPICH-V2 based on event logger.

To address the scalability issue, Ci et al [103] divided the system into zones (groups) where determinants are piggybacked only on intra-zones messages. They have also proposed an approach to eliminate information redundancy in an antecedence graph without using dependency matrix and dependency vectors [104].

A recent work by Meneses and Kalé [105] proposes an algorithm called CAMEL devoted for collective communication operation. it is based on multicast spanning tree and does not require multicast message determinants to be stored in logs of all multicast group members.

Table 2.6 summaries the related works of causal logging protocols

Table 2.6: Related works of causal logging protocols

Reference	System Model	Information	Number of integers piggybacked	Number of concurrent failures	Asynchronous recovery
Elnozahy and Zwaenepoel,1992 [97]	Point-to-Point	Antecedence graph	$O(D)$	N	Mostly
Alvisi et *al.* (1996, 2002) [98] [99]	Point-to point	Antecedence graph	$O(D)$[a]	f	Yes
		Antecedence graph + Log(m)	$O(D)$[a]		
		Antecedence graph + $\|Log(m)\|$	$O(D)$[a]		
		Antecedence graph + Dependency matrix	$O(D+N^2)$[a]		
		Antecedence graph + Stability matrix	$O(D+fN)$[a]		
		Antecedence graph + Stability vector	$O(D+N)$[a]		
Lee et al. 1998 [101]	Point-to-Point	Antecedence graph	$O(D)$	1	Yes
Bhatia et al. 2003 [100]	Tree	Antecedence graph	$O(D)$		Yes
Ahn et al. 2004 [102]	Point-to-Point	Antecedence graph	$O(D)$	f	Mostly
Lemarinier et al. 2004 [12]	Point-to-Point	Antecedence graph	$O(D)$		-
Ci et *al.* 2009 [103]	Group based	Antecedence graph	$O(D)$		-
Ci et *al.* 2010 [104]	Point-to-Point	Antecedence graph	$O(D)$	-	-
Meneses and Kalé, 2015 [105]	Spanning Tree	Antecedence graph	$O(D)$		-

f is the number of failures. D is the number of determinants piggybacked on the message.
$Log(m)$ is the set of processes that have a copy of determinant of the message m.

[a] The complexity is calculated by the protocol's authors

2.3 Techniques used for Checkpoint Interval selection

As shown in the foregoing sections, several studies are devoted to designing checkpointing protocols with the best tradeoff between overhead and performance. Since systems are prone to failures, designing a checkpointing protocol is not sufficient if it does not take into account the unpredictable failure time. The problem is that taking excessive checkpoints (checkpoint interval is too small) may lead to performance degradation due to the overhead incurred by transferring and storing a large number of checkpoints (especially if the network channels and the storage media are shared among checkpointing applications). Conversely, if the checkpoints are deficient (checkpoint interval is too large), it will enlarge the recovery time and increase the amount of work to redo in the event of a failure. Thus, it is worthwhile to strike a balance and find an optimal checkpoint interval.

The optimal checkpoint interval is the subject of several studies. Among these studies is that of Young [106], which proposed a first order model. In Young's model, failures occur with rate λ which is governed by a Poisson process. The Young's optimal interval is the following

$$\tau_{opt(Young)} = \sqrt{2\delta(M+R)} \tag{2.1}$$

where M is the mean time between two failures and R is the recovery time and δ is the checkpoint dump time

A more accurate model has been proposed by Daly [107], which is considered as an extension of Young previous work.

$$\tau_{opt(Daly)} = \begin{cases} \sqrt{2\delta M}\left(1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right) - \delta & \text{if } \delta < 2M \\ M & \text{if } \delta \geq 2M \end{cases} \tag{2.2}$$

However, these studies assume that no failure occurs during the recovery, thus, Xu et al. [108] have proposed a model that does not consider this latter assumption.

Unlike the mentioned studies that try to find out the optimal checkpoint interval to minimize the expected execution time. Other studies attempt to provide an optimal

checkpoint interval that minimizes the number of I/O checkpoints operations during an application execution [109, 110]. While other works aim to provide models that minimize energy consumption [111].

2.4 Techniques for stable storage contention avoidance

Several techniques are used to optimize the stable storage use, we limit ourselves to only incremental and diskless checkpointing.

2.4.1 Incremental Checkpointing

Incremental checkpointing attempts to reduce the size of checkpoint data by avoiding rewriting unchanged portions of the process states between two consecutive checkpoints. In the incremental checkpoint scheme, the first checkpoint is fully recorded. After that, only changed pages since the last checkpoint must be saved. Incremental checkpointing may be one of following categories: Page-based [112, 113] or Hash-based [114, 115]. Page-based incremental checkpointing requires paging support from hardware and the operating system. Page-based techniques might not scale well because if only one bit in a page changes, the entire page must be saved. Meanwhile, Hash-based Incremental Checkpointing, known also as probabilistic checkpointing tries to minimize the state saved in a checkpoint. It uses hashing algorithms to dynamically identify modified blocks in a checkpoint interval, rather than saving the dirtied pages as in standard incremental checkpointing.

2.4.2 Diskless Checkpointing

Diskless checkpointing approach was first proposed in [116]. It removes the physical disk from the stable storage operation replacing it with unused main memory on peer machines to avoid the excessive overhead associated with stable storage operation. Diskless checkpointing fall into four main categories: neighbor-based [117], parity-based [118, 119], Reed-Solomon code-based [116, 120] and Mutual-aid [121–123].

In the neighbor-based approach, Neighbor processors act as checkpoint processors whose memories are used as storage media to save checkpoints, In parity based checkpointing, processes cooperate in taking their checkpoint. Then, processes, for example,

uses XORing operations to calculate checkpoint parity which is stored in a dedicated process, In event of failure, processes with dedicated process coordinate to decode the checkpoint of failed process. Mutual-aid approach is a hybrid approach that combines neighbor and parity techniques. Reed-Solomon approach requires m processes to use Galois field arithmetic that generate k pieces for encoding information which will be then stored in k dedicated processes. The main advantage of this approach is tolerating m simultaneous failure but it suffers from more complex calculation problem and dedicated processes may increase failure likelihood.

2.5 Conclusion

Checkpointing is one of the fault tolerance techniques, which is an active area of research. It consists of saving periodically system state in stable storage. A myriad of checkpointing protocols has been proposed in the literature on which this chapter has shed light.

First, this chapter has started by describing coordinated checkpointing protocols. Such a class is characterized by the synchronization between processes. During the checkpointing procedure, the application may or may not blocked. Blocking the application implies a large overhead in terms of time. However, not blocking the application results in extra overhead in terms of synchronization messages and complicates the orphan messages handling. Hence to overcome such a problem, techniques such assuming specific network overlays, dependency tracking are used

Secondly, we have presented CIC protocols. CIC class use another form of synchronization between processes to ensure global checkpoint consistency while keeping a partial autonomy in taking local checkpoints. It is based on piggybacking extra control information on messages, which, in addition to local knowledge, helps to decide whether a forced checkpoint should be taken. CIC protocols can be broadly categorized based on the property ensured into those ensuring NZC property and those ensuring RDT property. However, The problem associated with CIC class is the large overhead added by appending extra information to messages, which is in some cases about $O(N^2)$. Besides, CIC class has the problem of keeping more than one checkpoint in stable storage because no bounds are made for the rollback propagation upon a failure.

As concerns uncoordinated class, processes have the full autonomy in taking local checkpoints. This chapter has not described techniques in such a class because it is generally used in conjunction with message logging techniques. Message logging techniques

are subdivided into three types: pessimistic, optimistic and causal. Logging techniques help to handle messages and make the recovery from a more recent state possible. Based on the order of logged messages, processes can re-execute in the same manner as before a failure. Nevertheless, logging techniques also have the problem of the overhead generated by recording messages in stable storage (pessimistic techniques) or by extra information transported by messages(optimistic and causal techniques).

Additionally, this chapter has described some techniques used to optimize checkpointing protocols performance such as interval selection, incremental and diskless checkpointing.

As a conclusion, this chapter has broadened the survey of Elnozahy et al. [2] with new references, techniques, discussion and comparisons. Such a survey helps us to explore and identify problem and weaknesses of techniques proposed so far and has motivated us to propose new solutions, which is the subject of the following chapters.

Chapter 3

Extending Index based NZC Protocols by Recovery Capability

Contents

3.1 INTRODUCTION

In distributed systems, communications between processes establish dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may direct some non-failed processes to rollback, creating what we call a rollback propagation. Consequently, processes may lose an important amount of work due to this phenomenon.

This chapter is devoted not only to extend CIC protocols but also to study the overhead incurred due to the rollback propagation. For this purpose, we will show how BRA (Basic Recovery Algorithm proposed by Manivanan and Singhal) can be changed in order to extend the following protocols: BCS, BCS-aftersend, LazyBCS, LazyBCS-aftersend, and LazyMS. Our work motivated by the fact that fewer research works in the literature study checkpointing protocols behavior in a failure-prone environment.

Over this chapter, we will present the characterization of NZC property based timestamping functions. Next, we will present index based checkpointing protocols that uses either classical indexing or lazy indexing. We will also describe the BRA protocol and see how it can work in conjunction with checkpointing protocol like BCS, BCS-aftersend, LazyBCS, LazyBCS-aftersend and LazyMS protocols. Finally, we will present our simulation study that compares protocols in failure-free and failure-prone environments.

3.2 NZC CHARACTERIZATION

No-Z-Cycle property postulates that there is no checkpoint involved in Z-Cycle. In other words, there is no useless checkpoint that cannot be part of any consistent global checkpoint.

The timestamping is used to characterize the NZC property. It allows the scheduling of events in a given checkpoint interval so that the reception events preceding the emission events. Hence it prevents the formation of Z-Cycle. If this is applied to all

distributed execution intervals we say that it ensures the virtual precedence property (virtual property precedence VP)

To characterize the NZC property, we will briefly describe the VP property

3.2.1 Virtual Precedence (VP) Property

The distributed execution can be modeled by a set of partially ordered events. A high level of abstraction introduced by Hélary et al [124] considers the execution of each process as a consecutive sequence of intervals, and each interval is a sequence of events such as

- each event belongs to a single interval $I_{i,x}$.
- each interval comprises at least one event

3.2.2 NZC Property and VP Property

Consider an abstraction of a distributed execution where each message and interval are associated with timestamps. Informally, such an abstraction satisfies VP property means that it is possible to timestamp messages and intervals as follows:

F1- For each pair of messages m and m' as $receive(m') \in I_{i,x}$ and $send(m) \in I_{i,x}$ then the timestamp of m' is less or equal to the timestamp of the message m.

F2- The timestamp associated with the interval $I_{i,x}$ is greater or equal to all timestamps associated with messages received in $I_{i,x}$ and less than that of messages transmitted in this interval.

This means that, via timestamps, communications can be seen causal in each interval. In other words, communication events can be reordered in any interval so that all reception events precede all emission events and timestamps increase along Z-paths.

An abstraction of a distributed execution satisfies the VP property if and only if we can design a timestamping function that has the following characteristics:

1. The intervals of the processes involved in a communication must be timestamped increasingly (safety property).
2. The timestamp of a process must be incremented after communications (liveness property).

Note: If the interval consists of a single event, the timestamping function reduces to the Lamport logical clock [125] or vector clocks Fidge-Mattern [126][127].

In the context of checkpointing problem, intervals correspond to checkpoint intervals, then, the abstraction of the distributed execution corresponds to a checkpoint and communication pattern. Helary et al. [124] proved the equivalence between VP property and NZC property.

Theorem 3.1

A distributed execution with checkpoints that satisfies the VP property satisfies also the NZC property and vice versa [128]

> ***Proof:*** The proof of the theorem can be found in citeHelary2002. ■

3.2.3 Timestemp based Characterization of NZC Property

This section presents a characterization based on timestamps of a distributed execution with checkpoints having no Z-Cycles.

Theorem 3.2

The two following properties of a distributed execution with checkpoints ($\hat{H}$, C) are equivalent [16]

1. *($\hat{H}$, C) has no Z-cycles.*
2. *It is possible to timestamp its local checkpoints so that $A \xrightarrow{Z} B \Rightarrow A.t < B.t$*

> ***Proof:***
>
> - Direction $\longrightarrow$. the proof is done by contradiction. Assume that there is a Z-Cycle from C to itself $C \xrightarrow{Z} C$ means that $C.t < C.t$, which is clearly impossible.
> - Direction $\longleftarrow$. Consider that (C,$\xrightarrow{Z}$) is acyclic. There is consequently a topological sort of its vertices. It follows that each vertex c (local checkpoint) can be labeled with an integer C.t such that $C_1 \xrightarrow{Z} C_2 \Longrightarrow C_1.t < C_2.t$
>
> ■

The inherent idea of this property can be seen as follows: if it is possible to design a protocol that guarantees the absence of useless checkpoints (no Z-Cycle), then, such a

protocol ensures a timestamp increasing along Z-paths, which must be consistent with the F1 and F2 rules.

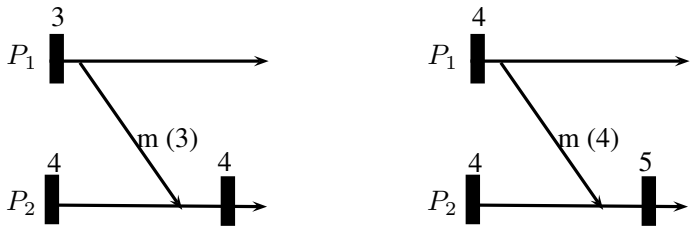
3.2.3.1 Classical Indexing (timestamping)

1. Each process P_i maintains a logical clock lc_i (or a sequence number sn_i) initialized to 0.
2. When P_i decides to take a local checkpoint, it saves its state with a copy of lc_i (or sn_i). Then, it increments lc_i.
3. When P_i sends a message m, it piggybacks its $lc_i(sn_i)$ on the message message noted $m.lc$ ($m.sn$).
4. Upon receiving such a message m, P_i updates its $lc_i = max(lc_i, m.lc)$ (or $sn_i = max(sn_i, m.sn)$.

3.2.3.2 Lazy Indexing (timestamping)

Vieira et al have introduced the lazy strategy [28] that strategy has the same rules as the classical one with modified rule 2.

1. Each process P_i maintains a logical clock lc_i (or a sequence number sn_i) initialized to 0 and a boolean variable $equiv_i$ initialized to false.
2. When P_i decides to take a local checkpoint, it saves its state with a copy of lc_i (or sn_i). Then, it increments lc_i if $equiv_i$ equals false .
3. When P_i sends a message m, it piggybacks its lc_i (sn_i) on the message noted $m.lc$ ($m.sn$).
4. Upon receiving such a message m, P_i updates its variables as follows: $equiv_i = (m.lc_i \leq lc_i)$ and $lc_i = max(lc_i, m.lc)$ (or $sn_i = max(sn_i, m.sn)$.



(a) Case1 (b) Case2

Figure 3.1: Lazy indexing, to increase or not to increase the timestamp

In Figure 3.1, when P_2 decides to take the next basic checkpoint, we have two cases:

Case 1: P_2 does not increment lc_2 for the next basic checkpoint, which equals to 4 because P_2 has received a message m timestamped by 3 and $lc_2 > m.lc$ (Figure 3.1(a)).

Case 2: P_2 increments lc_2 for the next basic checkpoint, which equals to 5 because P_2 has received a message m timestamped by 4 and $lc_2 = m.lc$ (Figure 3.1(b)).

3.3 RELATED WORKS: INDEX BASED NZC PROTOCOLS

3.3.1 Classical Indexing based protocols

In this section, we will present BCS [27], BCS-aftersend [28] and MS [58] protocols.

3.3.1.1 BCS Protocol

Take – Basic – Checkpoint: When a process P_i decides to take a basic checkpoint, it saves its state with a copy of its lc_i. Then it increments its lc_i

Send – Message: The process piggybacks its lc_i on the message and sends the message.

Receive – Message: When a process P_i receives a message m, it takes a forced checkpoint if the message carried new information. P_i then updates its lc_i ($lc_i = max(lc_i, m.lc)$ and consumes the message.

Procedure 1: When process P_i sends a message M

$M.lc = lc_i$;
send (M);

Procedure 2: When Process P_j receives a message from process P_i

if $lc_j < M.lc$ **then**
  Take checkpoint C;
  $lc_j = M.lc$;
Process the message;

Procedure 3: When it is time for process P_i to take a basic checkpoint

Take checkpoint C ;
$lc_i + +$;

Consider a system constituting of four process P_1, P_2, P_3 and P_4 as shown in Figure 3.2. Basic checkpoints are shown as black rectangular boxes while forced checkpoints are

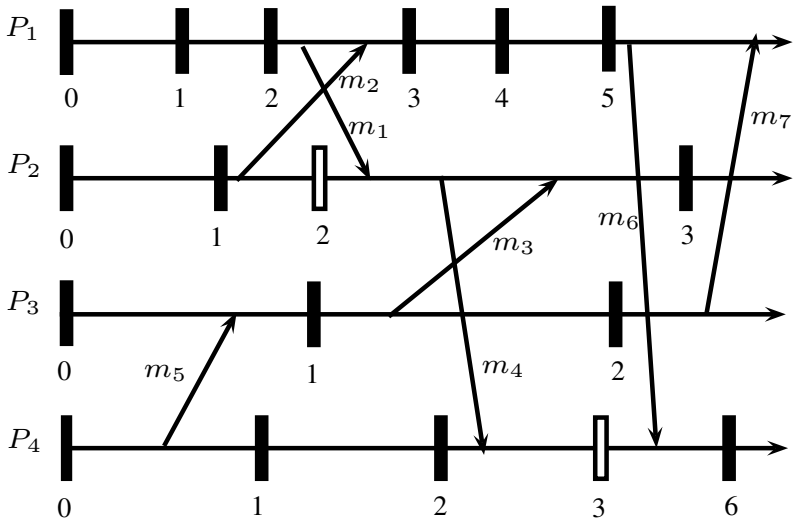


Figure 3.2: BCS protocol

shown as white rectangular boxes, Figure 3.2 is also enriched by local clocks lc_i. For the scenario depicted in Figure 3.2, each process increments its lc_i whenever it takes a basic checkpoint. Message m_1 forces process P_2 to take a forced checkpoint before processing it ($m_1..lc = 3 > lc_2 = 2$). Similarly, P_4 is forced to take checkpoint $C_{4,5}$ before processing the message m_6 whereas process P_1 does not take forced checkpoint upon receiving message m_7 ($m_7..lc = 3 < lc_1 = 6$).

3.3.1.2 BCS-aftersend Protocol

This protocol is a simple enhancement of BCS protocol. It is based on the following observation: if a process P_i has not sent messages since its last checkpoint, no Z-pattern can be formed. Consequently, no additional checkpoints are needed. As an addition to the data structure of BCS protocol, a boolean variable $sent_i$ is used to capture "no send" pattern.

Procedure 1: When process P_i sends a message M

$M.lc = lc_i$;
$sent_i = true$;
send (M);

Procedure 2: When Process P_j receives a message from process P_i

if $sent_j \wedge (lc_j < M.lc)$ **then**
    Take checkpoint C;
    $lc_j = max(lc_j, M.sn)$;
Process the message;

Procedure 3: When it is time for process P_i to take a basic checkpoint

Take checkpoint C ;
lc_i++;
$sent_i = false$;

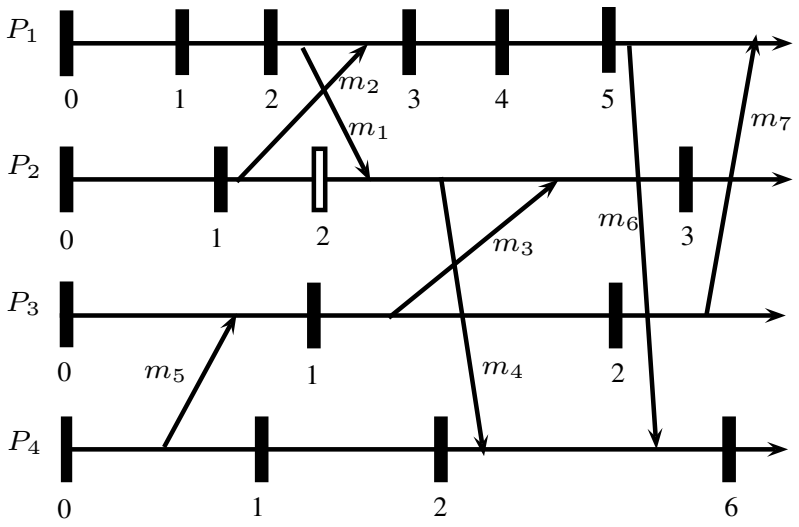


FIGURE 3.3: BCS-aftersend protocol

Likewise to Figure 3.2, Figure 3.3 shows an execution of BCS-aftersend protocol under the same scenario. When process P_2 receives message m_1 it takes a forced checkpoint before processing this message because it has already send a message in its current checkpoint interval and the variable lc carried by the message is greater than its lc_2 ($sent_2 = true \wedge m_1.lc = 3 > lc_2 = 2$). However, process P_1 does not take a forced checkpoint before processing message m_7 even though it has send a message in its current checkpoint interval because the variable lc brought by message m_7 is less than its lc_2.

3.3.1.3 MS protocol

MS protocol was proposed by Manivannan and Singhal, In this protocol, each process maintains two variables next and sn_i. *next* variable is auto increment and sn_i is used to timestamp checkpoints taken by a process.

Take – Basic – Checkpoint: When it is the time to take a basic checkpoint, a process P_i skips taking the basic checkpoint if it has already taken a forced checkpoint with $C.sn \geq next_i$. Otherwise, P_i updates its sn_i and saves its state as a basic checkpoint with a copy of its *sn*.

Send – Message: When a process P_i sends a message, it appends its current sn_i to this message.

Receive – Message: When a process P_i receives a message m, it checks for new information. If it is the case, it takes a forced checkpoint and assigns *sn* brought by the message to this checkpoint. Then, the process P_i consumes the message.

Procedure 1: When it is time for process P_i to increment $next_i$

$next_i = next_i + 1$;

Procedure 2: When process P_i sends a message M

$M.sn = sn_i$;
send (M);

Procedure 3: When Process P_j receives a message from process P_i

if $sn_j < M.sn$ **then**
    $sn_j = M.sn$;
    $C.sn = sn_j$;
    Take checkpoint C;
Process the message;

Procedure 4: When it is time for process P_i to take a basic checkpoint

if $next_i > sn_i$ **then**
    $sn_i = next_i$;
    $C.sn = sn_i$;
    Take checkpoint C;

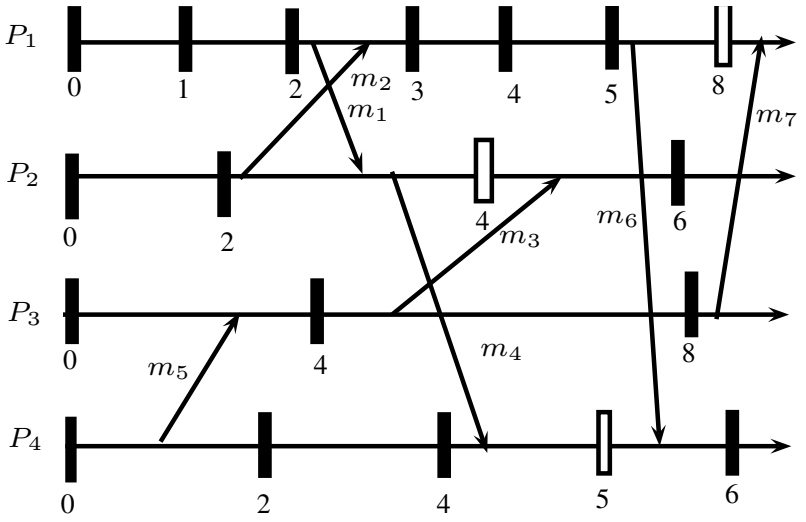


Figure 3.4: MS protocol

Figure 3.4 illustrates an execution of MS protocol under the same scenario depicted in Figure 3.2. Each process P_i increments its variable $next_i$ every x time units. Process P_1 takes a basic checkpoint every x units of time, processes P_2 and P_4 take their basic

checkpoints every $2 * x$ time units while process P_3 takes its basic checkpoints every $4 * x$ time units. Sequence numbers sn_i are picked from local counters $next_i$. As shown in Figure 3.4, process P_2 is directed by MS protocol to take a forced checkpoint before processing the message m_3 ($m_3.sn = 4 > sn_2 = 2$). Later, P_2 skips taking a basic checkpoint with $sn_2 = 4$ because it has already taken a forced checkpoint. Message m_6 forces process P_4 to take a forced checkpoint before processing it because $m_6.sn$ is greater than $P_4's$ sn. However, P_3 does not take a forced checkpoint before processing the message m_5 ($m_5.sn < sn_3$).

3.3.2 Lazy Indexing based protocols

In this section, we will present LazyBCS [28], LazyBCS-aftersend [28]protocols. In addition, we have applied MS protocol to lazy indexing, which results in LazyMS protocol.

3.3.2.1 LazyBCS Protocol

Procedure 1: When process P_i sends a message M

$M.lc = lc_i$;
send (M);

Procedure 2: When Process P_j receives a message from process P_i

if $lc_j < M.lc$ **then**
    Take checkpoint C
if $M.lc \geq lc_j$ **then**
    $equiv_j = false$
$lc_j = max(M.lc, lc_j)$;
Process the message;

Procedure 3: When it is time for process P_i to take a basic checkpoint

Take checkpoint C ;
$lc_i + +$;
$equiv_i = true$;

In Figure 3.5, we show an example of BCS protocol execution using lazy indexing. As shown, Process P_1 takes three consecutive basic checkpoints assigned with the same lc_1, which is equal to 0. Then, when P_1 receives the message m_2. it assigns the next basic checkpoint by $lc_1 = 1$. Similarly, other processes behave like P_1 when they receive messages assigned with lc greater or equal than their local lc_i. Process P_2, on receiving

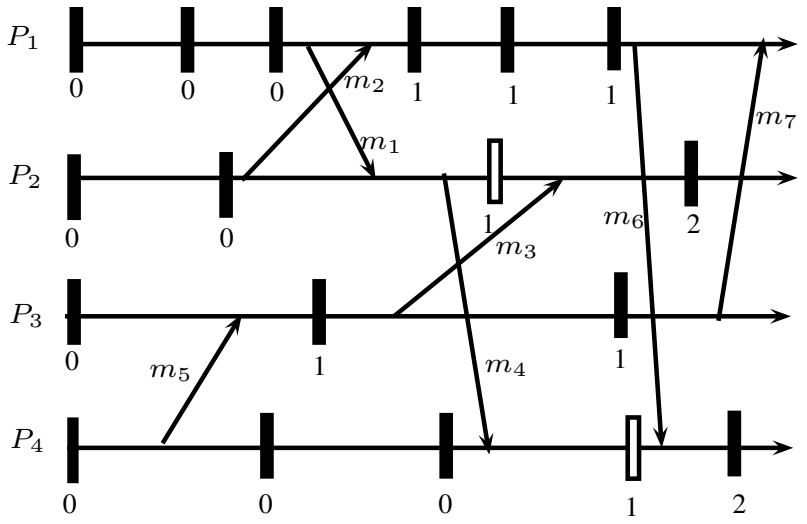


Figure 3.5: LazyBCS protocol

message m_3, takes a forced checkpoint and assigns it by $lc_2(=1)$ because its induction condition becomes true. By the same way, process P_4 takes its checkpoint $C_{4.1}$ before processing the message m_6 and assigns it with $lc_4(=1)$.

3.3.2.2 *LazyBCS-aftersend protocol*

Procedure 1: When it is time for process P_i to take a basic checkpoint

Take checkpoint C ;
lc_i++;
$equiv_i = true$;

Procedure 2: When process P_i sends a message M

H]
$M.lc = lc_i$;
send (M);

Procedure 3: When Process P_j receives a message from process P_i

if $sent_j \wedge lc_j < M.lc$ **then**
    Take checkpoint C
if $lc_j \geq M.lc$ **then**
    $equiv_j = false$
$lc_j = max(M.lc, lc_j)$;
Process the message;

Figure 3.6 depicts an execution of LazyBCS-aftersend protocol for the same scenario depicted in the above figures. Processes P_1 and P_2, on taking their basic checkpoints,

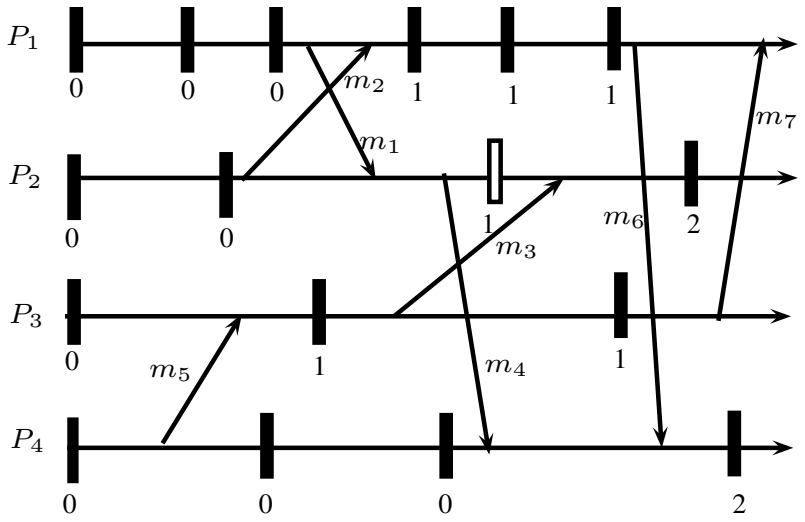


FIGURE 3.6: LazyBCS-aftersend protocol

do not change the lc_i value assigned until they receives m_2 and m_1 respectively, which piggyback lc equal to their local lc_1. After that, P_1 and P_2 will assign $lc_i(=1)$ to their next basic checkpoints. Moreover, upon receiving m_3, P_2 takes a forced checkpoint before processing it ($sent_2 = true \wedge m_3.lc = 2 > lc_2 = 1$). Contrariwise, P_4 does not need to take forced checkpoint before processing m_6 even thought it brought new information ($m_6.lc = 1$) because it has not sent a message in its current checkpoint interval.

3.3.2.3 *LazyMS protocol*

We have applied MS protocol to lazy indexing, which results in LazyMS protocol.

Procedure 1: When it is time for process P_i to increment $next_i$

$next_i = next_i + 1$;

Procedure 2: When process P_i sends a message M

$M.sn = sn_i$;
send (M);

Procedure 3: When Process P_j receives a message from process P_i

if $sn_j < M.sn$ **then**
  $sn_j = M.sn$;$C.sn = sn_j$;
  Take checkpoint C;
  $equiv_j = false$;
Process the message;

Procedure 4: When it is time for process P_i to take a basic checkpoint

```
if next_i > sn_i then
    if equiv_j = false then
        sn_i = next_i;
        C.sn = sn_i;
    Take checkpoint C;
    equiv_j = true;
```

In the context of lazy indexing, Figure 3.7 depicts an execution of LazyMS protocol. We recall that each process P_i increments its variable $next_i$ every x time units. Process P_1 takes a basic checkpoint every x units of time, processes P_2 and P_4 take their basic checkpoints every $2 * x$ time units while process P_3 takes its basic checkpoints every $4 * x$ time units. P_1, on taking the first three checkpoints, it assigns them with the same sn_1, which is equal to 0. Then, when P_1 receives the message m_2. it assigns the next checkpoint by $sn_1 = 3$ picked from its local counter $next_1$. In the same manner, other processes will take their basic checkpoints. Furthermore, when P_2 receives m_4, it takes a forced checkpoint assigned with $sn_2 = 4$. Similarly, P_1 and P_4 behave like P_2 upon receiving m_6 and m_7.

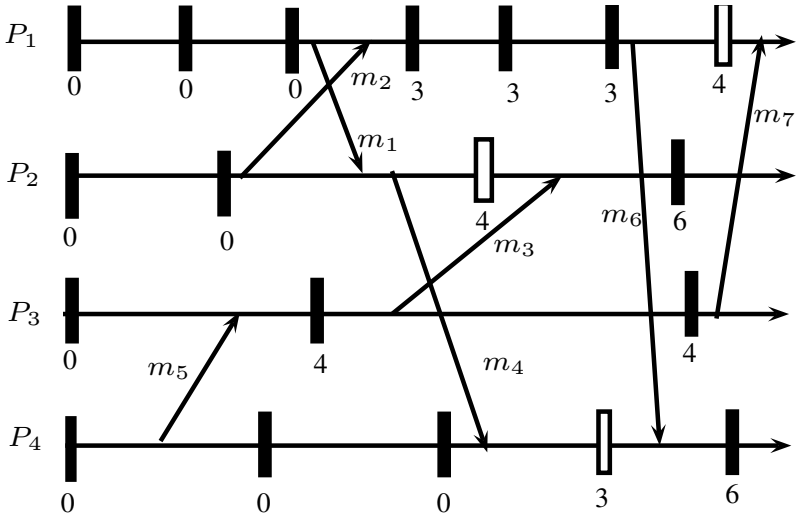


Figure 3.7: LazyMS protocol

Important note

- BCS, BCS-aftersend LazyBCS and LazyBCS-aftersend protocols share the following property.

Theorem 3.3

Let $C_{i,x}$ be a local checkpoint with $C_{i,x}.t = a$. The global checkpoint G_a ={C_{1,x_1} ... $C_{i;x_i}$ ... C_{n,x_n} }defined in the following way: ($\forall k : C_{k,x_k}$ is the last local checkpoint of P_k such that $C_{k,x_k}.t$ is the greatest integer $\leq a$. Then, G_a includes $C_{i,x}$ is a consistent global checkpoint [16].

> ***Proof:*** (The following proof is a sketch proof of above theorem, the reader can refer to [16, 129] for a complete one).
> The proof is done by contradiction. Assume that G_a is inconsistent and $A = C_{i,x}$, which means that $\exists B$ such that $A \xrightarrow{Z} B$. Hence, $B.t > A.t$ (By Theorem 3.2). On the other hand, we have $B.t \leq A.t$ due to the theorem assumption that $\forall C \in G_a \mid C.t \leq A.t$ which leads to a contradiction. So, G_A is consistent. ■

Using property 3.3, the task of finding the recovery line is facilitated. If a process P_i fails and returns to the checkpoint $C_{i,x} \mid C_{i,x}.t = a$, other processes need only to rollback to their last local checkpoints such that $C_{k,x_k}.t \leq a$.

- MS and LazyMS protocols share the following property.

Theorem 3.4

A local checkpoint $C_{i,x}$ with $C_{i,x}.sn = m$ associated with the global checkpoint G_m $C_{1,x_1} \ldots C_{i;x_i} \ldots C_{n,x_n}$. Then, G_m is a consistent global checkpoint if $C_{i;x_i}$ is the earliest local checkpoint of P_i such that $\forall i$ $C_{i;x_i}.sn_i \geq m$ [14].

> ***Proof:***
> Suppose G_m is not a consistent global checkpoint, which means there exist two checkpoints $C_{r,m_r}, C_{q,m_q} \in G_m|$ such that $C_{q,m_r} \xrightarrow{Z} C_{q,m_q}$. Let $M_1, M_2, \ldots, M_n$ be the Z-path from C_{r,m_r} to C_{q,m_q}
> On one hand, we have $M_n.sn \geq M_{n-1}.sn \geq \ldots \geq M_1.sn \geq m_r \geq m$, and receive (M_n) precedes C_{q,m_q}, which is impossible since P_q receives M_n only after taking a checkpoint whose sequence number $\geq M.sn$;
> On other hand, we have $\forall q;\ 0 \leq q \leq N-1$ $C_{q,x} \xrightarrow{C} C_{q,m_q} \Rightarrow x \leq m$, which means that no checkpoint with sequence number $\geq M.sn \geq m$ precedes C_{q,m_q}
> We get a contradiction, hence, the theorem.
>
> ■

Property 3.4 has an impact on facilitating the recovery line calculus. In other words, when a process P_i fails, it starts its execution from its last checkpoint $C_{i,x} \mid C_{i,x}.sn = m$, then, the other processes simply rollback from their last checkpoints such that $C_{i;x_i}.sn_i \geq m$.

3.4 Basic Recovery Algorithm (BRA)

BRA protocol has been proposed by Man Manivannan and Singhal [14]

Why BRA is used ?

BRA protocol has the following features

- The application is not blocked during recovery.
- Processes are asynchronous during the recovery, which means that each process decides when it must rollback or continue its execution.
- Processes recover from a consistent recovery line.
- It has a complete message handling procedure.

Assumption of BRA protocol

In BRA protocol, a process fails according to fail-stop mode and no other process fails until the rollback procedure terminates for the previous failure. In addition, failure does not propagates among processes.

3.4.1 Description of BRA protocol

In BRA protocol, rec_line_i is used to indicate the global checkpoint number, and the inc_i is incremented whenever a failure occurs. These two variables are stored with the checkpoint even if there is no failure. Later, these variables are used in the recovery process.

RollbackInitiation: When a process P_i fails, it returns to its latest checkpoint and broadcasts Roll_back message(inc_i, rec_line_i) to all other processes in the system.

Receive – Rollback – Message: at the arrival of Rollback message(inc, rec_line), There are two cases:

Case1 P_i rolls back to the earliest checkpoint C such that $C.sn \geq rec\_line$.

Case2 All the existing checkpoints have sequence numbers $< rec\_line$. In this case, P_i is directed to take a forced checkpoint assigned with received rec_line($C.sn = rec\_line$).

Procedure 1: Recovery initiated by process P_i after failure

Restore the latest checkpoint;
$inc_i = inc_i + 1$;
$rec\_line_i = sn_i$;
send rollback(inc_i, rec_line_i) to all other processes;
resume normal execution;

Procedure 2: Process P_j upon receiving Roll_Back(inc_i, rec_line_i) from P_i

if $inc_i > inc_j$ **then**
    $inc_j = inc_i$;
    $rec\_line_j = rec\_line_i$;
    Roll_Back(P_j);
    continue as normal;
else Ignore the rollback message;

Procedure 3: Roll_Back(P_j)

if $rec_line_j > sn_j$ **then**
    $C.sn = rec\_line$; $sn_j = C.sn$;
    Take checkpoint C;
else
    Find the earliest checkpoint C with $Csn \geq rec\_line_j$; $sn_j = C.sn$;
    Restore checkpoint C;
    Delete all checkpoints beyond C;

Figure 3.8 depicts an execution of BRA protocol. Assume that P_3 fails at the shown moment. When P_3 recovers, it increments its inc_2 to 1, sets rec_line_2 to sn_2(=9), restarts its execution from the latest checkpoint $C_{3.9}$ and broadcasts roll_back(1,9) messages to P_1 and P_2. P_1 will rollbacks to its earliest checkpoint $C_{1.10}$ whose sn = 10. However, P_2 will take a forced checkpoints because all its checkpoints having sequence numbers < 9 and assigns 9 as its sequence number. Therefore, {$C_{1.10}$, $C_{2.9}$, $C_{3.9}$} is the recovery line.

As one can see, the recovery is done by a simple and efficient way, and all processes rollback to a consistent global checkpoint, which is merely determined.

Theorem 3.5

If process P_i fails and initiates recovery by sending rollback ($inc_i recline_i$) message the system is restored to a consistent state after all the processes roll back in response to this message

Proof:

"A process knows about this rollback message either after receiving it directly

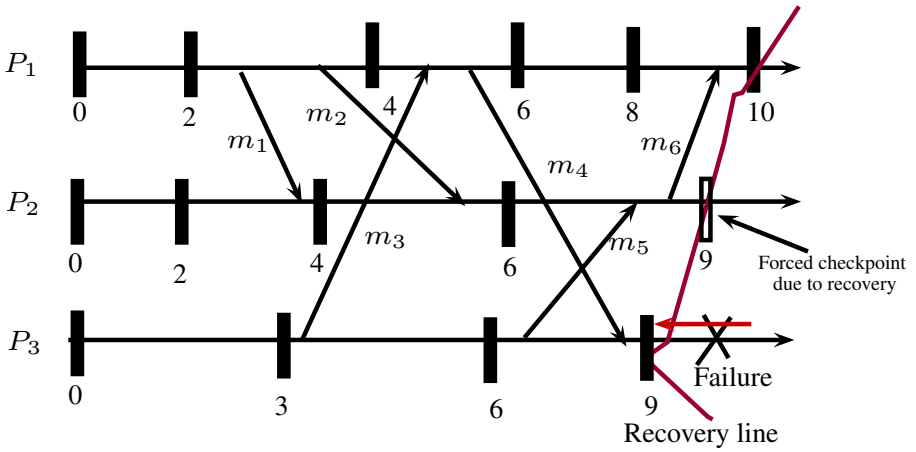


Figure 3.8: BRA protocol

> or after receiving a message sent by a process that already received this rollback message. In any case, as a result of this rollback message, all the processes rollback to their earliest checkpoint whose sequence number is ≥ rec_line By Theorem 3.4, the checkpoints to which the processes rollback form a consistent global checkpoint.Thus, the system is restored to a consistent state after all the processes rollback" [14]. ■

3.4.2 Message Handling

In failure recovery, it is important to bring the system to a consistent state, but it is also important to appropriately handle messages that are left in abnormal state due to the recovery. Hence, message handling becomes a real concern in the recovery process.

3.4.2.1 Handling received messages

With help of an example depicted in Figure 3.9, we show how messages are managed during recovery process (as shown in [14]).

a) M is a delayed message

A message is said delayed if it piggybacks an incarnation less than that of the destination process. In other words, the sender does not aware about the recovery at the sending moment. We have two cases.

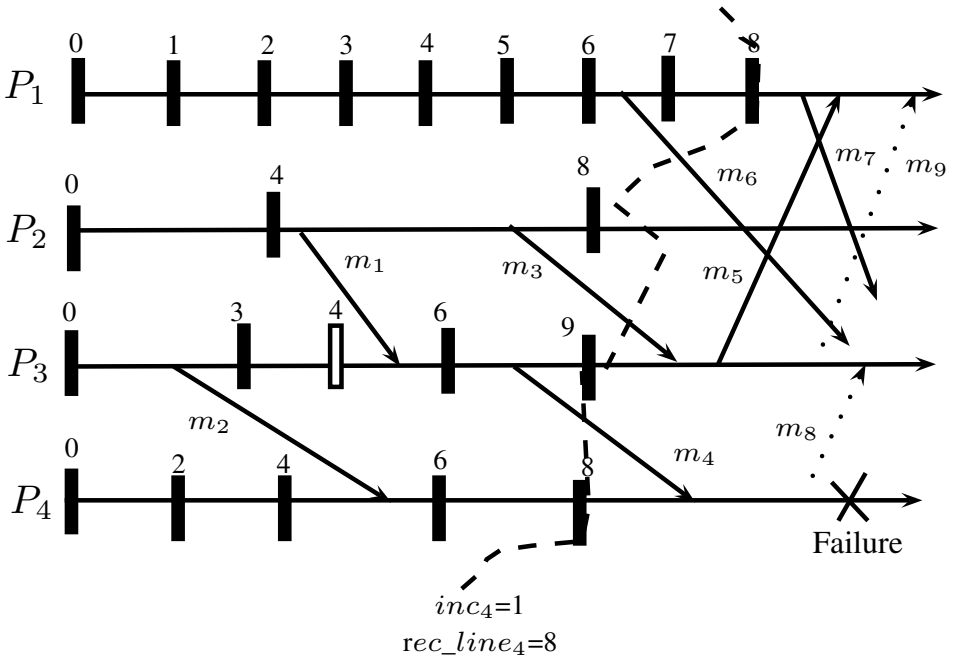


FIGURE 3.9: Different types of messages

Case 1 If $M.sn < rec\_line$, the message must be logged.

Case 2 If $M.sn > rec\_line$, the message must be discarded because it will be resend after recovery.

In Figure 3.9, Message m_6 must be logged because $m_6.sn = 6 < rec\_line = 8 \wedge m_6.inc = 0 < inc = 1$ while m_5 must be discarded because $m_6.sn = 9 > rec\_line = 8$

b) M was sent in the current incarnation

A process must log all messages having sequences numbers less than its one. If a failure occurs, these messages might need to be replayed.

Message m_8 has an incarnation equal to 1. At receiving this message, P_3 must log it because $m_8.inc = inc_3 = 1 \wedge m_8.sn = 8 < sn_3 = 9$

Thus, the logging rule is stated as follows:

A message M is logged if

1. $M.inc < inc_i \wedge M.sn < rec\_line_i$
2. $M.inc = inc_i \wedge M.sn < sn_i$

c) M was sent in the future incarnation

In this case, message M carries a new information about recovery, that is, M plays the role of Roll_back message because it was received before Roll_back message. Hence, P_i sets rec_line_i to M.rec_line and inc_i to M.inc, then, it rollbacks to its earliest checkpoint whose sequence number is greater or equal to rec_line. After that, M will be handled as in previous case ($M.inc = inc_i$).

3.4.2.2 Handling logged messages

To avoid lost message whose send event is done and receive event is undone, a process during recovery must replay messages from it log(note that, not all logged messages are replayed.

Thus, the following rule for replaying logged messages.

> A message M is replayed if it has received after checkpoint C (is a part of the recovery line) and $M.sn < rec\_line$.

In Figure 3.9, m_3 and m_4 must be replayed while m_2 must not.

Procedure 4: When process P_j receives a message M

```
if M.inc < inc_j then
    if M.sn > sn_j then
        log the message M;
        Process the message M;
    else Discard the Message M ;
else if M.inc == inc_j then
    if M.sn > sn_j then
        C.sn = M.sn;
        sn_j = C.sn;
        Take checkpoint C;
    else log the message M;
    Process the message M;
else if M.inc > inc_j then
    rec_line_j = M.rec_line;
    inc_j = M.inc;
    Roll_Back(P_j );
    Replay logged messages (i);
    Process the message;
```

Procedure 5: Replay logged messages (i)

for *each Message M*∈ *MessageLog* **do**
  if $M.sn < rec\_line_i$ **then**
    Replay Message M
  else
    Discard Message M

3.5 EXTENDING INDEX BASED CIC PROTOCOLS WITH BRA

After having described basic recovery algorithm(BRA) [14] and its mechanism of handling messages, we are now able to describe how Index based CIC protocols can work in conjunction with BRA.

3.5.1 Extending BCS-aftersend protocol

To extend protocols such as BCS, LazyBCS, BCS-aftersend and LazyBCS-aftersend, we need to modify BRA according to Property 3.3.This means that case1 of the procedure Roll_back must be modified as follows

Case1 P_i rolls back to the earliest checkpoint C such that $C.sn \leq rec\_line$.

and the procedure Roll_back is modified as follows:

Procedure 4: Roll_Back(P_j)

if $rec_line_j > sn_j$ **then**
  $C.sn = rec\_line$; $sn_j = C.sn$;
  Take checkpoint C;
else
  Find the earliest checkpoint C with $Csn \leq rec\_line_j$; $sn_j = C.sn$;
  Restore checkpoint C;
  Delete all checkpoints beyond C;

With the aid of an example depicted in Figure 3.10, we show how can BRA protocol can work with an another checkpointing protocol other than MS protocol. In this cases, we choose BCS-aftersend as checkpointing algorithm.

Similarly, we use the same scenario depicted in Figure 3.8. When P_3 recovers from the failure, it update its incarnation, rec_line ($inc_2 = 1, rec\_line = 3$). Then, it rollbacks to its checkpoint $C_{3,3}$ and sends Roll_back (1,3) message to all other processes. Upon receiving

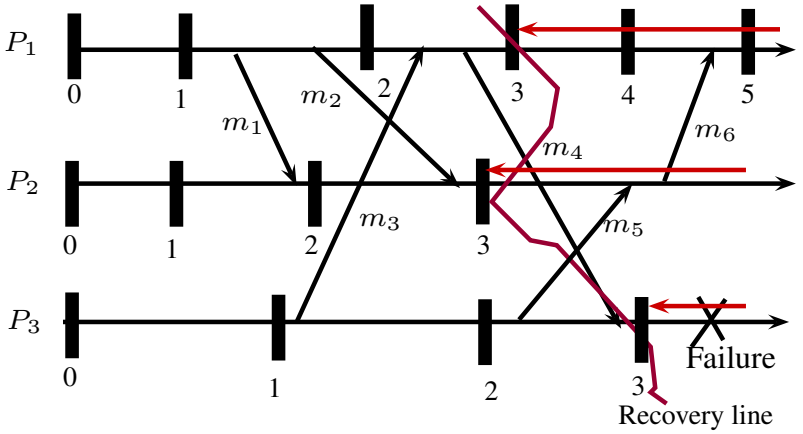


Figure 3.10: Adapted BRA protocol

such a message, P_1 deletes its checkpoints $C_{1.4}$ and $C_{1.5}$ and restarts its execution from its checkpoint $C_{1.3}$, which has a sequence number equal to 3. Whereas, P_2 will return to checkpoint $C_{3.3}$ assigned with $sn_2 = 3$. Thus, $\{C_{1.3}, C_{2.3}, C_{3.3}\}$ is the recovery line.

Note that, it is not necessarily that all checkpoints in the recovery line have the same sequence number.

3.5.2 Extending the other CIC protocols

BCS, LazyBCS and LazyBCS-aftersend protocols ensure the same property as BCS-aftersend protocol (Property 3.3). As a result, the extension made for BCS-aftersend can be used for such protocols

As concerns LazyMS protocol, it can work in conjunction with BRA protocol without any modification.

3.5.3 Factors impacting the recovery process

3.5.3.1 Impact of Timestamping functions

1. BCS-aftersend protocol (Figure 3.10) deletes more checkpoints than MS protocol (Figure 3.8). Additionally, the amount of work to redo when using BCS-aftersend protocol is greater than that of the case when using MS protocol. Moreover, MS protocol recovers from a more recent state than BCS-aftersend protocol. This is due to the property of timestamping function used by MS protocol (See Property

3.4. As a result, choosing timestamping function plays a key role in advancing or retrogressing the recovery line.

2. The same remarks of BCS-aftersend and MS protocols can be drawn when using LazyBCS-aftersend and LazyMS protocols.

3.5.3.2 Does skipping technique reduces the number of forced checkpoint?

To answer this question, let us consider the scenario shown in the Figure 3.11 and compare BCS-aftersend and MS protocols behaviors.

In Figure 3.11(a), we have a system composed by three processes P_1, P_2 and P_3. According to the BCS-aftersend protocol, when P_3 receives the message m_3, it takes a forced checkpoint $C_{3.2}$ before processing such a message (because P_3 has already sent a message m_2 and $m_3.lc(=2 > lc_3(=1)$. P_1 behaves like P_3 when it receives the message m_4.

In Figure 3.11(b), P_3, according to MS protocol, takes a forced checkpoint $C_{3.2}$ before processing the message m_3 because $m_3.sn(=2 > sn_3(=1)$. P_1. Then, when P_3 is about taking the basic checkpoint $C_{3.3}$, it skips taking because $(next_3 = sn_3 = 2)$. The checkpoint $C_{3.3}$ is delayed and is taken after sending the message m_4. In contrast to P_1's behavior shown in Figure 3.11(a), here, when P_1 it receives the message m_4, it does not take a forced checkpoint because the condition is not verified $(m_4.sn = sn_1 = 2)$.

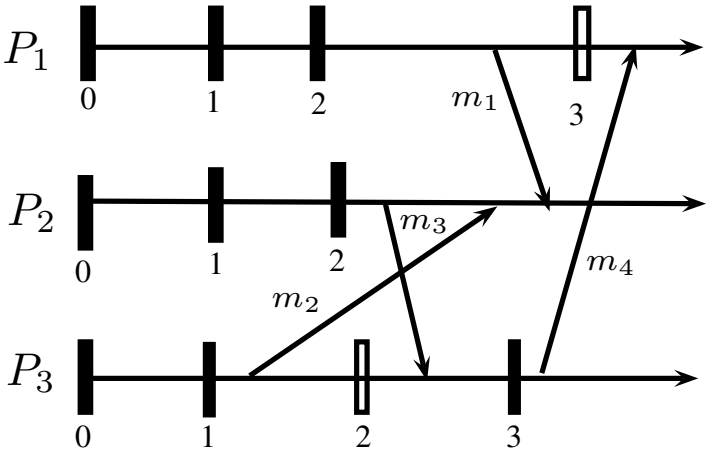
Let us consider again Figure 3.11, we assume that a failure occurs at $P_2's$ level after receiving the message m_1. P_2, in this case, returns to the checkpoint $C_{2,2}$. Consequently, P_1 and P_3, respectively, returns to their checkpoints $C_{1,2}$ and $C_{3,2}$. For the case of BCS-aftersend, two checkpoints are deleted ($C_{1,3}$ and $C_{3.3}$). In contrast, only one checkpoint is deleted for the case of MS protocol.

Important notes

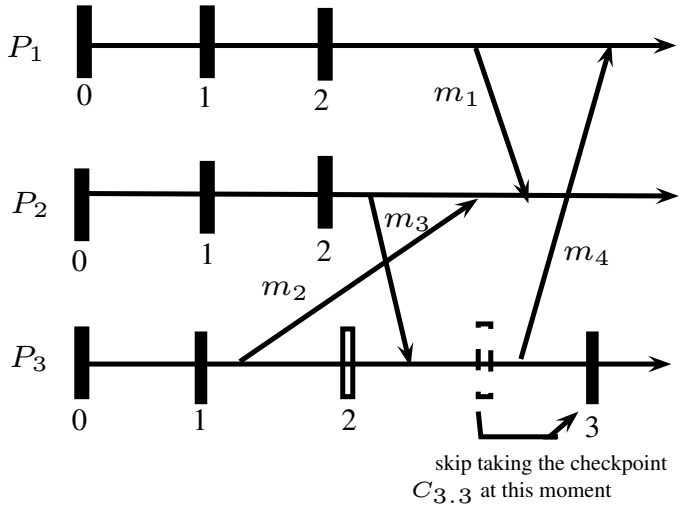
As shown in Figure 3.11, MS protocol, via the skipping technique, reduces the number of forced checkpoint, even that it has a weaker condition than that of BCS-aftersend.

The scenario shown Figure 3.11 is a counterexample to what have been found in the previous studies [61, 70, 71, 130], which postulates that comparing checkpoint induction conditions is enough to decide which protocol can perform better.

Via the skipping technique, not only the number of forced checkpoints is reduced, but also, the number of deleted checkpoints will be reduced.



(a) BCS-aftersend protocol



(b) MS protocol

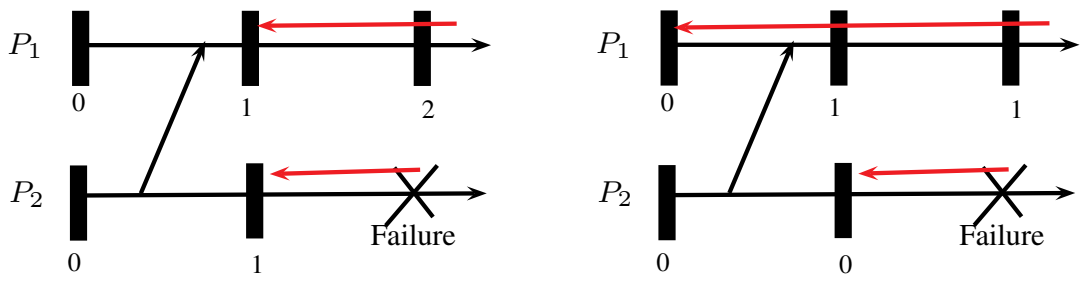
Figure 3.11: Effect of the skipping technique

3.5.3.3 *Impact of lazy indexing on recovery process*

Let us start from the scenario depicted in Figure 3.12.

Figure 3.12(a) shows a system composed by two processes P_1 and P_2. According to BCS protocol, checkpoints are timestamped using classical indexing. When P_2 fails, it resumes its execution from its checkpoint $C_{2.1}$ and sends Rollback(1.1). At receiving such a message, P_1 rolls back to its checkpoint $C_{1.1}$ and deletes its checkpoint $C_{1.2}$.

As shown in Figure 3.12(b), checkpoints are timestamped using lazy indexing. For example, the first two checkpoints of P_2 are timestamped by 0. When P_2 fails, it resumes its execution from its checkpoint $C_{2.1}$ and sends Rollback(1.0). According to the recovery



(a) Recovery using BCS protocol

(b) Recovery using LazyBCS protocol

FIGURE 3.12: Lazy indexing impact

protocol, P_1 resumes its execution from the checkpoint $C_{1.0}$ and deletes the two last checkpoints.

As a result, LazyBCS deletes more checkpoints than BCS protocol. This is due to the lazy indexing, which timestamps more than one checkpoints by the same timestamp (index).

Important Note

Using lazy indexing may lead to redo a great work in case of recovery because more than one checkpoints have the same timestamp.

3.6 SIMULATION STUDY

This section presents the performance evaluation of protocols described in foregoing sections via simulation experiments. For this purpose, we have chosen the Ns2 simulator. The simulation was conducted in for failure-free and failure-prone environments. Under failure-prone environment, we assume that process failures are exponentially distributed with a failure rate λ

3.6.1 Performance metrics

In our evaluation, we use the following metrics: 1) the number of forced checkpoints denoted by F, 2) the ratio of logged messages on the overall messages number denoted by Logged, 3) the number of deleted checkpoints denoted by Deleted, 4) Rollback distance denoted by Distance, which is defined as follows: Distance= $max_{i=1}^{n}|x-y|$ such that x is

the index of the last checkpoint $C_{i,x}$ of P_i and y is the index of the checkpoint $C_{i,y}$ in the recovery line.

3.6.2 Simulation scenarios

Failure free environment

Scenario 1 Under this scenario, the measurements are collected in function of processes number. Processes have the same interval length L which is equal to 40 events. Processes number ranges from 10 to 100 processes.

Scenario 2 In this scenario, we fix the processes number to 80 and calculated the forced checkpoints number in terms of interval length L which ranges from 10 to 50.

Failure prone environment

Scenario 1 This scenario is similar to scenario 1 of failure free environment except that failures occur with a rate equal to 0.1

Scenario 2 This scenario is similar to scenario 2 of failure free environment except that failures occur with a rate equal to 0.1

Scenario 3 In this scenario, we consider a system comprising 80 processes having an interval length L equal to 40 events. The measurements are collected in terms of failure rate ranging from 0.1 to 0.5 failure/s.

Table 3.1 summaries simulation scenarios for failure-prone environment

Table 3.1: Simulation scenarios for failure-prone environment

Scenario	type	nb-proc	Interval length (L)	Interval length (L')	Failure rate λ
Scenario1	Symmetric	10-100	40	-	0.1
	fast process	10-100	40	20	0.1
	slow process	10-100	40	60	0.1
Scenario2	Symmetric	80	40	-	0.1-0.5
	fast process	80	40	20	0.1-0.5
	slow process	80	40	60	0.1-0.5
Scenario3	Symmetric	80	10-50	-	0.1

3.6.3 Failure free environment

Figure 3.13 depicts simulation results for failure-free environment. As shown in Figure 3.13, the number of Forced checkpoints increases as the processes number increases. Moreover, the difference between simulated protocols becomes more obvious when the number of processes is very large. For example, when nb-proc is equal to 100 processes, forced checkpoints number is around 8000 for BCS and LazyBCS protocols and is around 6000 for BCS-aftersend and LazyBCS-aftersend protocols while forced checkpoints number for MS and LazyMS protocols do not exceed 2000 checkpoints. This is due to indexing and skipping strategies used by latter protocols which minimize checkpoints number.

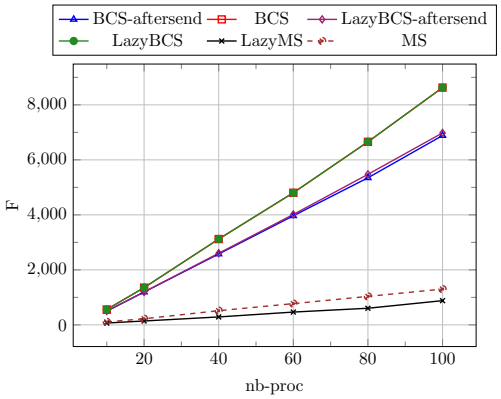


FIGURE 3.13: Number of forced checkpoints vs number of processes

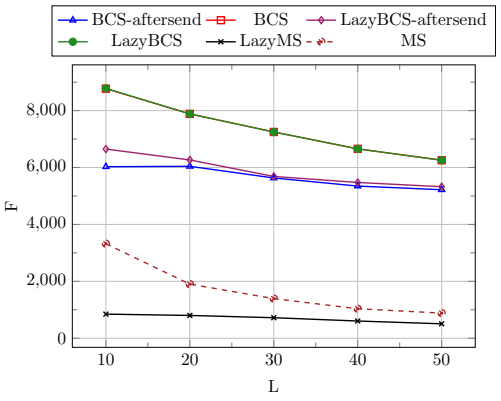


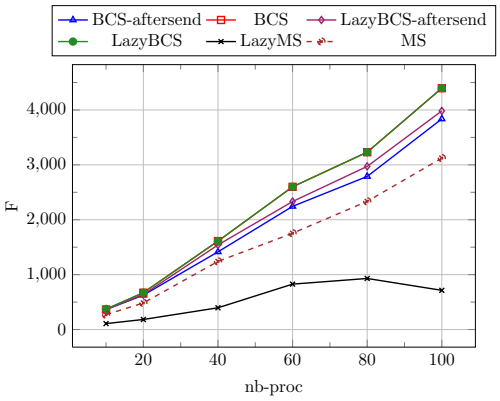
FIGURE 3.14: Number of forced checkpoints vs interval length

Figure 3.14 shows the variation of forced checkpoints number in terms of interval length. At first, we distinguish two classes. i) Class1 containing BCS, LazyBCS, BCS-aftersend and LazyBCS-aftersend, ii) Class3 containing MS and LazyMS protocols. As one can see LazyMS protocol has the lower checkpoints number in comparison to the remaining protocols which decreases slightly. Moreover, there is a great difference between Class1 and Class2 whose forced checkpoint number is less by 6000 checkpoints than Class2. An explanation of simulation results depicted in Figure 3.14 is as follows. Decreasing the forced checkpoints number is due to the decreasing the basic checkpoints number(longer interval means less basic checkpoints number). Recall that forced checkpoints are needed to ensure NZC property while the difference between protocols is due to the different ways in indexes management.

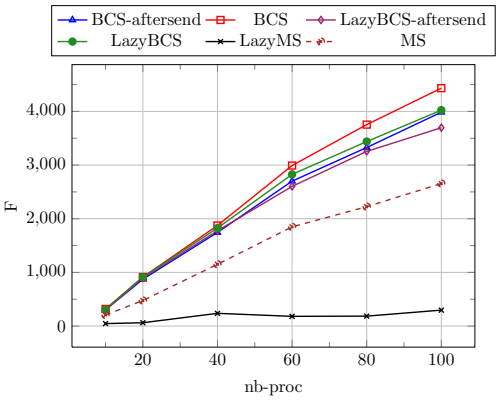
3.6.4 Failure prone environment

3.6.4.1 *Forced checkpoints*

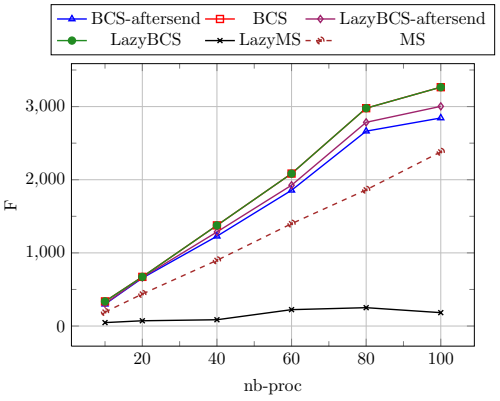
n this section, we discuss on how different protocols fare in the forced checkpoints number. Starting with symmetric case illustrated in Figure 3.15(a), an increase in processes number implies an increase in forced checkpoints number for all protocols. As can be seen, LazyMS protocol does not exceed 1000 forced checkpoints when processes number is equal to 100 processes; it then performs well than other simulated protocols. Additionally, the number of forced checkpoints is less than that of the failure-free environment; this is can be explained as follows: In a case of a failure, processes roll back to a previous state without taking forced checkpoints. Such remarks can be also drawn for the asymmetric cases depicted in Figures 3.15(b) and 3.14(c).



(a) Symmetric



(b) Fast process



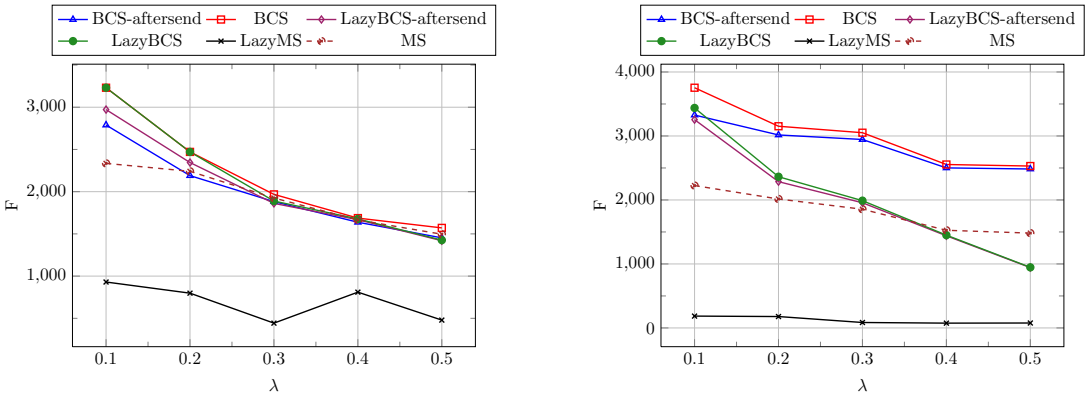
(c) Slow process

Figure 3.14: Number of forced checkpoints vs number of processes

Besides these results, we can clearly see the impact of asymmetry on protocols behavior, especially when there is a slow process whose interval length L' equals to 60 events (Figure 3.14(c)). The calculated number of forced checkpoint is less than that of the symmetric case (Figure 3.15(a)); this can be explained by the fact that slow process influences

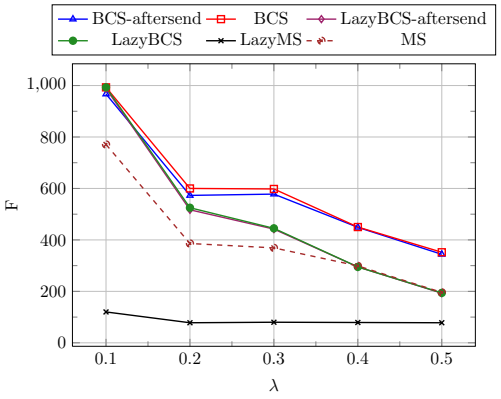
the diminishment of basic checkpoints, which has a direct impact on forced checkpoints number.

On other hands, we can see a slight impact of lazy indexing strategy on protocols performance, which was principally introduced to deal with asymmetry case. As a result, the lazy family does not perform well as expected in comparison to classical one, which means that lazy indexing does not achieve its main goal.



(a) Symmetric case

(b) Fast process case



(c) Slow process case

Figure 3.14: Number of forced checkpoints vs fault rate

Figure 3.14 depicts simulation results in terms of failure rates for symmetric and asymmetric cases. As can be seen, forced checkpoints number decreases as failure rate increases for all protocols. Moreover, forced checkpoints number for the slow process case is less than that of the fast process case which is in turn less than that of the symmetric case. Additionally, LazyMS protocol has the lowest checkpoint number for all cases. The decrease in the forced checkpoint is explained by processes are spending time rolling back to a consistent state than taking forced checkpoints, which means redoing the previous work. As a result, a higher rate of failures implies a slow progress of the system.

To show the impact of interval length on protocols performance, Figure 3.15 is given. As depicted, forced checkpoint number starts decreasing to attain under 3000 checkpoints for almost all protocols, which is due to the decrease in the number of basic checkpoints (large interval length implies less basic checkpoints number).

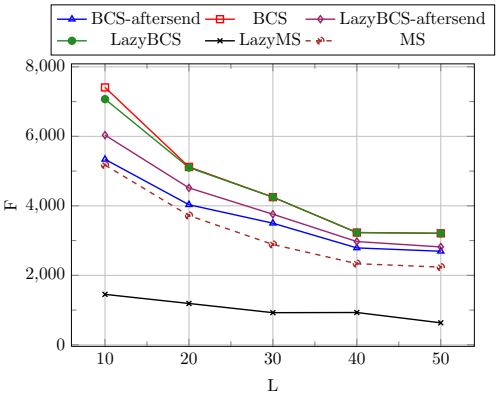


Figure 3.15: Forced checkpoints number vs Interval length

It is worth noting that BCS-aftersend and LazyBCS-aftersend have a stronger checkpoint induction condition than that of MS and LazyMS.However, the latter protocols outperform the former ones. This is due to the way of indexing and skipping techniques used by MS and LazyMS. The skipping technique, indeed, has an impact on reducing basic checkpoints number, which in turn, lowers the number of forced checkpoints taken (For theoretical argumentation, the reader can refer to the section 3.3.1.3). Whereas, having strongest induction condition is not enough to say that the protocol is the best as said in previous works [61, 70, 71, 130].

As concerns the scalability, CIC protocols do not perform well in large distributed systems including a large number of processes (Figure 3.14).

3.6.4.2 Logged messages

Logging messages is an important task in message handling. For this purpose, we measure the ratio of logged messages to the overall messages number with respect to processes number, failure rate, and interval length. Simulation results are shown in Figures 3.16, 3.18 and 3.17.

Firstly, increasing processes number results in an increase in the logged ratio. When processes number equals 100, logged ratio is around 40% for nearly all protocols. Whereas, LazyMS protocol's ratio is around 70%, which is due to the low number of checkpoints taken. Consequently, LazyMS protocol performs worst in this case.

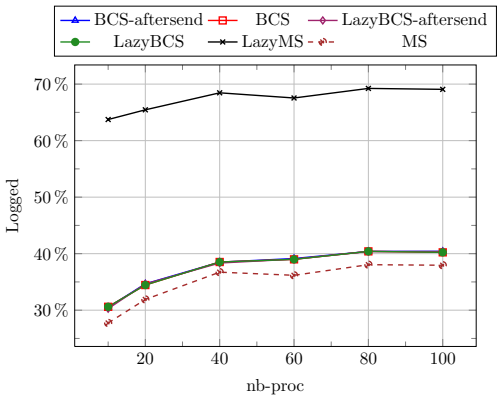


FIGURE 3.16: Ratio of logged messages vs number of processes

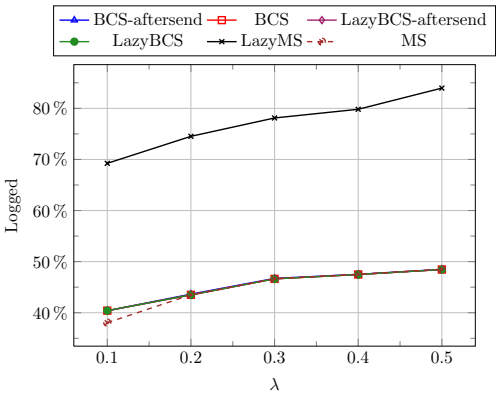


FIGURE 3.17: Ratio of logged messages vs failure rate

In addition, Figure 3.18 depicts the impact of failure rate on logged ratio. We can clearly see that an increase in failure rate causes an increase in logged ratio. An explanation is that processes, under the recovery, try to keep system state consistent, which results in a higher ratio of logged messages.

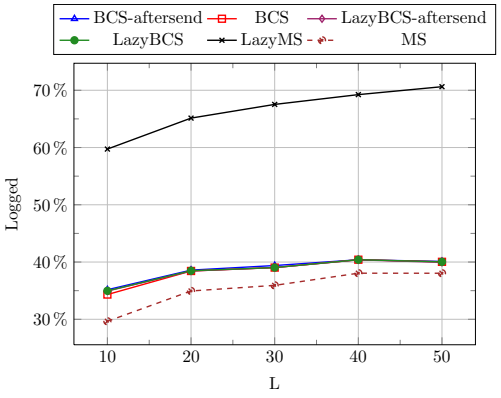


FIGURE 3.18: Ratio of logged messages vs interval length

Figure 3.17 illustrates interval length impact on logged messages ratio. As one can see, the augmentation of interval length implies the augmentation of logged messages ratio. As shown in Figure 3.17, logged ratio start increasing from 30% to 40% for almost all protocols except LazyMS whose ratio reaches 70%, and MS protocol whose ration is less than 40%. This can be explained by the fact that longer interval means less checkpoint number, which in turn implies high rate of logged messages

In the light of these results, we can see a tradeoff between checkpoints number and logged messages ratio, a protocol that takes less number of checkpoints logs a large number of messages.

Logging messages consumes storage space of process especially when using LazyMS protocol, which means that garbage collection protocol must be invoked in order to manage storage space.

3.6.4.3 *Deleted checkpoints during recovery*

Deleted checkpoints number during the recovery process gives us an idea about the amount of work to undo due to the failure occurrence.

Figure 3.19 shows the variation of deleted checkpoints number with respect to processes number. As expected, increasing processes number causes the increase in deleted checkpoints number. When processes number equals 100, deleted checkpoints number, for BCS, LazyBCS, BCS-aftersend and LazyBCS-aftersend protocols, exceeds 25 checkpoints. Whereas for MS and LazyMS protocols, deleted checkpoints number does not exceed 25 and 20 respectively. As a result, MS and LazyMS outperform other simulated protocols.

Additionally, Figure 3.21 depicts the variation of deleted checkpoints number with the variation of failure rate. As shown, increasing failure rate results in decreasing the deleted checkpoints number. This can be explained by the fact with a higher failure rate, processes take less checkpoint, which means less deleted checkpoints. Another reason lies in the fact that more than one failure may occur in the same checkpoint interval implying the rollback to the same recovery line.

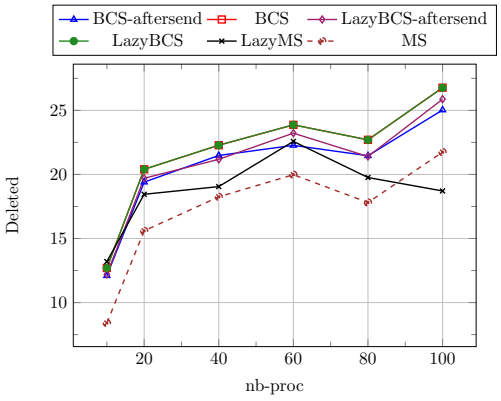


Figure 3.19: Number of deleted checkpoints vs number of processes

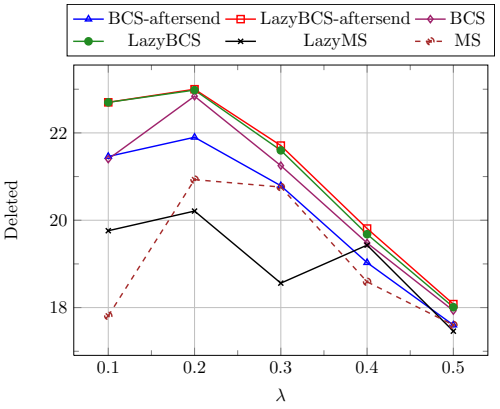


Figure 3.20: Number of deleted checkpoints vs failure rate

Under scenario 3, we measure deleted checkpoints number in terms of interval length. As illustrated in Figure 3.20, deleted checkpoints number decreases as interval length increases. The slope of the reduction in deleted checkpoints number for MS and LazyMS

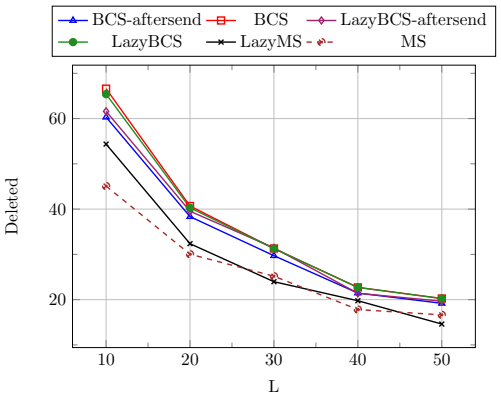


FIGURE 3.21: Number of deleted checkpoints vs interval length

protocols is not as large as remaining protocols, which is due to their indexing and skipping strategies.

As shown in Figures 3.19, 3.21 and 3.20, the protocols using the lazy indexing deletes larger number of checkpoints than protocols using the classical indexing. This is due to the fact that more than one checkpoints are timestamped by the same index and hence will be all deleted in the case of recovery (For theoretical argumentation, refer to the section 3.5.3.3).

Once again, even that BCS-aftersend and LazyBCS-aftersend have strongest induction conditions but these protocol do not perform better than MS and LazyMS protocols.

3.6.4.4 Rollback distance

An important metric in our evaluation is the rollback distance which measure the rollback propagation and its effect on protocols performance.

From Figure 3.22, 3.23 and 3.24, we can see that rollback distance for BCS, BCS-aftersend, LazyBCS and LazyBCS-aftersend varies between 2 and 4, whereas LazyMS has rollback distance equal to 2 for almost the cases, and MS protocol rollback distance equals 1. Our simulation results confirm the conjuncture made by MS protocol 's authors [13], which states that MS protocol is 1-rollback. In addition, we can say that BCS-aftersend, LazyBCS, LazyBCS-aftersend and LazyMS are also K-rollback. From these results, we can draw the following remarks: i) For MS protocol, garbage collection protocol can be invoked asynchronously whenever a process takes a new checkpoint by only keeping the two newest checkpoints; ii) For LazyMS, we can only keep three checkpoints while for other protocols, we keep more than three checkpoints.

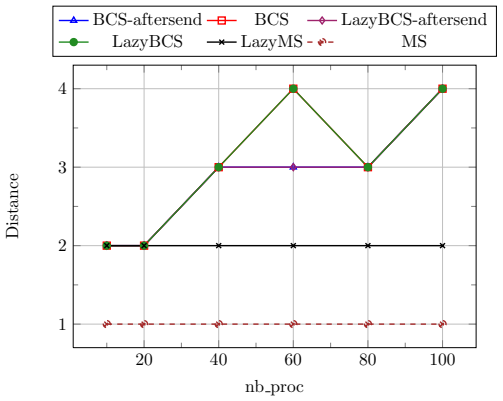


Figure 3.22: Rollback distance vs number of processes

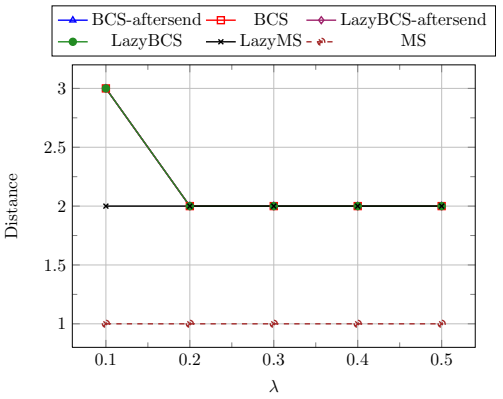


Figure 3.23: Rollback distance vs failure rate

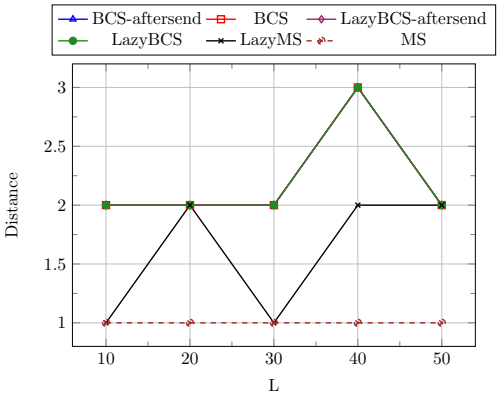


Figure 3.24: Rollback distance vs interval length

It is worth noting that LazyMS does not preserve the 1-rollback property of MS protocol due to the use of lazy indexing strategy (this can be explained by the number of deleted checkpoints by LazyMS protocol).

3.7 Conclusion

Through this chapter, we have shown a characterization of NZC property based on timestamping(indexing) technique. Via timestamping the communication can be seen causal, which facilitates designing checkpointing protocols. Timestamping can be done by either classical or lazy way. In classical timestamping, checkpoint logical clock (sequence number) is incremented each time a checkpoint was taken. While in lazy timestamping, checkpoint logical clock is not increased if the process does not receive new information.

Then, we have described protocols belonging to those using classical indexing and those using lazy indexing. Subsequently, we have shown how Index based NZC protocols are extended with BRA protocol. The choice of BRA protocol is justified by its salient features such as i) The application is not blocked during recovery; ii) Processes are asynchronous during the recovery, which means that each process decides when it must rollback or continue its execution; iii) It has a complete procedure of message handling.

We have also conducted excessive simulation experiments in order to study CIC protocols behavior in failure-prone environment. Such a simulation allows us to draw the following remarks: Index based CIC checkpointing protocols worsen their performance in large distributed systems, which means that these protocols do not scale well. Additionally, having a stronger induction condition is not sufficient to say that protocols can perform better than those having weaker conditions.

In addition, There is a trade-off between checkpoints number and the logged messages ratio. A large number of checkpoints makes the recovery simple, but it consumes storage space. However, logging messages consumes less storage space, but it makes the recovery more complex.

Besides these remarks, using lazy indexing may lengthen the rollback recovery time and enlarge the amount of the work to redo in the case of a failure.

Finally, knowing the rollback propagation bounds facilitate the implementation of the garbage collection during failure-free execution rather than during the recovery, thereby, avoiding the stable storage contention.

Chapter 4

New RDT protocols with constant size control information on messages

Contents

4.1 Introduction

A Z-path is a sequence of messages that establish dependency relations between checkpoints. It may be causal or non-causal. Non-causal Z-paths are undesirable in checkpoint and communication patterns because it creates hidden dependencies, which may results in global checkpoint inconsistency. If non-causal Z-paths are prevented from being formed or are causally doubled, we said that the checkpoint and communication pattern satisfy the rollback-dependency trackability property.

Since the introduction of Rollback dependency trackability property by Wang [18], many proposed RDT protocols in the literature try to give a minimal characterization in order to reduce the number of forced checkpoints. However, such protocols suffer from piggybacking extra control information, which depends on processes number [19] [18][20][21]. At this regard, this chapter presents two new RDT protocols that piggyback only a constant size of control information, which is independent of the number of processes. Thereby, lowering the overhead of RDT protocols.

First of all, this chapter presents RDT characterizations: i) based on Z-paths and ii) based on transitive dependency vectors. Then, a classification of RDT protocols followed by examples of some RDT protocols are given. After that, this chapter describes the direct dependency clocks and its utilization by our new RDT protocols. Finally, the present chapter ends by a comparison that studies the efficiency of the new RDT protocols.

4.2 RDT Property Characterization

RDT property stipulates that there are no hidden dependencies between checkpoints. Such property is characterized by transitive dependency vectors or based on Z-paths.

4.2.1 Transitive dependency vectors based characterization of RDT property

Wang was the first to define the RDT property. He has introduced a characterization of such property based on transitive dependency vectors. The following section describes such vectors and how it was used to track.dependencies.

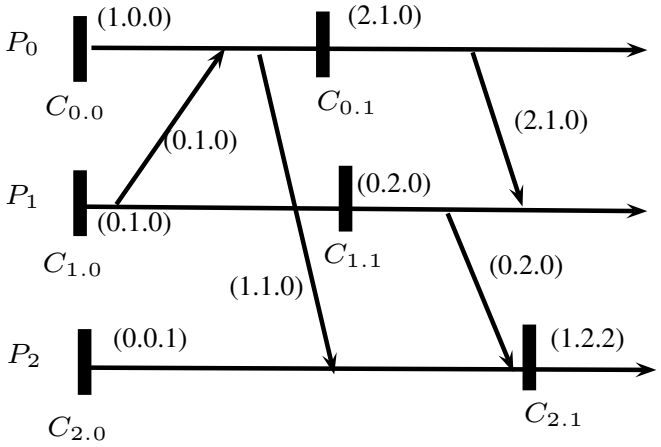


Figure 4.1: Timestamping with transitive dependency vectors

4.2.1.1 *Transitive dependency vectors*

We use the transitive dependency vectors (DV) mechanism to track causal dependencies between checkpoints. Each process P_i maintains DV_i vector of size N (where N is the number of processes), such that $\forall i\ DV_i[i]$ is set to '0'. The component $DV_i[i]$ represents the current checkpoint interval of P_i and incremented immediately after any checkpoint, Other components $DV_i[j](j \neq i)$ represents the knowledge of the process P_i on the existence of checkpoints on other processes. This vector is modified whenever a message m carrying new information is received by P_i $(\forall j DV_i[j] = max(m.DV[j], DV_i[j]))$.

Definition 4.1

The distributed execution $(\hat{H}, C)$ ensures the RDT property if all dependencies between checkpoints can be tracked online by using transitive dependency vectors $\forall C_{i,x}, C_{j,y} : C_{i,x} \xrightarrow{Z} C_{j,y} \Leftrightarrow DV_{j,y}[i] \geq DV_{i,x}[i]$*[18]*

Figure 4.1 shows a distributed execution with checkpoints enriched by DV vectors. At the beginning of the execution, processes P_0, P_1 and P_2 have, respectively, the following vectors (1.0.0), (0.1.0) and (0.0.1). $C_{2,1}$ associated vector reflects the existence of a Z-path from $C_{1.0}$ to $C_{2.1}$, but it does not reflect the existence of a Z-path from $C_{0.1}$ to $C_{2.1}$ because $DV_{2.1}[0] < DV_{0.1}[0]$. This example shows that not all dependencies are tracked online. Hence, the distributed execution shown in Figure 4.1 does not satisfy the RDT property. If an additional checkpoint is taken before processing the message coming from P_0 to P_1, then the execution satisfies the RDT property.

4.2.2 Z-paths based characterization of RDT property

This section presents a characterization of RDT property based on Z-paths, which has been introduced by Baldoni et al. [19].

Definition 4.2

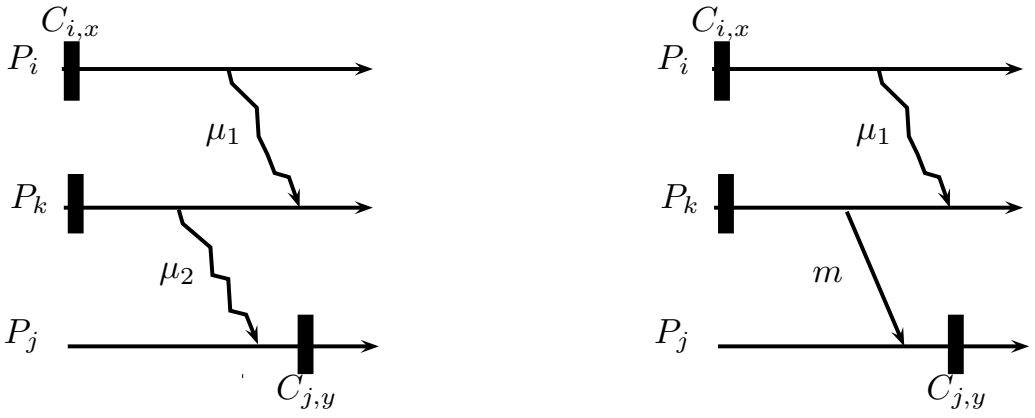
A distributed execution satisfied RDT property if and only if all Z-paths are causally doubled $\forall C_{i,x}, C_{j,y}: C_{i,x} \xrightarrow{Z} C_{j,y} \rightarrow C_{i,x} \xrightarrow{C} C_{j,y}$

4.2.2.1 PCM characterization

Given a distributed execution with checkpoints, it is unnecessary to check that all non-causal Z-paths are causally doubled to say that it ensures RDT property. Doubling subsets of non-causal Z-paths may be enough, such a subset is called characterization of RDT property.

a) Z-path of order 2 (CC-path) (Causal-Causal-path)

Each Z-path is a concatenation of causal paths, the order of Z-path is the number of Z-causal paths that form [41]



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

FIGURE 4.2: CC-path and CM-path

- A Z-path of order 2 (CC-path) if it composed by two Z-paths μ_1.and μ_2 (denoted by$\mu_1.\mu_2$. See Figure 4.2(a)).
- A Z-path of order 2 ($\mu_1.\mu_2$) such that μ_2 consists of only one message is called CM-path (causal-message-path).

b) PCM-path (Prime-Causal-Message path)

Given a set of all Z-paths from $I_{j,y}$ to $I_{i,x}$ denoted by CL(i,x).(j,y). This set is totally ordered by the following $(\mu \leq \mu') \Leftrightarrow (receive(\mu.last) \xrightarrow{hb} receive(\mu.last))$.

Definition 4.3 (*Prime Path*)

A Z-path μ from $I_{i,x}$ to $I_{j,y}$ is prime if μ is the minimum of the set CL(i,x).(j,y).

In other words, a Z-path μ is said prime from $I_{i,x}$ to $I_{j,y}$ if every Z-path μ' verifies the following relation $\mu \leq \mu'$. Intuitively, μ brings new information about P_i checkpoint to process P_j.

Definition 4.4 (*PCM-path*)

A PCM-path $\mu.\{m\}$ is CM-path whose μ is prime (see Figure 4.2(b)).

Theorem 4.1

A distributed execution satisfies RDT property if and only if all PCM-paths are causally doubled [41]

> ***Proof:*** Due proof length, the proof is omitted. The reader may consult [41]. ■

4.2.2.2 *Optimal Characterization of RDT Property*

The set of all EPSCM-paths [41] is the small subset of Z-paths, if this set is causally doubled, then, the distributed execution with checkpoints ensures the RDT property,

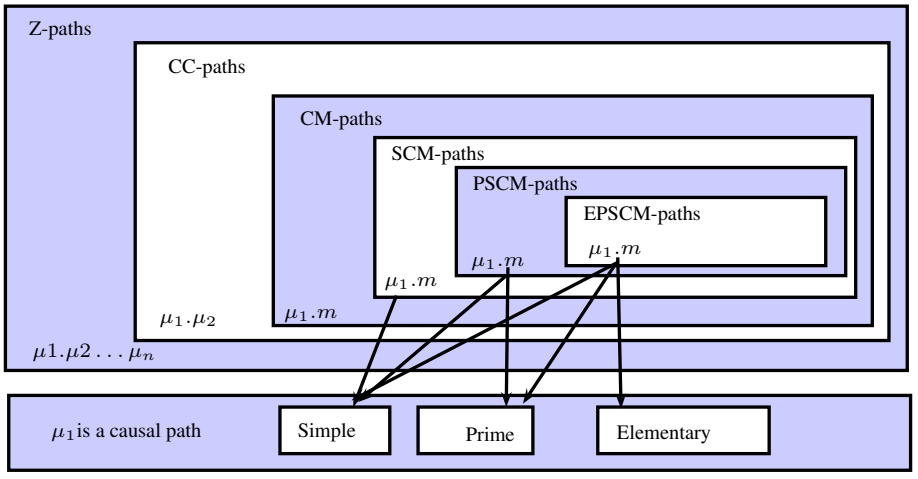
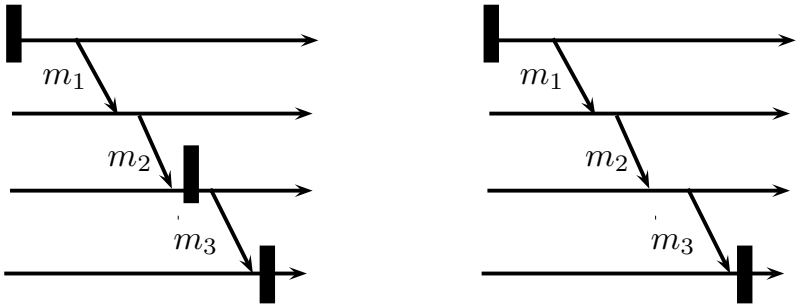


Figure 4.3: Subsets of Z-paths [41]

a) EPSCM-path (Elementary-Prime-Simple-Causal-Message path)



(a) Non-simple path (b) Simple path

FIGURE 4.4: Simple and non-simple paths

Definition 4.5 (*Simple Path*)

A Z-path $\mu = \{m_1, m_2, \ldots, m_q\}$ is said simple if the two events receive(m_k) and send(m_{k+1}) (in this order) occurred in the same interval.

Figure 4.4(a) represents a non-simple path while Figure 4.4(b) represents a simple path.

Definition 4.6 (*Elementary Path*)

Consider the processes sequence $\{P_1, P_2, \ldots, P_n\}$ crossed by a Z-path μ, we say that μ is elementary if the sequence $\{P_1, P_2, \ldots, P_n\}$ does not contain repetitions.

Definition 4.7 (*EPSCM-path*)

A EPSCM-path $\mu.\{m\}$ is PCM-path whose μ is elementary and simple.

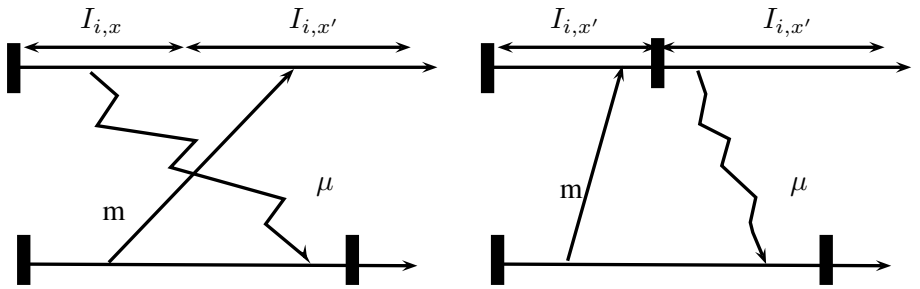
b) EPSCM-cycle

The EPSCM-path $\xi = \mu.m$ forms a cycle if the process that sends $\mu.first$ and the process that receives m are the same

Consider the EPSCM-path formed from $I_{i,x}$ to $I_{i,x}$. We have two cases :

- $x < x'$**:** In this cases $\xi = \mu.m$ is causally doubled by its self (Figure 4.5(a)).
- $x > x'$**:** In this cases $\xi = \mu.m$ forms a Z-cycle that can never be causally doubled (Figure 4.5(b)).

Likewise, A PCM-Cycle formed from $I_{i,x}$ to $I_{i,x}$ such that $x > x'$ can never be causally doubled.



(a) $x < x'$ (b) $x' < x$

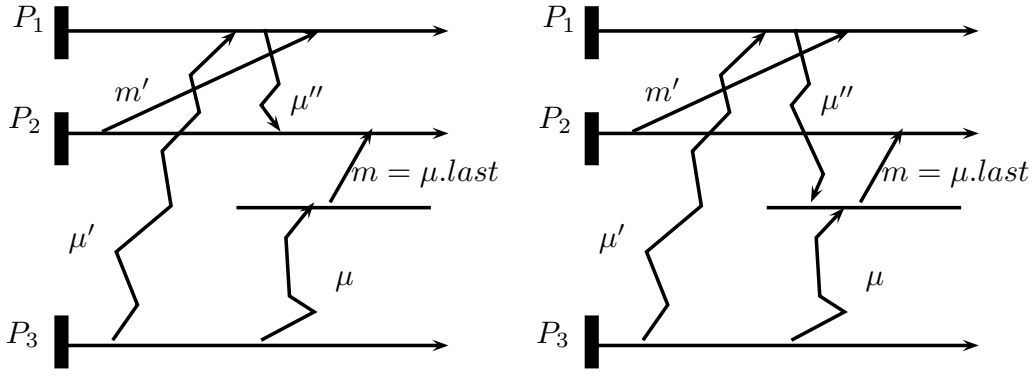
Figure 4.5: EPSCM-cycle

Theorem 4.2 (*Minimal RDT property characterization*)

A distributed execution satisfies RDT property if and only if all EPSCM-paths are causally doubled [41].

4.2.2.3 *Visibly doubled Property*

According to Theorem 4.1, to satisfy RDT property, each non-causally doubled EPSCM path must be prevented from being formed by taking forced checkpoints, as this information may be unavailable in the causal past of the process when P_i detects a EPSCM-path. The online decision of a protocol can be only based on the notion of visibly doubling.



(a) μ is not prime due to the Z-path $\mu'.\mu''$ (b) $receive(\mu.last) \xrightarrow{hb} send(\mu.last)$ (via μ'')

Figure 4.6: Visibly doubled PCM-path

Definition 4.8

Consider the EPSCM-path $\xi = \mu.m$ in $(\hat{H},C)$, ξ is visibly doubled by μ' if and only if:

1. *ξ is causally doubled by a Z-path μ'*
2. *$receive(\mu''.last) \xrightarrow{hb} send(\mu.last)$*

In Figure 4.6(a), $\mu.m'$ is causally doubled by μ', but μ is not prime due to $\mu'.\mu''$ (i.e.$receive(\mu'.last) \xrightarrow{hb} receive(\mu.last)$), which means that μ is not visibly doubled. Figure 4.6(b), P_2 learns that μ is causally doubled by μ' via the information brought by the path $\mu''.\mu.last$. As a result, we said that $\mu.m'$ is visibly doubled by μ'. The availability of such information makes the decision about taking forced checkpoint easier.

The following theorem states the characterization of RDT property by an online protocol

Theorem 4.3

A distributed execution with checkpoints generated by an online protocol ensures RDT property if and only if all EPSCM-paths are visibly doubled [41].

> ***Proof:*** Due proof length, the proof is omitted. The reader may consult [41]. ■

4.2.2.4 PMM characterization

Recently, a new characterization of the property called PMM-path (Prime-Message-Message) was proposed by Garcia et Buzato [131], which is a refinement of the previous characterization.

The PMM-path is defined as follows:

Definition 4.9 (*PMM Path*)

A PMM-path is a Z-path composed by only two messages {m,m'}such that

- *Message m is prime.*
- $send(m') \xrightarrow{hb} receive(m)$ *({m,m'} forms a Z-pattern).*

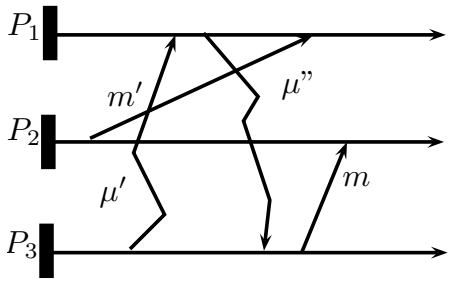


FIGURE 4.7: PMM-path m.m' is visibly doubled by $\mu'.\mu$"

In Figure 4.7, the PMM-path $m.m'$ is visibly doubled by the causal path μ' and $receive(\mu'.last) \xrightarrow{hb} send(m)$ due to the causal path μ".

Theorem 4.4

A distributed execution with checkpoints generated by a online protocol ensures RDT property if and only if all PMM-paths are visibly doubled [21].

4.3 Related work: RDT protocols

This section gives a classification of RDT protocols and some example of such protocols.

4.3.1 RDT protocols classification

Tsai [68] has proposed a classification of RDT protocols based on the characterization used as follows: i) PCM protocols, ii) EPSCM protocols and iii) PMM protocols.

4.3.1.1 PCM protocols

The most known protocol of this class is due to Baldoni et al. [19] and called **BHMR**, BHMR protocol prevents non visibly doubled PCM-paths and non-causally doubled PCM cycles from being formed. It is called also **NNDV-PCM (No-Non-Visibly-doubled)**. From this protocol, other protocols can be derived based on PCM-paths. The first variant is **No-PCM-Paths** [41], which prevents all PCM-paths formation. The second one is **No-PCM-Cycle** [41]. It inhibits the formation of all non visibly doubled PCM-paths and all PCM-cycles. The last one is called **No-PCM** [18] known as FDAS protocol, it avoids the formation of all **PCM-paths**.

4.3.1.2 EPSCM protocols

The proposed protocol in [41] prevents all non-visibly doubled **EPSCM-paths** and all non-causally doubled **EPSCM-cycles** to be occurring, It also called NNVDEPSCM (No-Non-Visibly-Doubled EPSCM) [41]. Similarly, many variants can be derived from this protocol. The first one is called **No-EPSCM-Path**, it avoids the formation of all **EPSCM-Paths** and all non causally doubled **EPSCM-Cycles**. The second one is called No-EPSCM-Cycle that prevents the formation of all **EPSCM-Cycles** and all non-visibly

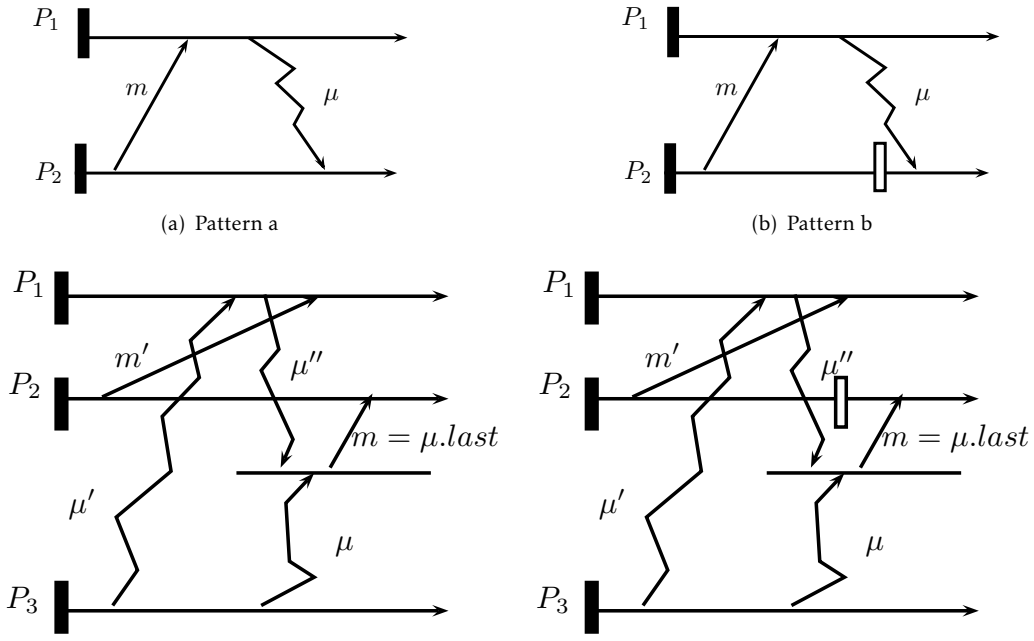


FIGURE 4.8: NNDV-EPSCM, No-EPSCM-Path, No-EPSCM and No-EPSCM-Cycle

doubled **EPSCM-paths**. The last one is called **No-EPSCM** that does not allow any **EPSCM-Path** or **EPSCM-Cycle** to be established.

In Figure 4.8, we can see that

Pattern a under this pattern, NNVD-EPSCM and No-EPSCM-path protocols do not induce a forced checkpoint before processing the message m.

Pattern b No-EPSCM and No-EPSCM-Cycle disallow the formation of the cycle $\mu.m$ by taking a forced checkpoint before processing the message m.

Pattern c $\mu.m$ is visibly doubled by the causal path $\mu'.\mu''$. In this case, NNVD-EPSCM and No-EPSCM-Cycle protocols do not need to take extra checkpoints to make this pattern satisfying RDT property.

Pattern d No-EPSCM and No-EPSCM-path protocols force P_2 to take a forced checkpoint before processing the message m even that $\mu.m$ is visibly doubled by the causal path. $\mu'.\mu''$.

4.3.1.3 PMM protocols

Garcia et al have proposed a protocol called RDT-Minimal [21] that avoids the non-visibly doubled PMM-paths and **non-doubled PMM** cycles from being formed. it is also called **NNVD-PMM**. RDT-Minimal protocol can have many variants. RDT-Partner is

one from its variants, it does not allow the formation of PMM-paths and non causally doubled PMM-Cycles. RDT-Partner is also called **No-PMM-Cycle**[68]. Another variant called **No-PMM** prevents all **PMM-paths** and **PMM-cycles** formation. In Figure 4.9, we

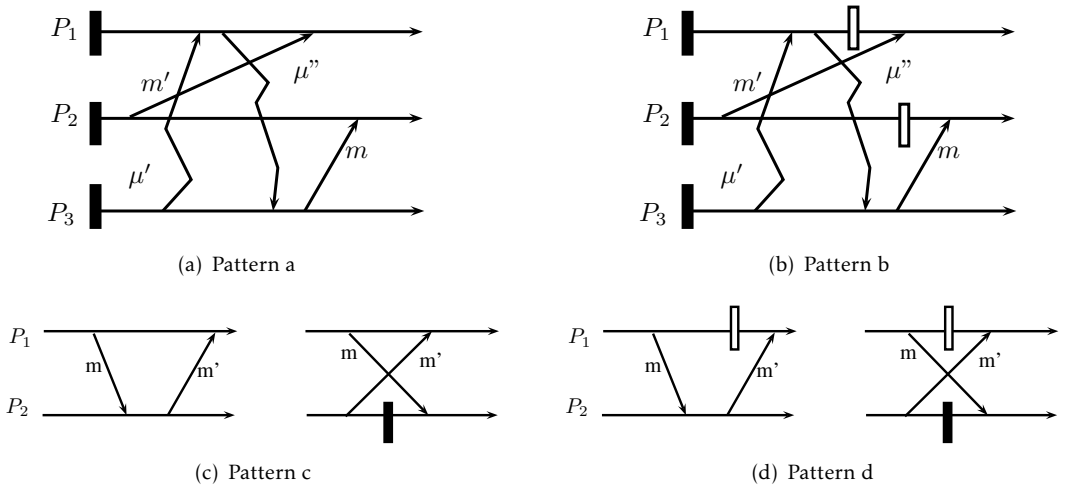


Figure 4.9: NNDV-PMM, No-PMM-Cycle, No-PMM-Path and No-PMM

can see that

Pattern a NNVD-PMM and No-PMM-Cycle protocols do not force P_1 or P_2 to take extra checkpoint before processing the message m or m' because the path $m.m'$ is visibly doubled by $\mu.\mu'$.

Pattern b this pattern describes No-PMM and No-PMM-path behavior. On receiving m and m', processes P_1 and P_2 are forced to take extra checkpoints before processing such messages.

Pattern c $m.m'$ is doubled by it self, this why NNVD-PMM and No-PMM-path protocols do not induce extra checkpoints.

Pattern d In contrary to NNVD-PMM and No-PMM-path protocols, No-PMM and No-PMM-cycle forces P_2 to take forced checkpoint before processing the message m'.

4.3.2 Example of RDT protocols

4.3.2.1 FDAS Protocol

Data structures

Every process P_i has the following data structures.

$Sent_i$: a boolean initialized to false. When a process P_i sends a message m, it sets $Sent_i$ to true.

DV_i : An integer vector initialized to "0" except $DV_i[i]$ which is initialized to "1".

Description of the protocol

Take – Basic – Checkpoint: When a process P_i decides to take a basic checkpoint, it increments its DV component ($DV_i[i] = DV_i[i]+1$) and saves its state with a copy of its DV vector.

Send – Message: the process P_i sets $Sent_i$ to true and piggybacks its vector on the message.

Receive – Message: When a process P_i receives a message m, it takes a forced checkpoint if it sends a message in this current interval and the message carried new information. Then, the process P_i updates its DV vector ($\forall k, DV_i[k] = max(DV_i[k].mDV_i[k])$).

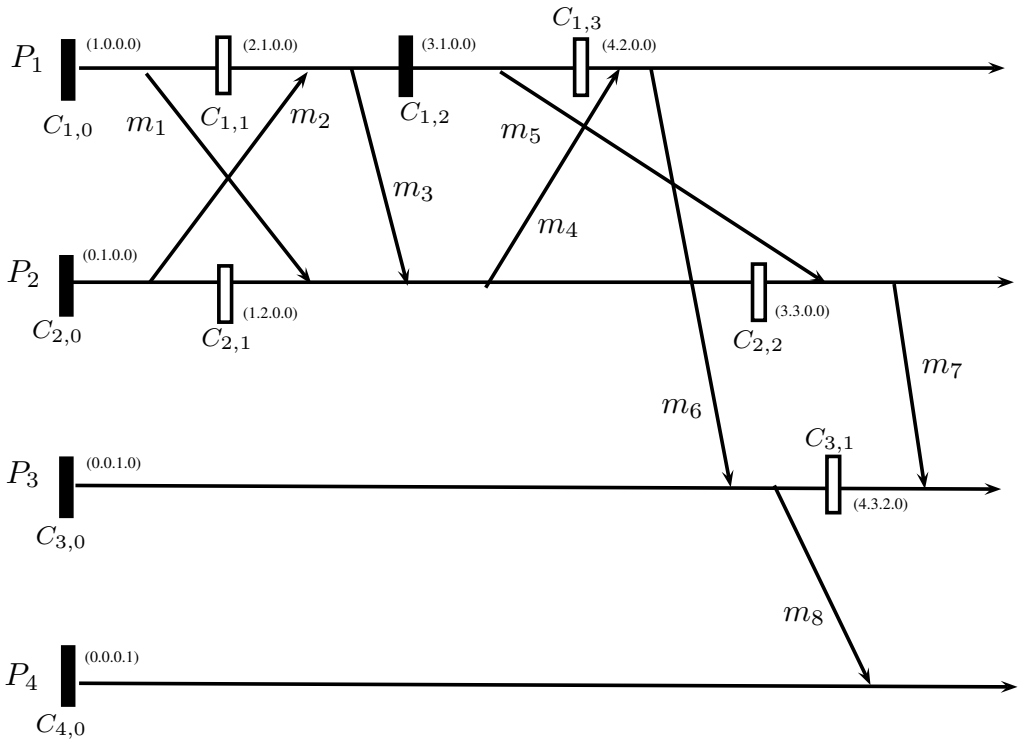


FIGURE 4.10: FDAS protocol

Figure 4.10 depicts an example of FDAS execution. The system composed by four processes P_1, P_2, P_3 and P_4. When P_1 sends the message to P_2, it piggybacks its DV vector on it. On receiving the message m_1 by P_2, P_2 checks for new information carried by m_1, then it takes a forced checkpoint because it has already sent a message m_2 to P_1

($sent_2 = true \wedge m_2.DV[1] > DV_2[1]$). The same actions will be taken by other process on receiving m_1,m_4, m_5 and m_7.

4.3.2.2 BHMR protocol

Data structures

$Sent-to_i$: A boolean vector initialized to false.

DV_i : An Integer vector (similar to that of FDAS protocol.

$Causal_i$: A boolean matrix initialized to false except its diagonal. $Causal_i[k,j]$ means that a there exists a causal path from P_k to P_j

$Simple_i$: A boolean vector initialized to false except $Simple_i[i]$. It is used to detect the formation of Z-Cycles.

Description of the protocol

$Take-Basic-Checkpoint$: When a process P_i decides to take a basic checkpoint, it rests its Sent-to and Simple vectors and its Causal matrix. then, it increments its DV entry and saves its state with a copy of its DV vector.

$Send-Message$: When P_i decides to send a message m to process P_j, it sets $Sent-to_i[j]$ to true and sends the message m with a copy of DV and Simple vectors, and Causal matrix.

$Receive-Message$: Upon receiving a message m by P_i, P_i takes a forced checkpoints in the following two cases:

1. P_i has already sent a message m' to a process P_j ($Sent-to_i[j]$), and the path μ ended by the message m (m= $\mu.last$) is prime path ($\exists k\ DV[k] > DV_i[k]$), and $\mu.m'$ is not causally doubled ($\neg Causal[k,j]$). Hence, the first part of induction condition is as follows:
$$(\exists j,\ Sent-to_i[j] \wedge \exists k\ DV_i[k] < m.DV[k] \wedge \neg Causal[k,j])$$
2. The Z-path $\mu.m$ may form a Z-cycle, which is detected by using the Simple vector. The second part of induction condition is as follows:
$$(DV_i[i] = m.DV_i[i] \wedge m.Simple[i]).$$

Finally P_i updates its data structures.

Note

1. When BHMR condition is reduced to only the following condition $\exists j, Sent-to_i[j] \wedge \exists k DV_i[k] < m.DV[k] \wedge \neg Causal[k,j])$, it is called No-PCM-cycle

2. When BHMR condition is reduced to only the following condition $Sent - to_i[j] \wedge \exists k DV_i[k] < m.DV[k] \wedge) or (DV_i[i] = m.DV_i[i] \wedge m.Simple[i])$, it is called No-PCM-path.
3. When BHMR condition is reduced to only the following condition $\exists j, Sent - to_i[j] \wedge \exists k DV_i[k] < m.DV[k]$, it is called No-PCM which is simply FDAS protocol.

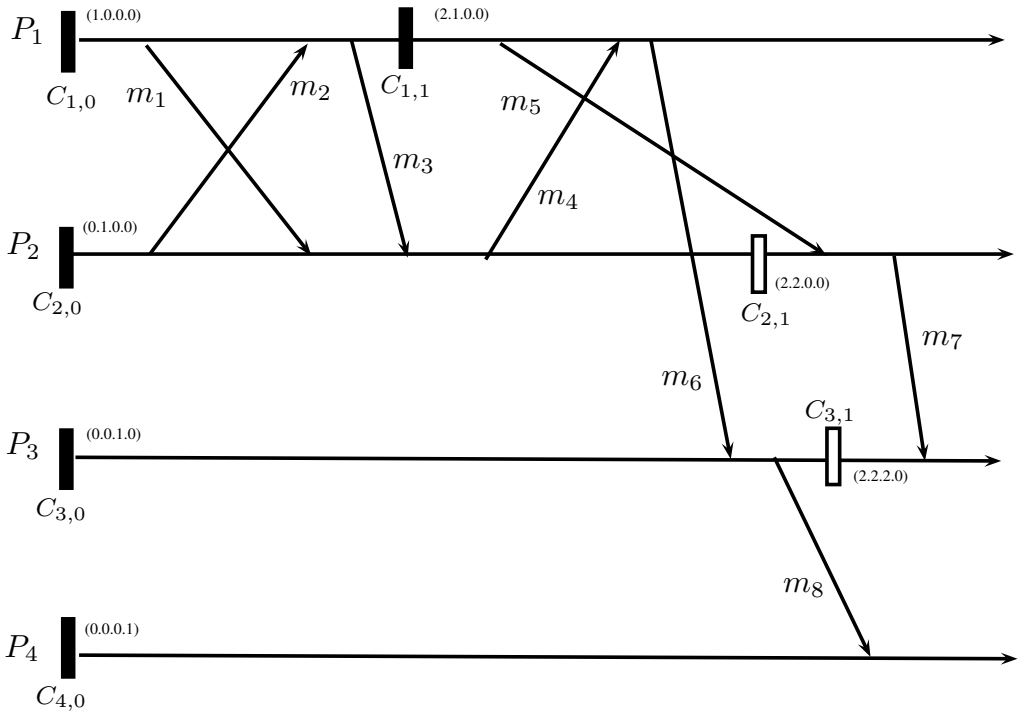


FIGURE 4.11: BHMR protocol

Figure 4.11 shows an example of BHMR execution for the same communication pattern presented in Figure 4.10. When P_2 receives the messages m_5, it detects that the checkpoint $C_{1.1}$ is involved in a Z-Cycle ($sent\_to_2[1] = true \wedge m_5.DV[2] = DV_2[2]$), then it takes a forced checkpoint before processing a such message. Likewise, P_3 takes a forced checkpoint before processing the message m_7 because it has detected that the path ending by the message m_7 is not visibly doubled ($sent\_to_3[4] = true \wedge m_5.DV[2] > DV_3[2] \wedge \neg m.Causl[2.4]$).

4.3.2.3 RDTPartner protocol

Data structures

DV_i : An Integer vector has the same role as in FDAS protocol.

$Simple_i$: A boolean vector initialized to false except $Simple_i[i]$. it is used to detect the formation of Z-Cycles.

Partner: An integer variable

Partner = No-Partner =-1 means that P_i does not send a message.

Partner = j means that P_i has sent a message to P_j.

Partner = More-Than-One-Partner = N+1 means that P_i has sent messages to more than one process.

Message: contains the following fields.

sender: Sender Identity.

receiver : Receiver Identity.

DV: An integer vector.

Simple: a boolean that indicates if there is a simple path between the sender and the receiver.

Description of the protocol

$Take-Basic-Checkpoint$: When a process P_i decides to take a basic checkpoint, it increments its DV entry and saves its state with a copy of it DV vector, it rests its Simple vectors and Partner to No-Partner.

$Send-Message$: Upon sending a message m by a process P_i from a process P_j, P_i piggybacks its DV vector and only $Simple_i[j]$. then it sets Partner to j if it has sent a message to only P_j else it sets Partner to More-Than-One-Partner.

$Receive-Message$: When P_i receives a message m, it takes a forced checkpoints before processing the message in the following two cases:

1. P_i has already sent a message to a process different from Partner or P_i has more than one partner.
2. If the path ended by m is not simple.

Finally P_i updates its data structures.

Similarly to Figures 4.10 and 4.11, Figure 4.12 presents an execution of RDTPartner. As one can see, RDTPartner protocol induces the same number of forced checkpoints as BHMR protocol. P_2 on receiving the message m_5, it detects and breaks the Z-Cycle m_5,m_2 ($partner = 2 \wedge m.DV[2] > DV_2[2]$) involving the checkpoint $C_{1.1}$ by taking the forced checkpoint $C_{2.1}$. Then, at receiving the message m_7, P_3 takes a forced checkpoint before processing it($partner \neq 2 \wedge DV_3[2] < m_7.DV[2]$).

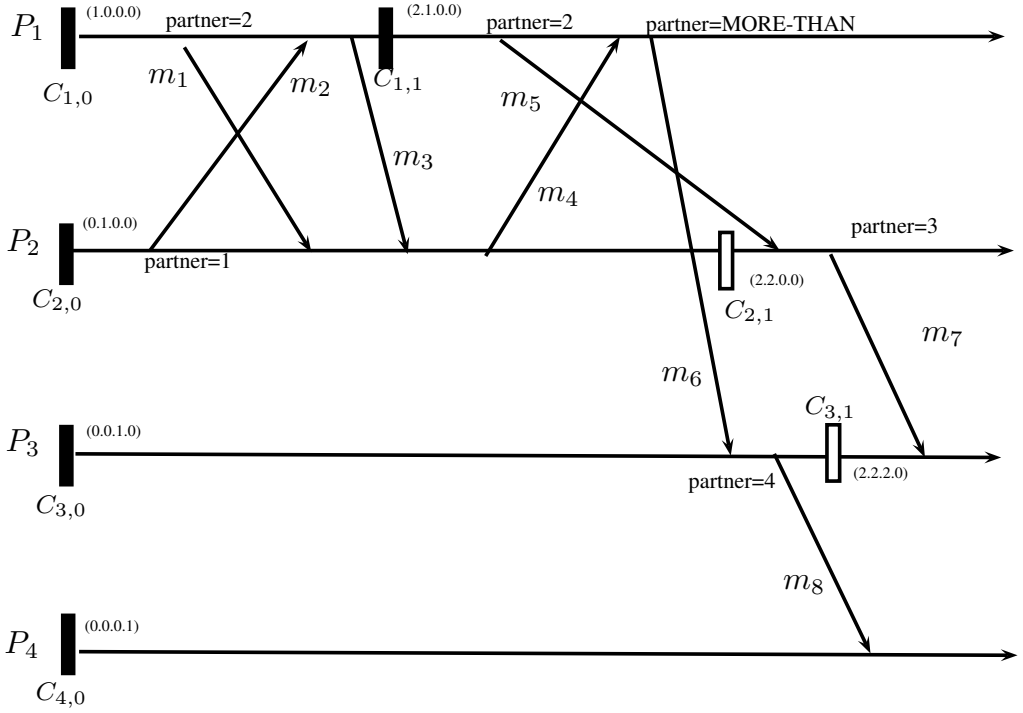


Figure 4.12: RDTPartner protocol

4.4 RDT protocols with constant size messages (our proposition)

In this section, we will describe our new RDT protocols.

4.4.1 Idea behind the new RDT protocols

Starting from the FDAS checkpoint induction condition, which is the following:

$$sent_i \wedge (\exists\ k \mid DV_i[K] < m.DV[k]$$

This condition allows a process P_i to decide whether it takes a checkpoint or not upon receiving a message m from a process P_j. Recall that $sent_i$ indicates if P_i has sent a message m' in its current interval while $\exists k \mid DV_i[k] < m.DV[k]$ is used to verify if there exists a new information brought by the message m (a process P_k has taken a new checkpoint that P_i does not know about it) .

The question is: "is it possible to ensure the RDT property without appending the entire vector DV on computation messages?"

To answer to this question ,we start by describing the direct dependency clocks.

4.4.1.1 Direct Dependency Clock

Fowler and Zwaenepoel [132] have proposed a weaker version of vector clocks [126, 127](called Direct Dependency Clock) that dates events and tracks the causality. The advantage of such timestamping technique is that does not require sending the entire vector, it, instead, appends only one component on the message. This key feature motivates us to adapt direct dependency clocks (DC) to timestamps checkpoints because it allows us to reduce the size of control information to piggyback on messages.

Each process P_i has a local vector clock DC_i with conventional increment and adjustment operations. Initially the clock of process P_i has the value $DC_i[i] = 1 \wedge \forall j \neq i :: DC_i[j] = 0$. the following class describe the basic operation Inc and Adjust.

```
class Clock {
        int[] cpt ; int PID = 0 ;
        // Read and increment the clock
        int[] Inc() { int[] hr = cpt.clone() ; cpt[PID]++ ; return hr ; }
        // Adjust the clock
        void Adjustself( int j, int vj ) { /* message from a process with PID=j */
                cpt[loc] = max(cpt[PID],vj ) ;
        }
        void Adjustsother( int j, int vj ) { /* message from a process with PID=j */
                cpt[j] = max(cpt[j],vj) ;
        }
        // Constructor : "PID" Process IDentity.
        Horloge(int PID, int N) {
                PID = getpid(); cpt = new int[N] ;
                for (int i = 0; i<N; i++) cpt[i]=0; cpt[PID] = 1 ;
        }
}
```

Finally, we manage the clock DC as follows:

Event at P_i	Update actions by P_i
Taking a checkpoint	DC_i.Inc()
Sending a message m	m.sn= $DC_i[i]$, send(m)
Receiving a message m by P_i from P_j	DC_i.Adjustself(j,m.sn),Adjustother(j,m.sn)

To keep the same behavior as in FDAS protocol when P_i receives a message m from P_j, we need only to piggyback a scalar sn. sn will be transported along the path μ from P_k to P_j (including the message m). To know if there is new information brought by the message m it is necessary to keep track of other processes states at P_i via the vector DC_i and Adjustself and Adjustother operations. Hence, P_i evaluates the following condition $sent_i \wedge (m.sn > DC_i[j]$.

4.4.2 CSFDAS (Constant-Size-FDAS)Protocol

Data structures

Every process P_i has the following data structures

sn_i : An integer initialized to "0".

$Sent_i$: A boolean initialized to false. When a process P_i sends a message m, it sets $Sent_i$ to true.

DC_i : An integer vector initialized to "0" except $DC_i[i]$ which is initialized to "1".

Description of the protocol

$Take-Basic-Checkpoint$: When a process P_i decides to take a basic checkpoint, it increments its sn_i, sets $DC_i[i]$ to sn_i and saves its state with a copy of its sn and its DC vector.

$Send-Message$: The process P_i sets $Sent_i$ to true and piggybacks its sn_i on the message.

$Receive-Message$: When a process P_i receives a message m from P_j, it takes a forced checkpoint if it has sent a message in its current interval and the coming message has carried new information $sent_i \wedge (m.sn > DC_i[j]$. Then, P_i updates its sn. $(sn_i = max(sn_i, m.sn)$.

Procedure 1: When P_i decide to take a basic checkpoint

```
sn_i++;
DC.Inc();
Take a basic checkpoint;
sent_i=false;
```

Procedure 2: When P_i decide to send a message m

sent$_i$=True;
m.sn= sn_i;
send the message m;

Procedure 3: When P_i receives a message m from process P_j

if $(sent_i \wedge (m.sn > DC_i[j]))$ **then**
    $sn_i = max(sn_i, m.sn)$+1;
    $DC_i.Adjustself(j, m.sn)$; $DC_i.Inc()$;
    Take a forced checkpoint;
    $sent_i$=false
else
    $sn_i = max(sn_i, m.sn)$;
    $DC_i.Adjustself(j, m.sn)$
$DC_i.Adjustother(j.m.sn)$;
Process the message m;

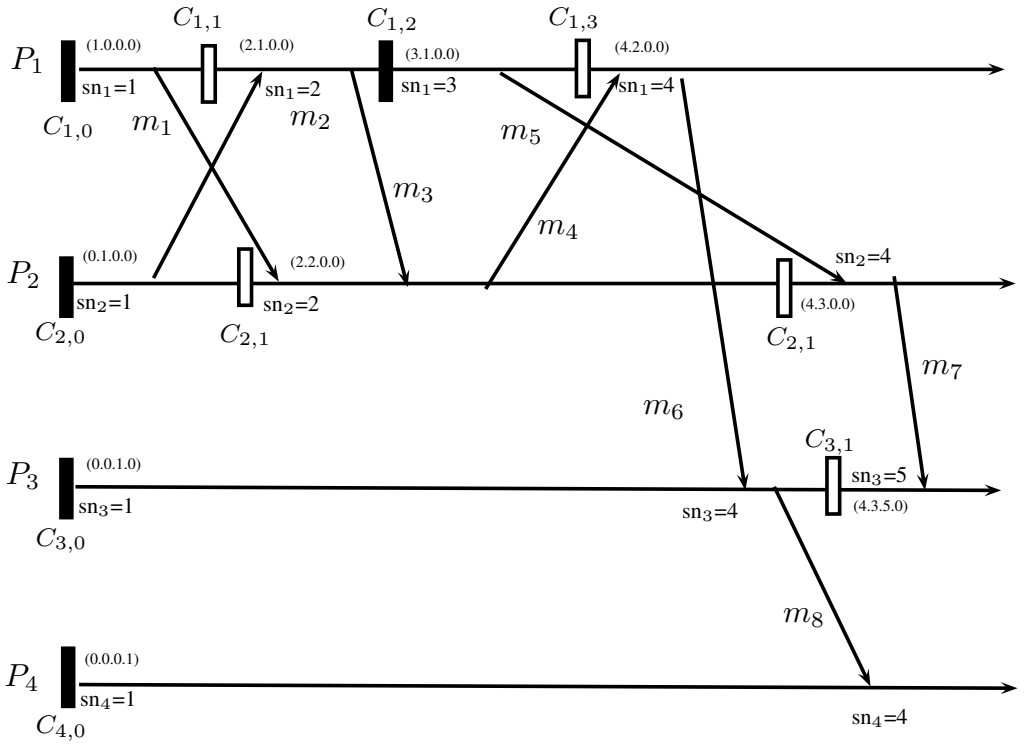


Figure 4.13: CSFDAS protocol

Example

In figure 4.13, CSFDAS protocol induces the same number of forced checkpoint as FDAS protocol. When P_1 sends the message to P_2, it piggybacks its sn variable on it. P_2, at receiving the message m_1, checks for new information carried by the received message,

then it takes a forced checkpoint $C_{2.1}$ because it has already sent a message m_2 to P_1 ($sent_2 = true \wedge m_2.sn > DC_2[1]$). Other processes take the same action as P_2 at receiving messages m_1,m_4, m_5 and m_7.

4.4.2.1 *Correctness proof*

Theorem 4.5

CSFDAS protocol ensures RDT property.

Proof: The proof is done by contradiction. Assume that RDT property is not ensured, that means there exists a non-causal Z-path $\mu.m$ such that last (μ) is sent from P_i to P_j ($\neg(sent \wedge DC_j[i] > last(\mu).sn)$. There exist two cases:

Case1 m is not sent in the current interval ($\neg sent$) that means there exists a checkpoint that breaks the Z-path $\mu.m$. So contradiction.

Case2 m is sent in the current interval ($DC_j[i] < (\mu.last).sn$) means that P_j has received a message m' before sending m. In this case $\mu.m$ is causally doubled by the causal path $\mu - (\mu.last).m'.m$. so contradiction

We obtain a contradiction, so, CSFDAS protocol ensures RDT property. ■

4.4.3 CSRDTPartner (Constant-size RDTPartner) protocol

By the same way as CSFDAS, we can modify the RDTPartner protocol in order to reduce the message overhead. We keep the same data structure except DV vector which is changed to DC vector, and we add sn variable. In message header, we substitute DV vector by only snsender and snreceiver variables.

4.4.3.1 *Description of the protocol*

Take – Basic – Checkpoint: When a process P_i decides to take a basic checkpoint, it increments its sn_i, sets $DC_i[i]$ to sn_i and saves its state with a copy of its sn and its DC vector.

Send – Message: When process P_i decide to send a message to P_j, it sets Partner equal to j or MORE-THAN-ONE.Then, it piggybacks its $Simple_i[i]$, and sn_i and $DC_i[j]$, respectively, as snsender and snreceiver on the message.

Receive – Message: When a process P_i receives a message m from P_j, it takes a forced checkpoint i) if it detects, via the received information (m.simple, m.snreceiver) that a PMM-Cycle is about creating a Z-Cycle; or ii) it has sent messages to more than one process. Then, P_i updates its data-structures.

Procedure 1: When P_i decide to take a basic checkpoint

sn_i++;
$DC_i.Inc()$;
Take a basic checkpoint;
partner pid = NO-PARTNER;
for *i:=0* **to** *N* **do** simple[i] = i == pid;

Procedure 2: When P_i decide to send a message m to P_j

sent$_i$=false;
m.sender=i;
m.receiver=j;
m.simple = simple[m.receiver];
m.snsender=sn_i;
m.snreceiver=$DC_i[m.receiver]$;
if *partner pid == NO-PARTNER* **then** partner pid = m.receiver;
else if *partner pid != m.receiver* **then** partner pid = MORE-THAN-ONE;
send the message m;

Procedure 3: When P_i receive a message m from process P_j

if $(m.snsender > DC_i[m.sender])\delta\delta(partnerpid! = NO-PARTNER)$ **then**
  if $(partnerpid! = m.sender) \,||\, (partnerpid == m.sender)\delta\delta((m.snreceiver == DC_i[m.receiver])\delta\delta!m.simple))$ **then**
    $sn_i = max(sn_i, m.snsender)+1$;
    $DC_i.Adjustself(j, m.snsender)$;
    $DC_i.Inc()$;
    Take Forced checkpoint
  simple[m.sender] = true;
else
  $sn_i = max(sn_i, m.snsender)$;
  $DC_i.Adjustself(j, m.snsender)$
$DC_i.Adjustother(j, m.snsender)$;
Process the message m;

Theorem 4.6

CSRDTPartner protocol ensures RDT property

Proof: The proof is similar to that of Theorem 4.5 ■

Example

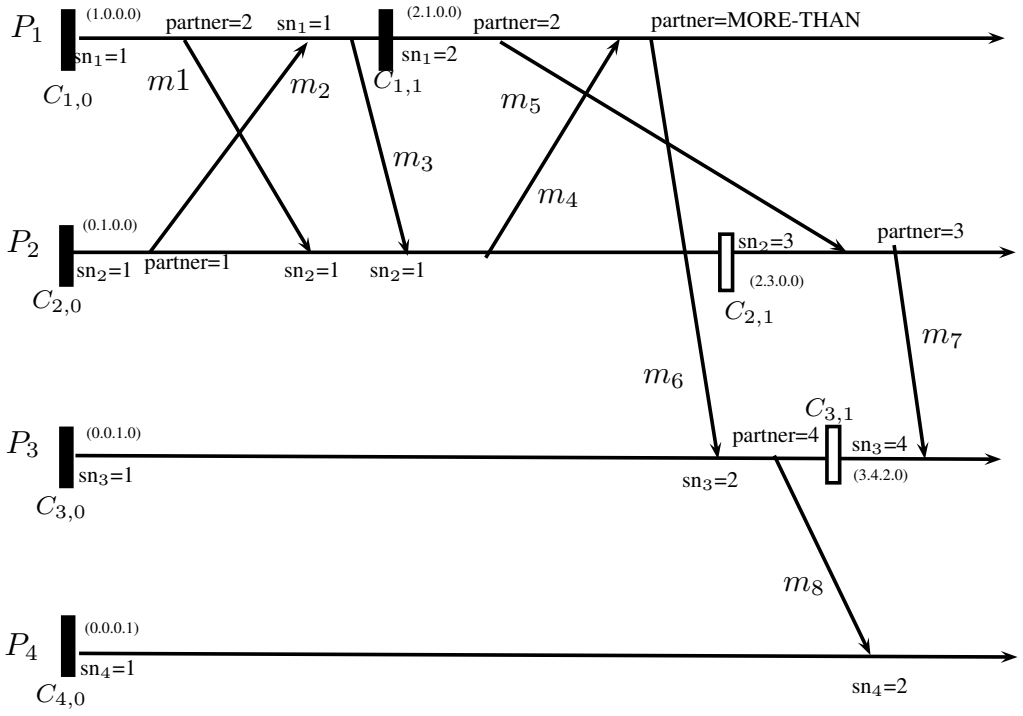


FIGURE 4.14: CSRDTPartner protocol

In the example depicted in Figure 4.14, CSRDTPartner induces the same number as RDTPartner protocol. For example, when P_2 receives the message m_5, it takes a forced checkpoint $C_{2,1}$ because it has detected that the checkpoint $C_{1,1}$ of process P_1 is involved in a Z-Cycle formed by $\{m_5, m_2\}$ ($partner = 1 \wedge m.snreceiver = DC_2[2] = 1 \wedge m.simple = true$). In turn, P_3 takes the forced checkpoint $C_{3.1}$ because the following condition is verified ($partner = 4 \neq m.snsender = 2 \wedge m.snreceiver = 3 > DC_3[2]$).

4.5 ON RDT PROTOCOLS COMPLEXITIES

Every CIC protocol is characterized by its checkpoint inducing condition. Using a stronger condition may on one hand reduce the forced checkpoint number but on other hand force CIC protocol to use complex data-structures and to piggyback it on the computation message. In terms of piggybacked information size, we calculate the complexity of described RDT protocols. BHMR is O(N^2) complexity because it need to piggyback $N^2 + N$ bits plus N integer, RDTPartner is $O(N)$ complexity piggybacks due to the fact that it superpose $Nbits + N$ integer. FDAS has the same RDTPartner complexity, namely $O(N)$ because it piggybacks only n integers.CSRDTPartner is $O(1)$ because it superpose

two integers on the computation message. CSFDAS protocol has lowest complexity of $O(1)$ since it piggyback only one integer.

4.6 Simulation study

In this section, we compare our proposals to RDT protocols presented in Section 4.3.1.3. For this purpose, we choose ChkSim simulator [133] to evaluate the performance of such protocols under two different topology: complete and ring. To cover the major communication patterns of CIC protocols, we have made different scenarios described as follows.

Scenario 1

Under this scenario, the measurements are collected in function of the processes number which ranges from 5 to 100. In this case, processes have the same interval length L=44 events.

Scenario 2

This scenario is similar to the previous scenario except 10% of process having an interval length L'=14 events.

Scenario 3

In this scenario, we fix the number of processes to 100 process and we vary the number of asymmetric process from 5 to 50 process.

Scenario 4

Under this scenario, measurements are collected in terms of interval length for a system composed of 100 process. Interval length ranges from 4 to 118 event

Scenario 5

Measurements, under this scenario, are made with interval length variation. The system constitutes of 100 processes where only 10 processes have an interval length L' less than other processes by 30 events. The interval length L' ranges from 4 to 118.

Scenario 6

In this scenario, we consider a system of 100 processes. All processes have an average interval length of 44 events, except that 10 processes have an interval length less than the others by 2 to 40 events. The data are collected in terms of the difference in interval lengths.

Table 4.1 summaries the simulation results (for complete and ring typologies).

TABLE 4.1: Summary of simulation scenarios

Scenario	nb-proc	interval length L	# asymmetric processes	interval length L'	Diff
Scenario1	5-100	44	-	-	-
Scenario2	5-100	44	10	14	30
Scenario3	100	44	5-50	14	30
Scenario4	100	4-118	-	-	
Scenario5	100	30-148	10	4-118	30
Scenario6	100	44	10	40-2	2-40

4.6.1 Simulated protocols

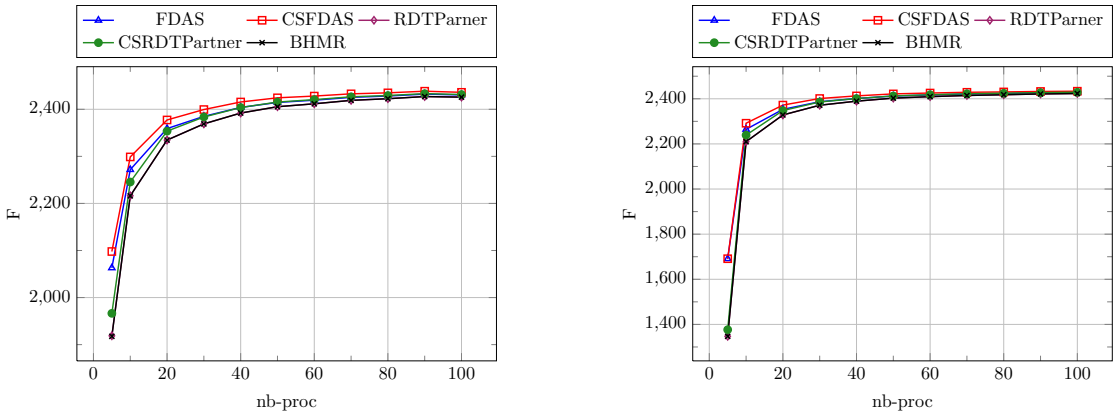
We evaluate the performance of the following protocols: FDAS, BHMR, RDTPartner, CSFDAS and CSRDTPartner.

4.6.2 Simulation results

4.6.2.1 Complete network

In this section, we study the behavior of RDT protocols under complete network topology. Figures 4.15(a) to 4.18 depict simulation results with the respect of forced checkpoints number F.

In figure 4.15(a) when nb-proc is between 5 and 40, we can see a fast increasing of forced checkpoints number. When the number of processes is becoming large, we see a



(a) Symmetric case

(b) Asymmetric case

Figure 4.15: Number of forced checkpoints vs number of processes

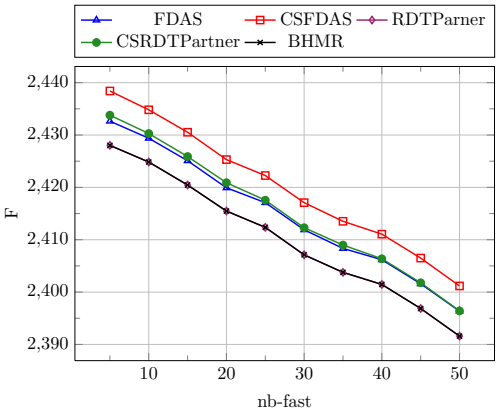
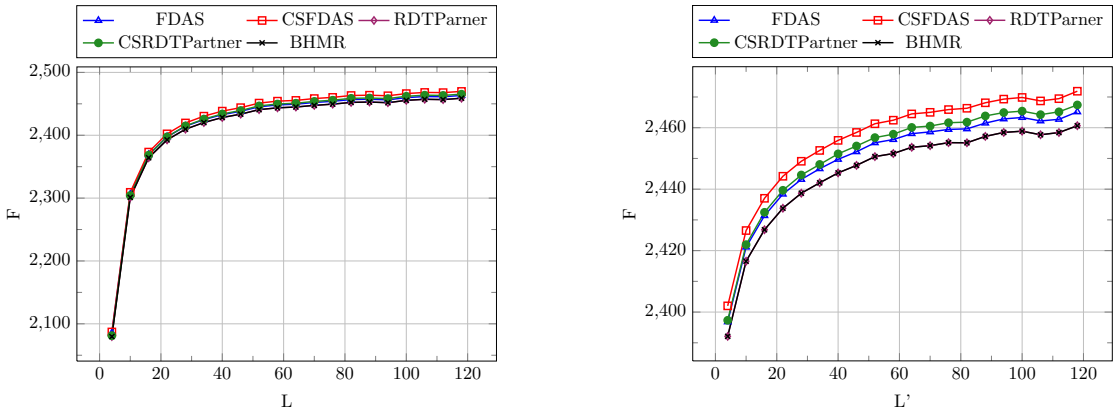


Figure 4.16: Number of forced checkpoints vs number of fast processes

slight increase. In addition, CSFDAS and CSRDTPartner protocols induce nearly the same number of checkpoints as RDTPartner and BHMR because the latter protocols search patterns involving Visibly doubled PCM-paths and PMM paths, which are seldom to occur. Consequently, BHMR and RDTPartner protocols have nearly the same performance as CSFDAS and CSRDTPartner protocols.

Under scenario 2, we study the impact of the asymmetry on protocols behavior where only 10% of processes have a different interval length from the others. The remarks drawn for the previous scenario remains the same for this scenario which, however, does not show clearly the asymmetry impact (Figure 4.15(b)). Hence the third scenario.

Scenario 3 shows the variation of forced checkpoints number in terms of fast processes (nb-fast). Due to the increase in the number of fast processes, the number basic checkpoints increases, and hence the decrease in the satisfaction chance of the induction condition as well the number of forced checkpoint (Figure 4.16).



(a) Symmetric case

(b) Asymmetric case

FIGURE 4.17: Number of forced checkpoints vs interval length

Under scenarios 4, 5 and 6, we study the impact of interval length on protocols performance for the symmetric and asymmetric cases. Figures 4.17(a) and 4.17(b) illustrate the forced number variation in terms of interval length. According to the results shown in Figure 4.17(a) when L is between 5 and 50, we see a fast increase in the number of forced checkpoints from 2300 to 2430. However, when L is between 50 and 120, the forced checkpoints number slightly increases; this can be explained by the fact that communication patterns inducing forced checkpoints are almost the same.

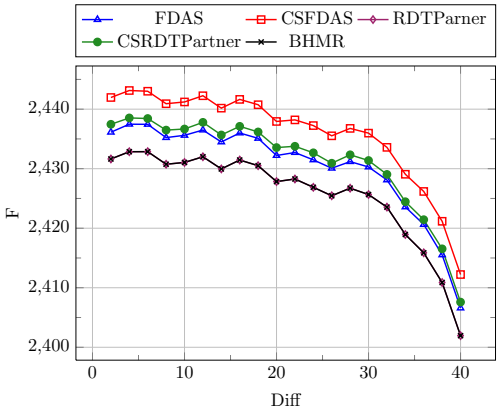


FIGURE 4.18: Number of forced checkpoints vs interval length differences

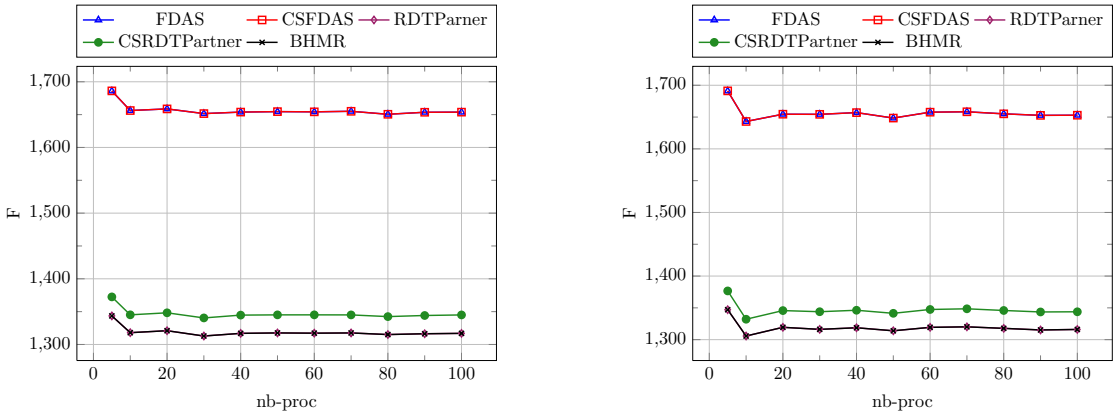
In comparison to Figure 4.17(a), Figure 4.17(b) shows the asymmetry influence on RDT protocols behavior, especially when L in the range of [5,50]. The forced checkpoints number is larger than that of the Figure 4.17(a), the reason lies in the difference in interval lengths.

Figure 4.18 confirms the results drawn about asymmetry in the foregoing scenario. Once again, increasing the difference in interval lengths results in the decrease in the forced checkpoints number. This can be explained by the fact that causally doubled

paths are more likely to appear, which implies the non-satisfaction of the checkpoint induction condition.

4.6.2.2 *Ring network*

The performance of simulated protocols under ring topology is measured with respect to forced checkpoints number. Likewise to the complete topology, we study the impact of the different simulation parameters on protocols behavior for the described scenarios.



(a) Symmetric case

(b) Asymmetric case

Figure 4.19: Number of forced checkpoint vs number of process (Ring topology)

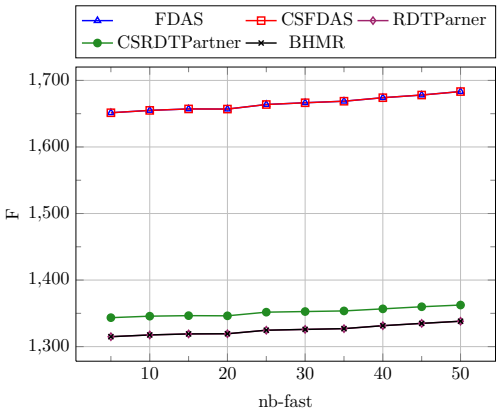


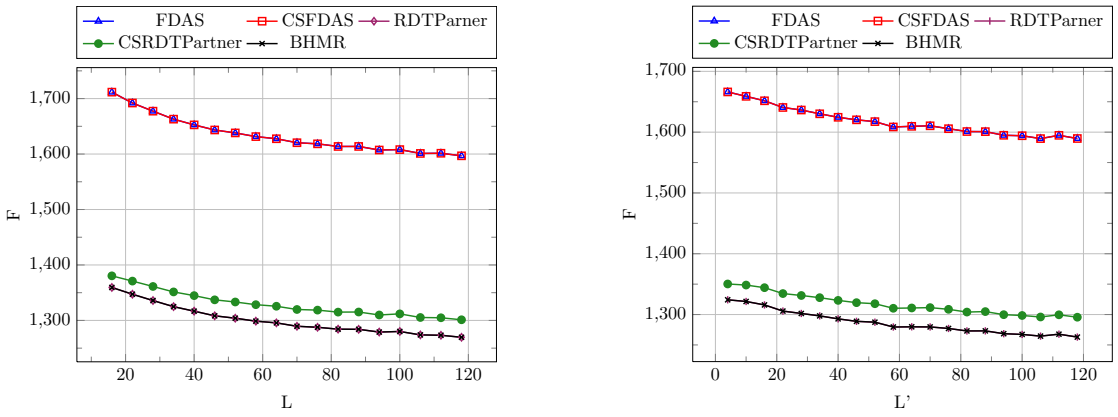
Figure 4.20: Number of forced checkpoints vs number of fast processes (Ring topology)

The measurements illustrated in Figures 4.19(a) and 4.19(b) show the impact of processes number variation on the forced checkpoints number for the symmetry and asymmetry cases. As one can see, the forced checkpoints number remains almost fixed even

though the processes number increases or having asymmetric behavior; this can be explained by the fact that each process communicates only with its two neighbors, which makes the communication patterns inducing forced checkpoints remain nearly the same.

In addition, as one can see, there is a great difference between FDAS and CSFDAS protocols, and the remaining protocols because the latter protocols have a stronger inducing checkpoints condition than the former ones. On other hands, in spite of having a weaker inducing condition, CSRDTPartner protocol show a good performance compared to that of BHMR and RDTPartner protocols.

In Figure 4.20, we see a slight impact of the increase in fast processes on the increase in the forced checkpoints number due to the communication nature of this specific topology.



(a) Symmetric case

(b) Asymmetric case

FIGURE 4.21: Number of total checkpoints vs interval length (Ring topology)

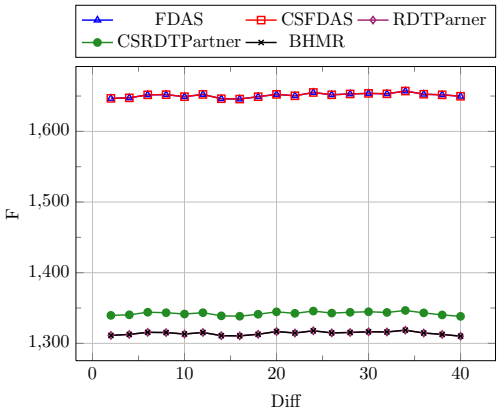


FIGURE 4.22: Number of forced checkpoints vs Interval length differences (Ring topology)

Besides studying the impact of processes number, we study the interval length impact on protocols performance. Figures 4.21(a) and 4.21(b) depict the simulation results of

scenarios 4 and 5. Increasing interval lengths implies the decrease in the forced checkpoint number, but with a slight impact. In contrary, increasing the difference in interval lengths does not have a great effect protocols behavior (Figure 4.22).

As a conclusion to the simulation study, we can draw the following remarks. First, RDT protocols, under large systems, worsen their performance even though they use a stronger checkpoint induction condition (like BHMR protocol). Secondly, RDT protocols having low message overhead (like CSFDAS, CSRDTPartner) show a good performance despite using weaker conditions, which make them a better choice than other protocols having heavy messages. Finally, frequency of taking basic checkpoints has a great effect on RDT protocols behavior, which is proved by studying the impact of interval length, asymmetry, and the number of processes involved in the communication patterns.

4.7 Conclusion

This chapter is devoted to special checkpointing protocols those based on a stronger consistency property called Rollback Dependency Trackability (RDT for short). RDT property states that there is no hidden dependency between checkpoints. Wang was the first who give a simple characterization of RDT property based on transitive dependency vectors. It has introduced FDAS protocol where process are induced to take additional checkpoints if such property is violated.

Another characterization has been introduced by Baldoni et al. called PCM (Prime causal message) and EPSCM .(Elementary and Simple PCM) characterizations. The idea behind such characterization is "if a subset of non-causal Z-paths are causally doubled, then, the property RDT will be ensured". The proposed protocols based on such characterization (for example BHMR protocol) require some knowledge of processes past to be available. Moreover, it needs piggybacking more control information on computation message. Such information is used to prevent PCM paths (or EPSCM paths) from establishing hidden dependencies.

A refinement of Baldoni et al.'s characterization has been proposed by Garcia and Buzeto called PMM(Prime Message Message). Protocols based on PMM characterization (for example RDT-Partner) require less information to be piggyback on computation messages while ensuring the efficiency.

FDAS, BHMR and RDT-Partner protocols suffer from overhead problem due to piggybacking O(N) to O(N^2) (N is the number of processes) control information, which has a negative impact on protocols performance. Thus, this chapter has presented two new

RDT (called CSFDAS and CSRDTPartner) protocols. CSFDAS and CSRDTPartner protocol are based on direct dependency clocks that help in the overhead reduction. Moreover, such protocols, through a simulation study, have shown their good performance compared to FDAS, BHMR and RDT-Partner protocols especially with a large number of processes as well as for the ring topology.

Chapter 5

Fast Non-Blocking Coordinated Checkpointing Protocol (FNB)

Contents

5.1 Introduction

Coordinated checkpointing protocols rely on the cooperation between processes by the mean of request messages to determine a consistent global checkpoint. This cooperation, however, may lead to the main drawback of synchronization overhead. Therefore, many techniques have been proposed to decrease such overhead. It is based on a specific topology or dependency tracking. Despite that, the problem is still laid.

In order to design a Non-blocking Coordinated Checkpointing protocol that does not suffer from the synchronization cost or the problem of a large number of mutable checkpoints, This chapter presents a new solution based on two techniques. The first one is piggybacking dependency information on computation messages and replies. The second policy is popular processes, i.e. a process that has a large dependency information percentage.

At first, we will illustrate the problem statement of the synchronization and how our techniques can solve it. Then, we give a full description of our new protocol (FNB) provided with correctness proof. After that, we will present our simulation study that compares FNB protocol to CSB and CSNB protocols.

5.2 Problem statement

First, we recall the concept of R vectors introduced by CSNB to track direct dependency between processes. Each process has a bit array R_i. In each checkpoint interval, this array is initialized by zero except $R_i[i] = 1$. If P_i has received a message from P_j then $R_i[j]$ is updated to 1. In Figure 5.1, P_0 has initially the vector (1000000), when P_0 receives a message from P_3, it updates $R_0[3]$ to 1 and its vector becomes (1001000). We say that P_0 depends on P_3.

Figure 5.1 shows an execution of the CSNB protocol. When P_2 initiates the checkpointing process; it sends a request to P_1. Then, P_1 takes its tentative checkpoint and sends requests to P_0 and P_5. Similarly, this procedure is repeated by P_0, P_5, P_3 and P_4 as soon as they receive requests. Note that P_3 has received a computation message m_4 indicating that a checkpointing procedure is running, thus, P_3 will take a mutable checkpoint before processing this message. After, when P_3 receives the request, it will turn its mutable checkpoint into tentative. As one can see, P_6 does not participate in the global checkpoint determination because it has not any dependency relation with the initiator P_2.

Notation	Description
T_{disk}	delay incurred in saving a checkpoint on the stable storage .
T_{data}:	delay incurred in transferring a checkpoint to the stable storage .
T_{msg}	delay incurred by control messages during a checkpointing process.
DFP	duration of the first phase $DFP = T_{msg} + T_{data} + T_{disk}$.

Figure 5.1 depicts clearly that CSNB protocol suffers from two main problems. The first problem is duplicate requests (P_5 has received three requests from P_1, P_3 and P_4) and the second one is the longest path taken by requests to reach a destination due to the dependency tree between processes (the longest path shown in Fig 5.1 is constituted by P_2, P_1, P_0, P_3 and P_4). Due to this problem, the CSNB protocol may have a high latency .

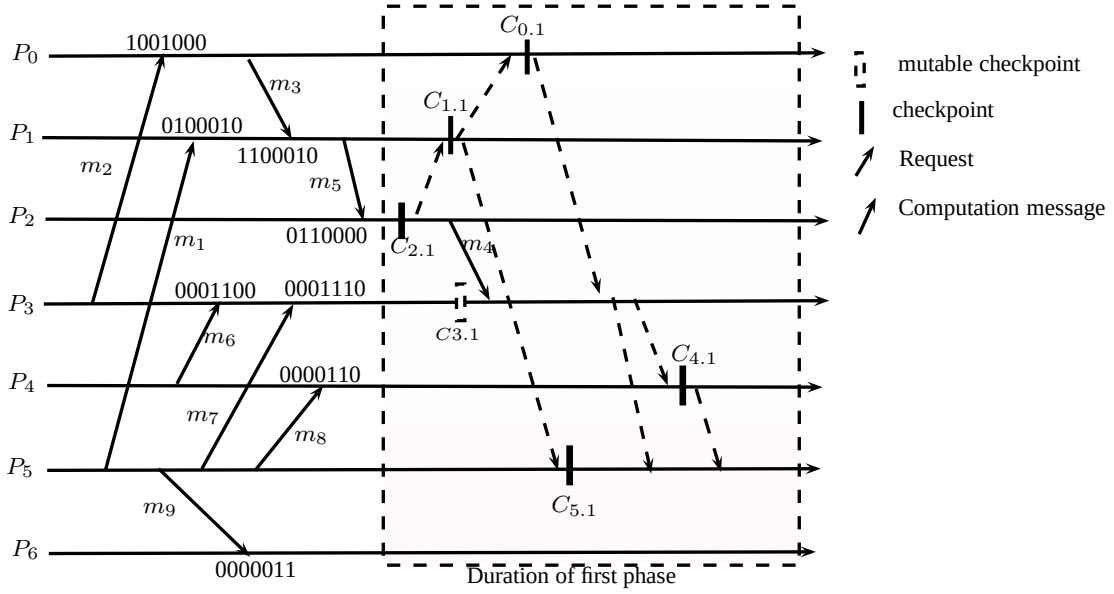


Figure 5.1: Execution of CSNB protocol

As shown in Fig5.1, CSNB first phase starts with sending the first request to P_1 and ends with receiving of the reply from P_5. As a consequence; the first phase duration involves the sending time of the following sequence $\{req_{21}, req_{10}, req_{03}, req_{34}, req_{45}, rep_{50}\}$. Under LogGP model (see Appendix A), we assume that each request and reply are of K bytes size. So, T_{msg} is equal to $T_{msg}(CSNB) = 6L + 12o + 2g + 8(k-1)G$. Hence, $DFP_{CSNB} = T_{data} + T_{disk} + 6L + 12o + 2g + 8(k-1)G$.

Moreover, the CSNB protocol may induce a large number of mutable checkpoints due to the problem of high latency. For example, after checkpointing initiation, if P_2 sends computation messages to P_4 and P_5, they must take mutable checkpoints before processing these messages (not shown in the figure)

5.3 FNB: FAST NON-BLOCKING CHECKPOINTING PROTOCOL

Our objective is minimizing the number of request messages, reducing the number of mutable checkpoints induced during the checkpointing process and reducing the latency. To fulfill these objectives, we propose a solution based on two mechanisms. The first one is piggybacking dependency vectors on reply and computation messages and the second is the notion of popular process.

5.3.1 Piggybacking dependency vectors on reply messages

Our first idea is piggybacking dependency vectors on reply messages, and making the initiator as the only responsible for sending requests. In Fig 5.2, P_2 initiates the

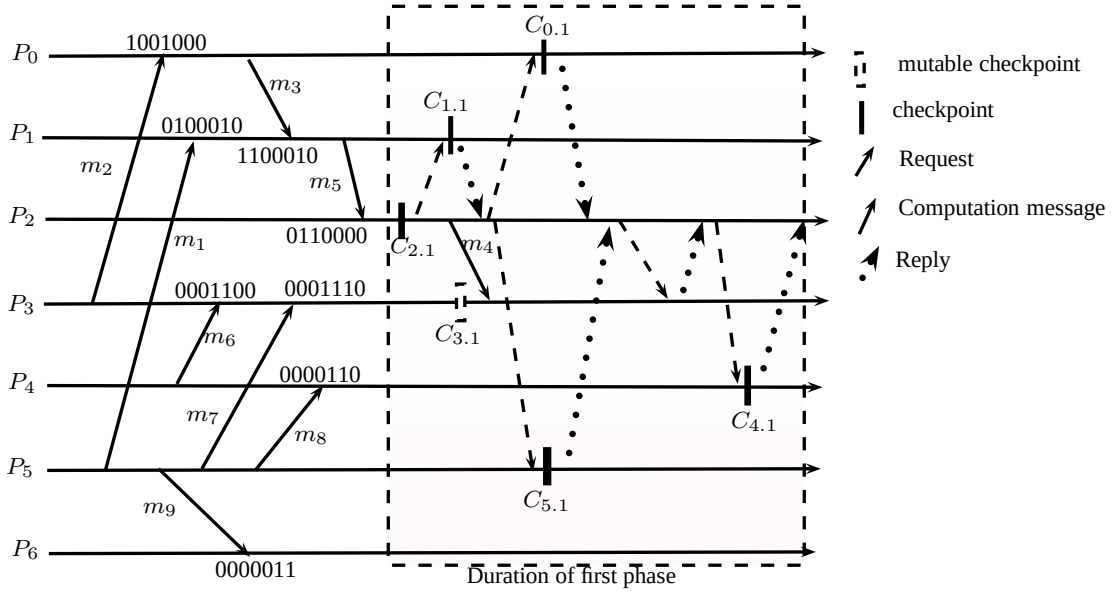


FIGURE 5.2: Piggybacking dependency vectors on reply messages

checkpointing process by sending a request to P_1. Then, P_1 takes its checkpoint and sends a reply with its dependency vector to P_2, P_2 checks the information piggybacked on the reply and sends requests to P_5 and P_0. In turn P_5 and P_0 take checkpoints and send replies to P_2. After receiving replies from these processes, P_2 sends a request to P_3. At the arrival of the request, P_3 makes its mutable checkpoint tentative and sends a reply back to P_2 (P_3 has taken a mutable checkpoint because it has received a message indicating that a checkpointing process is running). Finally, P_2 again has new information so it sends a request to P_4.

As we can see, this way solves the problem of duplicate requests. But the checkpointing process may have a high latency.

5.3.2 Piggybacking dependency vectors on computation and reply messages

Piggybacking dependency vectors on reply messages solves the problem of duplicate requests, but the number of mutable checkpoints is not reduced and the checkpointing process may have a high latency. Therefore, this scheme is not efficient, and piggybacking dependency vectors on computation messages can enhance more our protocol.

Piggybacking dependency vectors on computation and reply messages relies on the following management of dependency vectors. This management is different from the one used in CSNB protocol and similar to the protocol used in [26].

Based on the following DEP management:

Managing the dependency vector DEP
Initialization
$DEP_i[i] = 1$
$DEP_i[j] = 0$ such that $i \neq j$
Sending a computation (reply) message m
Piggybacking DEP on m (m.DEP)
Receiving a computation (reply) message m by P_i
For $i \neq j$ $DEP_i[j]$=max ($DEP_i[j]$, m. DEP[j])

we can track transitive dependencies between processes. Therefore, this management helps us to reduce the checkpointing process latency. Figure 5.3 shows that exchanging computation messages helps processes to track dependencies among themselves. For example, P_2 detects new dependencies, due to the piggybacked vector (1101000) on the message m_5; then, the DEP vector of P_2 is updated and became (1111010).

Hence, when P_2 starts checkpointing, it sends requests simultaneously to P_0, P_1, P_3 and P_5 based on its DEP vector. After receiving replies from this set of processes, a new dependency is discovered between P_2 and P_4 due to the piggybacked dependency vectors on replies. Therefore, P_2 will send a request to P_4. Finally, the first phase of checkpointing terminates when P_2 receives the reply from P_4.

Note that in Fig 5.3, P_3 takes a tentative checkpoint not a mutable checkpoint (unlike the case of Fig 5.1) because it is more likely that it receives a request before receiving

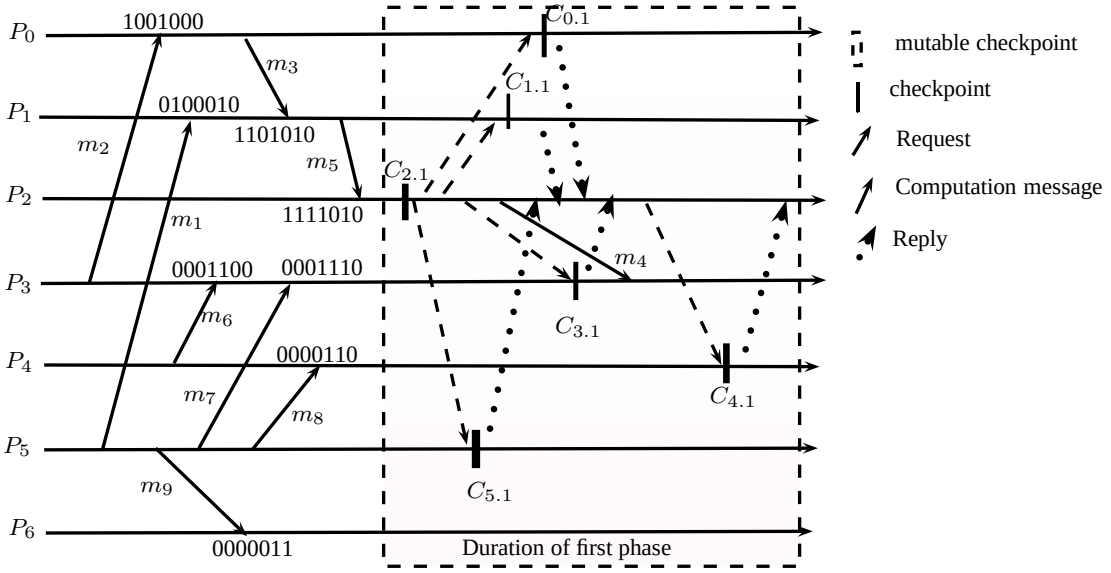


FIGURE 5.3: Piggybacking dependency vectors on computation and reply messages

m_4 (because P_3 receives request directly from P_2). As a result, piggybacking dependency vectors on computation messages may reduce the number of mutable checkpoints.

It is clear that the hidden dependencies cannot be tracked before initiating checkpointing, i.e., they are not trackable on-line, and this type of dependency is only tracked on receiving reply messages.

As a result, piggybacking dependency vectors on computation and reply messages solves the problem of duplicate requests and may reduce the checkpointing latency.

5.3.2.1 *Reduction of messages overhead*

With the help of an example (Figure 5.4), we show how the message overhead can be reduced. Suppose a system composed by three processes P_1, P_2 and P_3 with dependency vectors DEP_0 (100), DEP_1 (010) and DEP_2 (001). When P_1 sends m_1, it appends its DEP vector to this message. When P_1 sends m_3, it finds that its DEP vector has not been changed so it is unnecessary to append its vector with the message m_3.

This way of management helps us to reduce the overhead of computation and reply messages used in our protocol(see [134]for more efficient reduction).

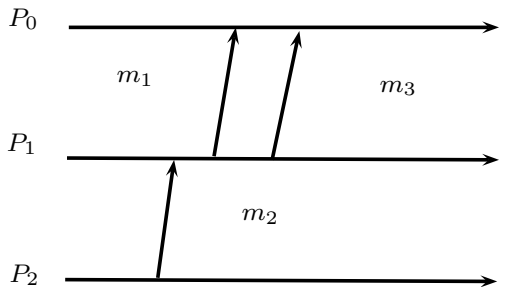


Figure 5.4: Reduction of message overhead

5.3.3 Popular process and checkpointing initiation

Each process P_i calculates its dependency percentage DP by the following formula $DP = \frac{DEPN}{N}$ (DEPN: number of DEP bits such that $DEP_i[i] = 1\ \forall i \neq j$. N: number of processes).

According to the dependency percentage, we can categorize processes into three groups. The first group, processes have a DP<20%. The second group has a percentage between 20% and 50%. The third group, named the group of popular processes, has a DP ≥50%.

When a process decides to initiate checkpointing, it starts a timer T (T= $\frac{1}{DEPN} + \Delta t$). At the expiration of its timer, the process can initiate checkpointing; this timer is stopped if the trigger has received a request or a computation message indicating that a checkpointing process is running.

To give more chance to popular processes to initiate checkpointing, we assign different Δt for each group such that the Δt of the third group is the smallest one.

We can benefit from Daly optimal interval τ_{opt} [107] by supposing that an initiator can trigger its timer after τ_{opt} interval from the last initiation. Where τ_{opt} is calculated as follows

$$\tau_{opt} = \begin{cases} \sqrt{2\delta M}\left(1 + \frac{1}{3}(\frac{\delta}{2M})^{\frac{1}{2}} + \frac{1}{9}(\frac{\delta}{2M})\right) - \delta & \text{if } \delta < 2M \\ M & \text{if } \delta \geq 2M \end{cases}$$

(M is the mean time between two failure and δ is checkpoint dump time)

Note that adding such a timer to τ_{opt} does not influence τ_{opt} delay for the following two reasons

i) When the system is very large having a higher communication rate, 1/DEPN becomes very small. In addition, Δt is calculated based on RTT (Ping Round Trip

Time, see Procedure 1) which is in the millisecond order. As a result, T is in the millisecond order

ii) τ_{opt} is calculated based on M which is in the hours order [135, 136], even if it is in the minutes order, that makes T$\ll \tau_{opt}$

In Fig 5.3, processes P_0, P_1, P_2, P_3, P_4, P_5 and P_6 having the following dependency percentages 14,2%, 28.5%, 57,1%, 28.5%, 14.2%, 0% and 14% respectively can be categorized into group1=$\{P_0, P_4, P_5, P_6\}$, group2=$\{P_1, P_3\}$and group3=$\{P_2\}$.

Suppose that at time t, processes P_2 and P_3 decide to initiate checkpointing; so, they start timers T_2 and T_3. T_2 expires first so P_2 can initiate checkpointing.

As shown in Fig 5.3, FNB first phase starts with sending of the first request to P_0 and ends with receiving of the P_4 reply. Under LogGP model , $T_{msg}(FNB)$ is equal to $4L + 8o + 8(k-1)G + 3g$. Then, $DFP(FNB) = 4L + 8o + 8(k-1)G + 3g + T_{data} + T_{disk}$. This duration is less than DFP (CSNB) which is equal to $T_{data} + T_{disk} + 6L + 12o + 2g + 8(k-1)G$.

5.3.4 Formal description of our Checkpointing algorithm

Our FNB checkpointing protocol has two phases; it is based on two policies. The first one is piggybacking dependency vectors on reply and computation messages and the second is the popular process.

Initiating checkpointing

When a process P_i decides to initiate checkpointing, it first starts a timer. At the timer expiration, the trigger can initiate checkpointing; it takes a tentative checkpoint and sends requests to every process P_j such that $DEP_i[j] = 1$.

Receiving a request message

On receiving checkpointing request from P_j, a process P_i compares its csn (cur_csn) with the request csn (req_csn). There are two possibilities: if (cur_csn $\leq$ req_csn), P_i will change its state, turn the mutable checkpoint to tentative (if it exists) or take a tentative checkpoint and send a reply message to the initiator. Otherwise, it just sends a reply to the initiator because it has already taken a checkpoint.

Receiving a reply message

On receiving a reply, the initiator checks it for new piggybacked dependencies, if it finds new information, it will send requests to those related processes. After receiving all the relevant replies, the initiator detects the end of the first phase and it starts the second phase by broadcasting commit messages.

Receiving a commit message

After receiving a commit message, each process turns its tentative checkpoint into permanent, changes its state and discards mutable checkpoints if they exist.

Receiving a computation message during checkpointing

When a process P_i receives a computation message from P_j, P_i will process this message if the received csn (m.csn) is less than csn_i[j]. Otherwise, It checks if the message carries checkpointing information. Then, P_i takes a mutable checkpoint if: i) P_i has not taken a checkpoint associated to this initiator (msg_trigger) ii) P_i has sent a message after its last checkpoint and iii) P_j is under checkpointing (like CSNB conditions).

5.3.4.1 Data structures

DEP_i : a bit array of length N. DEP_i[j] =1 indicates that P_i is transitively dependent upon P_j.

csn_i : an array of integers of length N. csn$_i[j]$ contains the current information of P_i about P_j's checkpoint sequence number.

V_i : an array of integers of length N. V_i[j] contains the version of DEP vector sent to process P_j. When P_i takes a checkpoint, it resets this vector. V_i[i] is increased if the DEP vector has been changed.

$nb\text{-}req$: integer counter of requests.

$nb\text{-}reply$: integer counter of replies.

S : boolean vector used to calculate the minimum set of involved processes.

$trigger_i$: a tuple (PID, inum). It contains initiator PID and the initiator csn.

$sent_i$: a boolean, $sent_i = 1$ means that P_i has sent a message in its current checkpoint interval.

cur_state_i: a boolean, $cur\_state_i = 1$ means that P_i is under checkpointing.

cur_csn: an integer that indicates the csn of the current tentative (permanent) checkpoint.

MP_i : a record used by each process P_i to save the mutable checkpoint and the necessary data structures in the following fields:

mutable: the mutable checkpoint of P_i.

DEP : dependency vector of P_i before taking the mutable checkpoint.

trigger : contains the initiator information for which P_i has taken the mutable checkpoint.

V : contains versions of DEP vector sent to processes before taking the mutable checkpoint.

sent : the sent value of P_i before taking the current mutable checkpoint.

5.3.4.2 *Checkpointing algorithm*

Procedure 1: When P_i triggers a timer in order to initiate checkpointing

for *k:=0* **to** N **do**
  if $(DEP_j[k] = 1) \wedge (j \neq k)$ **then** $DEPN++$;
$DP := DEPN/N$;
if $DP < 0.20$ **then**
  $\Delta t := \Delta t1$; /* $\Delta t1$ = Ping Echo Round Trip Time */
else if $DP < 0.50$ **then** $\Delta t := \Delta t1/2$;
else $\Delta t := \Delta t1/3$;
Start Timer $T := 1/(DEPN) + \Delta t$;
if *the Timer T has expired* **then** P_i can initiate checkpointing ;
if P_i *has received a request or a computation message carrying checkpointing information*
then Stop Timer T ;

Procedure 2: When P_i decides to initiate checkpointing

$csn_i[i]++$; mytrigger := $(P_i; csn_i[i])$; $cur\_state_i := 1$;
for *k;=0* **to** N **do** $Maxcsn[k] := csn_i$[k];
S_i[k]=0;
for *k:=0* **to** N **do**
  if $(DEP_i[k] = 1) \wedge (i \neq k)$ **then**
    $nb - req++$;
    send(P_i, request, $csn_i[i]$, $csn_i[k]$)
for *k:=0* **to** N **do** $S_i[k] := max(DEP_i[k], S_i[k])$;
take a tentative checkpoint (on stable storage);
$cur\_csn_i := csn_i[i]$; $sent_i := 0$; reset DEP_i; reset V_i;

Procedure 3: When P_i sends a computation message m to P_j

```
sent_i := 1;
if cur_state_i = 1 then
    if V_i[i] > V_i[j] then
        send(P_i, m, csn_i[i],DEP, mytrigger)
    else
        send(P_i, m, csn_i[i],NULL, mytrigger)
else
    if V_i[i] > V_i[j] then
        send(P_i, m, csn_i[i],DEP, NULL)
    else
        send(P_i, m, csn_i[i],NULL, NULL)
```

Procedure 4: When P_i receives a checkpoint request from P_j

```
receive(P_j , request, recv-csn, req-csn);
csn_i[ j] := recv-csn;
if cur_csn_i ≤ req − csn then
    cur_state_i := 1;
    if mytrigger =(PID(P_j), recv-csn) then
        if MP_i trigger=(PID(P_j, recv-csn) then
            save MP_i mutable on stable storage;
            cur_csn_i := csn_i[i];
            if MP.V[i]> MP.V[j] then
                for k:=0 to N do
                    MR[k].csn := MP_i csn[k];
                    MR[k].DEP :=MP_i DEP[k];
                send (P_i, reply, MR) to the initiator;
            else send (P_i, reply, NULL) to the initiator;
            MP_i := NULL;
    else
        csn_i[i]++; mytrigger :=( (PID(P_j), recv-csn));
        if V_i[i] > V_i[j] then
            for k:=0to N do
                MR[k].csn := csn_i[k]; MR[k].DEP:=DEP_i[k]
            send(P_i, reply, MR) to the initiator;
        else send (P_i, reply, NULL) to the initiator;
        take a tentative checkpoint (on stable storage);
        cur_csn_i :=csn_i[i]; sent_i := 0; reset DEP_i
else send(P_i, reply, NULL) to the initiator;
```

Procedure 5: When P_i receives a reply message

receive(P_i , reply, MR);
nb-reply++;
if $MR \neq NULL$ **then**
  for k:=0 **to** N **do** Maxcsn$_i$[k] := max (csn_i[k], MR[k]csn);
  for k:=0 **to** N **do**
    if $(MR[k].DEP=1) \wedge (Maxcsn_i[k]=MR[k].csn) \wedge (S_i[k] \neq MR[k].DEP) \wedge (i \neq k)$ **then**
      nb-req++;
      send(P_i, request, MR.csn$_i$[j], $Maxcsn_i$[k]);
  for k:=0 **to** N **do** $S_i[k]$:= max (MR[k].DEP, $S_i[k]$);
if *nb-reply=nb-req* **then**
  cur_state_i := 0;
  for k:=0 **to** N **do** send(commit, mytrigger) to P_k ;
nb-req:= 0; nb-reply:= 0; reset S_i;

Procedure 6: When process P_i receives a commit message

receive (commit, msg-trigger);
cur_state$_i$:= 0;
if $MP_i.trigger$ =*msg-trigger* $\wedge MP_i \neq NULL$ **then**
  $sent_i := sent_i \cup MP_i sent; DEP_i := DEP_i \cup MP_i.DEP$;
  MP$_i$:= NULL;$V_i[i] = max(MP_i.V[i], V_i[i]) + 1$;
if *there is a tentative checkpoint associated with msg-trigger* **then**
  make it permanent

Procedure 7: When P_i receives a computation message from P_j

receive (P_j, m, recv-csn, DEP, msg-trigger);
if $recv - csn > csn_i[j]$ **then**
  if $(msg\text{-}trigger \neq NULL) \wedge (sent_i = 1) \wedge (msg\text{-}trigger \neq mytrigger)$ **then**
    take a local checkpoint, save it in MP_i mutable;
    MP$_i$.trigger := msg-trigger; MP$_i$.DEP := DEP$_i$; MP$_i$.V := V$_i$;
    MP$_i$.sent := sent$_i$;
    sent$_i$:= 0; reset DEP$_i$; reset V$_i$;
  if $(msg\text{-}trigger \neq NULL) \wedge (state_i = 0)$ **then**
    cur_state$_i$:= 1; csn$_i$[i]++;mytrigger :=msg-trigger;
    csn$_i$[j]:= max (recv-csn, csn$_i$[j])
if $DEP \neq NULL$ **then**
  for k:=0 **to** N **do** DEP$_i$:= max (DEP, DEP$_i$[k]);
V$_i$[i]++;DEP$_i$[j];=1;
Process the message m

5.3.5 FNB protocol complexity

In the first phase, the initiator sends N_{dep} request and receives N_{dep} replies ($N_{dep}(N_{dep} \leq N)$ is the number of dependent processes involved in checkpointing process). In the second phase, the initiator broadcasts N commit messages so the total number of messages is $2 * N_{dep} + N$. Therefore, our protocol is O(N) complexity.

5.4 Correctness proof

Theorem 5.1

In the presence of one initiator at a time, FNB protocol creates a consistent global checkpoint

Proof: The proof is done by contradiction. Suppose that the global checkpoint is inconsistent. So, there is an orphan message such that its send event is not recorded and its receive event is recorded in the global checkpoint. In the other words, there exists a pair of checkpoints $(C_{i,x}, C_{j,y})$ such that $C_{i,x}$ precedes send(m) and receive(m) precedes $C_{j,y}$ There are two possibilities:

1. The message m does not carry checkpointing information (msg-trigger=NULL) In this case, after receiving the message m, P_j updates DEP_j[i]. When P_j receive a request from P_k, it sends a reply back to this initiator. As a result, P_k sends a request to P_i giving arise to two cases:
   - P_i takes a checkpoint associated to the initiator P_k so the send(m) is recorded (m is not orphan).
   - P_i has already taken a checkpoint (associated to an other initiator); this means that req-csn<cur_csn, so, send(m) is recorded (m is not orphan)
2. The message m carries checkpointing information(msg-trigger≠NULL) In this case, P_i has taken a checkpoint before send(m), there are two cases:
   - P_j has already taken a checkpoint. So, receive(m) is not recorded (m is not orphan).
   - P_j takes a mutable checkpoint before processing the message; so, after receiving checkpointing request, this mutable checkpoint is turned into tentative. Hence, the receive(m) is still not recorded (m is not orphan).

We get contradiction so the global checkpoint is consistent. ■

Theorem 5.2

In the presence of one initiator at a time, FNB protocol terminates within a finite time.

Proof: The following invariant holds during checkpointing
$(nb-req = \mid set_t \mid + \mid set_i \mid) \wedge (set_t \cap set_i = \emptyset) \wedge (\mid set_t \cup set_i \mid \leq N)$
Where set_t is the set of processes which have direct and transitive dependencies with the initiator, set_i is the set of processes which have hidden dependencies with the initiator.

At the beginning of checkpointing, the initiator sends requests to every process $P_i \mid P_i \in set_t$.In this time, set_i is not constructed, so $set_t = \emptyset$. As a result, the invariant holds.

After receiving replies form the processes of set_t, a new set is constructed due the information carried by reply messages. If $set_t \neq \emptyset$ then the initiator send requests to every process in this set such that $P_i \notin set_t$.

At the end of checkpointing, no requests are generated and the invariant holds after checkpointing.

Since, on one hand, the number of processes in the system and the transmission delays of the messages are finite, and on the other hand, the initiator is the only responsible of sending requests (with finite time of sending requests and receiving replies), therefore, we deduce that our protocol terminates within a finite time. ■

5.5 HANDLING CONCURRENT INITIATORS

In Large distributed systems, it is more likely to have concurrent initiators. To handle this case, two basic approaches are proposed: minimal and maximal. In minimal approaches, a process takes a checkpoint in response to the first initiator and defers its response for the others. The obtained global checkpoint is the intersection of global checkpoints calculated by concurrent initiators; this global checkpoint is still consistent [126]. The minimal approach is adopted by [55]

In maximal approach, a process takes a checkpoint for each request sent by each initiator. As a result, the combined global checkpoint is the union of calculated global checkpoints which is also consistent[126]. This approach has been used by [56].

The minimal approach has the advantage of minimal tentative checkpoints while the maximal approach has the advantage of the most recent global checkpoint.

5.6 Simulation results and discussion

In our study, We adopt the simulation strategy to compare FNB protocol to CSNB protocol. We have integrated these protocols in the NS2 simulator by creating a new agent for each protocol (cc, h files). For this purpose, we have made the necessary changes and recompiled NS2.

5.6.1 Point-to-point environment

In our simulation, we consider that the nodes of the systems are connected by Ethernet Network ($100MB/s$). Then, each process has been assigned to each node. The sending and receiving events are randomly generated. the destination of a message is uniformly distributed random variable among all the processes

Table 5.1: Simulation parameters used for performance evaluation

	Number of processes (nb-proc)	Number of Initiations (nb-init)	Number of Messages (nb-msg)
Scenario 1	[20-100]	16	5000
Scenario 2	100	[10-50)	5000
Scenario 3	100	16	[2000-6000]

To evaluate the performance of the protocols, we use request number per initiation (nb-req), mutable checkpoints number (nb-m) and mean duration of first phase (MDFP) as performance metrics

We have executed our simulation under various scenarios and parameters summarized in Table 5.1.

In addition, we have executed the scenario 1 under 40GB model to show the impact of now-days network technologies on protocols performance (Figs 5.8(b) and 5.11(b))

5.6.1.1 Number of Requests

Figure 5.5 depicts the request number variation in terms of process number. When the number of processes increases, the number of requests also increases. The difference

between the two protocols increases with increasing process number. CSNB protocol has 150 requests more than FNB when the process number is 100.

We can see from this scenario that CSNB protocol suffers from duplicate request problem. When the number of processes is 30, CSNB has more than 120 requests which means that a process may receive more than 4 requests during one single initiation. We can see also that our FNB algorithm doesn't have duplicate requests; in each case the request number is smaller than process number.

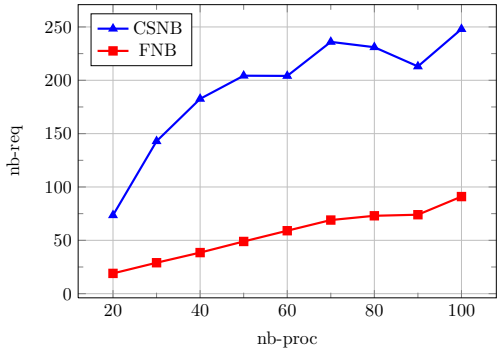


Figure 5.5: number of request vs number of processes

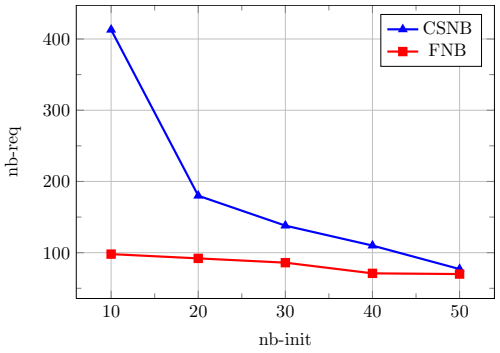


Figure 5.6: Number of request vs number of initiations

In both protocols, as shown in Fig 5.6, the increment of the initiation number implies the decreasing of the number of messages per checkpoint interval which results in the decrease of the number of requests. This is due to the decreasing of the amount of dependency information per checkpoint interval. Moreover, the number of requests in the FNB protocol is always lower than CSNB protocol.

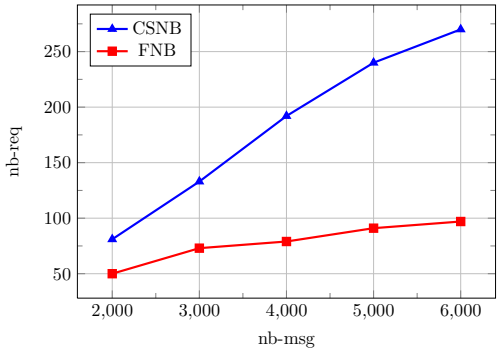


Figure 5.7: Number of request vs number of messages

With increasing message number, the request number of CSNB protocol drastically increases from 80 to 270 while the request number of FNB protocol increases from 50 to 97 (Figure 5.7). This can be explained by the fact that FNB does not induce duplicate requests. So, with our protocol the network is not congested with control messages.

In the light of Figure 5.6 and 5.7, we can conclude that FNB protocol is more efficient when we have a less number of dependencies per checkpoint interval.

Table 5.2: Overhead incurred by protocols with N=100 processes and the data payload is 3KB

nb-msg	Total number of requests		Total sizes of messages (KB)		Total sizes of requests (KB)		Total (KB)	Diff	Overhead ratio (%)	
	CSNB	FNB	CSNB	FNB	CSNB	FNB			CSNB	FNB
2000	1296	800	6032	6056	562	13	525		9,86%	1,14%
3000	2128	1168	9048	9084	923	18	868		10,74%	1,13%
4000	3072	1264	12064	12112	1333	20	1265		11,58%	1,09%
5000	3840	1456	15080	15140	1666	23	1583		11,58%	1,08%
6000	4320	1552	18096	18168	1875	25	1778		10,89%	1,06%

Total size of messages = (nb-msg message size)*
*Total size of requests =(total number of requests * request size)*
Total Diff=Total size(messages +requests)$_{CSNB}$ *-Total size(messages +requests)*$_{FNB}$

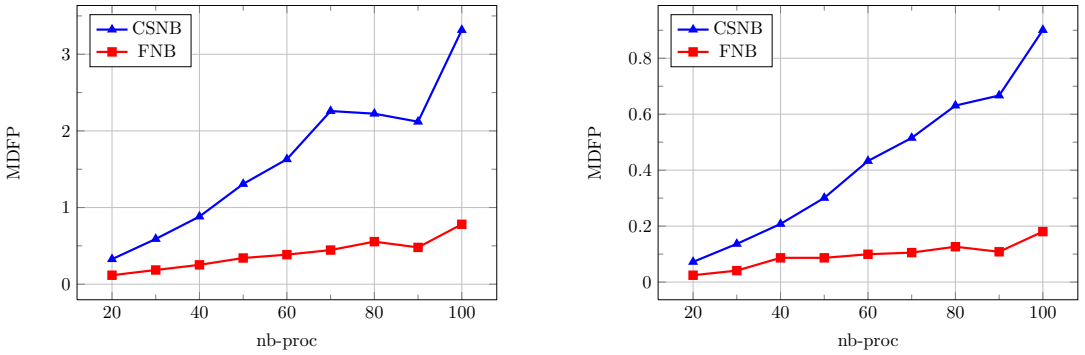
As each protocol uses a different technique to track dependency, it is important to calculate the overhead incurred by each protocol. In this regard, Table 5.2 gives an other view of protocols performance in terms of overhead (the two last columns).

First of all, as shown in Table 5.2, the total size of messages for FNB protocol, on one hand, is greater than that of CSNB protocol. On the other hand, FNB protocol has the lower total size of requests than the CSNB protocol. In addition, CSNB has the highest overhead ratio which is around 10%, whereas FNB's overhead ratio is around only 1%. In this case, CSNB protocol consumes more the bandwidth due to the request packet length and requests number. In comparison to CSNB, FNB protocol seems to have a little bandwidth consumption in spite of piggybacking of dependency vectors on computation messages

5.6.1.2 Duration of the first phase

In this section, we discuss the influence of the number of processes on first phase duration (MDFP). As shown in Figure 5.8, when the number of processes increases the MDFP for the two protocols also increases. Note also that when processes number becomes larger the difference between the two protocols becomes more obvious. This can be explained as follows. In our FNB protocol, the initiator sends requests simultaneously to the involved processes, and the request takes short paths to reach a destination due to piggybacking dependency information and the popular process techniques.

Under 40GB model, we can see a performance enhancement in terms of MDFP. As illustrated in Figure 5.8(b), MDFP(CSNB) does not exceed *1s* for *100 processes* while this duration exceeds 3s under 100MB model (Figure 5.8(a)). On other hand, MDFP(FNB) is



(a) Under 100MB (b) Under 40GB

FIGURE 5.8: Mean duration of first phase vs number of processes

about *0.2s* for *100 processes* (Figure 5.8(b)) which is much less than the duration under *100MB* for the same number of processes (see Figure 5.8(a)

As can be seen in Figure 5.9 increasing initiations number decreases the MDFP of the two protocols. In addition, FNB protocol has lower MDFP values, which means that FNB protocol performs well than CSNB.

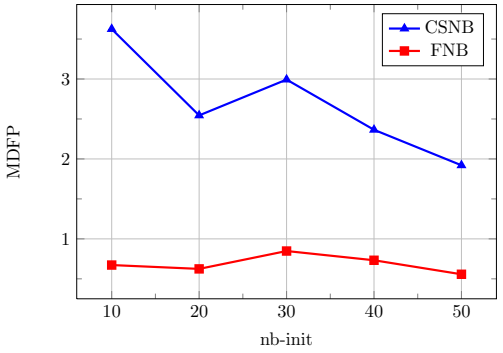


FIGURE 5.9: Mean duration of first phase vs number of initiations

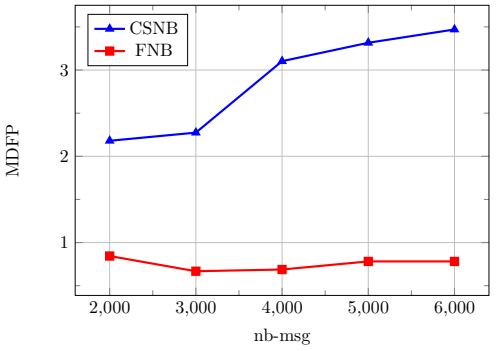


FIGURE 5.10: Mean duration of first phase vs number of messages

Besides the results shown in Figure 5.8 and 5.9, Figure 5.10 illustrate also the difference between the CSNB and FNB protocols. Note that in this case, CSNB takes 2s more than FNB protocol to accomplish the checkpointing first phase.

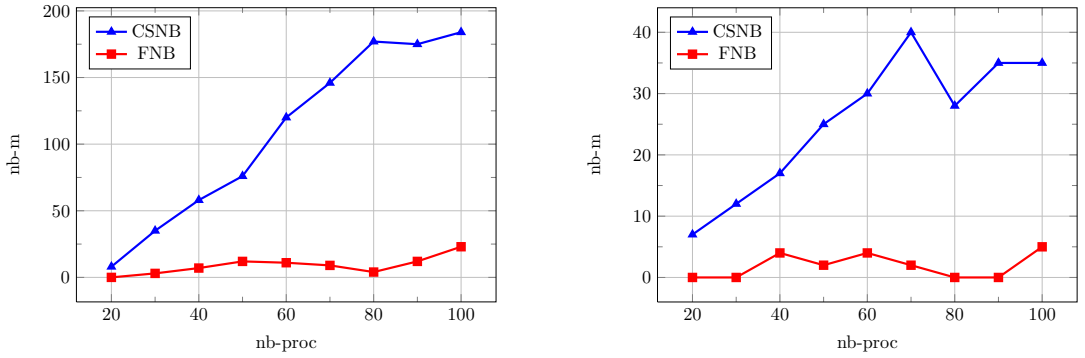
As a result, our FNB protocol is faster than CSNB protocol and it is more likely that the checkpointing in FNB protocol is not aborted due to the faults occurrence.

5.6.1.3 Number of mutable checkpoints

Figures 5.11, 5.12 and 5.13 show how different protocols fare in the number of mutable checkpoints. The results illustrated in Figure 5.11(a) show the increase of mutable

checkpoints number with the increment of processes number. Much more to our expectations, CSNB performs worst due to: i) longer paths taken by requests, ii) reception of computation messages carrying checkpointing information before receiving requests

We can point out that the MDFP has also an impact on mutable checkpoints number. this can be seen clearly under 40GB model (Figure 5.11(b). In a small duration, it is less likely that communication patterns involving mutable checkpoints to be established as mutable checkpoints are only needed during checkpointing process. As a result, 40GB model has an indirect impact on mutable checkpoints number reduction in comparison to 100MB model.



(a) Under 100MB

(b) Under 40GB

Figure 5.11: Number of mutable checkpoints vs number of processes

Figure 5.12 shows that our FNB outperforms CSNB protocol in the number of mutable checkpoints with a varying initiation number. Note that for 30 initiations the difference between the protocols is about 200 mutable checkpoints.

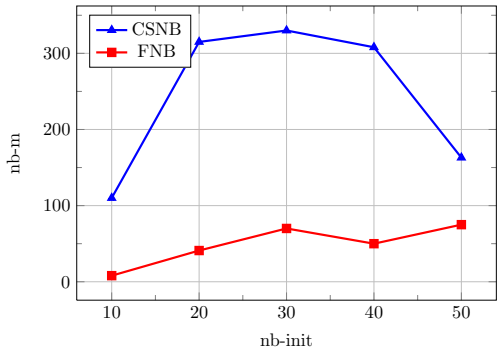


Figure 5.12: Number of mutable checkpoints vs initiations number

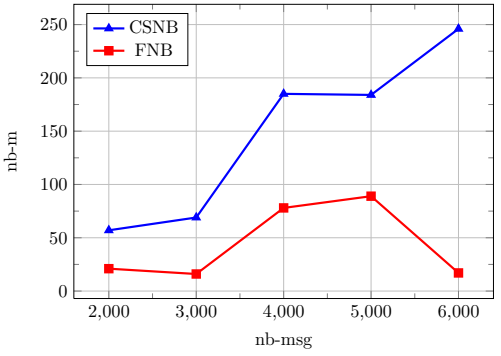


Figure 5.13: Number of mutable checkpoints vs number of messages

Finally, as one can see in Figure 5.13, the slope of the reduction in the number of mutable checkpoints for our FNB protocol is not as large as CSNB protocol. This can be explained as follows: as the number of messages increases, much more dependencies can

be established and can be tracked before initiating checkpointing. As a result, processes do not take mutable checkpoints because they may receive requests directly from the initiator.

5.6.2 Group communication environment

To evaluate CSNB and FNB protocols under group communication environment, we run the simulation for 100s. In this environment, processes are arranged into groups of 10 processes For intragroup communication, the destination of a message is a uniformly distributed random variable among processes of the same group. A process can communicate with processes from other groups, but with lower rates (the destination of a message is selected with a lower probability than a process in the same group).

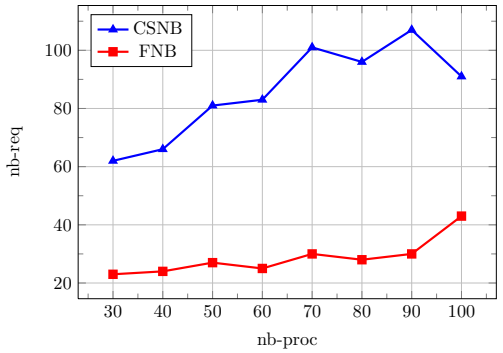


Figure 5.14: Number of requests vs number of processes

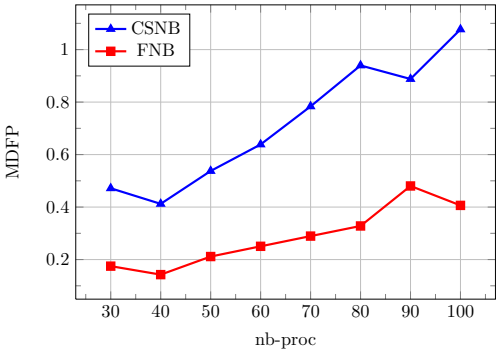


Figure 5.15: Mean first phase duration vs number of processes

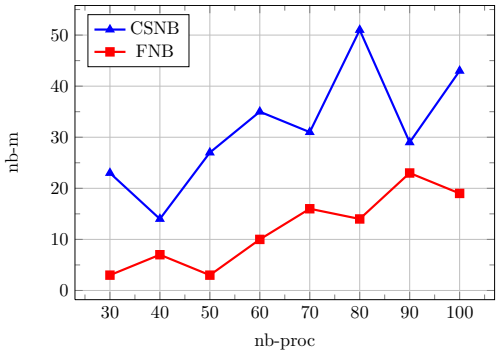


Figure 5.16: Number of mutable checkpoints vs number of processes

Figure 5.14 shows the variation of the request number in terms of process number. When nb-proc is equal to 100, the number of requests for FNB protocol is around 30 while for CSNB protocol is more than 130. Once again, we can clearly see that CSNB suffers from duplicate requests.

In contrast to point-to-point environment, FNB and CSNB protocols scale well. As one can see, the reduction of requests number is around 60% when we have an important number of groups. This is due to the nature of such an environment where it is more likely that the initiator has more dependencies with processes in its group.

Dividing the systems into groups has also a significant impact on the enhancement of protocols performance in terms of mutable checkpoints number and the first phase mean duration (Figure 5.15and 5.16).

5.6.3 Comparison between blocking and non-blocking protocols

In this section, we compare non-blocking protocol (FNB and CSNB) with the blocking protocol (CSB). We consider the number of synchronization messages (requests, replies and commits) and MDFP as performance metrics. We vary the number of processes from 20 to 100 and fix the messages number to 5000, and the initiation number to 16.

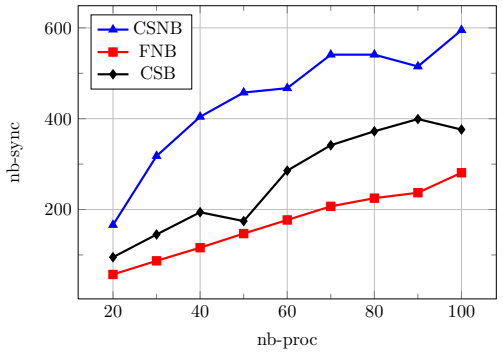


Figure 5.17: Number of synchronization messages vs number of processes

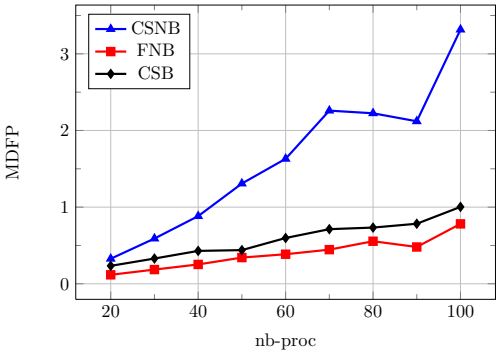


Figure 5.18: Mean first phase duration vs number of processes

Figure5.17 depicts the variation of synchronization messages number (nb-sync) as a function of processes number. When the number of processes increases,the number of synchronization messages increases also. We can see that FNB protocol behaves better than CSB protocol which is in turn behaves better than CSNB protocol. When the number of processes is 100, the difference between FNB protocol and CSB protocol is about 100 messages in favor of FNB protocol this is due to its reducing strategy of synchronization messages.

Figure 5.18 presents the measurement of MDFP time with a varying processes number. Once again, FNB protocol outperforms CSNB and CSB protocols. Despite the fact that CSB protocol is designed to reduce the checkpointing latency by tracking all dependencies before checkpointing initiation, it takes more time than FNB protocol to accomplish the checkpointing

In the light of the simulation study, we can conclude that FNB protocol outperforms CSNB and CSB protocols. In addition, FNB is more efficient when there is less dependency. Moreover, FNB protocol scales well in clustered systems.

5.7 CONCLUSION

Coordinated checkpointing is one of the most attractive domain in fault tolerance. As its name indicates, processes cooperate to determine a consistent global checkpoint either in blocking or non-blocking way. Since the blocking protocols suffer from blocking time overhead, non-blocking protocols seem to be the best choice. However, non-blocking protocols also face the problem of overhead due to the synchronization messages (requests, replies and commit messages). Such a problem can be solved by dependency tracking technique.

Due to communication patterns, dependency relation may be direct, transitive or hidden, Earlier works based on tracking direct dependencies shows its inefficiency, and the problem of overhead remains still unsolved. Thus, this chapter has presented a new coordinated non-blocking checkpointing protocol called FNB. During normal execution, FNB protocol tracks transitive dependencies which are appended to computation messages. Moreover, FNB protocol uses popular process technique which plays a key role in the checkpointing initiation. Thanks to these techniques, FNB protocol eliminates the problem of duplicated requests and reduces the number of mutable checkpoints. In addition, it has low checkpointing latency. Compared to other protocols, FNB protocol shows a good performance which is demonstrated through the simulation study.

Conclusion and Future Works

Checkpointing protocols provide application-transparent fault-tolerance to distributed systems, which are based on the periodic saving of system state in the stable storage. In the event of a failure, the system can resume its execution from a previously saved state instead of restarting from the beginning. In this regard, three main approaches are proposed: uncoordinated, communication-induced (CIC), and coordinated checkpointing.

Uncoordinated checkpointing class is characterized by the autonomy of processes in taking checkpoints, which is susceptible to entail the domino-effect problem. Coordinated class is considered as an alternative to uncoordinated one, where processes synchronize among them by the mean of control messages to construct a consistent global checkpoint. In CIC class, each process takes two types of checkpoints (basic and forced). Basic checkpoints are taken at process's pace while forced checkpoint are taken based on information piggybacked on computation messages to ensure the consistency.

Through this thesis, we have principally focused our efforts on CIC and coordinated classes. More precisely, we have tackled the problem of overhead incurred by Index based CIC protocols ensuring NZC property, CIC protocols ensuring RDT property and Non-blocking protocols, which direct us to study and propose new checkpointing protocols.

At first, we have extended (BCS, BCS-aftersend, LazyBCS, LazyMS, LazyBCS-aftersend) protocols by BRA protocol. BRA protocol ensures asynchronous recovery without blocking the application, and the existence of a recovery line consistent with the latest checkpoint of any process all the time. It also bounds the rollback propagation. These features allow us to study the behavior of index based CIC protocols under the failure-prone environment; We have shown that i) comparing checkpoint inducing conditions is not a sufficient criteria to decide which CIC protocol is the best. ii) the

timestamping function and skipping technique play a key role in index based CIC protocol design and the reduction of checkpoint induced during the checkpointing process, iii)Lazy indexing was shown to enlarge the recovery time and consequently, the waste is increased due to the amount of useful work to redo, iii) CIC protocols worsen their performance in large systems and are not suitable for application with a higher rate of failures, iv) quantifying the rollback is an important issue in protocols performance because rolling back to the most recent global checkpoint minimizes the amount of useful work wasted in the event of failures and helps to decide when the garbage collection should be invoked to reduce the stable storage contention.

Another contribution of this thesis is the introduction of two new RDT protocols called CSFDAS and CSRDTPartner. These protocols ensure the RDT property while keeping low overhead messages. In other words, messages carry only a constant size of control information due to the use of direct dependency clocks rather than using transitive dependency vectors, which is independent on the number of processes in the system. The simulation study shows that CSFDAS and CSRDTPartner have a good performance compared to other simulated RDT protocols (FDAS, BHMR, and RDTPartner). This is a surprising result since that CSFDAS and CSRDTPartner have weaker induction condition than the others. Thus, CSFDAS and CSRDTPartner provide best trade-offs between low overhead and performance, which make them best choice to achieve fault tolerance.

We have show that most of the search in the field of checkpointing concentrates on coordinated protocols. However, these protocols typically require the participation of all processes in the system, a large number of requests used for the synchronization and a large number of saved checkpoints; thus, such protocols are not efficient. Hence, we have presented our fast non-blocking coordinated checkpointing protocol called FNB, which is based on two strategies. The first technique is piggybacking dependency information on computation messages, which permit to track the transitive dependencies and gives maximum information about processes involved in checkpointing procedure to the initiator. Since processes may have hidden dependencies among themselves, piggybacking dependency information on reply messages has the goal of completing such list and gives full information to the initiator. Thereby, reducing the synchronization overhead. The second technique is imitating checkpointing by a specific process called popular process. Thanks to the second technique, the checkpointing operation is accelerated.

Compared to the previous works, FNB protocol reduces both the number of requests and that of checkpoints and minimizes the checkpointing latency. Besides, it incurs a low overhead. Moreover, FNB protocol is more efficient in the case of less dependency per checkpoint interval. In addition, FNB scales well in clustered system. Besides that,

it bears more advantages such as a reduced likelihood of the fault occurrence during the checkpointing process and a no flooded network by control messages.

Future Works

This thesis has covered several aspect of coordinated and CIC protocols. However, this research gives rise to some open lines and future work which are:

- **Coordinated checkpointing with concurrent initiators**
  Checkpointing protocols with concurrent initiators proposed so far have been shown to have a large overhead even using union or intersection approaches. The question is how we can benefit from concurrent initiators to design more sophisticated protocols while keeping a low overhead.
- **Combination of coordinated checkpointing and logging techniques**
  Message Logging provides a tool to replay the most frequent of non-deterministic events: message receptions. We can exploit message logging to advance more the consistent global checkpoint.
- **Combination of FNB protocol with time-based coordinated checkpointing protocols**
  In time-based coordinated checkpointing, processes take their checkpoints at previously agreed dates..Similarly, we can take advantage from this feature by taking mutable checkpoints independently at checkpoint intervals.
- **Study of the correlated failures case**
  In this thesis, we have only studied the behavior of Index based CIC protocols in the case of one failure at time. The question is: do checkpointing protocols preserve their performance in the case of correlated failures?. What should we change to enhance more checkpointing protocols? The research in this direction will strengthen the value of presented work.
- **Fault tolerance and energy consumption**
  Together with fault tolerance, energy consumption is expected to be a major challenge. Such a challenge opens another direction for future work, which is the study of checkpointing techniques impact on energy consumption. As FNB protocol does not require the participation of all processes in the determination of the global checkpoint, it is interesting to examine the effect of this feature in the energy consumption reduction.

Appendix A

LogP model overview

LogP is a model designed for parallel computation analysis [137]. The main parameters of LogP are described in Table A.1. The basic model assumes all messages are of a small size. Thus, we use its extension for long messages LogGP model [138].

Sending a K bytes message (Figure A.1) with LogGP model involves

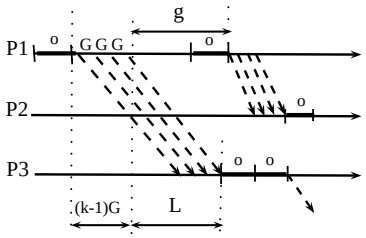


Figure A.1: Sending and receiving messages under the LogGP model.

- o cycles to get the first byte into the network.
- o+(K-1)G cycles for the other subsequent bytes.
- L cycles for traveling each byte through the network.
- other o cycles to be spent at receiving processor
- g cycles to be spent before sending the next message.

The entire time for sending K bytes message is equal to $2o + (K-1)G + L$.

Table A.1: List of notations

	LogGP model parameters
L	Upper bound for the *latency*, incurred in sending a message from its source processor to its target processor
o	*Overhead*, defined as the length of the time that a process is engaged in the transmission or reception of a message.
g	*Gap*, defined as the minimum time interval between consecutive message transmissions or receptions
G	*Gap* per bytes for long messages
P	number of processes

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, *et al.*, *Fundamental concepts of dependability*. University of Newcastle, Computing Science Newcastle , UK, 2001. (Cited on pages 1 and 12.)

[2] E. N. M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput Surveys*, vol. 34, no. 3, pp. 375–408, 2002. (Cited on pages 2, 4, 28, 43 and 62.)

[3] B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Trans Software Eng*, vol. 1, no. 2, pp. 220–232, 1975. (Cited on page 2.)

[4] R. E. Strom and S. Yemini, "Optimistic recovery in distributed systems," *Trans. Comput. Systems*, vol. 3, no. 3, pp. 204–226., 1985. (Cited on pages 2, 53 and 55.)

[5] A. Borg, J. Baumbach, and S. Glazer, "A message system supporting fault tolerance," in *Symp on Operating Systems Principles (ACM SIGOPS)*, pp. 90–99, 1983. (Cited on page 2.)

[6] G. Bosilca, A. Bouteiller, S. Cappello, F.and Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *ACM/IEEE conf on Supercomputing, ser. Supercomputing '02*, (Los Alamitos, CA, USA: IEEE Computer Society Press), 2002. (Cited on page 2.)

[7] A. Bouteiller, T. Cappello, F.and Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *ACM/IEEE conference on Supercomputing*, (New York, NY, USA), 2003. (Cited on pages 2 and 52.)

[8] Q. Jiang, Y. Luo, and D. Manivannan, "An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems," *J. Parallel Distrib. Comput*, vol. 68, no. 12, pp. 1575–1589, 2008. (Cited on

pages 2, 54 and 55.)

[9] A. P. Sistla and J. L. Welch, "Efficient distributed recovery using message logging," *IEEE/ACM Trans. Networking*, vol. 4, no. 5, pp. 785–795, 1996. (Cited on pages 2, 54 and 55.)

[10] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv*, vol. 37, no. 1, pp. 42–81, 2005. (Cited on page 2.)

[11] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998. (Cited on page 2.)

[12] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *IEEE Int Conf on Cluster Comput*, pp. 115–124, 2004. (Cited on pages 2, 57 and 58.)

[13] D. Manivannan and M. Singhal, "A low-overhead recovery technique using quasi-synchronous checkpointing," in *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pp. 100–107, IEEE, 1996. (Cited on pages 2, 47, 50 and 94.)

[14] D. Manivannan and M. Singhal, "Asynchronous recovery without using vector timestamps," *Journal of Parallel and Distributed Computing*, vol. 62, no. 12, pp. 1695–1728, 2002. (Cited on pages 4, 44, 47, 50, 76, 77, 79 and 82.)

[15] J.-M. Helary, A. Mostefaoui, and M. Raynal, "Communication-induced determination of consistent snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, pp. 865–877, Sep 1999. (Cited on pages 44 and 47.)

[16] J. M. Hélary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal, "Communication-based prevention of useless checkpoints in distributed computations," *Distrib Comput*, vol. 13, no. 1, pp. 29–43, 2000. (Cited on pages 44, 47, 50, 66 and 76.)

[17] Y. Luo and D. Manivannan, "FINE: A Fully Informed aNd Efficient communication-induced checkpointing protocol for distributed systems," *J Parallel Distributed Comput*, vol. 69, no. 2, pp. 153–167, 2009. (Cited on pages 2, 44, 45, 48 and 50.)

[18] Y. M. Wang, "Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints," *IEEE Trans on Computers*, vol. 46, no. 4, pp. 456–468, 1997. (Cited

on pages 2, 5, 22, 45, 48, 50, 98, 99 and 105.)

[19] R. Baldoni, J.-M. Hélary, A. Mostefaoui, and M. Raynal, "A communication-induced checkpointing protocol that ensures rollback-dependency trackability," in *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pp. 68–77, IEEE, 1997. (Cited on pages 5, 46, 48, 98, 100 and 105.)

[20] I. C. Garcia, G. M. Vieira, and L. E. Buzato, "RDT-Partner: An efficient checkpointing protocol that enforces rollback-dependency trackability," in *Proc. 19 ^{t}h Brazilian Symp. Computer Networks*, 2001. (Cited on pages 46, 48 and 98.)

[21] I. C. Garcia and L. E. Buzato, "An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pp. 126–135, IEEE, 2004. (Cited on pages 2, 5, 46, 48, 50, 98, 105 and 106.)

[22] G. Cao and M. Singhal, "Checkpointing with mutable checkpoints," *Theoretical Computer Science*, vol. 290, no. 2, pp. 1127–1148, 2003. (Cited on pages 3, 31 and 40.)

[23] G. Li and L. Shu, "Design and Evaluation of a Low-Latency Checkpointing Scheme for Mobile Computing Systems," *The Computer Journal.*, vol. 49, pp. 527–540, 2006. (Cited on pages 31 and 40.)

[24] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery," *IEEE Trans.in Software Engenering*, vol. 13, no. 1, pp. 23–31, 1987. (Cited on pages 3, 30 and 40.)

[25] T. C. Sakata and I. C. Garcia, "Non-Blocking Synchronous Checkpointing Based on Rollback-Dependency Trackability," in *25th IEEE Symp Reliable Distributed Systems*, pp. 4–11, oct. 2006. (Cited on pages 3, 37 and 40.)

[26] P. Kumar and A. Khunteta, "A Minimum-Process Coordinated Checkpointing Protocol For Mobile Distributed System," *IJCSI Internat. J. Computer Science Issues*, vol. 7, no. 3, 2010. (Cited on pages 3 and 131.)

[27] D. Briatico, A. Ciuffoletti, and L. Simoncini, "A Distributed Domino-Effect free recovery Algorithm.," in *Symposium on Reliability in Distributed Software and Database Systems*, vol. 84, pp. 207–215, 1984. (Cited on pages 4, 44, 47, 50 and 68.)

[28] G. M. Vieira, I. C. Garcia, and L. E. Buzato, "Systematic analysis of index-based

checkpointing algorithms using simulation," in *Proceedings of IX Brazilian Symposium on Fault-Tolerant Computing*, pp. 31–41, 2001. (Cited on pages 4, 44, 45, 47, 49, 50, 67, 68 and 72.)

[29] Z. Abdelhafidi, M. Djoudi, and M. B. Yagoubi, "An improved schema of coordinated checkpointing protocol for distributed systems based on Popular process," in *12^{th} Internat Conf Innovations in Information Technology (IIT)*, pp. 367–372, march 2012. (Cited on page 5.)

[30] Z. Abdelhafidi, M. Djoudi, N. Lagraa, and M. B. Yagoubi, "FNB: Fast Non-Blocking Coordinated Checkpointing for Distributed Systems," *Theory of Computing Systems*, vol. 57, no. 2, pp. 397–425, 2015. (Cited on page 5.)

[31] A. Tanenbaum and M. V. Steen, *Distributed Systems principals and Paradigms*, vol. 40. Pearson Prentice Hall, Second ed., 2001. (Cited on page 8.)

[32] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems Concepts and Design*. Fifth ed., 2005. (Cited on pages xix, 8 and 10.)

[33] A. D. Kshemkalyani and M. Singhal, *Distributed Computing Principles, Algorithms, and Systems*. Cambridge University Press, 2008. (Cited on pages 8 and 9.)

[34] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 241–299, 2000. (Cited on page 13.)

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005. (Cited on page 13.)

[36] W. Bartlett and L. Spainhower, "Commercial fault tolerance: A tale of two systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 87–96, 2004. (Cited on page 13.)

[37] I. Koren and C. M. Krishna, *Fault-tolerant systems*. Morgan Kaufmann, 2010. (Cited on page 13.)

[38] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991. (Cited on page 13.)

[39] S. Poledna, "Replica determinism in distributed real-time systems: A brief survey," *Real-Time Systems*, vol. 6, no. 3, pp. 289–316, 1994. (Cited on page 14.)

[40] R. H. B. Netzer and J. Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165–169, 1995. (Cited on pages 17, 19, 21, 25 and 44.)

[41] R. Baldoni, J.-M. Hélary, and M. Raynal, "Rollback-dependency trackability: A minimal characterization and its protocol," *Information and Computation*, vol. 165, no. 2, pp. 144–173, 2001. (Cited on pages xvi, 23, 47, 50, 100, 101, 103, 104 and 105.)

[42] Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," in *In 1984 International Conference on Parallel Processing*, Citeseer, 1984. (Cited on pages 30 and 40.)

[43] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985. (Cited on pages 33, 38 and 40.)

[44] T. H. Lai and T. H. Yang, "On distributed snapshots," *Information Processing Letters*, vol. 25, no. 3, pp. 153–158, 1987. (Cited on page 34.)

[45] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.," *J. Parallel Distrib. Comput.*, vol. 18, no. 4, pp. 423–434, 1993. (Cited on pages 34 and 40.)

[46] R. Garg, V. K. Garg, and Y. Sabharwal, "Efficient Algorithms for Global Snapshots in Large Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 21, no. 5, pp. 620–630, 2010. (Cited on pages 35 and 40.)

[47] A. Kshemkalyani, "Fast and Message-Efficient Global Snapshot Algorithms for Large-Scale Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 21, no. 9, pp. 1281–1289, 2010. (Cited on pages 35 and 40.)

[48] J. Tsai, "Flexible Symmetrical Global-Snapshot Algorithms for Large-Scale Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 24, no. 3, pp. 493–505, 2013. (Cited on pages 35 and 40.)

[49] Z. Tong, R. Y. Kain, and W. Tsai, "Rollback recovery in distributed systems using loosely synchronized clocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 246–251, 1992. (Cited on pages 37, 38 and 41.)

[50] S. Neogy, A. Sinha, and P. Das, "Distributed checkpointing using synchronized clocks," in *Computer Software and Applications Conference, 2002. COMPSAC 2002.*

Proceedings. 26th Annual International, pp. 199–204, 2002. (Cited on pages 37, 38 and 41.)

[51] A. K. Singh, "On mobile checkpointing using index and time together," *World Academy Science, Engineering and Technology*, vol. 26, pp. 144–151, 2007. (Cited on pages 38 and 41.)

[52] N. Neves and W. K. Fuchs, "Coordinated checkpointing without direct coordination," in *Proceedings. IEEE International Computer Performance and Dependability Symposium, IPDS'98.*, pp. 23–31, IEEE, 1998. (Cited on pages 38 and 41.)

[53] J. Surender, S. Arvind, K. Anil, and S. Yashwant, "Low Overhead Time Coordinated Checkpointing Algorithm for Mobile Distributed Systems," in *Computer Networks & Communications (NetCom)*, pp. 173–182, Springer, 2013. (Cited on pages 38 and 41.)

[54] C.-Y. Lin, S.-C. Wang, and S.-Y. Kuo, "An efficient time-based checkpointing protocol for mobile computing systems over mobile IP," *Mobile Networks and Applications*, vol. 8, no. 6, pp. 687–697, 2003. (Cited on pages 38 and 41.)

[55] M. Spezialetti and P. Kearns, "Efficient distributed snapshots," in 6^{th} *Internat. Conf. on Distributed Computing Systems*, (Boston), pp. 382–388, 1986. (Cited on pages 38, 41 and 140.)

[56] R. Prakash and M. Singhal, "Maximal global snapshot with concurrent initiators," in 6^{th} *IEEE Symposium on Parallel and Distributed Processing*, pp. 344–351, IEEE Comput. Soc. Press, 1994. (Cited on pages 38, 39, 41 and 140.)

[57] P. S. Mandal and K. Mukhopadhyaya, "Concurrent checkpoint initiation and recovery algorithms on asynchronous ring networks," *Journal of Parallel and Distributed Computing*, vol. 64, pp. 649–661, May 2004. (Cited on pages 38 and 41.)

[58] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: models, characterization, and classification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 703–713, 1999. (Cited on pages 43, 44 and 68.)

[59] J. Tsai, "An efficient index-based checkpointing protocol with constant-size control information on messages," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 4, pp. 287–296, 2005. (Cited on pages 45, 47 and 50.)

[60] J. Tsai and J.-W. Lin, "On the fully-informed communication-induced checkpointing protocol," in *11^th Pacific Rim International Symposium on Dependable Computing, 2005. Proceedings*, pp. 8–16, IEEE, 2005. (Cited on pages 45, 47 and 50.)

[61] Y. Luo and D. Manivannan, "Theoretical and experimental evaluation of communication-induced checkpointing protocols in F_E and F_{Lazy-E} families," *Performance Evaluation*, vol. 68, no. 5, pp. 429–445, 2011. (Cited on pages 45, 48, 49, 50, 84 and 91.)

[62] Y.-W. Ci, Z. Zhang, D.-C. Zuo, Z.-B. Wu, and X.-Z. Yang, "A multi-cycle checkpointing protocol that ensures strict 1-rollback," *Information Processing Letters*, vol. 112, no. 20, pp. 788–793, 2012. (Cited on pages 45, 48 and 50.)

[63] F. Quaglia, R. Baldoni, and B. Ciciani, "On the No-Z-cycle property in distributed executions," *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 400–427, 2000. (Cited on pages 45, 47 and 50.)

[64] I. C. Garcia, L. E. Buzato, I. C. Garcia, and L. E. Buzato, "Checkpointing using local knowledge about recovery lines," *Tech. Rep. IC-99-22*, 1999. (Cited on pages 45 and 47.)

[65] J. Wu and D. Manivannan, "An enhanced model-based checkpointing protocol for preventing useless checkpoints," *J Parallel, Emergent and Distributed Systems*, vol. 24, no. 5, pp. 383–406, 2009. (Cited on pages 45, 48 and 50.)

[66] R. Baldoni, J.-M. Hélary, and M. Raynal, "Rollback-dependency trackability: visible characterizations," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pp. 33–42, ACM, 1999. (Cited on page 46.)

[67] R. Baldoni, J.-M. Hlary, and M. Raynal, "Rollback-dependency trackability: A minimal characterization and its protocol," *Information and Computation*, vol. 165, no. 2, pp. 144 – 173, 2001. (Cited on pages 46 and 48.)

[68] J. Tsai, "Systematic comparisons of RDT communication-induced checkpointing protocols," in *Proceedings. 10th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 66–75, IEEE, 2004. (Cited on pages 46, 49, 105 and 107.)

[69] R. Baldoni, F. Quaglia, and P. Fornara, "An index-based checkpointing algorithm for autonomous distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 181–192, 1999. (Cited on pages 47 and 50.)

[70] J. Tsai, Y.-M. Wang, and S.-Y. Kuo, "Evaluations of domino-free communication-induced checkpointing protocols," *Inf. Process. Lett.*, vol. 69, pp. 31–37, jan 1999. (Cited on pages 49, 84 and 91.)

[71] J. Tsai and J.-W. Lin, "On characteristics of def communication-induced checkpointing protocols," in *Proceedings. Pacific Rim International Symposium on Dependable Computing,*, pp. 29–36, IEEE, 2002. (Cited on pages 49, 84 and 91.)

[72] J. Tsai, "Performance comparisons of index-based communication-induced checkpointing protocols," *Journal of the Chinese Institute of Engineers*, vol. 29, no. 6, pp. 1113–1118, 2006. (.)

[73] Z. Abdelhafidi, M. Djoudi, M. Yagoubi, A. Alliliche, and M. Kebir, "Simulation by NS2 of Checkpointing Protocols Satisfying NZC Property based on indexing strategies," in *Confrence Internationale des Technologies de lInformation et de la Communication (CITIC)*, march 2009. (.)

[74] G. M. Vieira and L. E. Buzato, "Distributed checkpointing: Analysis and benchmarks," in *Proceedings of the 24th Brazilian Symposium on Computer Networks, SBRC*, vol. 6, 2006. (Cited on page 49.)

[75] J. Tsai, S.-Y. Kuo, and Y.-M. Wang, "Theoretical analysis for communication-induced checkpointing protocols with rollback-dependency trackability," *Transactions onParallel and Distributed Systems*, vol. 9, no. 10, pp. 963–971, 1998. (Cited on page 49.)

[76] J. Tsai, "On properties of RDT communication-induced checkpointing protocols," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 8, pp. 755–764, 2003. (Cited on page 49.)

[77] J. Tsai, S.-Y. Kuo, and Y.-M. Wang, "More Properties of Communication-Induced Checkpointing Protocols with Rollback-Dependency Trackability," vol. 257, pp. 239–257, 2005. (Cited on page 49.)

[78] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg, "Quantifying rollback propagation in distributed checkpointing," *Journal of Parallel and Distributed Computing*, vol. 64, no. 3, pp. 370 – 384, 2004. (Cited on pages 49 and 50.)

[79] A. Agbaria and R. Friedman, "Model-based performance evaluation of distributed checkpointing protocols," *Perform Evaluation*, vol. 65, no. 5, pp. 345–365, 2008. (Cited on page 50.)

[80] B. Bhargava and S. R. Lian, "Independent checkpointing and concurrent rollbackfor recovery-An optimistic approach.," in *Seventh Symposium on Reliable Distributed Systems*, pp. 3–12, 1988. (Cited on page 50.)

[81] Y. M. Wang, *Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems*. PhD thesis, University of Illinois, Department of Computer Science, 1993. (Cited on page 50.)

[82] E. Feller, J. Mehnert-Spahn, M. Schoettner, and C. Morin, "Independent checkpointing in a heterogeneous grid environment," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 163–170, 2012. (Cited on page 50.)

[83] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under UNIX," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 1–24, 1989. (Cited on page 51.)

[84] D. B. Johnson and W. Zwaenepoel, "Sender-based message logging," in *Proc. of Fault Tolerant Computing Systems*, pp. 14–19, 1987. (Cited on pages 51 and 52.)

[85] J. Xu, R. H. Netzer, and M. Mackey, "Sender-based message logging for reducing rollback propagation," in *IEEE Symposium on Parallel and Distributed Processing*, pp. 602–602, IEEE Computer Society, 1995. (Cited on page 52.)

[86] J.-m. Yang, K. F. Li, W.-w. Li, and D.-f. Zhang, "Trading off logging overhead and coordinating overhead to achieve efficient rollback," no. August 2008, pp. 819–853, 2009. (Cited on pages 52 and 53.)

[87] E. Meneses, C. L. Mendes, and L. V. Kalé, "Team-based message logging: Preliminary results," in *2010 10 ^{t}h IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 697–702, IEEE, 2010. (Cited on pages 52 and 53.)

[88] H. Meyer, D. Rexachs, and E. Luque, "Hybrid Message Logging. Combining advantages of Sender-based and Receiver-based Approaches," *Procedia Computer Science*, vol. 29, pp. 2380–2390, 2014. (Cited on pages 52 and 53.)

[89] J. Ahn, "How to lift SBML's limitation on fault-tolerance based on group-based communication links (WIP)," in *Proceedings of the 2014 Summer Simulation Multiconference*, p. 68, 2014. (Cited on pages 52 and 53.)

[90] S. L. Peterson and P. Kearns, "Rollback based on vector time," in *12^{t}h Symposium onReliable Distributed Systems, 1993.*, pp. 68–77, IEEE, 1993. (Cited on pages 54

and 55.)

[91] T. Park, N. Woo, and H. Y. Yeom, "An efficient optimistic message logging scheme for recoverable mobile computing systems," *IEEE Transactions on Mobile Computing*, vol. 1, no. 4, pp. 265–277, 2002. (Cited on pages 54 and 55.)

[92] O. P. Damani, Y.-M. Wang, and V. K. Garg, "Distributed recovery with K-optimistic logging," *Journal of Parallel and Distributed Computing*, vol. 63, no. 12, pp. 1193–1218, 2003. (Cited on pages 54 and 55.)

[93] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and check-pointing," *Journal of Algorithms*, vol. 11, no. 3, pp. 462–491, 1990. (Cited on pages 54 and 55.)

[94] T. Ropars and C. Morin, "O2P : An Extremely Optimistic Message Logging Protocol," *INRIA Research Report 635*, November 2007. (Cited on pages 54 and 55.)

[95] R. Wang, B. Salzberg, and D. Lomet, "Log-based middleware server recovery with transaction support," *The VLDB Journal*, vol. 20, no. 3, pp. 347–370, 2011. (Cited on pages 54 and 55.)

[96] Y. Luo and D. Manivannan, "HOPE: A Hybrid Optimistic checkpointing and selective Pessimistic mEssage logging protocol for large scale distributed systems," *Future Generation Computer Systems*, vol. 28, pp. 1217–1235, Oct. 2012. (Cited on pages 54 and 55.)

[97] E. N. M. Elnozahy, D. B. Johnson, and W. Zwarnepoel, "The performance of consistent checkpointing,," in 11^{th} *Symp Reliable Distributed Systems*, pp. 39–47, 1992. (Cited on pages 56, 57 and 58.)

[98] L. Alvisi, *Understanding the message logging paradigm for masking process crashes*. PhD thesis, Cornell University, 1996. (Cited on pages 56 and 58.)

[99] L. Alvisi, K. Bhatia, and K. Marzullo, "Causality tracking in causal message-logging protocols," *Distributed Computing*, vol. 15, pp. 1–15, jan 2002. (Cited on pages 56 and 58.)

[100] K. Bhatia, K. Marzullo, and L. Alvisi, "Scalable causal message logging for wide-area environments," *Concurrency Computation Practice and Experience*, vol. 15, no. 10, pp. 873–889, 2003. (Cited on pages 57 and 58.)

[101] B. L. B. Lee, T. P. T. Park, H. Yeom, and Y. C. Y. Cho, "An efficient algorithm for causal message logging," in *Proceedings Seventeenth IEEE Symposium on Reliable*

Distributed Systems (Cat. No.98CB36281), 1998. (Cited on pages 57 and 58.)

[102] J. Ahn, S. G. Min, and C. S. Hwang, "A causal message logging protocol for mobile nodes in mobile computing systems," *Future Generation Computer Systems*, vol. 20, no. 4, pp. 663–686, 2004. (Cited on pages 57 and 58.)

[103] Y. W. Ci, Z. Zhang, D. C. Zuo, and X. Z. Yang, "Message fragment based causal message logging," *Journal of Parallel and Distributed Computing*, vol. 69, no. 11, pp. 915–921, 2009. (Cited on pages 57 and 58.)

[104] Y.-W. Ci, Z. Zhang, D.-C. Zuo, Z.-B. Wu, and X.-Z. Yang, "Dependency mining-based causal message logging," *Information Processing Letters*, vol. 110, no. 5, pp. 182–187, 2010. (Cited on pages 57 and 58.)

[105] E. Meneses and L. V. Kalé, "CAMEL: collective-aware message logging," *The Journal of Supercomputing*, pp. 1–23, 2015. (Cited on pages 57 and 58.)

[106] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974. (Cited on page 59.)

[107] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, pp. 303–312, Feb. 2006. (Cited on pages 59 and 133.)

[108] Z. Xu, C. Men, W. Li, and X. Li, "Checkpoint scheduling model for optimality," *Inf. Process. Lett.*, vol. 111, no. 19, pp. 979–984, 2011. (Cited on page 59.)

[109] R. Subramaniyan, E. Grobelny, S. Studham, and A. D. George, "Optimization of checkpointing-related I/O for high-performance parallel and distributed computing," *The Journal of Supercomputing*, vol. 46, pp. 150–180, dec 2007. (Cited on page 60.)

[110] S. Arunagiri, J. T. Daly, and P. J. Teller, "Modeling and analysis of checkpoint i/o operations," in *Analytical and Stochastic Modeling Techniques and Applications*, pp. 386–400, Springer, 2009. (Cited on page 60.)

[111] S.-H. Lim, S. W. Lee, B.-H. Lee, S. Lee, and H. W. Lee, "Energy-aware checkpoint intervals in error-prone mobile networks," in *Proceedings of the 6 ^{t}h International Conference on Queueing Theory and Network Applications - QTNA '11*, pp. 128–133, ACM Press, 2011. (Cited on page 60.)

[112] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pp. 9–9, Nov 2005. (Cited on page 60.)

[113] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2009. (Cited on page 60.)

[114] N. Hyochang, K. Jong, S. J. Hong, and L. Sunggu, "Probabilistic checkpointing," *IEICE Transactions on Information and Systems*, vol. 85, no. 7, pp. 1093–1104, 2002. (Cited on page 60.)

[115] K. B. Ferreira, R. Riesen, P. Bridges, D. Arnold, and R. Brightwell *Future Generation Computer Systems*, vol. 30, no. 1, pp. 66–77, 2014. (Cited on page 60.)

[116] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998. (Cited on page 60.)

[117] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *Proceedings of the 9^{t}h ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 213–223, ACM, 2005. (Cited on page 60.)

[118] J. S. Plank, "A new MDS erasure code for RAID-6," *Technical Report CS-07-602*, 2007. (Cited on page 60.)

[119] Z. Chen and J. Dongarra, "A scalable checkpoint encoding algorithm for diskless checkpointing," in *High Assurance Systems Engineering Symposium*, pp. 71–79, IEEE, 2008. (Cited on page 60.)

[120] D. Hakkarinen, S. Member, Z. Chen, and S. Member, "Multilevel Diskless Checkpointing," *IEEE Trans on Comput*, vol. 62, no. 4, pp. 772–783, 2013. (Cited on page 60.)

[121] J. F. Chiu and W. H. Hao, "Mutual-aid: Diskless checkpointing scheme for tolerating double faults," *10 ^{t}h IEEE International Conference on High Performance Computing and Communications, HPCC 2008*, pp. 540–547, 2008. (Cited on page 60.)

[122] G. M. Chiu and J. F. Chiu, "A new diskless checkpointing approach for multiple processor failures," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4, pp. 481–493, 2011. (.)

[123] J.-F. Chiu, "Double Mutual-Aid Checkpointing for Fast Recovery," in *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 1015–1020, 2012. (Cited on page 60.)

[124] J. Helary, a. Mostefaoui, and M. Raynal, "Interval Consistency of Asynchronous Distributed Computations," *J. Comput. Syst. Sci.*, vol. 64, 2002. (Cited on pages 65 and 66.)

[125] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978. (Cited on page 66.)

[126] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989. (Cited on pages 66, 113 and 140.)

[127] C. J. Fidge, *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science, 1987. (Cited on pages 66 and 113.)

[128] J.-M. Hélary, A. Mostéfaoui, and M. Raynal, "Virtual precedence in asynchronous systems: Concept and applications," in *11th International Workshop on Distributed Algorithms*, pp. 170–184, Septemebr 1997. (Cited on page 66.)

[129] M. Raynal, *Distributed Algorithms for Message-Passing Systems*. Springer Berlin Heidelberg, 2013. (Cited on page 76.)

[130] L. Alvisi, S. Rao, S. A. Husain, A. De Mel, and E. Elnozahy, "An analysis of communication-induced checkpointing," in *International Symposium on, Fault-Tolerant Computing*, p. 242, IEEE, 1999. (Cited on pages 84 and 91.)

[131] I. C. Garcia and L. Buzzato, "On the minimal characterization of the rollback-dependency trackability property," in *Distributed Computing Systems, 2001. 21st International Conference on.*, pp. 342–349, IEEE, 2001. (Cited on page 104.)

[132] J. Fowler and W. Zwaenepoel, "Causal distributed breakpoints," in *Distributed*

Computing Systems, 1990. Proceedings., 10th International Conference on, pp. 134–141, IEEE, 1990. (Cited on page 113.)

[133] G. M. Vieira and L. E. Buzato, "Chksim: A distributed checkpointing simulator," *Technical Report IC-05-034*, 2005. (Cited on page 119.)

[134] A. Khunteta, P. Sharma, and R. Garg, "New & efficient low overheads algorithm for mobile distributed systems," in *International Conference & Workshop on Emerging Trends in Technology - ICWET '11*, (New York, New York, USA), pp. 447–450, ACM Press, 2011. (Cited on page 132.)

[135] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010. (Cited on page 134.)

[136] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Euro-Par 2011: Parallel Processing Workshops*, pp. 302–311, Springer, 2012. (Cited on page 134.)

[137] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Fourth ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, (San Diego, California, USA), pp. 1–12, 1993. (Cited on page 153.)

[138] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the LogP model for parallel computation," *Journal of parallel and distributed computing*, vol. 44, no. 1, pp. 71–79, 1997. (Cited on page 153.)