Instituto Nacional de Astrofísica, Óptica y Electrónica

Algoritmo de *checkpointing* de comunicación-inducida para sistemas heterogéneos

por

M. en C. Alberto Calixto Simón

Tesis sometida como requisito para obtener el grado de

DOCTOR EN CIENCIAS EN EL ÁREA DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Supervisada por:

Dr. Saúl Eduardo Pomares Hernández

Tonantzintla, Puebla,
Diciembre 2013

Resumen

La necesidad de resolver problemas complejos eficientemente nos ha forzado a combinar diversos ambientes de cómputo (sistemas heterogéneos), sin embargo esta solución adoptada por muchos sistemas acarrea otros problemas tal como el problema de tolerancia a fallas. *Checkpointing* es una técnica eficiente para la tolerancia a fallas en sistemas distribuidos y paralelos, esta es utilizada además para solucionar un amplio rango de problemas en sistemas distribuidos y paralelos, tales como: la depuración de software, balance de carga, migración de procesos, entre otros. Dentro de los algoritmos de *checkpointing*, los algoritmos de *checkpointing* de comunicación inducida (CIC) se caracterizan por su bajo *overhead*, generación de *checkpoints* asíncronos y eliminación del efecto dominó. Para lograr esto, los algoritmos CIC acarrean información en los mensajes de las aplicaciones y generan *checkpoints* forzados cuando detectan patrones potencialmente peligrosos (e.g. *z-paths*). Las principales desventajas de los algoritmos CIC son el *overhead* por mensaje y el *overhead* de almacenamiento inducido (cantidad de *checkpoints* forzados).

En está investigación exponemos un nuevo algoritmo de comunicación inducida de *checkpointing* HSDC (*Heterogeneous Scable Delay Checkpointing*) para sistemas heterogéneos con modelos de ejecución síncrono y asíncrono. A diferencia de los trabajos existentes, nuestro trabajo soporta de manera simultánea ambos tipos de ejecuciones, tiene un bajo *overhead* de mensajes, no inhibe la ejecución, es escalable, permite que cada proceso genere *checkpoints* asíncronamente y elimina el efecto dominó.

El algoritmo HSDC utiliza un orden parcial de conjunto de eventos para establecer una representación compacta y coherente de la ejecución causal del sistema heterogéneo, la cual permite disminuir considerablemente el *overhead* haciendolo escalable. El algoritmo HSDC también reduce el número de *checkpoints* forzados detectando ciertas condiciones que nosotros llamamos *Condiciones Seguras para el Retraso de Checkpoint* (CSRC).

Abstract

The need to solve complex problems efficiently has us forced to combine different computing environments (systems heterogeneous), however this solution adopted by many systems brings other problems such as: the problem of tolerance failures. Checkpointing is an efficient fault tolerance technique used in distributed and paralel systems, this is also used to solve a wide range of problems in parallel and distributed systems, such as debugging software load balance, migration of processes, among others. Inside checkpointing algorithms, we find the algorithms of communication-induced checkpointing (CIC) which are characterized by their low overhead, it allows processes to take asynchronous checkpoints and avoids the domino effect. To achieve these, CIC algorithms piggyback information on the application messages and take forced local checkpoints when they recognize potentially dangerous patterns (z-paths). The main disadvanges of CIC algorithms are the amount of overhead per message and the induced storage overhead.

In this research we present a new communication-induced checkpointing HSDC (Heterogeneous Scable Delay Checkpointing) algorithm for heterogeneous systems with synchronous and asynchronous execution model. Unlike the related work, our work supports simultaneously both types of executions, our work has low overhead message, does not inhibit execution, is scalable, allows each process take checkpoints asynchronously and eliminates the domino effect.

The HSDC algorithm uses a partial order set of events for establishing a compact representation and consistent execution of the causal heterogeneous system, which helps to considerably reduce the overhead making it scalable. HSDC algorithm also reduces the number of forced checkpoints to detect certain conditions that we call “Safe Conditions for Delayed Checkpoint” (CSRC).

Agradecimientos

Quiero agradecer a mi esposa **Laura Calzada Ruiz** por todo el apoyo que me dio en la realización de este proyecto, a mis hijos **Tonantzin Alejandra**, **Laura Itzel** y **Juan Pablo** por el tiempo que me otorgaron para realizar este trabajo de investigación, jamas podre recompensarlos; pero gracias a esto, hoy comprendo lo que tengo y soy, y lo que quiero hacer para el resto de mis días.

Quiero agradecer a mis padres **Isabel Juana Simón Chino** y **Miguel Calixto López**, así como a mis hermanos **Matilde**, **Miguel** y **Maria Elena** por todo su apoyo y comprensión, por todos los momentos felices que tuvimos durante el tiempo en que estuve de regreso en casa, jamas los voy a olvidar.

De manera muy especial quiero agradecer a mi asesor **Dr. Saul Eduardo Pomares Hernández** por todo el apoyo y comprensión durante todo el proceso de mis estudios de doctorado. A mi amigo **José Roberto Perez Cruz** por el apoyo a este trabajo de investigación. No tengo palabras para expresar mis agradecimientos.

También quiero agradecer a mis sinodales: **Dra. Claudia Feregrino Uribe** (INAOE), **Dra. María Del Pilar Gómez Gil** (INAOE), **Dr. Gustavo Rodríguez Gómez** (INAOE), **Dr. Jesús González Bernal** (INAOE) y **Dr. Victor Manuel Larios Rosillo** (UDG, Universidad de Guadalajara) por su apoyo a este trabajo de investigación. Muchas gracias por todos sus comentarios y aportaciones que enriquecieron y mejoraron este trabajo de investigación.

Finalmente quiero agradecer a la *Universidad del Papaloapan (UNPA)*, al *Instituto Nacional de Astrofísica Óptica y Electrónica (INAOE)* y al *Consejo Nacional de Ciencia y Tecnología (CONACYT)* por el apoyo y financiamiento para la realización de mis estudios de doctorado, sin estos, este trabajo de investigación no sería una realidad.

Para ustedes abuelitas, como un tributo a su gran carriño, en donde quiera que ustedes estén:

† *Aurora Chino Ramos*
y
† *Victoria López Velazquez*

Nunca puede entenderlas pero siempre las respecté, y ahora, las recordaré hasta mis últimos días.

Atentamente,

Alberto Calixto Simón

Lista de figuras v

Lista de cuadros vii

Lista de acrónimos ix

1. Introducción 1

1.1. Descripción del problema de investigación 5

1.1.1. Modelo de ejecución heterogéneo 5

1.1.2. El problema de *checkpointing* 7

1.2. Objetivo general 10

1.3. Propuesta de solución 10

1.4. Organización de la tesis 11

2. Modelo de sistema y marco conceptual 13

2.1. Modelo de sistema 13

2.1.1. Modelo de ejecución asíncrono 15

2.1.2. Modelo de ejecución síncrono 16

2.2. Definición del orden causal y su implementación 17

2.3. Fundamentos del checkpointing 19

3. Trabajos relacionados 25

3.1. Algoritmos de *checkpointing* que simulan condiciones de ejecución 27

3.1.1. Algoritmos de *checkpointing* para sistemas distribuidos 27

3.1.2. Algoritmos de *checkpointing* para sistemas paralelos 31

3.1.3. Simulación de condiciones de ejecución síncrono en sistemas de ejecución asíncrono 31

3.1.4. Simulación de condiciones de ejecución asíncrono en sistemas de ejecución síncrono 32

3.2. Algoritmos de *checkpointing* que utilizan un Middleware 32

3.3. Algoritmos de *checkpointing* para ambientes heterogéneos 33

4. Algoritmo S-FI de comunicación inducida 35

4.1. Descripción del algoritmo de referencia FI 36

4.2. Condición de *checkpoint* forzado del algoritmo S-FI 38

4.2.1. Condición de *checkpoint* forzado del algoritmo S-FI con estructuras dinámicas 41

4.3. Especificación del algoritmo S-FI 44

4.3.1. Descripción del algoritmo S-FI 44

4.4. Análisis de *overhead* del algoritmo S-FI 45

4.5. Simulación del algoritmo S-FI 52

4.6. Análisis de S-FI 54

4.6.1. Análisis formal de S-FI 55

4.6.2. Análisis de la simulación 56

5. Algoritmo DCFI para el retraso de *checkpoints* 59

5.1. Análisis de la información causal en el algoritmo FI 60

5.2. Análisis de *overhead* de la información causal entre procesos 61

5.2.1. Caracterización de *z-cycles* causales 65

5.3. Enfoque de retraso de *checkpoint* 69

5.3.1. Condiciones Seguras para el Retraso de *Checkpoint* (CSRC) 73

5.4. Especificación y descripción del algoritmo DCFI 75

5.5. Simulación del algoritmo DCFI 78

5.6. Conclusiones 79

6. Algoritmo HSDC para ambientes heterogéneos **83**

6.1. Principios de agrupación de *checkpoints* 85

6.1.1. Relación Z-Dependencia Inmediata 85

6.1.2. Método de agrupación de *checkpoints* 86

6.2. Agrupación de *checkpoints* en el algoritmo HSDC 87

6.3. Especificación del algoritmo HSDC 91

6.4. Conclusiones 92

7. Conclusiones y trabajos a futuro **97**

7.1. Conclusiones 97

7.2. Trabajo a futuro 100

7.2.1. Algoritmo de *checkpointing* híbrido 100

7.2.2. Aplicación de la estrategia *lazy indexing* 101

7.2.3. Algoritmo *rollback recovery* (recuperación hacia atrás) 101

Bibliografía **103**

A. Demostraciones **109**

A.1. Demostración del Teorema 2 109

A.2. Demostración del Teorema 3 113

B. Algoritmos S-FI y DCFI **119**

B.1. SFI.java 119

B.2. DCFI.java 125

Lista de figuras

1.1. Esquema general de un sistema heterogéneo 3

1.2. Diagrama de un modelo de ejecución asíncrono. 6

1.3. Diagrama de un modelo de ejecución síncrono. 7

1.4. Esquema abstracto del modelo heterogéneo. 8

1.5. Diagrama del modelo de ejecución heterogéneo. 8

1.6. Ejemplo del modelo de ejecución heterogéneo. 9

1.7. Conjuntos de *checkpoints* en el modelo de ejecución heterogéneo. 10

2.1. Esquema abstracto del modelo heterogéneo. 14

2.2. Gráfica IDR del patrón de comunicación y checkpoints de la Figura 2.3. . . 18

2.3. Patrón de comunicación y *checkpoints*. 20

2.4. *Snapshot* global consistente y no consistente. 21

2.5. *z-path* causal y no causal. 22

2.6. *z-paths* y *z-cycles* en un patrón de comunicación y *checkpoints*. 22

3.1. Taxonomía de algoritmos de *checkpointing*. 26

4.1. Detección de cualquier *z-path* causal en la recepción en un mensaje. 38

4.2. Resultados de la simulación de S-FI. 54

5.1. *z-path* no causal y su eliminación utilizando el algoritmo FI. 61

5.2. Esquemas utilizados en el análisis del algoritmo FI en [22]. 62

5.3. *z-path* no causal de C_j^y a C_k^z. 63

5.4. Escenarios posibles respecto a cómo el reloj lógico de p_k puede ser acarreado hasta el proceso p_i. 64

5.5. Dos diferentes perspectivas de un *z-cycle*. 66

5.6. Diversos esquemas de un *z-cycle*. 67

5.7. Caracterización de *z-cycle* rastreable. 68

5.8. *z-cycle* rastreable que detecta la formación de un *z-cycle*. 69

5.9. *z-cycle* no rastreable que necesita de al menos dos *checkpoints* forzados para eliminar un *z-cycle*. 70

5.10. Esquemas de intervalos consistentes y no consistentes. 71

5.11. *z-cycle* rastreable y formas de removerlo. 71

5.12. Ejemplos con *checkpoints* forzados en cascada en algoritmos CIC. 72

5.13. Ejemplos sin efecto cascada al aplicar el retraso de *checkpoint*. 72

5.14. Condiciones seguras para el retraso de *checkpoint*. 74

5.15. Resultados de la simulación de DCFI para 1,000 y 2,500 mensajes. 80

5.16. Resultados de la simulación de DCFI para 5,000 y 7,500 mensajes. 81

5.17. Resultados de la simulación de DCFI para 10,000 y 50,000 mensajes. . . . 82

6.1. Esquema base de la relación $ZIDR$. 86

6.2. Ejemplo de conjunto de *checkpoints* con $ZIDR$. 89

6.3. Ejemplo de patrón de comunicación y *checkpoints* heterogéneo. 90

Lista de cuadros

3.1. Comparativo de algoritmos de *checkpointing* para sistemas distribuidos. . . 31

3.2. Comparación de algoritmos de *checkpointing* con características heterogéneas. 34

4.1. Algoritmo S-FI (ω_0 y ω_1). 45

4.2. Algoritmo S-FI (ω_2). 46

4.3. Procedimientos y funciones usados en el algoritmo S-FI. 47

4.4. Valores de t respecto al número de procesos n. 49

4.5. *Overhead* por mensaje (bits) para S-FI, FI y FINE. 52

4.6. *Overhead* en FI, porcentajes de *overhead* para FINE y S-FI respecto a FI, y puntos en la gráfica de S-FI donde se invierte la pendiente. 55

5.1. Secciones de envío o recepción en la estructura genérica de un *z-path* no causal. 63

5.2. Características de los escenarios de la Figura 5.4. 63

5.3. Algoritmo DCFI (σ_0 y σ_1). 76

5.4. Algoritmo DCFI (σ_2). 77

5.5. Procedimiento para generar un *checkpoint* en el algoritmo DCFI. 78

6.1. Método para la agrupación de *checkpoints* usando ZIDR. 88

6.2. Algoritmo HSDC heterogéneo (η_0). 92

6.3. Algoritmo HSDC heterogéneo (ρ_0 y ρ_1). 93

6.4. Algoritmo HSDC heterogéneo (ρ_2). 95

6.5. Procedimientos y funciones usados en el algoritmo HSDC. 96

Lista de acrónimos y siglas

API	*Application Programming Interface*
CC	*Coordinated checkpointing*
CCP	*Comunnication and Checkpoint Pattern*
CIC	*Communication-Induced Checkpointing*
CSRC	Condiciones Seguras de Retraso de *Checkpoint*
DCFI	*Delay Checkpoint Fully Informed* (algoritmo CIC)
FI	*Fully Informed* (algoritmo CIC)
FINE	*Fully Informed aNd Efficient* (algoritmo CIC)
HB	*Happened-Before* (relación de Lamport, sección 2.2)
HSDC	*Heterogeneuos Scalable Delay Checkpoint*
IDR	*Immediate Dependency Relation* (sección 2.2)
IPT2	*Immediate Predecessor Tracking 2*
MPI	*Message-Passing Interface*
RSC	*Realizable with Synchronous Communication*
S-FI	*Scalable-Full Informed* (algoritmo CIC)
SG	*Snapshot* Global
SGC	*Snapshot* Global Consistente
SGCs	*Snapshots* Globales Consistentes
TCKPT	*Total Checkpoint*
ZCF	*Z-Cycle Free*
ZIDR	*Z-Depends Immediate Dependency Relation*

Capítulo 1

Introducción

La evolución de los sistemas de cómputo y de los sistemas de comunicación durante la mitad del siglo pasado nos ha dejado una gran diversidad tecnológica. Esta evolución se ha dado en los sistemas de cómputo debido a la necesidad de hacer más eficiente el uso de los recursos disponibles y la necesidad de incrementar la capacidad de cómputo, mientras que la evolución en los sistemas de comunicación se ha dado por la necesidad de transmitir cada vez más datos y más rápidamente. Podemos mencionar que una consecuencia directa de esta evolución es la aparición de los sistemas llamados *sistemas heterogéneos*. En términos generales, un sistema heterogéneo está constituido de *sistemas distribuidos* y de *sistemas paralelos*. Un sistema distribuido es un conjunto de entidades independientes que cooperan para solucionar un problema en común que no puede resolverse de manera individual. Un sistema paralelo, por otra parte, es un conjunto de entidades que procesan de manera simultánea las partes de una tarea dividida previamente [29]. La principal diferencia entre ambos sistemas es la manera en que se comunican las entidades del sistema (e.g. procesos, usuarios, agentes, ect). En un sistema distribuido, las entidades se comunican a través del intercambio de mensajes, asumiendo una red de computadoras, mientras que en los sistemas paralelos, las entidades asumen una comunicación a través del acceso a memoria compartida. A continuación presentamos una definición aceptada por la comunidad científica acerca de lo que se considera es un sistema heterogéneo.

Khokhar et al. [26] definen a un sistema heterogéneo de la siguiente forma:

Un sistema heterogéneo es un intento por combinar diversos ambientes de cómputo de alto desempeño[1] *para solucionar eficientemente un problema complejo.*

[1]En este contexto, un ambiente de cómputo de alto desempeño puede ser una red de alta velocidad, interfaz, sistema operativo, protocolo de comunicación, entorno de programación, etcétera.

En la Figura 1.1 mostramos el esquema general del sistema heterogéneo introducido por Khokhar et al. [26]. En este esquema podemos observar diversas interconexiones entre *nodos*. Un nodo es una representación o caracterización abstracta del cómputo que desarrolla uno o más procesadores bajo un mismo modelo de ejecución; por ejemplo: *paso de mensajes*, *memoria compartida*, *SIMD*[2], *MIMD*[3], entre otros. Además, con el objetivo de proporcionar una alta velocidad de procesamiento, cada nodo cuenta con un hardware especializado: *cluster*, *CPU-multicore*, *máquinas CRAY*, *máquinas CM-5*, *CPU-GPU*, etcétera.

A su vez, el desarrollo de tecnologías como *GPUs (Graphics Processing Units)*, *procesadores con múltiples núcleos (multi-core)* y *FPGAs (Field Programmable Gate Arrays)*, han renovado el interés por las *arquitecturas paralelas* y el *cómputo concurrente* [8, 32]. El cómputo concurrente existe desde hace años. En las últimas décadas fue ampliamente desplazado por el *cómputo secuencial*[4] debido a la estrategia utilizada por esta tecnología para aumentar su capacidad de cómputo (constantes aumentos en la velocidad de los procesadores de un sólo núcleo a un bajo costo); sin embargo, los límites de la tecnología CMOS (*Complementary Metal-Oxide-Semiconductor*) están rezagando el aumento de velocidad (comparado con el crecimiento previo) de los procesadores de un sólo núcleo [2]. Actualmente, las micro-arquitecturas de un sólo núcleo clásico están siendo desplazadas por micro-arquitecturas multi-core, con multiproceso en cada núcleo [25, 27]. De esta forma, las tecnologías emergentes (mencionadas anteriormente) han retornado el interés de estudio hacia el cómputo concurrente y los sistemas heterogéneos; a tal grado, que estas áreas de estudio son consideradas como alternativas sólidas y viables para explotar esta tecnología de forma eficiente, y con grandes posibilidades de continuar aumentando la capacidad de cómputo de los sistemas.

La *capacidad de cómputo* de los sistemas heterogéneos está vinculada con la cantidad de nodos y la capacidad de cómputo de cada uno de ellos. A mayor cantidad de nodos en el sistema, en general, esperamos incrementar la capacidad de cómputo; sin embargo, a mayor número de nodos también incrementamos la complejidad de interacción entre nodos y la probabilidad de fallas en el sistema. No obstante, a esto, muchas aplicaciones

[2]SIMD (Single Instruction, Multiple Data), en este modelo de cómputo, todos los procesadores ejecutan la misma instrucción de manera sincronizada sobre diferentes datos almacenados en su memoria local.

[3]MIMD (Multiple Instruction, Multiple Data), en este modelo de cómputo, los procesadores ejecutan de manera independiente un código almacenado en la memoria local.

[4]El *cómputo secuencial* es el modelo básico ordinario, en donde una máquina procesa una única instrucción o un único dato a la vez, y siguiendo un programa almacenado en memoria.

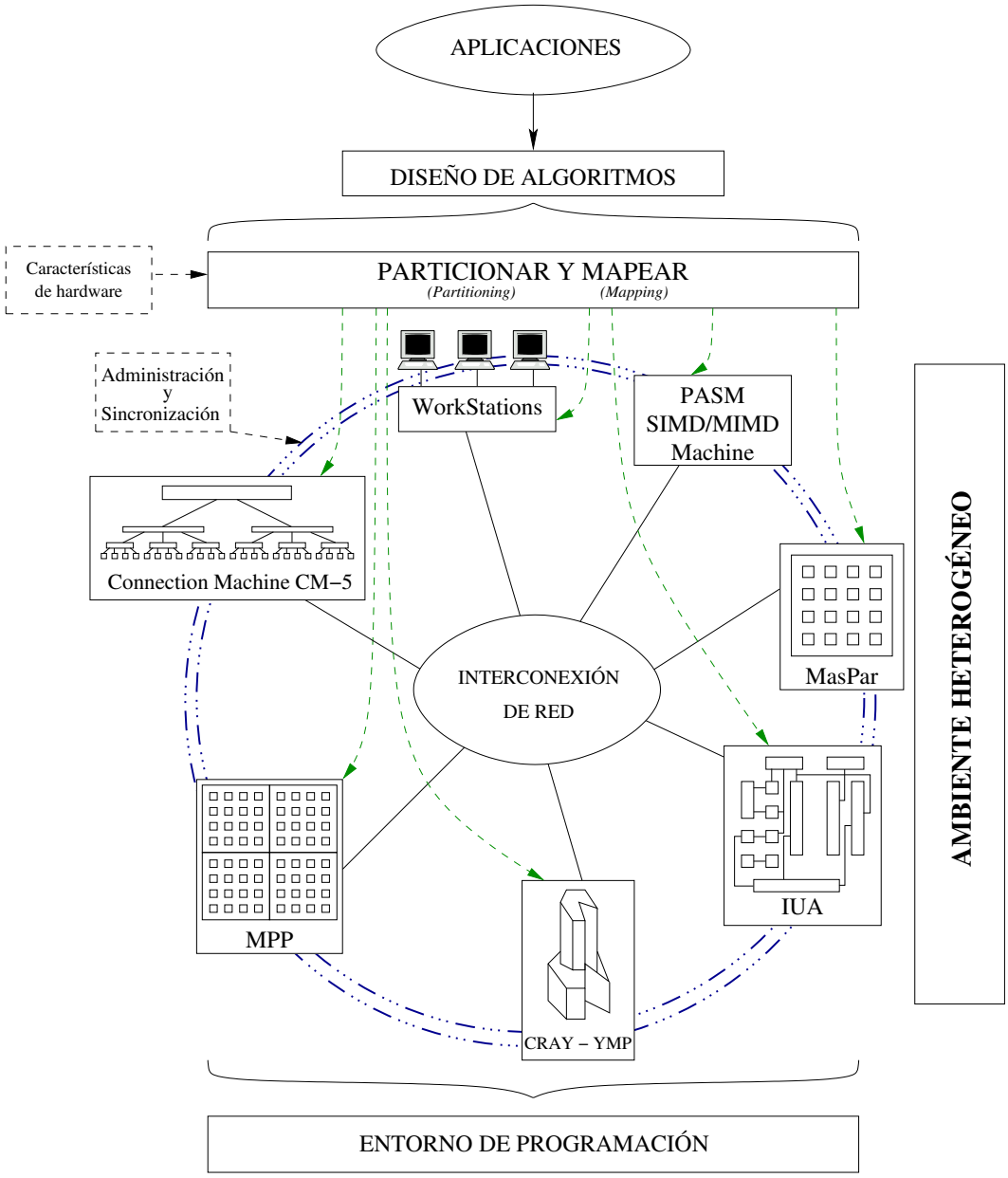


Figura 1.1: Esquema general de un sistema heterogéneo, Khokhar et al. [26].

requieren (cada vez más) grandes cantidades de cómputo; puesto que estas aplicaciones puede durar días, semanas, meses, ó incluso años procesando algún trabajo. Un ejemplo de estas aplicaciones es la identificación de estructuras proteínicas [40]; sin embargo, las simulaciones que acoplan múltiples fenómenos físicos son de las aplicaciones con mayores requerimientos en cómputo [18].

Debido al tiempo de procesamiento de algunas aplicaciones (mencionadas anteriormente), la probabilidad de fallas en un sistema heterogéneo es alta, por lo que estos sistemas

requieren del diseño de sistemas tolerantes a fallas. Un sistema *tolerante a fallas* realiza su función correctamente aún en presencia de fallas en el sistema [24]. En general, una falla es de tipo *permanente* o *temporal*. Una falla permanente se produce por el daño en uno o más componentes del sistema, mientras que una falla temporal es producida por cambios de condiciones en el sistema. Las fallas permanentes pueden corregirse a través de la reparación o sustitución de los componentes. Las fallas temporales perduran por un corto período de tiempo, son difíciles de detectar y tratar, por lo que, es más factible y rentable hacer frente a éstas por software que por hardware.

Los algoritmos de *checkpointing* son una de las técnicas preferidas para proporcionar tolerancia a fallas temporales. Además, estos son ampliamente utilizados para solucionar una gama de problemas, dentro de estos problemas tenemos: *depuración* (*debugging*), *balance de carga* (*workload balancing*) o *migración de procesos* (*process migration*), entre otros [19, 24]. Un algoritmo de *checkpointing* resguarda información del estado local de un proceso (*checkpoint*) durante su tiempo de ejecución, para que en caso de ocurrir fallas en el sistema, cada proceso (que falla) pueda recuperar su último estado estable. De esta forma, el sistema puede recuperarse de una o más fallas y sin perder el cómputo desarrollado.

Numerosos algoritmos de *checkpointing* han sido utilizados en sistemas distribuidos y en menor medida en sistemas paralelos, sin embargo, pocos han sido desarrollados para sistemas heterogéneos. Los algoritmos convencionales de *checkpointing*, por lo general, consideran que los sistemas sólo manejan un modelo de ejecución (asíncrono, para sistemas distribuidos; y síncrono, para sistemas paralelos). Las investigaciones realizadas en el ámbito heterogéneo, desde la perspectiva del *checkpointing*, han sido enfocadas a la portabilidad de *checkpoints* entre sistemas [41, 46, 43]. Por otra parte, los trabajos de *Cao* [13] y *Tantikul* [47] son una primera iniciativa por generar algoritmos de *checkpointing* heterogéneo; estos dos trabajos de investigación han expuesto las carencias de los algoritmos convencionales de *checkpointing*, cuando intentan resolver el problema del *checkpointing* con más de un modelo de ejecución, revelando con ello, la necesidad de una clase de algoritmos de *checkpointing* diferente al convencional.

El trabajo de *Cao* muestra la necesidad de algoritmos de *checkpointing* para sistemas distribuidos formados por subsistemas que ejecutan un mismo modelo de ejecución. Por su parte, *Tantikul* desarrolla un algoritmo de *checkpointing* para *sistemas multi-hilos*[5]. Los

[5]Un *sistema multi-hilos* es un sistema con la capacidad de soportar subprocesos dentro de un proceso. A los subprocesos por lo general se les conoce como hilos; por lo que, un proceso puede contener múltiples hilos.

algoritmos convencionales de *checkpointing* están pensados para trabajar a nivel proceso. El trabajo de *Tantikul* pone en evidencia la necesidad de algoritmos de *checkpointing* orientados a modelos de ejecución. En su trabajo, considera dos modelos de ejecución, un modelo de ejecución asíncrono para procesos y un modelo de ejecución síncrono para hilos.

En está investigación desarrollamos un algoritmo de *checkpointing* para sistemas heterogéneos con modelos de ejecución síncrono y asíncrono. A diferencia de los trabajos existentes, nuestro trabajo soporta de manera simultánea ambos tipos de ejecuciones, tiene un bajo *overhead*, no inhibe la ejecución y es escalable. Lamentablemente, en está investigación, no llegamos a establecer un orden parcial a nivel conjunto de eventos para ambos modelos de ejecución. Pero, por medio de un orden parcial de conjunto de eventos para ejecuciones asíncronas establecimos una representación compacta y coherente de la ejecución causal del sistema. Esta representación nos permitió disminuir el *overhead* en el modelo de ejecución asíncrono y hacer escalable a nuestro algoritmo heterogéneo. En los siguientes capítulos ampliamos los detalles de nuestro algoritmo de *checkpointing* para sistemas heterogéneos.

1.1. Descripción del problema de investigación

Nuestro problema radica en la generación de *Snapshot globales consistentes* (SGC) del sistema heterogéneo. Con el objetivo de comprender la naturaleza del problema, iniciamos esta sección con la descripción del modelo de ejecución heterogéneo. Después, describimos el problema de *checkpointing* en sistemas heterogéneos y las delimitaciones del problema dentro de nuestra investigación.

1.1.1. Modelo de ejecución heterogéneo

Lynch [34] establece que se pueden realizar varias suposiciones respecto al tiempo de ejecución de los eventos en un sistema, lo que reflejará distintos tiempos que pueden ser utilizados por los algoritmos. Por un lado, tenemos a los eventos que son ejecutados por un conjunto de procesadores en una completa sincronización (síncronos), realizando comunicaciones y cómputo en perfecta sincronía; y por el otro, a los eventos que son ejecutados por un conjunto de procesadores no sincronizados (asíncronos), realizando comunicaciones y cómputo a velocidades arbitrarias y en un orden aleatorio.

Modelo de ejecución asíncrono

Un *modelo de ejecución asíncrono* considera que los eventos de procesos diferentes se ejecutan de manera independiente; el tiempo de ejecución de cada evento se desconoce y es arbitrario; los eventos de diferentes procesos se comunican por medio de mensajes, y estos, tienen un retardo finito que no puede despreciarse.

En la Figura 1.2 mostramos un diagrama *tiempo-espacio*[6], este diagrama es una representación gráfica del modelo de ejecución asíncrono. En el diagrama, un evento interno es representado por un círculo y un mensaje por una flecha. Note que las flechas no son verticales, lo que significa que transcurre un tiempo finito entre el envío y la recepción de un mensaje.

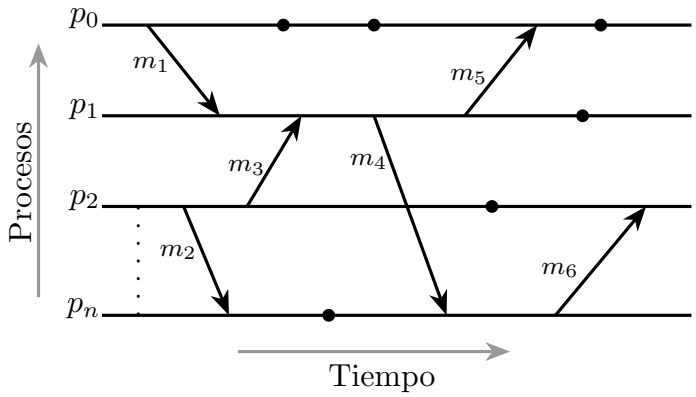


Figura 1.2: Diagrama de un modelo de ejecución asíncrono.

Modelo de ejecución síncrono

El *modelo de ejecución síncrono* considera que el tiempo entre la ejecución de dos o más eventos de procesos diferentes es despreciable, de tal forma, que dos o más eventos (de diferentes procesos) se realizan de forma *simultanea*. En la Figura 1.3 mostramos un ejemplo de un modelo de ejecución síncrono; la representación de eventos es igual al modelo de ejecución asíncrono, pero los mensajes se representan con flechas verticales. Las flechas verticales, indican que el envío y la recepción de un mensaje se realiza en un mismo tiempo.

[6]Un diagrama tiempo-espacio [31] es una gráfica bidimensional que ilustra en el eje Y al conjunto de procesos, y en el eje X, la evolución del conjunto de eventos en la línea de tiempo de cada proceso.

Aunque la figura no muestra eventos simultáneos o concurrente de forma explicita, estos eventos pueden ocurrir dentro del diagrama.

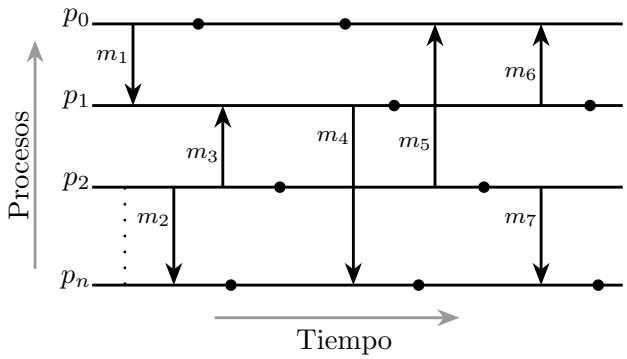


Figura 1.3: Diagrama de un modelo de ejecución síncrono.

Modelo de ejecución heterogéneo

En la Figura 1.4 mostramos el esquema abstracto de nuestro modelo heterogéneo. Este fue desarrollado a partir de las ideas de Khokhar et al. [26] y está formado por un conjunto de *nodos*; cada nodo ejecuta un modelo de ejecución síncrono o asíncrono, como lo ilustramos en la Figura 1.5. Los eventos producidos dentro de un nodo, los podemos agrupar en subconjuntos disjuntos de eventos. Los eventos en un subconjunto se ejecutan de manera síncrona (simultanea o concurrente) o asíncrona, mientras que los eventos de diferentes subconjuntos se ejecutan de manera asíncrona. En la Figura 1.6 mostramos un ejemplo de un modelo de ejecución heterogéneo; la representación de estos eventos la realizamos de la misma forma que los dos modelos anteriores (síncrono y asíncrono).

1.1.2. El problema de *checkpointing*

El problema de *checkpointing* radica en la generación de uno o más *Snapshots Globales Consistentes* (SGC). Un SGC es un conjunto de *checkpoints*[7], un *checkpoint* por cada proceso en el sistema, y con la característica de que ningún par de *checkpoints* (en el SGC) tiene una relación causal. En otras palabras, si A y B son dos *checkpoints* en un SGC, A no ocurre antes que B y B no ocurre antes que A.

[7]Un *checkpoint* es un conjunto de información local que resguarda un proceso en un tiempo especifico de su ejecución (ver sección 2.3).

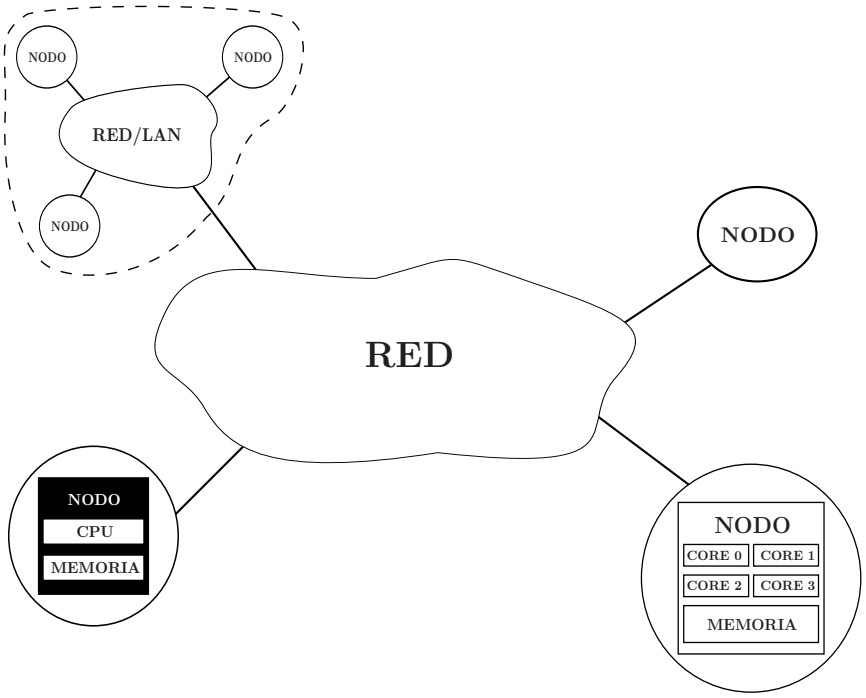


Figura 1.4: Esquema abstracto del modelo heterogéneo.

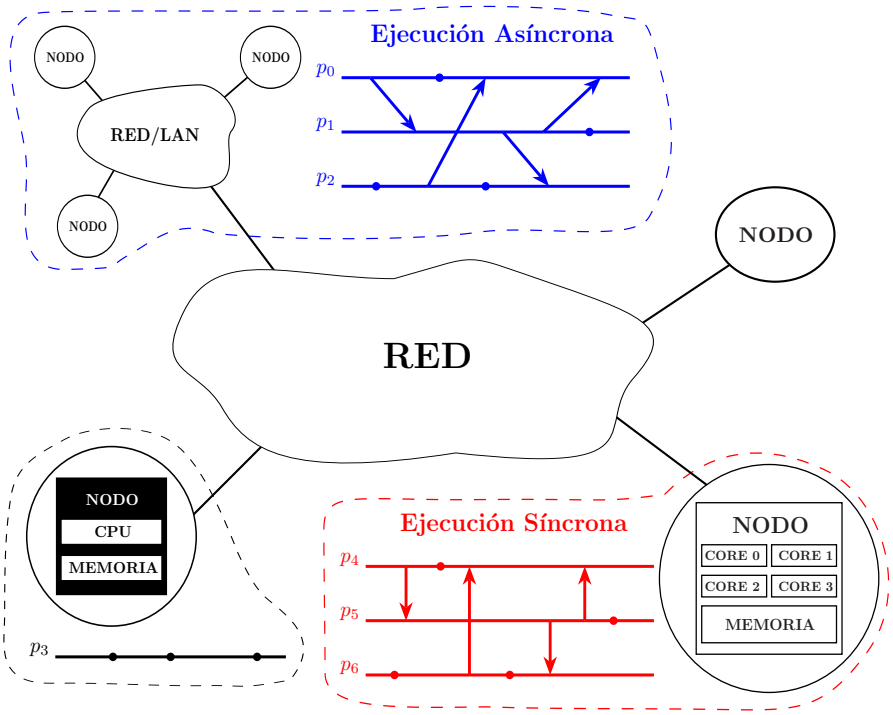


Figura 1.5: Diagrama del modelo de ejecución heterogéneo.

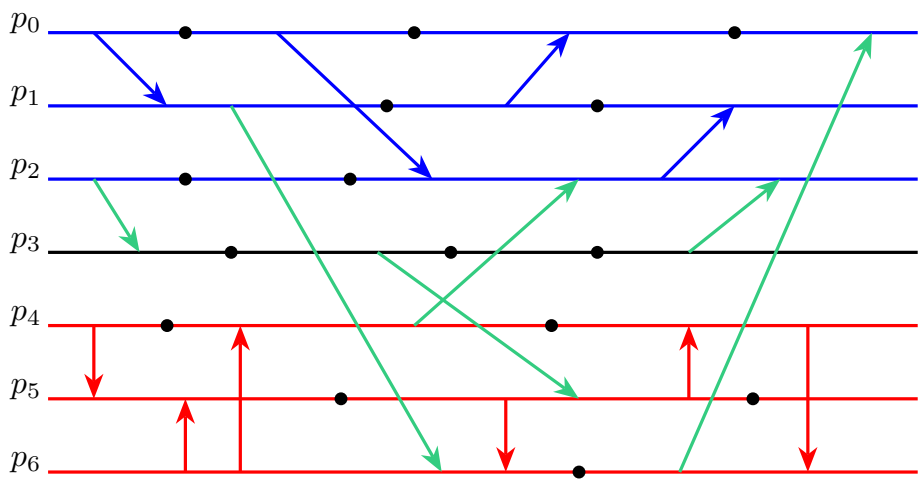


Figura 1.6: Ejemplo del modelo de ejecución heterogéneo.

Los *checkpoints* generados por los procesos del sistema, en el mejor de los casos, podrán agruparse para formar SGCs, pero en el peor de los casos, algunos de ellos no podrán ser parte de ningún SGC del sistema. De esta forma, algunos *checkpoints* no serán utilizados en la recuperación de fallas, y por lo tanto, únicamente deterioran el desempeño del sistema.

La solución del problema de *checkpointing* no es trivial. Los procesos se ejecutan de forma concurrente; las dependencias introducidas por los mecanismo de comunicación entre procesos generan *checkpoints* causales que no pueden ser parte de un mismo SGC, e incluso, la no causalidad entre un par de *checkpoints* no asegura que ambos sean parte de un mismo SGC [37]. Si esto no fuera suficiente, en ejecuciones asíncronas, la falta de sincronización (reloj común) entre procesos hace difícil establecer un SGC; las comunicaciones por medio de mensajes genera un sistema no determinista (debido a los retardos en los mensajes), aumentando la complejidad del problema. Además, los algoritmos de *checkpointing* por lo regular son algoritmos en línea (*online*), es decir, trabajan con información parcial del sistema (no tiene un visión global del estado del sistema).

En la Figura 1.7 mostramos un diagrama con grupos de *checkpoints* en un modelo de ejecución heterogéneo. En este caso, hay procesos realizando cómputo con diferentes modelos de ejecución (síncrono o asíncrono).

En resumen, el problema de *checkpointing* es un problema transversal en el que participan:

a) La eficiente creación de *checkpoints* por cada proceso en el sistema.

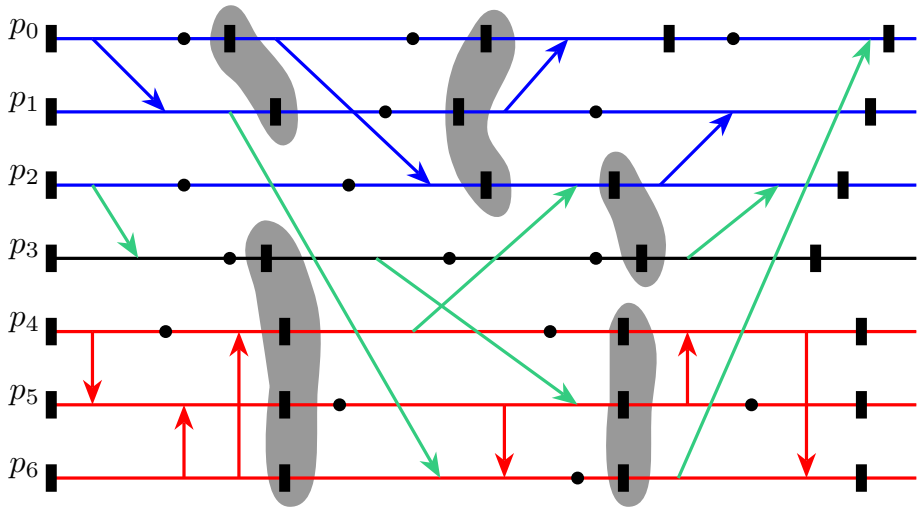


Figura 1.7: Conjuntos de *checkpoints* en el modelo de ejecución heterogéneo.

b) El traslado de la información causal en el sistema.

c) La eficiente construcción de *snapshots* globales consistentes en el sistema.

Estos tres puntos son los ejes principales de la investigación y bajo un ambiente heterogéneo con modelos de ejecución síncrono y asíncrono.

1.2. Objetivo general

Desarrollar un algoritmo de checkpointing eficiente para sistemas heterogéneos con modelos de ejecución síncrono y asíncrono.

1.3. Propuesta de solución

En esta tesis proponemos un nuevo algoritmo de *checkpointing* de comunicación inducida para sistemas heterogéneos con modelos de ejecución síncrono y asíncrono, sin inhibición de cómputo, con un *overhead*[8] bajo, escalable y sin efecto dominó.

[8]En ciencias computacionales, *overhead* es un costo adicional o el exceso de uso de algún recurso para lograr un particular objetivo. En particular, el *overhead* de un mensaje, en redes de computadoras, es el costo adicional por el envío de información de control.

La idea principal consiste en establecer una representación compacta del cómputo del sistema heterogéneo, que permita relacionar los eventos de *checkpoints* y agruparlos para formar *snapshot* globales consistentes.

La contribución principal de esta tesis es un algoritmo de *checkpointing* de comunicación inducida para sistemas heterogéneos con modelos de ejecución asíncrona y síncrona. A grandes rasgos, desarrollamos los siguientes puntos para la obtención del algoritmo de *checkpointing*.

- Determinamos una representación compacta que capturará la causalidad de los *checkpoints* del sistema.
- Aseguramos que todos los *checkpoints* generados en el sistema fueran parte de un *snapshot* global consistente.
- Optimizamos la *información de control* (*overhead*) mínimo y necesario para asegurar la causalidad de los *checkpoints* en el sistema.
- Disminuimos la cantidad de *checkpoints* para optimizar el *overhead* de almacenamiento y reducir la complejidad de agrupar *checkpoints* para formar *snapshot* globales consistentes.
- Finalmente, establecimos un mecanismo para agrupar *checkpoints* y formar *snapshot* globales consistentes.

1.4. Organización de la tesis

Esta tesis esta organizada en siete capítulos y dos apéndices.

En el capítulo 2 presentamos el marco conceptual. Introducimos el modelo de sistema, la notación y definiciones utilizadas en el documento y la teoría base utilizada en el ámbito de los algoritmos de *checkpointing*.

En el capítulo 3 presentamos el estado del arte de nuestro problema de investigación.

En el capítulo 4 desarrollamos el algoritmo de *checkpointing* de comunicación inducida S-FI (Scalable-Fully Informed) para un modelo de ejecución asíncrona. El algoritmo S-FI fue desarrollado con el objetivo de establecer la información de control mínima para mantener la causalidad de *checkpoints* en el sistema, y optimizar con ello, el *overhead*

de mensajes en el sistema. En este capítulo, primero presentamos el algoritmo FI [22] y su *condición de checkpoint forzado* [9]. Después desarrollamos la condición forzada de nuestro algoritmo S-FI en base a estructuras estáticas (como FI). Posteriormente, redefinimos la condición forzada de S-FI en base a estructuras dinámicas, para finalmente definir el algoritmo S-FI. Finalizamos, el capítulo, presentando un análisis de los resultados de la simulación del algoritmo S-FI junto con los de otros dos algoritmos de comunicación inducida.

En el capítulo 5 desarrollamos el algoritmo de *checkpointing* de comunicación inducida DCFI (Delay Checkpoint Fully Informed) que utiliza un *enfoque de retraso de checkpoint* desarrollado en esta investigación. El enfoque, permite disminuir la cantidad de *checkpoint forzados* a nuestro algoritmo de *checkpointing*. Este capítulo, inicia con un análisis de la información causal *piggyback* (acarreada o transportada) en cada mensaje que envía un proceso. Continuamos con la caracterización de una clase de *z-cycle* [37] que llamamos *z-cycle rastreable* e introducimos el enfoque de retraso de *checkpoint* para disminuir el número de *checkpoints* forzados. En esta parte del documento, definimos las condiciones viables para aplicar el enfoque de retraso de *checkpoint*. Posteriormente, presentamos el desarrollo del algoritmo DCFI que implementa al enfoque de retraso y presentamos los resultados de la simulación de DCFI junto con la simulación de otros dos algoritmos de comunicación inducida (analizados en el capítulo anterior). Finalizamos, el capítulo, presentando nuestras conclusiones de esta parte de la investigación.

En el capítulo 6 introducimos el modelo del sistema heterogéneo, la relación ZIDR para la agrupación de *checkpoints* y damos un esbozo del algoritmo de *checkpointing* de comunicación inducida para sistemas heterogéneos con modelos de ejecución síncrono y asíncrono.

En el último capítulo presentamos nuestras conclusiones finales, los trabajos a futuro de esta investigación y las referencias bibliográficas.

El apéndice del documento está organizado en dos partes. La primera parte, contiene las demostraciones de los teoremas 2 y 3 de la teoría del algoritmo S-FI. Y en la segunda parte presentamos el código fuente en lenguaje JAVA de los algoritmos S-FI y DCFI desarrollados en esta investigación.

[9] La condición de *checkpoint* forzado (en algoritmos de *checkpointing* de comunicación inducida) es una proposición que permite a cada proceso determinar (de manera local) si debe de generar un *checkpoint*, juntamente antes, de la entrega de un mensaje al proceso.

Capítulo 2

Modelo de sistema y marco conceptual

Con el objetivo de comprender los antecedentes de nuestra investigación, iniciamos este capítulo con la descripción del modelo de sistema y el marco conceptual definido en los algoritmos de *checkpointing* de comunicación inducida (CCI).

2.1. Modelo de sistema

El modelo de nuestro sistema heterogéneo está compuesto por un conjunto finito de nodos $N = \{N_0, N_1, \ldots, N_n\}$ (ver Figura 2.1). Cada nodo en el sistema tiene un modelo de ejecución (síncrono o asíncrono) asociado, y este a su vez, está formado por uno o más procesos que desarrollan un cómputo. Cuando un proceso $p_i \in P$ falla, este se comporta de acuerdo al modelo *fail-stop* [45]. En el modelo *fail-stop*, si un proceso falla entonces este simplemente se detiene.

Un *evento* es la ocurrencia de una acción dentro de un proceso. Sea e_i^x el x-ésimo evento producido por el proceso p_i. La secuencia finita o infinita $h_i = e_i^0 e_i^1 \ldots e_i^x \ldots$ constituye el cómputo local e historial de p_i, denotada por H_i.

En nuestro modelo consideramos dos tipos de eventos: *interno* y *externo*. Un evento interno es una acción única que ocurre en un proceso p y está cambia únicamente el estado local del proceso. El conjunto finito de eventos internos es denotado por E_i. En esta investigación, consideramos únicamente *checkpoints* como eventos internos y usamos la notación C_i^x para denotar al x-ésimo *checkpoint* del proceso p_i. Para el problema de

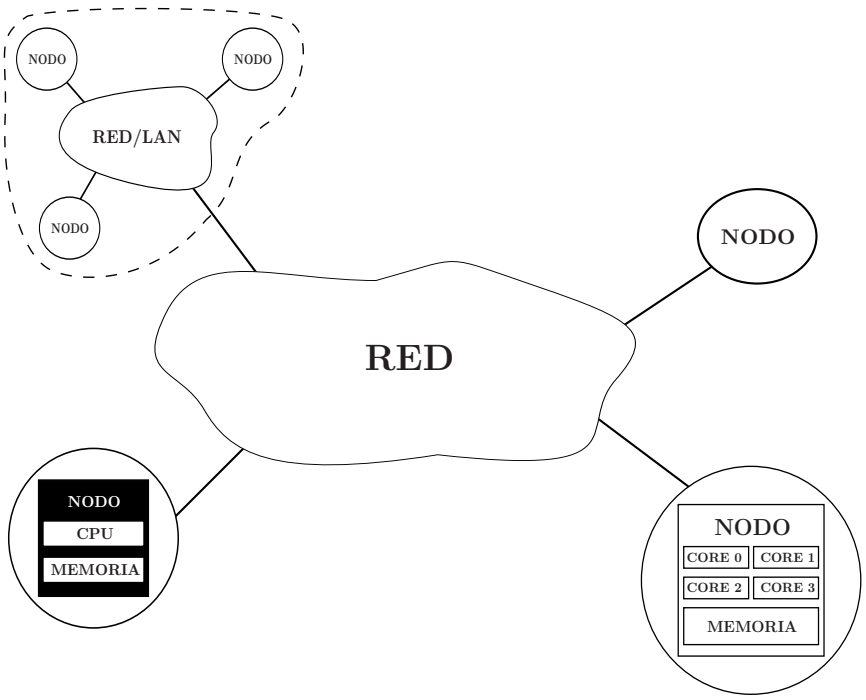


Figura 2.1: Esquema abstracto del modelo heterogéneo.

checkpointing, el conjunto E_i representa a un conjunto de *eventos relevantes*[1] a ser considerados. Además, suponemos que cada proceso genera un *checkpoint* antes de iniciar su cómputo (*checkpoint inicial*) y después de finalizar su cómputo (*checkpoint final*). Por otra parte, un evento externo es también una única acción que ocurre en un proceso, pero esta es vista por otros procesos, por lo que afecta al estado global del sistema.

La comunicación entre procesos del sistema heterogéneo es por *paso de mensajes* y/o *memoria compartida*. La comunicación entre procesos de diferentes nodos es exclusivamente por paso de mensajes, mientras que la de procesos en un mismo nodo son por paso de mensajes o por memoria compartida, pero no ambas a la vez. Un mensaje entre procesos produce los tipos de dependencia *inter-nodo* e *inter-proceso*. Las dependencias inter-nodo se generan cuando los procesos pertenecen a diferentes nodos; mientras que las dependencias inter-proceso cuando los procesos pertenecen a un mismo nodo. De esta forma, un nodo en el sistema está caracterizado por uno de los siguientes tres casos:

a) *Un nodo con un sólo proceso.*

En este caso, el proceso del nodo sólo genera dependencias inter-nodo. La generación

[1]Un conjunto E_R de eventos relevantes es un subconjunto de eventos del cómputo del sistema, tal que, E_R constituye un mejor nivel de abstracción del sistema.

de *snapshots* globales consistentes (SGC) en este tipo de nodos es relativamente simple; el nodo lo podemos representar como un simple proceso y cada *checkpoint* tomado por el proceso es un SGC dentro del nodo. El algoritmo de *checkpointing* heterogéneo que presentamos en la Sección 6.3 puede manejar este tipo de nodos que generan dependencias inter-nodo.

b) *Un conjunto de procesos con un modelo de ejecución asíncrono.*
En este caso, los procesos del nodo generan dependencias inter-nodo e inter-proceso. La generación de un SGC en este tipo de nodo es difícil; las dependencia inter-proceso generan patrones causales y no causales entre los *checkpoints* del mismo nodo como de otros nodos; sin embargo, ambas dependencias son generadas a partir del mecanismo de comunicación de paso de mensajes. Por lo que, el cómputo desarrollado por los procesos del nodo es similar al cómputo desarrollado por los nodos del sistema (cómputo asíncrono). De ahí que, el cómputo que desarrollan los nodos del sistema, lo visualizamos como un cómputo de primer nivel; mientras que el desarrollado por los procesos dentro de un nodo lo visualizamos como un cómputo de segundo nivel. El algoritmo de *checkpointing* heterogéneo, que presentamos en la Sección 6.3, puede manejar las dos dependencias (inter-nodo e inter-proceso) de este caso.

c) *Un conjunto de procesos con un modelo de ejecución síncrono.*
En este último caso, al igual que el anterior, los procesos del nodo generan dependencias inter-nodo e inter-proceso; sin embargo, en este caso, las dependencias inter-proceso se generan por medio de un mecanismo de comunicación de memoria compartida. Por lo que, la generación de un SGC en este tipo de nodos es relativamente fácil. Los procesos del nodo se sincronizan para formar un conjunto de *checkpoints*, y el sistema puede procesar a este conjunto como si el nodo tuviera un sólo proceso (primer caso que analizamos). Al igual que los casos anteriores, el algoritmo de la Sección 6.3, puede manejar este caso.

A continuación describimos los modelos de ejecución asíncrono y síncrono que se pueden ejecutar en un nodo.

2.1.1. Modelo de ejecución asíncrono

El modelo de ejecución asíncrono está compuesto por un conjunto finito de *procesos* $P = \{p_1, p_2, ..., p_n\}$. Los procesos presentan una ejecución asíncrona y se comunican úni-

camente por paso de mensajes.

Los eventos externos que consideramos para este modelo son los eventos *send* y *delivery*. Además, consideramos a un conjunto finito M de mensajes en el sistema. Cada mensaje $m \in M$ es enviado a través de una red asíncrona confiable, la cual es caracterizada por transmitir: sin limites de tiempo, entrega no ordenada y sin pérdida de mensajes.

Sea m un mensaje, denotamos la emisión de m por $send(m)$ y la entrega de m al proceso $p_j \in P$ por $delivery(p_j, m)$. El conjunto de eventos asociados a M es el conjunto $E_m = \{send(m) : m \in M\} \cup \{delivery(p, m) : m \in M \wedge p \in P$. El conjunto completo de eventos en el sistema es el conjunto finito $E = E_i \cup E_m$. De esta forma, el cómputo es modelado por el conjunto parcialmente ordenado $\widehat{E} = (E, \rightarrow)$, donde "$\rightarrow$", denota la relación *happened-before* de Lamport [31] (ver sección 2.2).

2.1.2. Modelo de ejecución síncrono

La descripción del modelo de ejecución síncrono es similar a la realizada del modelo asíncrono de la sección 2.1.1, excepto que los procesos de este modelo utilizan el mecanismo de comunicación por *memoria compartida* y comparten un reloj común. Esto caracteriza al modelo de ejecución síncrono de la siguiente forma:

- Cada nodo, con modelo de ejecución síncrona en el sistema, tiene un conjunto de variables compartidas que permiten la comunicación entre procesos del mismo nodo. En lo siguiente, nosotros suponemos dos casos: 1. El propietario de una variable compartida es definido como el último proceso que escribió en esta variable, y 2. El cómputo es secuencialmente consistente, es decir, en todo tiempo hay un único propietario de cada variable compartida.

- Existe un reloj global o común para todos los procesos de un nodo. De modo que, la generación de un *snapshot* global de los procesos en el nodo, se puede realizar de forma simple. Por ejemplo, por medio de alguno de los algoritmo de *checkpointing* introducidos en [24, 1, 53, 6, 5].

- La comunicación entre procesos de diferentes nodos es por paso de mensajes.

2.2. Definición del orden causal y su implementación

La relación **Happened-Before** (HB) fue definida por Lamport [31] y establece una dependencia causal de precedencia entre un conjunto de eventos. La relación HB es un orden parcial estricto (i.e. transitiva, irreflexiva y antisimétrica) definido de la siguiente forma:

Definición 1. *La relación happened-before "$\rightarrow$" es la relación mínima de un conjunto de eventos E que satisface las siguientes condiciones:*

a) Si a y b son eventos en un mismo proceso y a ocurre antes que b, entonces $a \rightarrow b$.

b) Si a es el evento de envío de un mensaje m por un proceso y b es el evento de recepción del mensaje m por otro proceso, entonces $a \rightarrow b$.

c) Si $a \rightarrow b$ y $b \rightarrow c$ entonces $a \rightarrow c$.

Lamport establece también la noción de eventos concurrentes en [31]. La definición formal es la siguiente:

Definición 2. *Dos eventos distintos a y b se dicen concurrentes si $a \nrightarrow b$ y $b \nrightarrow a$, esto es denotado por $a||b$.*

Note usted que en esta definición $a \nrightarrow b \equiv \neg(a \rightarrow b)$, en otras palabras, el evento a no ocurre antes que el evento b.

La relación de **Dependencia Causal Inmediata** (IDR) es la *reducción transitiva* de la relación HB [38]. Denotamos la relación IDR por "$\downarrow$". En particular, usamos "$\downarrow_i$"para indicar que los eventos relacionados por la relación IDR provienen de un sólo proceso. La definición formal de IDR es la siguiente:

Definición 3. *Dos eventos $a, b \in E$ tienen una relación de dependencia causal inmediata si la siguiente restricción se satisface.*

$$a \downarrow b \;\; \textbf{si} \;\; a \rightarrow b \;\wedge\; \forall c \in E, \;\; \neg(a \rightarrow c \rightarrow b)$$

En nuestro contexto, estamos interesados en identificar la relación de dependencia inmediata entre el conjunto de eventos relevantes, $E_i \subset E$, los cuales contienen a los

eventos de *checkpoints*. Por lo tanto, decimos que un par de *checkpoints* (eventos relevantes) $x, y \in E_i$ están IDR relacionados, si y sólo si, no hay otro evento relevante $z \in E_i$, tal que, z pertenece al futuro causal de x y al pasado causal de y. En la Figura 2.2 mostramos la gráfica IDR del escenario de la Figura 2.3 que presentamos mas adelante.

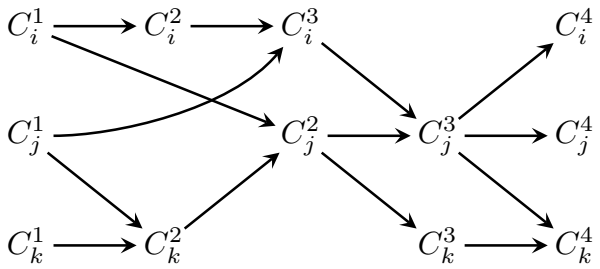


Figura 2.2: Gráfica IDR del patrón de comunicación y checkpoints de la Figura 2.3.

Implementación del orden causal

Mattern [36] y Fidge [20] desarrollaron el concepto de vector de tiempo para eliminar el defecto del *reloj lógico*[2] de Lamport [31]. La definición formal de un vector de tiempo es la siguiente:

Definición 4. *Un vector de tiempo para un sistema de n procesos es un vector v de longitud n ($v = (c_1, c_2, \ldots, c_n)$), en donde cada elemento c_i en el vector pertenece a un mismo dominio T ($c_i \in T$) y cada c_i representa el reloj local del proceso p_i. Por lo general, el dominio T es el conjunto de números enteros.*

*Cada proceso p_i mantiene su propio vector de tiempo $VTL_i = (c_1, c_2, \ldots, c_n)$. Este es utilizado como etiqueta de tiempo (*timestamp*) en sus eventos y es actualizado con base a las siguientes reglas:*

- **R1** *El vector de tiempo es inicializado en ceros. $\forall\ i, j : 1, 2, \ldots n$, $VTL_i[j] = 0$.*
- **R2** *El reloj local $VTL_i[i]$ del proceso p_i es incrementado justamente antes de colocar su etiqueta de tiempo a un evento. $VTL_i[i] = VTL_i[i] + 1$.*
- **R3** *El proceso p_i envía su vector de tiempo VTL_i en cada mensaje.*
- **R4** *Cuando el proceso p_i recibe un vector de tiempo VTL_j, en un mensaje, este actualiza su vector de tiempo VTL_i, de la siguiente forma:*

  $\forall\ k = 1, 2, \ldots, n$, $VTL_i[k] = max(VTL_i[k], VTL_j[k])$.

[2]Un reloj lógico es una abstracción del tiempo en términos de causalidad.

El defecto del reloj lógico de Lamport radica en que dados dos eventos e y e', con sus correspondientes relojes lógicos $L(e)$ y $L(e')$, si se cumple que $L(e) < L(e')$, esto no necesariamente implica que $e \to e'$. Por el contrario, si v y u son los vectores de tiempo de los eventos e y e', respectivamente, y además $v < u$, entonces no podemos implicar que el evento e ocurre antes que el evento e'; es decir $e \to e'$.

Propiedades de los vectores de tiempo

Si u y v son dos vectores con n procesos, éstos cumplen las siguientes propiedades:

a) $u \leq v$ Si $\exists i$ tal que $u[i] \leq v[i]$
b) $u \neq v$ Si $\exists i$ tal que $u[i] \neq v[i]$
c) $u < v$ Si $u \leq v \wedge u \neq v$
d) $u||v$ Si $\neg(u < v) \wedge \neg(v < u)$

La propiedad *(a)* establece que un vector de tiempo u es menor o igual que un vector de tiempo v, si existe un i-ésimo reloj lógico de u que es menor o igual que el i-ésimo reloj lógico de v; la propiedad *(b)*, que dos vectores de tiempo u y v son diferentes, si al menos un i-ésimo reloj lógico de los vectores es diferente ($u[i] \neq u[i]$); la propiedad *(c)*, que un vector u es menor que otro vector v, siempre y cuando ambos vectores cumplen con las dos propiedades anteriores; y por último, la propiedad *(d)*, que dos vectores (u y v) son concurrentes, si u no es menor que v y v no es menor que u.

2.3. Fundamentos del checkpointing

Iniciamos esta sección con la definición de lo que se entiende por *checkpoint*.

Definición 5. *Un* checkpoint *es un conjunto de información local que resguarda un proceso en un tiempo específico de su ejecución.*

El proceso resguarda la información de un *checkpoint* en un dispositivo de almacenamiento no volátil (*stable store*). La información resguardada corresponde al estado del proceso en un tiempo específico de su ejecución (libre de fallas). De esta forma, un proceso puede retornar a un punto especifico de su ejecución (instante en que realizó el *checkpoint*). El conjunto de información que se almacena de un *checkpoint* se compone de dos partes:

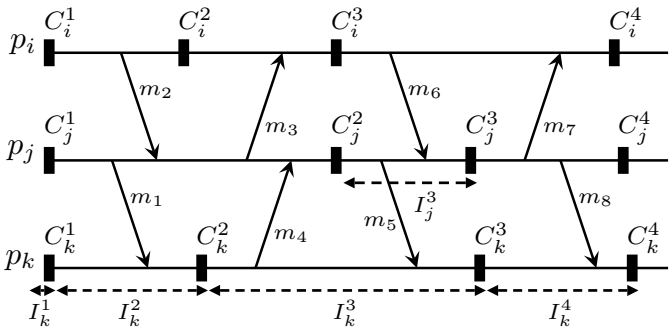


Figura 2.3: Patrón de comunicación y *checkpoints*.

el estado del proceso al momento de hacer el *checkpoint* y el estado del mecanismo de comunicación del proceso.

Definición 6. *Un patrón de comunicación y checkpoints (CCP, Communication and Checkpoint Pattern) es un par $(\widehat{E}, E_i)$ [54], $\widehat{E}$ es un conjunto parcialmente ordenado que modela un cómputo distribuido mientras que E_i es un conjunto de checkpoints locales definidos en $\widehat{E}$.*

En la Figura 2.3 mostramos un ejemplo de un CCP. En esta figura, un proceso es representado por una línea horizontal de tiempo; un *checkpoint* por un rectángulo, rectángulos sin rellanar para *checkpoints* forzados y rectángulos negros para los *checkpoints* locales, iniciales y finales; un mensaje es representado por una flecha, el inicio de la flecha representa al evento *send* y la cabeza o punta de la flecha al evento *delivery*; por último, el x-ésimo *intervalo de checkpoint* de un proceso p_i es denotado por I_i^x y representa la secuencia de eventos ocurridos entre los *checkpoints* C_i^{x-1} y C_i^x $(x > 1)$.

Definición 7. *Un snapshot global (SG) es un conjunto de checkpoints, uno por cada proceso que participa en el cómputo del sistema.*

Definición 8. *Un snapshot global se dice consistente (SGC) si no contiene checkpoints relacionados por la relación HB (Definición 1) [37, 14]. Por lo que, para cualquier par de checkpoints C_i^x y C_j^y de un SGC, se cumple lo siguiente:*

$$\neg(C_i^x \rightarrow C_j^y) \wedge \neg(C_j^y \rightarrow C_i^x)$$

Chandy and Lamport [14] fueron los primeros en definir la noción de *snapshot* global consistente para sistemas distribuidos. Para ellos, un *snapshot* global consistente es un

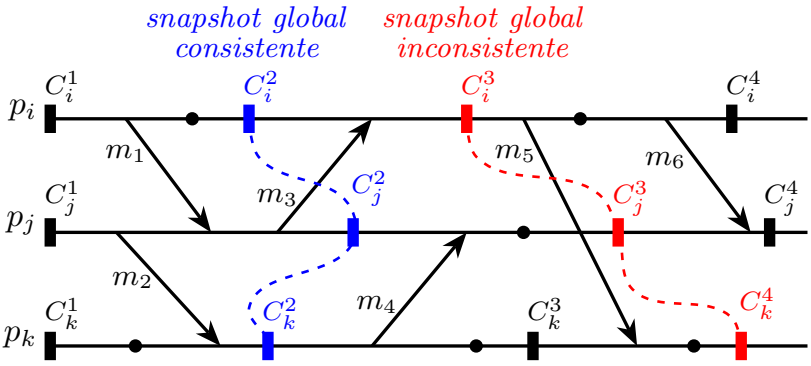


Figura 2.4: *Snapshot* global consistente y no consistente.

conjunto de estados (*checkpoints*, ver Figura 2.4), uno por cada proceso participante en el cómputo del sistema; en donde, si un estado almacena la recepción de un mensaje m, entonces, otro estado del conjunto, almacena el envío del mensaje m. De lo contrario, decimos que es un *snapshot* global inconsistente.

Netzer y Xu definieron la noción de *z-path* y *z-cycle* en [37]. Un *z-path* es una generalización de la relación HB (Definición 1), su definición formal es la siguiente:

Definición 9. *Un z-path(zigzag path) existe del checkpoint C_p^i al checkpoint C_q^j, si hay una secuencia de mensajes $m_1, m_2, \ldots, m_\ell$ tal que:*

a) m_1 es enviado por el proceso p después de C_p^i,

b) Si $m_k (1 \leq k < \ell)$ es recibido por el proceso r, entonces m_{k+1} es enviado por r en el mismo intervalo de checkpoint o posterior (m_{k+1} puede ser enviado antes o después de recibir m_k), y

c) m_ℓ es recibido por el proceso q antes de C_q^j.

Hay que notar la diferencia entre un *z-path* y un camino causal (*causal path*). Un camino causal relaciona a dos *checkpoints* por medio de la relación HB. Un *z-path* no siempre representa causalidad; de modo que, si hay una relación *z-path* entre dos *checkpoints*, esto no significa que hay un camino causal entre ellos. En este sentido, llamaremos *z-path causal* a un *z-path* formado por un camino causal de mensajes ($m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_\ell$), de lo contrario, lo llamaremos *z-path no causal*. En la figura 2.5 mostramos los dos tipos de *z-paths*. El *z-path* formado del *checkpoint* C_i^1 a C_k^2 es causal debido a que los mensajes m_α y $m_{\alpha+1}$, que lo forman, son causales ($m_\alpha \rightarrow m_{\alpha+1}$). Por otra parte, el *z-path* formado de

C_i^2 a C_k^3 es un *z-path* no causal, porque los mensajes m_β y $m_{\beta+1}$, que forma al *z-path*, son no causales, es decir, el envío del mensaje $m_{\beta+1}$ ocurre antes de la recepción del mensaje m_β en el proceso p_j.

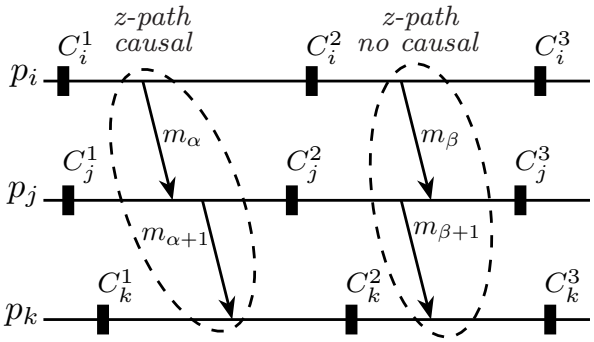


Figura 2.5: *z-path* causal y no causal.

Definición 10. *Un z-cycle (zigzag cycle) se forma cuando hay un z-path de un checkpoint C_i^x a si mísmo.*

Los *checkpoints* C_i^3, C_j^2, C_j^3 y C_k^2 de la Figura 2.6 tienen un *z-path* a si mismos cada uno, por lo que cada *checkpoint* está envuelto en un *z-cycle*. Por ejemplo, la secuencia de mensajes no causales $[m_6, m_5, m_4, m_3]$ forma el *z-path* de C_i^3 a si mismo y $[m_5, m_4]$ forma el *z-path* de C_j^2 a si mismo.

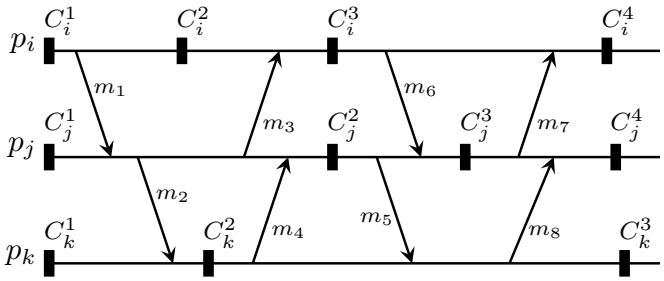


Figura 2.6: *z-paths* y *z-cycles* en un patrón de comunicación y *checkpoints*.

Con el objetivo de construir *snapshot* globales consistentes, a partir de las nociones de *z-path* y *z-cycle*, Netzer y Xu [37] introducen los siguientes Corolarios.

Corolario 1. Un *checkpoint* C_i^x puede pertenecer a un *snapshot* global consistente si C_i^x no tiene un *z-cycle* a si mismo.

Corolario 2. Dos *checkpoints* C_i^x y C_j^y (pertenecientes a diferentes procesos) pueden ser parte de un *snapshot* global consistente, si estos satisfacen lo siguiente:

a) Los *checkpoints* C_i^x y C_j^y no tienen un *z-cycle*, y

b) No existe un *z-path* entre los *checkpoints* C_i^x y C_j^y.

Para finalizar la sección, enunciamos dos definiciones y un teorema desarrollados por Hélary et al. en [22]. El teorema es una parte fundamental del desarrollo de esta investigación y es mencionado en muchas ocasiones en el resto del documento.

Definición 11. *Un checkpoint local C_j^y tiene una z-dependencia (z-depends) a un checkpoint local C_i^x, denotado por $C_i^x \xrightarrow{Z} C_j^y$, si se cumple una de las siguientes condiciones:*

a) $j = i \ \wedge \ y > x$, *ó*

b) hay un z-path de C_i^x a C_j^y.

Definición 12. *Un* z-cycle *es una* z-depends *de un* checkpoint *local C_i^x a sí mismo: $C_i^x \xrightarrow{Z} C_i^x$.*

Teorema 1. Las siguientes propiedades de un patrón de comunicación y checkpoints $(\widehat{E}, R_{\widehat{E}})$ son equivalentes:

a) $(\widehat{E}, R_{\widehat{E}})$ no tiene *z-cycle*.

b) Es posible etiquetar a los *checkpoints* locales de tal forma que:

$$A \xrightarrow{Z} B \Rightarrow A.t < B.t$$

donde $A.t$ y $B.t$ son relojes lógicos (Lamport [31]) de los *checkpoints* locales A y B, respectivamente.

Capítulo 3

Trabajos relacionados

En este capítulo describimos algunos trabajos relacionados con nuestra investigación. Algoritmos de *checkpointing* propuestos en diferentes ámbitos, y que en combinación con algunos métodos o técnicas desarrolladas para los sistemas distribuidos, intentan resolver la problemática de *checkpointing* en un ambiente heterogéneo.

En la Figura 3.1 mostramos una taxonomía jerárquica para los algoritmos de *checkpointing*. Los primeros niveles de esta taxonomía fueron introducidos por *Kalaiselvi* y *Rajaraman* en [24]. El nivel mas alto está estructurado por el número de procesadores. En este nivel tenemos a los *sistemas uniprocesador* y *sistemas multiprocesador*. Los algoritmos de *checkpointing* para sistemas uniprocesador son muy simples, estos por lo regular sólo consideran un conjunto de procesos que se ejecutan en un sólo procesador, por lo que un único *checkpoint* resguarda el estado de todos los procesos. Por otra parte, los algoritmos de *checkpointing* para sistemas multiprocesador son más complejos. En este caso, los procesos se ejecutan en varios procesadores, por lo que, se realizan tantos *checkpoints* simultáneos como procesadores tengamos.

En el segundo nivel de la Figura 3.1 tenemos a los algoritmos de *checkpointing* para sistemas multiprocesador, estos se dividen en *estáticos* y *dinámicos*. En los algoritmos estáticos, la generación de *checkpoints* se establece durante la *compilación*[1] de la aplicación. En este caso, los *checkpoints* se generan de forma automática y periódicamente durante la ejecución de la aplicación, por lo que no son muy versátiles. Por otra parte, los algoritmos dinámicos son más versátiles; establecen la generación de *checkpoints* en tiempo de ejecución, esto hace que se adaptan fácilmente a los cambios en el sistema.

[1] *Compilación* es el procedimiento por el cual, el código fuente de una aplicación pasa a ser código ejecutable de algún tipo de hardware.

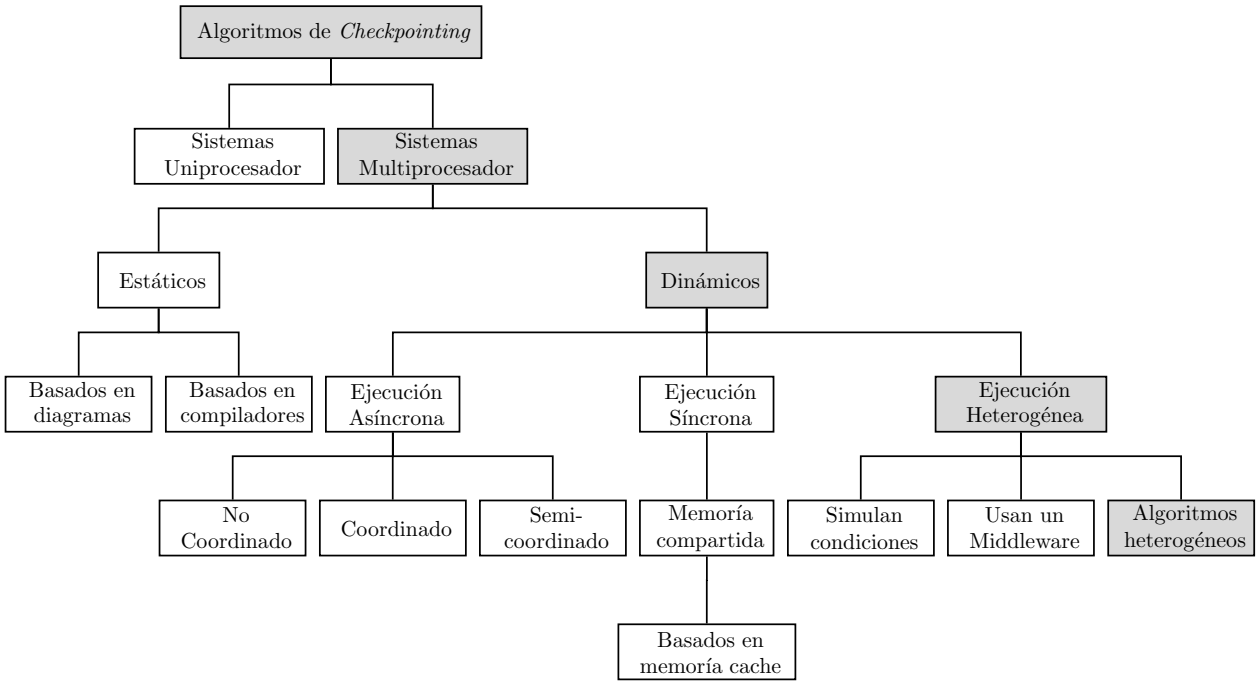


Figura 3.1: Taxonomía de algoritmos de *checkpointing*.

Los algoritmos de *checkpointing* dinámicos los podemos agrupar en tres grupos: *ejecución asíncronos*, *ejecución síncronos* y *ejecución heterogénea*.

Nuestro trabajo de investigación lo posicionamos en la categoría de algoritmos de *checkpointing* dinámicos con ejecuciones heterogéneas. En la Figura 3.1 ilustramos toda la rama jerárquica, con un tono gris obscuro, en la que posicionamos a nuestro trabajo de investigación. Por tal motivo, en lo que resta de este capítulo, describiremos los trabajos relacionados en la categoría; sin embargo, al realizar esta descripción, describimos también (de manear indirecta) a los algoritmos de *checkpointing* de ejecución síncrona y asíncrona.

Organizamos a los algoritmos de *checkpointing* para ejecuciones heterogéneas en tres clases:

Algoritmos de *checkpointing* que simulan condiciones de ejecución. En este grupo, tenemos a los trabajos que establecen condiciones para acoplar algoritmos de ejecución síncrono en asíncrono y asíncrono en síncrono.

Algoritmos de *checkpointing* que utilizan un Middleware. En este segundo grupo, tenemos a los trabajos que establecen, diseñan o desarrollan arquitecturas de capas o API (*Application Programming Interface*) para facilitar el acoplamiento entre diversos algoritmos de *checkpointing*.

Algoritmos de *checkpointing* para ambientes heterogéneos. En este último grupo, tenemos a los trabajos que desarrollan principios para algoritmos heterogéneos; es decir, algoritmos que se comportan de forma independiente al tipo de ejecución (síncrono o asíncrono) y que desde su diseño consideran diversos tipos de ejecución.

3.1. Algoritmos de *checkpointing* que simulan condiciones de ejecución

Los algoritmos de *checkpointing* han sido desarrollados tanto para *sistemas distribuidos* como para *sistemas paralelos* (ver [19, 24, 21, 30]). Estos algoritmo han madurado en las últimas década como mecanismos de *tolerancia a fallas*, y además, han sido utilizados ampliamente para la solucionar diversos problemas en los sistemas distribuidos; entre estos tenemos, la *depuración* (*debugging*), *balance de carga* (*workload balancing*) o *migración de procesos* (*process migration*), entre otros [19, 24].

Con el objetivo de comprender la naturaleza de la simulación de condiciones de ejecución. Introducimos primero a los algoritmos de *checkpointing* para sistemas distribuidos y paralelos, en seguida, abordamos el ámbito de la simulación de condiciones de ejecución.

3.1.1. Algoritmos de *checkpointing* para sistemas distribuidos

Los algoritmos de *checkpointing* para sistemas distribuidos, en general, asumen que los procesos se comunican únicamente por el paso de mensajes. Estos sistemas son prácticamente asíncronos, es decir:

a) El tiempo de ejecución de los eventos en un proceso se desconoce.

b) El límite de tiempo para la transmisión de mensajes entre procesos es arbitrario, se considera finito y no despreciable.

c) Los procesos no tiene un reloj global para sincronizar sus eventos.

d) Los procesos no cuentan con una memoria compartida entre ellos.

Los algoritmos de *checkpointing* para sistemas distribuidos son clasificados en tres categorías: *coordinado*, *no coordinado* y *comunicación inducida* [19]. A continuación, describiremos a cada una de estas clases:

Checkpointing coordinado

En los algoritmos de *checkpointing coordinado* (CC, *coordinated checkpointing*), los procesos participantes en el cómputo se coordinan para establecer en un tiempo especifico un *Snapshot Global Consistente* (SGC, ver Definición 8). La principal ventaja de estos algoritmos es que requieren almacenar únicamente un o dos SGC, mientras que sus principales desventajas son:

- *Overhead* adicional debido al intercambio de mensajes durante la generación de un SGC; este es no deseable si las fallas son poco frecuentes.
- Si un proceso falla durante la generación de un SGC, este puede bloquear las acciones del algoritmo de *checkpointing*.
- Existe la posibilidad de que uno o varios procesos sean bloqueados, no realicen acciones de cómputo, mientras el algoritmo genera un SGC.

Algunos ejemplos de algoritmos de *checkpointing* coordinado son los desarrollados en [11, 12]

Checkpointing no coordinado

Los algoritmos de *checkpointing no coordinados* (UCC, *uncoordinated checkpointing*) son más simples que los algoritmos de *checkpointing* coordinados. En estos algoritmos, cada proceso crea un *checkpoint* cada cierto tiempo de manera autónoma. Los procesos no necesitan coordinar acciones para generar un SGC durante su cómputo; sólo cuando una falla ocurre, estos se coordinan y utilizan sus *checkpoints* (generados) para establecer un SGC del sistema. Las principales ventajas de estos algoritmos son:

- La autonomía de cada proceso para realizar un *checkpoint*.
- La eliminación del *overhead* en los mensajes para coordinar un SGC (característica típica de estos algoritmo).

Por otra parte, las desventajas de los algoritmos de *checkpointing* no coordinados son:

- La formación del *efecto dominó* [42, 44]. El efecto dominó consiste en la imposibilidad de generar un SGC a partir de los *checkpoints* generados por cada proceso, lo cual se traduce como una pérdida del cómputo desarrollado por el sistema.

- La generación de una gran cantidad de *checkpoints* no útiles (no son parte de ningún SGC) que sólo degradan el desempeño del sistema.

Algunos ejemplos de algoritmos de *checkpointing* coordinado son los desarrollados en [7, 50]

Checkpointing de comunicación inducida

Los algoritmos de *checkpointing de comunicación inducida* (CIC, *communication-induced checkpointing*) son un intento por combinar los algoritmos de *checkpointing* coordinados y no coordinados. Los algoritmos CIC eluden el efecto dominó de los algoritmos no coordinados y permiten la generación de *checkpoints* autónomos y de forma asíncrona. Para lograr esto, los algoritmos CIC adicionan información de control (*piggyback*) en cada mensaje que envía un proceso, con el objetivo de identificar patrones de *checkpoints* potencialmente peligrosos. Un patrón peligroso es eliminado antes de que este ocurra por medio de la generación de un *checkpoint forzado*. Los patrones peligrosos son los *z-cycles* identificados por Netzer [37]. Así, un *checkpoint* forzado es inducido por el intercambio de información entre procesos, en un afán por prevenir y/o eliminar el efecto dominó.

Algunos algoritmos de este tipo generan suficientes *checkpoints* forzados para asegurar que todo *checkpoint* (local y forzado) sea parte de al menos un SGC [39, 35].

Los algoritmos de *checkpointing* de comunicación inducida conservan, en su mayoría, las ventajas de los algoritmos coordinados y no coordinados, mientras que las principales desventajas, de acuerdo al análisis desarrollado en [3], son la cantidad de *checkpoints* forzados que generan (*induced storage overhead*) y el *Overhead* en los mensajes. En general, todo algoritmo de comunicación inducida, desconoce de antemano, la cantidad de *checkpoints* forzados que generará [19].

Hélary et al. [22] desarrollaron el algoritmo FI (*Fully Informed*) de *checkpointing* de comunicación inducida, este es considerado uno de los mejores algoritmos CIC en la historia, debido a que hace uso de toda la información causal posible [48]. FI utiliza el Teorema 1 (sección 2.3) para eliminar los *z-cycles* de cualquier CCP, por lo que todo *checkpoint*

que genera el algoritmo pertenecerá a un SGC. FI elimina todos los posibles *z-cycles* permitiendo que los procesos evalúen localmente y de manera independiente una condición, esta condición determina cuando un patrón de comunicación entre *checkpoints* forman o tiene posibilidad de formar un *z-cycle*; de manera que cuando un proceso detecta la formación de un posible *z-cycle* genera un *checkpoint* forzado para romper este patrón. Las principales desventajas de este algoritmo CIC son el *overhead* de mensajes y el número de *checkpoints* generados. En el capítulo 4 describimos más detalladamente el funcionamiento y estructuras de datos de este algoritmo.

Por otra parte, Luo y Manivannan [33] desarrollaron el algoritmo FINE (*Fully Informed aNd Efficient*). FINE es un algoritmo basado en FI, utiliza un mecanismo de reloj lógico diferente al tradicional (reloj lógico de Lamport [31]) que llama *TDE-timestamp* (*Transitive Dependency Enabled timestamp*), pero con el mismo significado de causalidad que los relojes lógicos. Este mecanismo utiliza un vector $TDE\_TS_i[]$ de tamaño n (n es el número de procesos) que permite enumerar intervalos de *checkpoints* e incrementos en cada intervalo. Con el objetivo de disminuir el *overhead* en los mensajes del algoritmo, el mecanismo TDE-*timestamp* divide el número de bits de cada reloj lógico del vector $TDE\_TS_i[]$ en dos partes: una parte para etiquetar el último *checkpoint* (intervalo de *checkpoint*) del k-ésimo proceso y otra para etiquetar un incremento desde el último *checkpoint* del k-ésimo proceso (incremento dentro del intervalo). En este sentido, los incrementos dentro de un intervalo de *checkpoint* están limitados, y en consecuencia, estos incrementos pueden desestabilizar al algoritmo. Por ejemplo, si en un cómputo un proceso recibe más de los incrementos que puede manejar en un intervalo de *checkpoint* esto introduciría problemas en la captura de la causalidad del sistema. La función del mecanismo *TDE-timestamp* de FINE es con el afán de detectar *z-cycles* en un intervalo de *checkpoints* por medio de un sólo vector de relojes, lo cual FI hace con un vector de relojes, dos vector de n bits y un reloj lógico; sin embargo, FI tiene la flexibilidad de poder incrementar su reloj lógico local tanto como le permita su estructura de reloj lógico, algo que FINE esta limitado a realizar. Por otra parte, otra diferencia marcada entre FINE y FI radica en la condición o proposición que utilizan para detectar *z-cycles*, FINE parte de la condición desarrollada en FI, sin embargo, después de un análisis (que desarrollan los autores), determinan que esta condición detecta falsos *z-cycles* y generan una nueva condición que descarta a estos falsos *z-cycles*, reduciendo con esto la cantidad de *checkpoints* generados por el algoritmo FINE.

Para finalizar esta sección, en el cuadro 3.1 mostramos una comparación cualitativa

entre los tres tipos de algoritmos de *checkpointing* mencionados anteriormente.

Cuadro 3.1: Comparativo de algoritmos de *checkpointing* para sistemas distribuidos.

Característica	*Algoritmo de checkpointing*		
	coordinado	no coordinado	comunicación-inducida
overhead	alto	muy bajo	bajo
checkpoint/proceso	uno	varios	algunos
recuperación	difícil	fácil	fácil
efecto dominó	no	posible	no

3.1.2. Algoritmos de *checkpointing* para sistemas paralelos

Los algoritmos *checkpointing* paralelos, como los desarrollados en [1, 53, 6], por lo general, asumen que los procesos se comunican exclusivamente por una memoria compartida y que el sistema es prácticamente síncrono; es decir, hay una memoria compartida y un reloj global que permite sincronizar acciones entre procesos, el tiempo de comunicación entre procesos es despreciable ó se tiene límites establecidos. Por lo que estos algoritmos se orientan más a un modelo de ejecución síncrono [24].

3.1.3. Simulación de condiciones de ejecución síncrono en sistemas de ejecución asíncrono

La simulación de condiciones de ejecución síncrono en sistemas de ejecución asíncrono, es una técnica usada en sistemas distribuidos llamada *sincronizadores* (*Synchronizers*), en algunos casos también ha sido usada a nivel hardware [34]. Esta técnica es utilizada principalmente en sistemas distribuidos para dar soporte a algoritmos de naturaleza síncrona sobre sistemas de naturaleza asíncrona. De acuerdo a Lynch [34] podemos encontrar tres configuraciones llamadas *alfa*, *beta* y *gamma*. Cada configuración tiene un determinado grado de sincronización y *overhead* de mensajes. La desventaja principal de este tipo de mecanismo es el *overhead* de los mensajes, que en algunas situaciones es bastante alto. Además, la técnica de sincronizadores no es apropiada para algoritmos tolerantes a fallas debido al *problema de convenios o acuerdos* (*agreement*) en sistemas asíncronos (Lynch [34] capítulos 16 y 21). El problema de convenios en sistemas síncronos es relativamente fácil debido al nivel de sincronización entre procesos del sistema; sin embargo, en sistemas

asíncronos, por lo general, no es posible asegurar que el problema de convenios pueda resolverse en presencia de fallas en el sistema.

3.1.4. Simulación de condiciones de ejecución asíncrono en sistemas de ejecución síncrono

Charron-Bost et al. [15] establecen que no es posible acoplar mecanismos de comunicaciones asíncronos a través de mecanismos de comunicación síncronos de manera arbitraria. En su trabajo de investigación, establecen una jerarquía de las clases de cómputo distribuido. Esta jerarquía indica que todo cómputo síncrono puede ser desarrollado por un cómputo asíncrono; pero no todo cómputo asíncrono puede ser desarrollado por un cómputo síncrono. En su investigación, analizan y caracterizan el cómputo RSC (*Realizable with Synchronous Communication*), entre otros. Un cómputo RSC no forma dependencias cíclicas con los mensajes del sistema; no hay mensajes que se crucen, lo que permite que la gráfica de un cómputo asíncrono de clase RSC puede transformarse en una gráfica de cómputo síncrono; simplemente, con el movimiento de las recepciones de los mensaje (del cómputo RSC) al punto del envío correspondiente de cada mensaje, formándose mensajes con flechas verticales en la gráfica (gráfica de cómputo síncrono).

Debido a lo anterior, no resulta viable el acoplamiento de algoritmos de *checkpointing* en todos los casos; posiblemente sólo en algunos casos de CCP (patrones de comunicación y *checkpoints*) que generen un cómputo RSC. En otras palabras, sólo en aplicaciones que realicen un cómputo RSC podríamos utilizar esta técnica o mecanismo, por lo que un algoritmo de *checkpointing* de este tipo tendría muchas limitaciones y muy poca flexibilidad.

3.2. Algoritmos de *checkpointing* que utilizan un Middleware

El término *Middleware* se aplica a una capa de software que proporciona una abstracción de la programación y oculta la heterogeneidad de redes, hardware, sistemas operativos y lenguajes de programación [16].

Tsujita et al. [49] desarrollaron una librería MPI (*Message-Passing Interface*) flexible para dar soporte a operaciones de cómputo en un entorno heterogéneo, por lo que, los usuarios pueden usar las funciones de la librería sin conocimiento del mecanismo de comunicación. Sin embargo, este esquema tiene los mismos problemas de los algoritmos de *checkpointing* que simulan condiciones de ejecución.

Kovács et al. [28] introducen el prototipo TCKPT (*TotalCheckpoint*) desarrollado para un ambiente de *ClusterGrid*[2]. El prototipo desarrolla un *Middleware* para proporcionar a las aplicaciones de un *ClusterGrid* un mecanismo de tolerancia a fallas y un mecanismo de migración de tareas, ambos basados en un *checkpointing* a nivel de librería de usuario.

Los algoritmos basados en *Middleware* comparten las mismas desventajas que los algoritmos que simulan condiciones, pero una de las desventajas más criticas es la informalidad de la interacción entre las diversas primitivas de comunicación (síncrona-asíncrona y asíncrona-síncrona). Por lo regular, está permanece oculta o no está definida claramente, o bien, tienen grandes limitaciones [17].

Por otra parte, las principales ventajas de este enfoque radican en la reducción de complejidad, ahorro de trabajo, y la incorporación de diversos algoritmos de *checkpointing* de manera transparente a la aplicación, al programador y posiblemente a otros *Middlewares*.

3.3. Algoritmos de *checkpointing* para ambientes heterogéneos

Con base en el estudio realizado, identificamos qué algoritmos de *checkpointing* para ambientes heterogéneos con modelos de ejecución síncrono y modelos de ejecución asíncrono no han sido desarrollados hasta el momento. Los trabajos que más se acercan a lo que podríamos considerar un algoritmo heterogéneo son los trabajos realizados por Tantikul y Manivannan [47] y Cao et al. [13].

El trabajo de Tantikul y Manivannan [47], desarrolla un algoritmo de *checkpointing* para sistemas distribuidos multi-hilos. El algoritmo de *checkpointing* que proponen, resuelve la problemática de realizar *checkpoints* en procesos e hilos. Los algoritmos tradicionales de *checkpointing* para sistemas distribuidos asumen que cada proceso se ejecuta en un procesador; si aplicamos estos algoritmos a procesos e hilos, generaremos un falso problema de causalidad entre procesos que ejecuten hilos y un *overhead* injustificado; debido a que un proceso contiene a un conjunto finito de hilos.

Consideramos que el trabajo de investigación de Tantikul y Manivannan está relacionado con el nuestro, debido a que los hilos (que utiliza su algoritmo de *checkpointing*)

[2]Un *ClusterGrid* es un *Grid* que contiene *clusters* como componentes indivisibles. Un *Grid* es una infraestructura que permite la integración y el uso de equipo de alto desempeño, y está administrado por dos o más instituciones u organizaciones. Un *cluster*, por otra parte, es un conjunto de computadoras o nodos constituidos mediante la utilización de componentes de hardware comunes y que se comportan como un sólo equipo de alto desempeño.

tienen una ejecución síncrona (comparten una memoria, un reloj global, etcétera —los recursos del proceso son compartidos por todos los hilos—), mientras que los procesos sin hilos (del mismo algoritmo) tienen una ejecución asíncrona (no comparten memoria, no tiene un reloj global, etcétera); sin embargo, nuestro problema es aun más general, en el caso de los hilos (ejecución síncrona), no hay concurrencia entre ellos (sólo se ejecuta uno a la vez); mientras que para nuestro problema, nosotros si tenemos concurrencia entre los modelos de ejecución (síncrono y asíncrono).

Cao et al. [13] por su parte, desarrollan un algoritmo de *checkpointing* para sistemas distribuidos híbridos. Para ellos, un sistema distribuido híbrido es aquel sistema que contiene un cierto número de subsistemas que colaboran en la ejecución de un programa distribuido. La comunicación entre subsistemas, y, entre procesos, es a través de paso de mensajes, por lo que sólo se tiene un modelo de ejecución (asíncrono). Los subsistemas son heterogéneos en la clase de algoritmo de *checkpointing* que usan. Por lo que, el algoritmo de *checkpointing* desarrollado para todo el sistema es una combinación de algoritmos de *checkpointing*. En este trabajo, sólo se manejan algoritmos de *checkpointing* coordinado y no coordinado. Cada subsistema utiliza un algoritmo de *checkpointing* coordinado, mientras que un algoritmo de *checkpointing* no coordinado toma los resultados de cada subsistema y forma un SGC (*snapshot* global consistente) del sistema. De tal forma que cada SGC de un subsistema es tratado por el algoritmo de *checkpointing* no coordinado como un *checkpoint* del sistema.

En el cuadro 3.2 mostramos una comparación cuantitativa de las características de los dos algoritmos mencionados anteriormente.

Cuadro 3.2: Comparación de algoritmos de *checkpointing* con características heterogéneas.

Algoritmo	Bloqueo	Ejecución asíncrona	Ejecución síncrona	Tipo de algoritmo	*Overhead* en mensajes	*Overhead* en espacio
Tantikul [47]	no	si	si *(hilo)*	comunicación inducida	$O(1)$	*no definido*
Cao [13]	posible	si	no	Híbrido Coordinado a nivel subsistema No coordinado a nivel sistema global	$O(M)$	$O(M)$ a nivel *subsistema* *no definido* a *nivel global*

N es el número de procesos en el sistema.

M es el número máximo de procesos de un subsistema, tal que $M \leq N$.

Capítulo 4

Algoritmo S-FI de comunicación inducida

En este capítulo presentamos el algoritmo S-FI (*Scalable Fully-Informed*) de comunicación inducida que desarrollamos en nuestra investigación. El contenido que presentamos en este capítulo, fue utilizado para la publicación del artículo "*A Scalable Communication-Induced Checkpointing Algorithm for Distributed Systems*" [9].

El algoritmo S-FI es una parte fundamental de nuestro algoritmo de *checkpointing* para ambientes heterogéneos; su objetivo es atacar el problema de *overhead* de mensajes que tienen los algoritmos de comunicación inducida.

El algoritmo S-FI se basa en los principios introducidos por el algoritmo FI (*Fully-Informed*) de *checkpointing* propuesto por Hélary et al. [22] y el protocolo IPT2 (Immediate Predecessor Tracking 2) de rastreo propuesto por Anceaume et al. [4]. Específicamente, S-FI usa el Teorema 1 de la Sección 2.3 y la *condición de checkpoint forzado*[1] $\mathscr{C}2''$ del algoritmo FI (que describimos más adelante a detalle) para prevenir la formación de *z-cycles*, y hace uso del IPT2 con base en la relación IDR (ver Sección 2.2) para reducir el *overhead* de mensaje.

Para fusionar los principios de FI e IPT2 en S-FI, definimos primero una condición de *checkpoint* forzado inicial, que llamamos $\mathscr{D}$. Esta condición la formulamos al igual que $\mathscr{C}2''$, con estructuras estáticas, pero en términos de la relación IDR entre *checkpoints*, lo

[1] La *condición de checkpoint forzado*, de un algoritmo CIC, es una proposición que puede ser evaluada por un proceso de manera local, con el objetivo de generar un *checkpoint* local antes de la entrega de un mensaje recibido, esto con el afán de eliminar una posible formación de un *z-cycle* (ver Sección 2.3).

que significa, que el tamaño de las estructuras usadas en ambas condiciones es constante e igual. Posteriormente, demostramos que la condición $\mathscr{D}$ es equivalente a $\mathscr{C}2''$, y con esto, satisfacemos el Teorema 1. Después, redefinimos la condición $\mathscr{D}$ con el objetivo de poder utilizar estructuras dinámicas, a esta nueva condición la llamamos $\mathscr{D}'$. En este caso, el tamaño de las estructuras de datos a analizar se adaptan dinámicamente y acorde al comportamiento de la relación IDR entre *checkpoints* del sistema. Con base en esta última condición, diseñamos el algoritmo S-FI que presentamos en los cuadros 4.1 y 4.2. En la parte final de este capítulo, presentamos un análisis formal del *overhead* de mensajes, la simulación de nuestro algoritmo S-FI y las conclusiones de esta parte de la investigación.

Dado que la condición $\mathscr{C}2''$ de FI es fundamental para nuestra investigación, iniciamos la sección con una descripción detallada de los principales componentes de esta condición.

4.1. Descripción del algoritmo de referencia FI

Con el objetivo de satisfacer el Teorema 1, el algoritmo FI toma dos clases de *checkpoints*: *checkpoints* locales y *checkpoints* forzados. Los *checkpoints* locales son tomados por cada proceso en el sistema y únicamente dependen de la aplicación. Los *checkpoints* forzados son tomados por cada proceso en el sistema para asegurar que todo *checkpoint* (local y forzado) pueda pertenecer a algún *snapshot* global consistente (Definición 8).

Para lograr esto, en el algoritmo FI, cada proceso p_i evalúa la *condición de checkpoint forzado*(CCF) $\mathscr{C}2''$ después de la recepción de un mensaje. Si la condición $\mathscr{C}2''$ es verdadera, entonces p_i está forzado a tomar un *checkpoint* local. Esta acción rompe un *z-path* que contiene a un *checkpoint* que eventualmente puede formar un *z-cycle*.

Hélary et al. [22] definen varias CCF en [22], específicamente usamos la condición $\mathscr{C}2''$ porque esta necesita una menor cantidad de información para ser evaluada, es la mas óptima en este sentido. La condición $\mathscr{C}2''$ fue definida como:

$$\begin{aligned}\mathscr{C}2'' \equiv ((\exists k : sent\_to_i[k] \wedge m.greater[k]) \wedge m.lc > lc_i) \\ \vee (ckpt_i[i] = m.ckpt[i] \ \wedge \ m.taken[i]),\end{aligned}$$

donde:

- $sent\_to_i[1 \ldots n]$ es un arreglo booleano. $sent\_to_i[k]$ es *true* si p_i ha enviado un mesaje al proceso p_k desde su último *checkpoint*.

- lc_i es un entero que representa un reloj lógico de Lamport. Este es administrado por el proceso p_i, cuando p_i envía un mensaje m, el valor actual de lc_i es incluido en m (denotado por $m.cl$).

- $greater_i[1 \ldots n]$ es un arreglo booleano. $greater_i[k]$ es *true* si $lc_i > lc_k$. $greater_i[i]$ siempre mantiene un valor $false$. Este arreglo es actualizado de la siguiente forma:

  - Cuando p_i toma un *checkpoint* (local o forzado), para cada $k \neq i$, $greater_i[k]$ es igual a *true*. Cuando p_i envía un mensaje m, este arreglo es incluido en m (denotado por $m.greater[]$).

  - Cuando p_i recibe un mensaje m, este ejecuta las siguientes actualizaciones:

    case
    $m.lc > lc_i \rightarrow \forall k \neq i$ **do** $greater_i[k] := m.greater[k]$; **enddo**
    $m.lc = lc_i \rightarrow \forall k$ **do** $greater_i[k] := greater_i[k] \wedge m.greater[k]$; **enddo**
    $m.lc < lc_i \rightarrow$ **skip**
    endcase

- $ckpt_i[1...n]$ es un vector de relojes [31] que cuentan cuantos *checkpoints* han sido tomados por cada proceso. $ckpt_i[k]$ es el número de *checkpoints* tomados por p_k y conocido por p_i. Cuando p_i envía un mensaje m, este vector es incluido en m (denotado por $m.ckpt[]$).

- $taken_i[1...n]$ es un arreglo booleano. $taken_i[k]$ es *true* si hay un *z-path* causal del último *checkpoint* de p_k conocido por p_i a el siguiente *checkpoint* de p_i, y este *z-path* causal incluye un *checkpoint*. Este arreglo es administrado en la siguiente forma:

  - Cuando p_i toma un *checkpoint*, para cada $k \neq i$, $taken_i[k]$ es *true*. $taken_i[i]$ mantiene siempre el valor de $false$. Cuando p_i envía un mensaje m, este arreglo es incluido en m (denotado por $m.taken[]$).

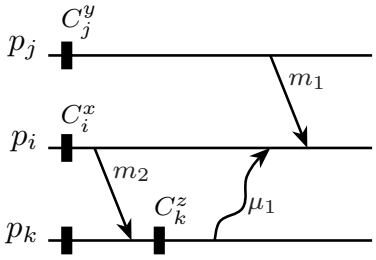
  - Cuando p_i recibe m, este actualiza $taken_i[]$ en la siguiente forma:

    $\forall k \neq i$ **do**
    case
    $m.ckpt[k] > ckpt_i[k] \rightarrow taken_i[k] := m.taken[k]$;
    $m.ckpt[k] = ckpt_i[k] \rightarrow taken_i[k] := (m.taken[k] \vee taken_i[k])$;
    $m.ckpt[k] < ckpt_i[k] \rightarrow$ **skip**
    endcase
    enddo

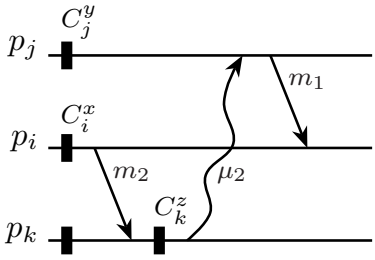
La condición $\mathscr{C}2''$ puede ser organizada en tres partes, y expresada como:

$$\mathscr{C}2'' \equiv (FI_a \wedge FI_b) \vee FI_c,$$
$$donde:$$
$$FI_a \equiv (\exists k : sent\_to_i[k] \wedge m.greater[k])$$
$$FI_b \equiv m.lc > lc_i$$
$$FI_c \equiv ckpt_i[i] = m.ckpt[i] \ \wedge \ m.taken[i]$$

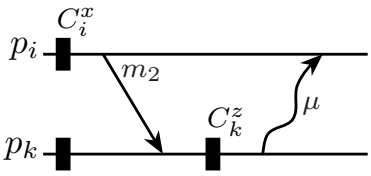
El objetivo de FI_a y FI_b es detectar *z-paths* no causales, mientras FI_c está orientado a identificar *z-paths* causales. Hélary et al. [22] usan la parte FI_C para identificar un *z-path* causal que envuelve a un *z-cycle*, ver Figura 4.1(c). Note que este *z-path* es una generalización de los *z-paths* causales que mostramos en las Figuras 4.1(a) y 4.1(b). En otras palabras, si la cadena de mensajes μ de la Figuras 4.1(c) es igual a la cadena de mensajes μ_1 entonces ambos esquemas representan el mismo escenario; de la misma forma, si $\mu = [\mu_2, m_1]$ los esquemas de las Figuras 4.1(b) y 4.1(c) serían los mismos.



(a) *z-path* con una cadena de mensajes μ_1.



(b) *z-path* con una cadena de mensajes μ_2 más un mensaje m_1.



(c) Abstracción de un *z-path* causal.

Figura 4.1: Detección de cualquier *z-path* causal en la recepción en un mensaje.

4.2. Condición de *checkpoint* forzado del algoritmo S-FI

Con el objetivo de capturar el mismo comportamiento que la condición $\mathscr{C}2''$ pero con los beneficios de la relación IDR, nosotros definimos la condición de *checkpoint* forzado

$\mathscr{D}$. Hay dos principales diferencias entre la condición $\mathscr{D}$ y $\mathscr{C}2''$. Primero, el vector $ckpt_i[]$, el cual tiene un crecimiento estricto monotónico, es reemplazado en S-FI por el vector $lc\_ckpt_i[]$ que presenta también un crecimiento estricto no monotónico. Segundo, el arreglo $taken_i[]$, usado en FI, es remplazado por el arreglo booleano $idr\_ckpt_i[]$. A través de $idr\_ckpt_i[]$ nosotros identificamos si un par de consecutivos *checkpoints* están IDR relacionados. Dos *checkpoints* relacionados vía IDR, significa que: *a)* hay un camino causal de mensajes entre ellos o son dos *checkpoints* locales consecutivos de un proceso; *b)* no hay un *checkpoint* intermedio entre ellos. Por otra parte, si dos *checkpoints* locales consecutivos de un proceso no están IDR relacionados, esto indica que hay un *z-path* causal con un *checkpoint* intermedio entre ellos. Este último comportamiento es sumamente importante para nosotros, debido a que la ruptura de la relación IDR entre *checkpoints* locales consecutivos, significa que hemos detectado un *z-cycle*.

La condición $\mathscr{D}$ es definida como sigue:

$$\mathscr{D} \equiv (SFI_a \wedge SFI_b) \vee SFI_c,$$

$$\begin{aligned}
&donde: \\
&SFI_a \equiv (\exists k : sent_to_i[k] \wedge m.greater[k]) \\
&SFI_b \equiv \mathbf{max}(m.lc_ckpt) > lc_i \\
&SFI_c \equiv lc_ckpt_i[i] = m.lc_ckpt[i] \wedge \neg m.idr_ckpt[i]
\end{aligned}$$

SFI_a y SFI_b tienen el mismo objetivo que FI_a y FI_b de la $\mathscr{C}2''$, respectivamente. SFI_c al igual que FI_c es usado para detectar *z-paths* causales (ver Figura 4.1), con la diferencia de que SFI_c hace uso de las relaciones IDR entre *checkpoints*. Note que SFI_b y SFI_c comparten la estructura $lc\_ckpt[]$. Esto evita incluir el reloj lógico del emisor de m, como lo detallamos más adelante. Las variables y estructuras de datos usadas por $\mathscr{D}$ son las siguientes:

- lc_i es un reloj lógico usado como en $\mathscr{C}2''$; sin embargo, este no es incluido en los mensajes que envía p_i.
- $lc\_ckpt_i[1 \ldots n]$ es un vector de relojes lógicos. $lc\_ckpt_i[i]$ tiene el valor del reloj lógico lc_i de p_i cuando este tomó su último *checkpoint*. $lc\_ckpt_i[k]$ tiene el valor del reloj lógico lc_k de p_k cuando este tomó su último *checkpoint*, y que es conocido por p_i. El vector es administrado en la siguiente forma:
  - Cuando p_i toma un *checkpoint*: $lc_i := lc_i + 1$, y $lc\_ckpt_i[i] := lc_i$. Cuando p_i envía un mensaje m, $lc\_ckpt_i[]$ es incluido en m (denotado por $m.lc\_ckpt[]$).

- Cuando p_i recibe un mensaje m, este actualiza al vector $lc\_ckpt_i[]$ como sigue:
  $\forall k \neq i$ **do**
  case
  $m.lc\_ckpt[k] > lc\_ckpt_i[k] \rightarrow lc\_ckpt_i[k] := m.lc\_ckpt[k];$
  $m.lc\_ckpt[k] < lc\_ckpt_i[k] \rightarrow$ **skip**
  $m.lc\_ckpt[k] = lc\_ckpt_i[k] \rightarrow$ **skip**
  endcase
  enddo

  max(u) es una función que obtiene el máximo valor almacenado en un arreglo u. Note que el reloj lógico del emisor es determinado a partir del vector $lc\_ckpt[]$ incluido en m $(max(m.lc\_ckpt[]))$.

- $idr\_ckpt_i[1 \ldots n]$ es un arreglo booleano. El valor de $idr\_ckpt_i[k]$ es $true$, si hay una relación IDR entre el último *checkpoint* de p_k conocido por p_i y el siguiente *checkpoint* de p_i. Este arreglo es administrado en la siguiente forma:

  - Cuando p_i toma un *checkpoint*, este establece su $idr\_ckpt_i[i]$ a $true$. Y para cada $k \neq i$, $idr\_ckpt_i[k] := false$.

    Cuando p_i envía un mensaje m, este incluye el arreglo $idr\_ckpt_i[]$ a m (denotado por $m.idr\_ckpt[]$).

  - Cuando p_i recibe un mensaje m, este actualiza al vector $idr\_ckpt_i[]$ como sigue:
    $\forall k \neq i$ **do**
    case
    $m.lc\_ckpt[k] > lc\_ckpt_i[k] \rightarrow idr\_ckpt_i[k] := m.idr\_ckpt[k];$
    $m.lc\_ckpt[k] = lc\_ckpt_i[k] \rightarrow idr\_ckpt_i[k] := (m.idr\_ckpt[k] \wedge idr\_ckpt_i[k]);$
    $m.lc\_ckpt[k] < lc\_ckpt_i[k] \rightarrow$ **skip**
    endcase
    enddo

Ahora, con el siguiente Teorema establecemos la equivalencia de las condiciones $\mathscr{D}$ y $\mathscr{C}2''$.

Teorema 2. La condición $\mathscr{D}$ es equivalente a la condición $\mathscr{C}2''$.

La demostración del Teorema anterior la mostramos en el anexo, para no desviar la atención del lector.

La equivalencia entre $\mathscr{D}$ y $\mathscr{C}2''$, desde la perspectiva de un algoritmo de *checkpointing*, significa que ambas condiciones habilitan la generación de un mismo número de *checkpoints* forzados, cuando procesan un patrón de comunicación y *checkpoints* en particular.

4.2.1. Condición de *checkpoint* forzado del algoritmo S-FI con estructuras dinámicas

Desde el punto de vista algorítmico, la evaluación de la condición $\mathscr{D}$ en un proceso, requiere de los arreglos booleanos $greater[]$ y $idr\_ckpt[]$, y del vector $lc\_ckpt[]$. Esto implica, un *overhead* constante por cada mensaje enviado e igual a n enteros más $2n$ bits; sin embargo, con los principios introducidos en el protocolo IPT2 [4], podemos evaluar la condición $\mathscr{D}$ con menos información. Esta información necesaria y suficiente para evaluar $\mathscr{D}$, la determina el conjunto de *checkpoints* IDR relacionados en un intervalo de *checkpoints*. Esto implica que la información que envía un proceso se adapta y es determinada dinámicamente, dando como resultando una reducción significativa del *overhead* de mensajes. La condición basada en la relación IDR y expresada con estructuras dinámicas es definida como:

$$\mathscr{D}' \equiv (SFI'_a \wedge SFI'_b) \vee SFI'_c,$$

donde :

$SFI'_a \equiv [\exists k : sent\_to_i[k] \ \wedge \ ((\exists y \in m.\psi, \ y.id = k : y.greater) \ \vee \ (\nexists y \in m.\psi, \ y.id = k))]$

$SFI'_b \equiv \mathbf{max}(m.\psi) > lc_i$

$SFI'_c \equiv (\exists z \in m.\psi, \ z.id = i : lc\_ckpt_i[i] = z.cl\_ckpt \ \wedge \neg z.idr\_ckpt)$

Las partes SFI'_a, SFI'_b y SFI'_c en la condición $\mathscr{D}'$ corresponden a las partes SFI_a, SFI_b y SFI_c de $\mathscr{D}$, respectivamente. Las estructuras de datos y variables usadas en esta condición son:

- El arreglo $sent\_to_i[]$, el vector $lc\_ckpt_i[]$ y el reloj lógico lc_i tienen el mismo significado y manejo como en la condición $\mathscr{D}$.

- $m.\psi$ es una estructura de datos formada por tuplas. Cada tupla contiene: un identificador de proceso, id; un reloj lógico, lc_ckpt; y dos valores booleanos, idr_ckpt y $greater$.

$$tuple \equiv (id, lc\_ckpt, idr\_ckpt, greater)$$

$m.\psi$ es construida a partir de las estructuras $lc\_ckpt_i[]$, $idr\_ckpt_i[]$ y $greater_i[]$; por lo que, $m.\psi$ es una copia parcial o completa de tales estructuras.

La función $\mathbf{max}(m.\psi)$ obtiene el reloj lógico máximo ($y.lc\_ckpt$) incluido en alguna tupla $y \in m.\psi$.

Para el problema de *rastreo de predecesores inmediatos*, Anceaume et al. [4] identificaron la información de control que cada proceso p_i debe incluir en un mensaje. Para esto, ellos definen la condición abstracta $K(m,k)$ y la condición $K2(m,k)$. K identifica la información que no se necesita incluir en un mensaje. K2 es una implementación que aproxima K, y puede ser evaluada por un proceso localmente. Basándonos en K2, definimos la condición K3 que es también una aproximación de la condición abstracta K. K3 está orientada a satisfacer $\mathscr{D}'$ y la definimos de la siguiente forma:

Definición 13.

$$K3(m,k) \equiv ((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k] = 1)) \vee (send(m).lc\_ckpt_i[k] = 0)$$

donde:

$send(m).x$ denota el valor de una variable x, al momento que un proceso ejecuta el evento $send(m)$. Así, $send(m).lc\_ckpt_i[k]$ debe interpretarse como: el k-ésimo valor del vector de reloj $lc\_ckpt_i[k]$ al momento que p_i envía m. De forma similar, interpretamos los valores para $send(m).idr\_ckpt_i[k]$ y $send(m).T_i[j,k]$.

T_i es una matriz booleana que cada proceso administra con el objetivo de satisfacer la siguiente propiedad:

Propiedad 1. Para cada mensaje que p_i envía a p_j,

$$\begin{aligned}(send(m).T_i[j,k] = 1) \quad \Rightarrow \quad & (send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]) \wedge \\ & (\mathbf{max}(send(m).lc\_ckpt_i[]) > send(m).lc\_ckpt_i[k])\end{aligned}$$

donde:

$pred(receive(m))$ denota al *checkpoint* C_j^x, en la secuencia H_j de p_j, que precede inmediatamente a la recepción de m. Note que $pred(receive(m).lc\_ckpt_j[k])$ es el valor más reciente de $lc\_ckpt_j[k]$ conocido por p_i, al momento que este envía m.

La propiedad anterior captura la noción de conocimiento de información en el sistema. Cuando $send(m).T_i[j,k] = 1$, significa que el proceso p_i no conoce información del procesos p_k más actual (ó reciente) que la que conoce p_j.

En general, cuando K3 se satisface, significa que la tupla:

$$(k, lc\_ckpt[k], idr\_ckpt[k], greater[k])$$

no es útil para actualizar la información en el proceso receptor. Por lo que, el proceso emisor no debería adjuntar la tupla a los mensajes que envía.

La demostración de $K3(m,k) \Rightarrow K(m,k)$ la presentamos en el apéndice B.

Con el objetivo de satisfacer la Propiedad 1, la matriz T_i es administrada de la siguiente forma:

T0 T_i es inicializada a *true*. $\forall(j,k) : T_i[j,k] := 1$.

T1 Cuando p_i toma un *checkpoint*, este reinicia la i-ésima columna de su matriz T_i. $\forall j \neq i : T_i[j,i] := 0$. Cuando p_i envía un mensaje, T_i no se actualiza o modifica.

T2 Cuando p_i recibe un mensaje m de p_j, este actualiza a T_i de la siguiente forma:

$\forall w \in m.\psi$ **do**

case

$w.lc\_ckpt > lc\_ckpt_i[w.id]$ $\rightarrow \forall \ell \neq i$ **do** $T_i[\ell, w.id] := 0$; **enddo**
if (**max**$(m.\psi) > w.lc\_ckpt) \vee (lc_i > w.lc\_ckpt)$ **then**
$T_i[j, w.id] := 1$;
endif

$w.lc\_ckpt = lc\_ckpt_i[w.id]$ $\rightarrow$ **if** (**max**$(m.\psi) > w.lc\_ckpt) \vee (lc_i > w.lc\_ckpt)$ **then**
$T_i[j, w.id] := 1$;
endif

$w.lc\_ckpt < lc\_ckpt_i[w.id]$ $\rightarrow$ **skip**

endcase

enddo

En este caso, la variable w obtiene las tuplas contenidas en el mensaje m, por lo que, esta variable maneja todos los valores de una tupla (estos fueron descritos anteriormente).

El siguiente teorema establece la equivalencia entre condiciones:

Teorema 3. La condición $\mathscr{D}'$ es equivalente a la condición $\mathscr{D}$.

La demostración de este teorema se desarrolla en la apéndice B. Este resultado junto con los resultados de la simulación que presentamos más adelante en la sección 4.5, muestran que para todos los casos la condición $\mathscr{D}'$ del algoritmo S-FI genera el mismo número de *checkpoints* forzados que la condición $\mathscr{C}2''$ del algoritmo FI.

4.3. Especificación del algoritmo S-FI

El algoritmo S-FI es un algoritmo de *checkpointing* de comunicación inducida. A diferencia del algoritmo FI, S-FI utiliza la condición $\mathscr{D}'$, desarrollada en la sección anterior, para eliminar patrones que puedan formar *z-cycles* o que no cumple el Teorema 1. A continuación realizamos la descripción del algoritmo S-FI.

4.3.1. Descripción del algoritmo S-FI

El algoritmo S-FI está compuesto de tres partes: ω_0, ω_1 y ω_2. En los cuadros 4.1 y 4.2 mostramos el pseudocódigo de estas tres partes. En el cuatro 4.3 mostramos el pseudocódigo del procedimiento *taken_checkpoint* y la función **max** que se utilizan en el pseudocódigo del algoritmo S-FI. A continuación realizamos una descripción general de cada parte de nuestro algoritmo.

ω_0 Inicializa los valores del algoritmo y genera el primer *checkpoint* local de cada proceso. El reloj lógico lc_i y las estructuras de datos: $lc\_ckpt_i[]$, $idr\_ckpt_i[]$, $greater_i[]$ y $T_i[][]$, son inicializadas de acuerdo a lo descrito en las Secciones 4.2 y 4.2.1 (ver líneas 2-6, Cuadro 4.1). El procedimiento *taken_checkpoint*() (línea 7, Cuadro 4.1), definido en el Cuadro 4.3, genera el primer *checkpoint* de un proceso (ver Secciones 4.2 y 4.2.1).

ω_1 Registra el envío de un mensaje y determina la información (adicional) que se anexará a este. Cuando un proceso p_i envía un mensaje m a un proceso p_j, este actualiza su arreglo booleano $sent\_to_i[j]$ a $true$, construye el conjunto ψ de tuplas y determina si el conjunto ψ (construido) o las estructuras ($lc\_ckpt_i[]$, $idr\_ckpt_i[]$, $greater_i[]$) serán anexadas a m (ver líneas 8-20, Cuadro 4.1). Esta decisión de anexar ψ o las estructuras al mensaje lo determina el menor costo de *overhead* entre ambas entidades (ver líneas 15-19, Cuadro 4.1), mientras que la construcción de ψ (ver líneas 11-13, Cuadro 4.1) se realiza a través de la evaluación de la condición K3 (Definición 13, Sección 4.2.1).

Cuadro 4.1: Algoritmo S-FI (ω_0 y ω_1).

```
   (ω0) Inicialización del proceso p_i.
 1 k, l : 1 ... n,   donde n es el número de procesos.
 2 ∀k do lc_ckpt_i[k] := 0; enddo
 3 ∀k, l do  T_i[k, l] := true; enddo
 4 idr_ckpt_i[i] := true;
 5 greater_i[i] := false;
 6 lc_i := 0;
 7 taken_checkpoint();
   (ω1) Cuando p_i envía un mensaje m a p_j.
 8 sent_to_i[j] := true;
 9 ψ_i ← ∅;
10 ∀k do
11     if [(¬T_i[j, k] ∨ ¬idr_ckpt_i[k]) ∧ (lc_ckpt_i[k] > 0)] then
12         ψ_i ← ψ_i ∪ (k, lc_ckpt_i[k], idr_ckpt_i[k], greater_i[k]);
13     endif
14 enddo
15 s := 32; //s es el #-bits para representar a reloj lógico (lc_ckpt_i).
   //size(ψ_i) regresa la cardinalidad de ψ_i.
16 if size(ψ_i) > (n)(s + 2)/(2s + 2) then
17   ψ_i ← ∅;
18   ∀k do ψ_i ← ψ_i ∪ (−, lc_ckpt_i[k], idr_ckpt_i[k], greater_i[k]); enddo
19 endif
20 send(m:=(ψ_i, Data)) to p_j;
```

ω_2 Actualiza la información de un proceso p_i cuando recibe un mensaje y determina si este debe tomar un *checkpoint* forzado. En **ω_2** evaluamos la condición $\mathscr{D}'$ descrita en la Sección 4.2.1 (líneas 22-25, Cuadro 4.2). Si $\mathscr{D}'$ se satisface, entonces el proceso toma un *checkpoint* forzado. Finalmente, con la información IDR recibida en el mensaje, p_i actualiza las estructuras $lc\_ckpt_i[]$, $idr\_ckpt_i[]$, $greater_i[]$ y $T_i[][]$, además de su reloj lógico, tal como lo describimos en la Sección 4.2.1.

4.4. Análisis de *overhead* del algoritmo S-FI

El *overhead* de mensajes, en el algoritmo S-FI, lo determina la cantidad de tuplas en ψ o las estructuras de datos ($lc\_ckpt_i[]$, $idr\_ckpt_i[]$ y $greater_i[]$; ver Cuadro 4.1) anexadas en cada mensaje.

Sea $t = |\psi|$ el número de tuplas (que se anexan en un mensaje), y s el número de bits para representar un entero, entonces la cantidad de bits que envía un proceso por tupla es:

Cuadro 4.2: Algoritmo S-FI (ω_2).

(ω_2 *When* p_i *receives the message* $m := (\psi, Data)$ *from* p_j.

21 $max\_lc\_ckpt := \mathbf{max}(\psi)$;
22 **if** $\big[\big(\exists k : sent\_to_i[k] \ \wedge \ (\exists y \in \psi, \ y.id = k : y.greater \vee$
23 $\nexists y \in \psi, \ y.id = k)\big) \ \wedge \ max\_lc\_ckpt > lc_i\big] \vee$
24 $\big[\exists z \in \psi, z.id = i : lc\_ckpt_i[i] = z.lc\_ckpt \ \wedge \ \neg z.idr\_ckpt\big]$
25 **then** take_checkpoint();
26 **endif**
27 $\forall w \in \psi$ **do**
28 **case**
29 $w.lc\_ckpt > lc\_ckpt_i[w.id] \rightarrow$
30 $lc\_ckpt_i[w.id] := w.lc\_ckpt$;
31 $idr\_ckpt_i[w.id] := w.idr\_ckpt$;
32 $\forall \ l \neq i$ **do** $T_i[l, w.id] := false$; **enddo**
33 **if** $(max\_lc\_ckpt \neq w.lc\_ckpt) \ \vee \ (lc_i > w.lc\_ckpt)$ **then**
34 $T_i[j, w.id] := true$;
35 **endif**
36 $w.cl\_ckpt = cl\_ckpt_i[w.id] \rightarrow$
37 $idr\_ckpt_i[w.id] := (idr\_ckpt_i[w.id] \wedge w.idr\_ckpt)$;
38 **if** $(max\_lc\_ckpt \neq w.lc\_ckpt) \ \vee \ (lc_i > w.lc\_ckpt)$ **then**
39 $T_i[j, w.id] := true$;
40 **endif**
41 $w.cl\_ckpt < cl\_ckpt_i[w.id] \rightarrow$ **skip**
42 **endcase**
43 **enddo**
44 **case**
45 $max\_lc\_ckpt > lc_i \rightarrow$
46 $lc_i := max\_lc\_ckpt$;
47 $\forall k \neq i$ **do** $greater_i[k] := true$; **enddo**
48 $\forall \ell \in \psi, \ell.id \neq i$ **do** $greater_i[\ell.id] := \ell.greater$; **enddo**
49 $max\_lc\_ckpt = cl_i \rightarrow$
50 $\forall \ell \in \psi$ **do** $greater_i[\ell.id] := greater_i[\ell.id] \wedge \ell.greater$; **enddo**
51 $max\_lc\_ckpt < lc_i \rightarrow$ **skip**
52 **endcase**
53 **delivery**(m);

Cuadro 4.3: Procedimientos y funciones usados en el algoritmo S-FI.

```
   // Procedimiento taken_checkpoint()
   // Genera o toma un checkpoint local o forzado.
54 procedure taken_checkpoint()
55   ∀k do sent_to_i[k] := false; enddo
56   ∀ k ≠ i do
57     idr_ckpt_i[k] := false;
58     greater_i[k] := true;
59     T_i[k, i] := false;
60   enddo
61   lc_i := lc_i + 1;
62   lc_ckpt_i[i] := lc_i;
63 endprocedure
```

```
   // Función max(α) obtiene el máximo reloj lógico en α.
64 function max(α)
65 max := 0;
66 ∀x ∈ α do
67   if x.lc_ckpt > max then max := x.lc_ckpt; endif
68 enddo
69 endfunction
```

$t(2s+1)$, debido a que cada tupla está formada por un identificador de proceso (entero), un reloj lógico (entero) y dos bits (ver Sección 4.2.1).

Haciendo uso de un análisis de complejidad por casos, tenemos lo siguiente:

- *En el mejor de los casos*, $t = 1$. Lo que significa que un proceso enviaría $2s+1$ bits.
- *En el peor de los casos*, $t = n$. Aquí, n representa al número de procesos en el sistema, por lo que, un proceso enviaría $(n)(2s+2) = 2sn+2n$ bits; sin embargo, recuerde que no siempre se envía ψ, sino, el costo menor entre ψ y las estructuras (mencionadas anteriormente, ver líneas 16-19, Cuadro 4.1). Si enviamos las estructuras tenemos un costo de $sn+2n$ bits, por lo que, en el peor de los casos es mejor enviar la información completa de las estructuras, que las tuplas en ψ ($sn+2n$ bits es menor que $2sn+2n$ bits). De esta forma, para el peor de los casos tenemos la siguiente ecuación:

$$(t)(2s+2) < sn + 2n \tag{4.1}$$

Por lo tanto, necesitamos detectar cuando:

$$|\psi| = t < \frac{sn + 2n}{2s + 2} \tag{4.2}$$

De esta forma, cada vez que la ecuación 4.2 se satisface enviamos las tuplas de ψ, de lo contrario, enviamos toda la información de las estructuras (ver líneas 16-19, Cuadro 4.1). De acuerdo a lo anterior, el costo para el peor de los casos es cuando enviasmos todas las tuplas, es decir, $sn + 2n$ bits.

- *En el caso promedio.* Tenemos que promediar los n posibles tamaños de t; es decir, el *overhead* cuando $t = 1$, $t = 2$, ..., $t = n$. Note que el valor de t no es una constante fija, su valor, lo determina la ecuación 4.1. En el cuadro 4.4, mostramos algunos valores de t, para $n = 2, \ldots, 7$. Note que $n \geq 2$, porque si $n = 1$ sólo tendríamos un único proceso, y por lo tanto, este no podría intercambiar mensajes con otro proceso.

  En el cuadro 4.4 mostramos algunas aproximaciones de t para algunos valores de n. El valor de t es aproximadamente $n/2$; aunque para alguna n "grande" $t > n/2$. Por ejemplo, si $s = 32$ bits y $n = 1024$, de la ecuación 4.1 tenemos:

$$t < \frac{sn + 2n}{2s + 2} < \frac{(32)(1024) + 2(1024)}{2(32) + 2} < \frac{34816}{66} \approx 527, \quad \text{mientras que: } \frac{n}{2} = 512$$

  Para este caso específico, t toma valores desde 1 hasta 526, $1 \leq t \leq 526$. Mientras que la aproximación de $t \approx n/2 = 512$, obtiene un valor por debajo de 526. La desigualdad $t < 527$, significa que podemos enviar hasta 526 tuplas antes de considerar enviar toda la información de las estructuras. Note que la diferencia entre 512 y 527 es 15. Además, $1024/66 \approx 15$. De manera informal, podemos inferir que el valor máximo de t lo podemos aproximar por $n/2 + n/(2s + 2) - 1$.

  Una manera más formal de obtener el valor de t es por medio de la ecuación 4.1, $t < \frac{sn + 2n}{2s + 2}$. Por lo que $t = \frac{sn + 2n}{2s + 2} - 1$ es un valor que satisface esta ecuación. Y por lo tanto, t es igual a:

$$\begin{aligned} t &= \frac{sn + 2n}{2s + 2} - 1 \\ &= \frac{sn + n + n}{2s + 2} - 1 \\ &= \frac{sn + n}{2s + 2} + \frac{n}{2s + 2} - 1 \\ &= \frac{n(s + 1)}{2(s + 1)} + \frac{n}{2s + 2} - 1 \\ &= \frac{n}{2} + \frac{n}{2s + 2} - 1 \end{aligned}$$

Cuadro 4.4: Valores de t respecto al número de procesos n.

n	t	costo en bits para:		Valor de t respecto a n para el peor de los casos.
		ψ $t(2s+2)$	Estructuras $n(s+2)$	
$n=2$	$t=1$	$2s+2$	$2s+4$	
	$t=2$	$4s+4$	$2s+4$	$t=n-1$
$n=3$	$t=1$	$2s+2$	$3s+6$	
	$t=2$	$4s+4$	$3s+6$	$t=n-2$
	$t=3$	$6s+6$	$3s+6$	
$n=4$	$t=1$	$2s+2$	$4s+8$	
	$t=2$	$4s+4$	$4s+8$	
	$t=3$	$6s+6$	$4s+8$	$t=n-2$
	$t=4$	$8s+8$	$4s+8$	
$n=5$	$t=1$	$2s+2$	$5s+10$	
	$t=2$	$4s+4$	$5s+10$	
	$t=3$	$6s+6$	$5s+10$	$t=n-2$
	$t=4$	$8s+8$	$5s+10$	
	$t=5$	$10s+10$	$5s+10$	
$n=6$	$t=1$	$2s+2$	$6s+12$	
	$t=2$	$4s+4$	$6s+12$	
	$t=3$	$6s+6$	$6s+12$	
	$t=4$	$8s+8$	$6s+12$	$t=n-3$
	$t=5$	$10s+10$	$6s+12$	
	$t=6$	$12s+12$	$6s+12$	
$n=7$	$t=1$	$2s+2$	$7s+14$	
	$t=2$	$4s+4$	$7s+14$	
	$t=3$	$6s+6$	$7s+14$	
	$t=4$	$8s+8$	$7s+14$	$t=n-3$
	$t=5$	$10s+10$	$7s+14$	
	$t=6$	$12s+12$	$7s+14$	
	$t=7$	$14s+14$	$7s+14$	

*El menor costos está marcado en fondo gris.

De esta forma, para el análisis del *overhead* de mensajes en el caso promedio, tenemos:

$$\frac{1}{n}\left[\sum_{i=1}^{t} i(2s+2) + \sum_{j=t+1}^{n} (sn+2n)\right], \quad t = \frac{n}{2} + \frac{n}{2s+2} - 1 \tag{4.3}$$

Y por lo tanto:

$$\begin{aligned}
&\frac{1}{n}\Big[\sum_{i=1}^{t} i(2s+2)+\sum_{j=t+1}^{n}(sn+2n)\Big] = \frac{1}{n}\Big[(2s+2)\sum_{i=1}^{t} i+(sn+2n)\sum_{j=t+1}^{n} 1\Big]\\
&= \frac{1}{n}\Big[(2s+2)\frac{t(t+1)}{2}+(sn+2n)(n-(t+1))\Big]\\
&= \frac{1}{n}\Big[(2)(s+1)\frac{t(t+1)}{2}+(sn+2n)(n-t-1)\Big]\\
&= \frac{1}{n}\Big[(s+1)\Big[\frac{n}{2}+\frac{n}{2s+2}-1\Big]\Big[\frac{n}{2}+\frac{n}{2s+2}-1+1\Big]\Big]\\
&\quad +\frac{1}{n}\Big[(sn+2n)(n-\frac{n}{2}-\frac{n}{2s+2}+1-1)\Big]\\
&= \frac{1}{n}\Big[(s+1)\Big[\frac{n}{2}+\frac{n}{2s+2}-1\Big](n)\Big[\frac{1}{2}+\frac{1}{2s+2}\Big]\Big]+\frac{1}{n}\Big[(n)(s+2)(\frac{n}{2}-\frac{n}{2s+2})\Big]\\
&= (s+1)\Big[\frac{n}{2}+\frac{n}{2s+2}-1\Big]\Big[\frac{1}{2}+\frac{1}{2s+2}\Big]+(s+2)(\frac{n}{2}-\frac{n}{2s+2})\\
&= (s+1)\Big[\frac{n}{4}+\frac{n}{2(2s+2)}-\frac{1}{2}+\frac{n}{2(2s+2)}+\frac{n}{(2s+2)(2s+2)}-\frac{1}{2s+2}\Big]\\
&\quad +\Big[\frac{sn}{2}-\frac{sn}{2s+2}+\frac{2n}{2}-\frac{2n}{2s+2}\Big]\\
&= (s+1)\Big[\frac{n}{4}+\frac{n}{(2s+2)}-\frac{1}{2}+\frac{n}{(2s+2)^2}-\frac{1}{2s+2}\Big]+\Big[\frac{sn}{2}-\frac{sn}{2s+2}+n-\frac{n}{s+1}\Big]\\
&= \frac{sn}{4}+\frac{sn}{2s+2}-\frac{s}{2}+\frac{sn}{(2s+2)^2}-\frac{s}{2s+2}+\frac{n}{4}+\frac{n}{2s+2}-\frac{1}{2}+\frac{n}{(2s+2)^2}-\frac{1}{2s+2}\\
&\quad +\frac{sn}{2}-\frac{sn}{2s+2}+n-\frac{n}{s+1}\\
&= \Big[\frac{sn}{4}+\frac{sn}{2s+2}+\frac{sn}{(2s+2)^2}+\frac{sn}{2}-\frac{sn}{2s+2}\Big]+\Big[-\frac{s}{2}-\frac{s}{2s+2}\Big]\\
&\quad +\Big[\frac{n}{4}+\frac{n}{2s+2}+\frac{n}{(2s+2)^2}+n-\frac{n}{s+1}\Big]+\Big[-\frac{1}{2}-\frac{1}{2s+2}\Big]\\
&= \Big[\frac{sn}{4}+\frac{sn}{(2s+2)^2}+\frac{sn}{2}\Big]+\Big[\frac{5n}{4}-\frac{n}{2s+2}+\frac{n}{(2s+2)^2}\Big]-\frac{s}{2}-\frac{s}{2s+2}-\frac{1}{2}-\frac{1}{2s+2}\\
&= \Big[\frac{3}{4}sn+\frac{sn}{(2s+2)^2}\Big]+\Big[\frac{5n}{4}-\frac{n}{2s+2}+\frac{n}{(2s+2)^2}\Big]-\frac{s}{2}-\frac{s}{2s+2}-\frac{1}{2}-\frac{1}{2s+2}\\
&= \Big[\frac{3}{4}sn+\frac{sn}{(2s+2)^2}\Big]+\Big[\frac{5n}{4}+\frac{-n(2s+2)+n}{(2s+2)^2}\Big]+\frac{-s(s+1)-s-(s+1)-1}{2s+2)}\\
&= \Big[\frac{3}{4}sn+\frac{sn}{(2s+2)^2}\Big]+\Big[\frac{5n}{4}+\frac{-2sn-2n+n}{(2s+2)^2}\Big]+\frac{-s^2-s-s-s-1-1}{(2s+2)}\\
&= \frac{3}{4}sn+\frac{sn}{(2s+2)^2}+\frac{5n}{4}-\frac{2sn+n}{(2s+2)^2}-\frac{s^2+3s+2}{(2s+2)}
\end{aligned}$$

Entonces, para nuestro caso promedio, el *overhead* está determinado por la ecuación:

$$\frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5n}{4} - \frac{2sn+n}{(2s+2)^2} - \frac{s^2+3s+2}{(2s+2)} \tag{4.4}$$

Ahora, vamos a demostrar que este *overhead* no es mayor a $sn + n$, por lo que demostraremos lo siguiente:

$$\frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5n}{4} - \frac{2sn+n}{(2s+2)^2} - \frac{s^2+3s+2}{(2s+2)} < sn + n \tag{4.5}$$

Para esto demostraremos las dos siguientes ecuaciones:

$$\frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5n}{4} - \frac{2sn+n}{(2s+2)^2} - \frac{s^2+3s+2}{(2s+2)} < \frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5}{4}n \tag{4.6}$$

$$\frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5}{4}n < sn + n \tag{4.7}$$

La demostración de la ecuación 4.6 es trivial, por lo que, sólo probaremos la desigualdad de la ecuación 4.7. La demostración es la siguiente:

$$\begin{aligned}
\frac{3}{4}sn + \frac{sn}{(2s+2)^2} + \frac{5}{4}n < sn + n \Leftrightarrow\ & \frac{sn}{(2s+2)^2} + \frac{1}{4}n < \frac{1}{4}sn \\
\Leftrightarrow\ & \left[\frac{s}{(2s+2)^2} + \frac{1}{4}\right](n) < \frac{1}{4}sn \\
\Leftrightarrow\ & \frac{s}{(2s+2)^2} + \frac{1}{4} < \frac{1}{4}s \\
\Leftrightarrow\ & \frac{s}{4(s+1)^2} + \frac{1}{4} < \frac{1}{4}s \\
\Leftrightarrow\ & \frac{1}{4}\left[\frac{s}{(s+1)^2} + 1\right] < \frac{1}{4}s \\
\Leftrightarrow\ & \frac{s}{(s+1)^2} + 1 < s \\
\Leftrightarrow\ & s < (s-1)(s+1)^2 \\
\Leftrightarrow\ & s < (s-1)(s^2+2s+1) \\
\Leftrightarrow\ & s < s^3 + 2s^2 + s - s^2 - 2s - 1 \\
\Leftrightarrow\ & s < s^3 + s^2 - s - 1 \\
\Leftrightarrow\ & 1 < s^3 + s^2 - 2s \\
\Leftrightarrow\ & 1 < s(s^2 + s - 2) \\
\Leftrightarrow\ & 1 < s \wedge 1 < s^2 + s - 2 \qquad \blacksquare
\end{aligned}$$

Debido a que s es el número de bits en un reloj lógico, s toma valores enteros, por lo que, la ecuación 4.7 se cumplen siempre que $s > 1$. De esta forma, el *overhead* de mensajes en S-FI para el caso promedio es menor a $sn + n$ (ecuaciones 4.6 y 4.7).

En el Cuadro 4.5 mostramos el *overhaed* de mensajes del algoritmo S-FI (los resultados del análisis anterior) y el de los algoritmos FI [22] y FINE [33] reportados. Utilizamos el algoritmo FINE en nuestro análisis debido a que es un algoritmo reciente; además, este hace uso de los principios introducidos por el algoritmo FI. El *overhead* de nuestro algoritmo S-FI está por debajo de FI para todos los casos. En comparación con FINE, S-FI sólo tiene un *overhead* mayor para el peor de los casos. En el mejor caso y caso promedio, S-FI está por debajo de FINE.

No obstante, a los resultados del análisis, consideramos que el peor de los casos en S-FI es poco probable. El caso promedio por otra parte, es únicamente una aproximación bajo la hipótesis de que todos los casos son probables, en general, esto no siempre es cierto, y consideramos que S-FI tiene todavía un mejor desempeño. En la siguiente sección simularemos estos tres algoritmos, para mostrar que el *overhead* de mensajes en S-FI es siempre menor que en FI y FINE.

Para finalizar esta sección, hacemos una anotación respecto al algoritmo FINE, este algoritmo utiliza un mecanismo de etiquetado de reloj lógico diferente al de FI y S-FI, el cual permite a FINE reducir su *overhead* de mensajes. Sin embargo, esto limita también al conteo de *checkpoints* en un intervalo, una desventaja de FINE en comparación a FI y S-FI.

Cuadro 4.5: *Overhead* por mensaje (bits) para S-FI, FI y FINE.

Algorithm	Best-Case	Average-Case	Worst-Case
S-FI	$2s+2$	$\frac{3sn}{4} + \frac{sn}{(2s+2)^2} + \frac{5n}{4} - \frac{2sn+n}{(2s+2)^2} - \frac{s^2+3s+2}{(2s+2)}$	$(n)(s+2)$
FI [22]	$(n)(s+2)+s$		
FINE [33]	$(n)(s+1)$		

s-número de bits para representar un entero.
n-número de procesos .

4.5. Simulación del algoritmo S-FI

Para analizar el desempeño de nuestro algoritmo S-FI, lo comparamos con dos algoritmos de *checkpointing* de comunicación-inducida (CIC): FI [22] y FINE [33]. Elegimos

FI, porque en la literatura es conocido como uno de los mejores algoritmos CIC [48], y porque S-FI utiliza su teoría de base. FINE, porque es un algoritmo CIC reciente; además, utiliza la teoría de FI. Los algoritmos FI, FINE y S-FI fueron simulados y analizados con el simulador ChkSim [51]. ChkSim implementa un modelo de simulación determinista, lo que permite reproducir varias veces una simulación con la misma información de un escenario, esto con el objetivo de poder comparar el desempeño de dos o más algoritmos, en un entorno controlado y bajo las mismas condiciones. Para nuestro análisis, utilizamos dos medidas: *overhead* de mensajes y número de *checkpoints* forzados.

Analizamos el desempeño para cuatro escenarios: 1000, 2500, 5000 y 50000 mensajes. En cada escenario utilizamos una distribución uniforme de mensajes y variamos el número de procesos de 10, 20,. . . , 120. Además, por cada escenario ejecutamos 100 iteraciones con diferentes patrones de comunicación y *checkpoints*.

Los resultados analizados son las siguientes:

a) *Número de checkpoints forzados.* Los resultados de nuestras simulaciones, para los cuatro escenarios, indican que tanto S-FI como FI generan el mismo número de *checkpoints* forzados, mientras que FINE genera una cantidad menor, ésta representa (en promedio) únicamente un 1.5 % menos que FI y S-FI, a pesar que FINE reporta un 3 % menos que FI.

b) *Overhead de mensajes.* En la Figura 4.2 mostramos el promedio de *overhead* de mensajes para las 100 iteraciones de cada escenario. Para S-FI, este es dinámico, debido a la dependencia sobre la densidad de mensajes en el sistema y no del número de procesos. El promedio de *overhead* para FI y FINE, presenta un crecimiento constante y lineal respecto al número de procesos. En este sentido, S-FI es más escalable que FI y FINE.

   El cuadro 4.6 muestra los datos de la Figura 4.2. En este cuadro, la columna de FI muestra el costo por mensaje (en bits) para $10, 20, 30, \ldots, 120$ procesos en los escenarios analizados (1000, 2500, 5000 y 50000 mensajes). Por ejemplo, el costo de enviar un mensaje con 20 procesos en los cuatro escenarios es de 712 bits, mientras que el costo de enviar un mensaje con 40 procesos es de 1392 bits. Por otra parte, las columnas de S-FI y FINE son porcentajes relativos a los costos de FI. Así, el costo de enviar un mensaje con 20 procesos en FINE es sólo el 92.7 % del costo de FI, aproximadamente 660 bits (para los cuatro escenarios), mientras que en S-FI, este mismo costo, depende del escenario: 88.67 % ($\approx$ 631 bits), 92.77 % ($\approx$ 661 bits),

94.14 % (≈ 670 bits) y 95.37 % (≈ 679 bits) para 1000, 2500, 5000 y 50000 mensajes respectivamente.

Es importante notar que el promedio de *overhead* de los mensajes en la gráfica de S-FI, tiende a disminuir conforme la densidad de mensajes permanece constante y el número de procesos crece. Esto significa que nuestra gráfica tiene un punto máximo en costo. En el Cuadro 4.6 marcamos los puntos máximos con un fondo negro. Por ejemplo, para 5000 mensajes, el costo máximo por mensaje es cuando tenemos 30 procesos; mientras que para el resto (10,30,40,...,120 procesos) es menor. De esta forma, S-FI se adapta mejor al crecimiento de procesos que FI y FINE.

4.6. Análisis de S-FI

S-FI es un algoritmo en línea, utiliza los principios introducidos en [22] para eliminar *z-cycles* y el mecanismo de comunicación presentado en [4] para disminuir el *overhead* de control en cada mensaje. Analizamos S-FI de manera formal y a través de una simulación.

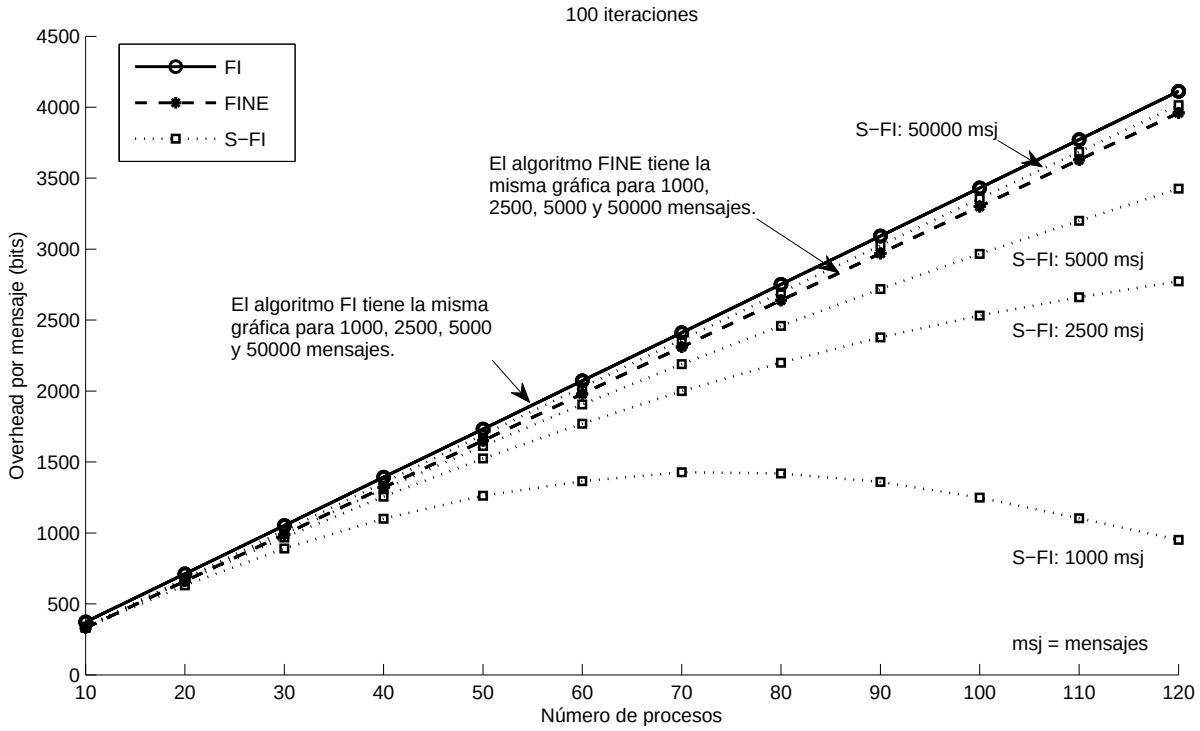


Figura 4.2: Resultados de la simulación de S-FI.

Cuadro 4.6: *Overhead* en FI, porcentajes de *overhead* para FINE y S-FI respecto a FI, y puntos en la gráfica de S-FI donde se invierte la pendiente.

#Procesos	*Overhead* por mensaje(bits)	Porcentajes de *overhead* para S-FI y FINE respecto a FI.				
	FI	**FINE**	**S-FI**			
	1000, 2500, 5000, 50000	1000, 2500, 5000, 50000	1000	2500	5000	50000
10	372	88.71 %	**89.20 %**	90.51 %	90.95 %	91.34 %
20	712	92.70 %	88.67 %	**92.77 %**	94.14 %	95.37 %
30	1,052	94.11 %	84.56 %	92.00 %	**94.48 %**	96.71 %
40	1,392	94.83 %	79.07 %	92.25 %	93.97 %	97.33 %
50	1,732	95.27 %	72.85 %	88.03 %	93.09 %	97.65 %
60	2,072	95.56 %	65.84 %	85.41 %	91.93 %	97.80 %
70	2,412	95.77 %	59.20 %	82.88 %	90.78 %	97.88 %
80	2,752	95.93 %	51.59 %	79.93 %	89.38 %	**97.89 %**
90	3,092	96.05 %	43.95 %	76.90 %	87.93 %	97.86 %
100	3,432	96.15 %	36.39 %	73.78 %	86.42 %	97.80 %
110	3,772	96.24 %	29.26 %	70.55 %	84.85 %	97.72 %
120	4,112	96.30 %	23.13 %	67.43 %	83.32 %	97.63 %

4.6.1. Análisis formal de S-FI

El análisis formal se realizó en dos contextos:

1. **Análisis de complejidad**. Utilizamos un análisis por casos para realizar esta parte. En el Cuadro 4.5 resumimos los resultados obtenidos. Los resultados muestran lo siguiente:

   a) *Peor de los casos.* Nuestro algoritmo S-FI tienen un *overhead* mayor, respecto al mejor algoritmo reportado (FINE). S-FI acarrea n bits más en cada mensaje (n representa al número de procesos en el sistema), sin embargo, S-FI no está limitado en los incrementos dentro de un intervalo de *checkpoint* como lo ésta FINE (ver sección 3.1.1). Por ejemplo, si ambos algoritmos usaran un reloj lógico de s bits y S-FI tuviera la misma limitante de FINE, el overhead de mensajes del algoritmo S-FI tendría mucho menos que n bits (los n bits más que tiene S-FI en un principio). Por ejemplo, si FINE estuviera limitado a realizar incrementos de $2^{10} = 1024$ (entre *checkpoints*) entonces no podría enviar un mensaje solicitando que el reloj lógico de un proceso se incremente

en más de 1024, restringiendo con ello la interacción entre procesos y la cantidad de estos. Para este caso, el algoritmo S-FI solo necesitaría relojes lógicos con 2^{10} bits en lugar de los 2^{16} bits (un entero) que se consideran para un reloj lógico; sin embargo, considerando que este es el peor de los casos, y que tiene una probabilidad baja de ocurrencia, sólo con que S-FI utilizará relojes lógicos con un bit menos, entonces el *overhead* por mensaje para S-FI y FINE sería el mismo, pero con la diferencia de que S-FI sólo estaría limitado a incrementos con 2^{s-1} (con s igual al número de bits de un reloj lógico).

b) Caso promedio. Nuestro algoritmo S-FI tiene el *overhead* más bajo que los algoritmos FI y FINE.

c) Mejor de los casos. Nuestro algoritmo, también presenta el menor *overhead*.

2. **Análisis de la condición de checkpoint forzado**. En esta parte del análisis de S-FI, enunciamos y demostramos los teoremas 2 y 3 (ver Apéndice A.1 y A.2). La demostración del teorema 2 establece que las condiciones de *checkpoint* forzado en los algoritmos FI y S-FI son equivalentes, y por lo tanto, ambos algoritmos generan el mismo número de *checkpoint* forzados. Por otra parte, la demostración del Teorema 3 establece que la información de control que enviamos en cada mensaje es redundante, que en muchos de los casos, la información causal de *checkpoints* con dependencias inmediatas puede ser descartada en algunos mensajes, y a pesar de esto, poder evaluar la condición de *checkpoint* forzado en S-FI. En otras palabras, el teorema 3 nos permite enviar menos *overhead* por mensaje y seguir evaluando la condición de *checkpoint* forzado en S-FI, al igual que lo realiza el algoritmo FI.

4.6.2. Análisis de la simulación

Con el objetivo de evaluar el desempeño de nuestro algoritmo S-FI, lo simulamos y comparamos junto con los algoritmos FI y FINE. Los resultados obtenidos de esta simulación son los siguientes:

1. *Overhead por mensaje.* Los resultados muestran que el *overhead* en FI y FINE presentan un crecimiento lineal constante respecto al número de procesos. El *overhead* para S-FI es dinámico y presenta un crecimiento por debajo de lo lineal. Esto se debe a que el *overhead* en S-FI no es directamente proporcional al número de procesos, este depende de la densidad de mensajes y la relación IDR entre *checkpoints*. Por lo

tanto, el algoritmo S-FI soporta un número mayor de procesos que FI y FINE. Los resultados del análisis formal presentado también apoyan los resultados de nuestra simulación.

2. *Número de checkpoints forzados.* El número de *checkpoints* forzados en S-FI y FI es el mismo, mientras que FINE presenta un 3 % menos en promedio que FI y S-FI.

Capítulo 5

Algoritmo DCFI para el retraso de *checkpoints*

En este capítulo introducimos un enfoque de retraso en *checkpoints* para algoritmos de *checkpointing* de comunicación inducida (CIC). El retraso de *checkpoints* está orientado a disminuir el problema del número de *checkpoints* forzados [3]. Nosotros logramos disminuir el número total de *checkpoints* (locales y forzados), debido a que no todos los *checkpoints* forzados generados en algunos algoritmos CIC son necesarios. Nuestro enfoque tiene su origen en la detección de una clase particular de *z-cycles* causales bajo ciertas condiciones. Llamamos *z-cycles rastreables* a esta clase de *z-cycles* causales y *condiciones seguras de retraso de checkpoint* (CSRC) a las condiciones que permiten aplicar nuestro enfoque. Por lo que, cuando un proceso p_i identifica las condiciones CSRC puede retrasar su último *checkpoint* no forzado y prevenir, con este retraso, que otro proceso genere un *checkpoint* forzado. De esta forma, conseguimos reducir la cantidad de *checkpoints* generados en el sistema.

En este capítulo presentamos el algoritmo DCFI (*Delay Checkpoint with Fully-Informed*), este algoritmo es una implementación del enfoque de retraso para *checkpoints* no-forzados y fue presentado en "22th IEEE WETICE conference 2013" [10].

El algoritmo DCFI utiliza el modelo de ejecución asíncrono presentado en la Sección 2.1.1. Para analizar la desempeño del algoritmo DCFI, simulamos el algoritmo con la herramienta ChkSim [51], junto con los algoritmos FI [22] y FINE [33]. Los resultados de la simulación de DCFI muestran la viabilidad de nuestro enfoque de retraso de *checkpoint*; y un número menor de *checkpoints* forzados que los algoritmos FI y FINE.

Debido a que nuestro enfoque hace uso de la información causal (detección de *z-cycles* rastreables); y que el algoritmo FI (*Fully-Informed*) es uno de los mejores algoritmos CIC que hace uso de está [48]. Iniciamos este capítulo con una descripción general del manejo de la información causal en FI; para después realizar un análisis del traslado de la información causal en el sistema e introducir nuestro enfoque de retraso en *checkpoints* no-forzados.

5.1. Análisis de la información causal en el algoritmo FI

La información causal que administra el algoritmo FI, está completamente relacionada con la administración del reloj lógico de cada proceso. En FI cada proceso tiene un reloj lógico lc_i administrado en la siguiente forma (reloj lógico de Lamport, [31]):

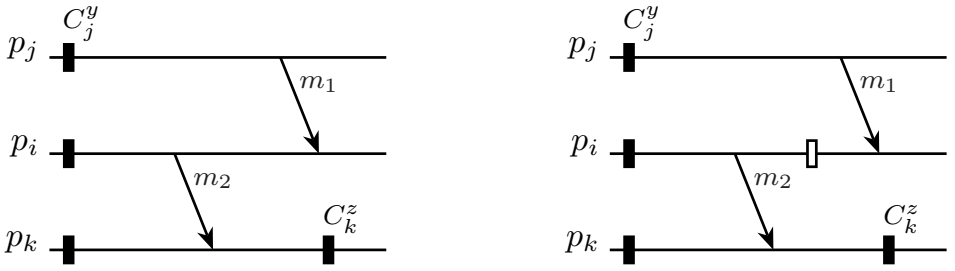
- Cada vez que p_i toma un *checkpoint*, este incrementa en uno su reloj lógico y lo asocial al nuevo *checkpoint*.
- Cada mensaje enviado por p_i es etiquetado con el valor del reloj lógico del emisor, $m.t$.
- Cuando un proceso p_i recibe un mensaje m, p_i actualiza su reloj lógico con el valor máximo de los relojes conocidos, $lc_i = max(lc_i, m.t)$.

FI utiliza el Teorema 1 (ver, Sección 2.3) para detectar *z-paths* que eventualmente podrían formar un *z-cycle*. El Teorema 1 establece que: *si hay un* z-path *de* C_j^y *a* C_k^z*, entonces los relojes lógicos asociados a cada checkpoint (*$C_j^y.t$ *y* $C_k^z.t$*), satisfacen la desigualdad* $C_j^y.t < C_k^z.t$.

En el algoritmo FI, cada vez que un proceso toma un *checkpoint*, incrementa su reloj lógico, esta acción asegura que todo *z-path* causal satisface el Teorema 1; sin embargo, en los *z-paths* no causales, tenemos problemas porque estos no siempre satisfacen el Teorema 1.

En la Figura 5.1(a) mostramos un *z-path* no causal formado de C_j^y a C_k^z. En este caso, el proceso p_i detecta la formación del *z-path* al recibir el mensaje m_1; en este punto, p_i puede tener información relativa de los procesos p_k y p_j, y puede determinar si el Teorema 1 es consistente ($C_j^y.t < C_k^z.t$), de lo contrario, debe eliminar el *z-path* por medio de un *checkpoint* forzado, Figura 5.1(b).

La información causal que conoce el proceso p_i es trasladada por medio de mensajes. Por ejemplo, el mensaje m_1 traslada el reloj lógico asociado a C_j^y; sin embargo, p_i no



(a) *z-path* no causal de C_j^y a C_k^z.

(b) Eliminación del *z-path* no causal de C_j^y a C_k^z utilizando el algoritmo FI.

Figura 5.1: *z-path* no causal y su eliminación utilizando el algoritmo FI.

siempre conoce el reloj lógico de C_k^z. Por lo que, el Teorema 1 puede no cumplirse, debido a que:

1. La desigualdad $C_j^y < C_k^z$ no se satisface.

2. Porque p_i desconoce el reloj lógico de C_k^z.

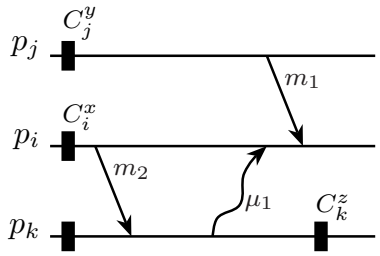
En ambos casos, p_i elimina el *z-path* para satisfacer el Teorema 1.

Los esquemas de la Figura 5.2 muestran cómo el proceso p_i, en FI, puede aproximar el reloj lógico asociado a C_k^z, si este conoce el reloj lógico del proceso p_k.

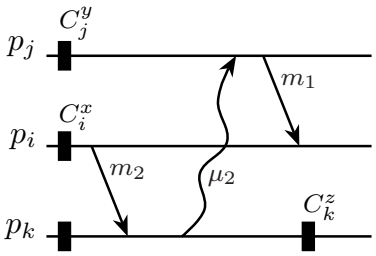
Los esquemas de la Figura *5.2(a)* y *5.2(b)* muestran que el valor lc_k (reloj lógico de p_k) traído hasta p_i es menor que el reloj lógico asociado a C_k^z, $lc_k < C_k^z.t$; por lo que, si $C_j^y.t \leq lc_k$, entonces satisface la desigualdad $C_j^y.t < C_k^z$ y el Teorema 1. Los esquemas de la Figura *5.2(c)* y *5.2(d)* muestran que $lc_k \geq C_k^z$; sin embargo, en estos casos, se forma un *z-cycle* que involucra al *checkpoint* C_k^z (existe una *z-path* de C_k^z a sí mismo). En la Figura *5.2(e)* mostramos una abstracción de este *z-cycle*. Si sustituimos la cadenas de mensajes μ por μ_1 obtenemos el esquema *5.2(c)*, pero si sustituimos μ por $[\mu_2, m_1]$ entonces obtenemos el esquema *5.2(d)*.

5.2. Análisis de *overhead* de la información causal entre procesos

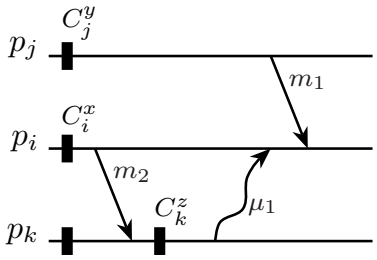
En la sección anterior mostramos cómo el algoritmo FI considera el envío de los relojes lógicos entre procesos; sin embargo, el análisis es incompleto porque no considera la llegada de la información a posteriori (después de la entrega del mensaje m_1, Figura 5.2).



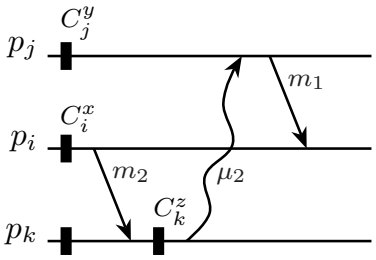
(a) p_i conoce lc_k por μ_1 y $lc_k < C_k^z$.



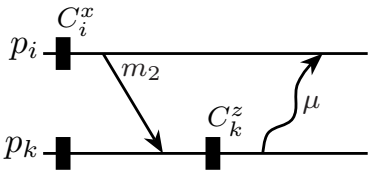
(b) p_i conoce lc_k por $[\mu_2, m_1]$ y $lc_k < C_k^z$.



(c) p_i conoce lc_k por μ_1 y $lc_k \geq C_k^z$.



(d) p_i conoce lc_k por $[\mu_2, m_1]$ y $lc_k \geq C_k^z$.



(e) Abstracción de los *z-cycles* de los esquemas *(c)* y *(d)* que envuelven a C_k^z.

Figura 5.2: Esquemas utilizados en el análisis del algoritmo FI en [22].

Al analizar el *z-path* de la Figura 5.3, podemos notar que existen ocho secciones entre los diversos eventos que están formando el *z-path*; dos en el proceso p_j (una sección antes del envío de m_1 y otra después del envío), tres en el proceso p_i y tres en el proceso p_k. En el Cuadro 5.1 describimos los límites de estas ocho secciones. Para nuestro análisis, estamos interesados en los mensajes que provienen del proceso p_k y eventualmente llegan al proceso p_i o p_j. De esta forma tenemos tres secciones de envío de mensajes y cinco secciones de recepción. Las tres secciones de envío en p_k las llamamos A, B y C; mientras que las secciones de recepción: 1, 2 y 3 para p_i y, 4 y 5 para p_j (ver Cuadro 5.1).

Al realizar las posibles combinaciones de las tres secciones de envío y cinco secciones de recepción, obtenemos 15 combinaciones que mostramos en la Figura 5.4. De estos 15 escenarios, los escenarios B1 y C1 no son posibles, debido a que la causalidad formada por el mensaje m_2 y la cadena de mensajes μ (en estos escenarios) es imposible de generar; los siete escenarios: A1, A2, A4, B2, B4, C2 y C4 son los utilizados o analizados por el

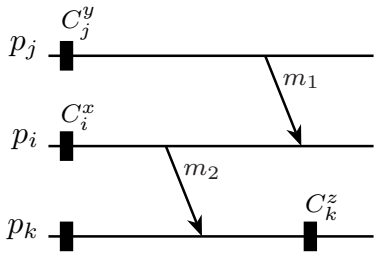


Figura 5.3: *z-path* no causal de C_j^y a C_k^z.

Cuadro 5.1: Secciones de envío o recepción en la estructura genérica de un *z-path* no causal.

Sección	Proceso	Límites de sección
A	p_k	antes de $delivery(p_k, m_2)$
B	p_k	entre $delivery(p_k, m_2)$ y C_k^z
C	p_k	después de C_k^z
1	p_i	entre C_i^x y $send(m_2)$
2	p_i	entre $send(m_2)$ y $delivery(p_i, m_1)$
3	p_i	después de $delivery(p_i, m_1)$
4	p_j	entre C_j^y y $send(m_1)$
5	p_j	después de $send(m_1)$

algoritmo FI, mientras que los seis escenarios: A3, A5, B3, B5, C3 y C5 no son utilizados por FI. De estos seis últimos escenarios, el escenario B3 es un escenario potencial, porque puede utilizarse por un algoritmo de CIC para corregir en el caso que pudiéramos reconsiderar, en un futuro cercano, la formación de un *checkpoint* forzado, ó bien, sustentar la generación de un *checkpoint* forzado antes de la entrega del mensaje m_1. En el Cuadro 5.2 resumimos algunas características de los 15 escenarios obtenidos. En particular, estamos interesados en caracterizar los escenarios C2 y C3 que forman un *z-cycle*, debido a que el principio del retraso de *checkpoints*, que desarrollamos más adelante, utiliza casos

Cuadro 5.2: Características de los escenarios de la Figura 5.4.

Característica	Escenarios														
	A_1	A_2	A_3	A_4	A_5	B_1	B_2	B_3	B_4	B_5	C_1	C_2	C_3	C_4	C_5
No valido						✓					✓				
Usado en FI	✓	✓		✓			✓		✓			✓		✓	
Posibilidad de uso en un CIC								✓				✓	✓		
Utilizado en nuestro enfoque												✓	✓		

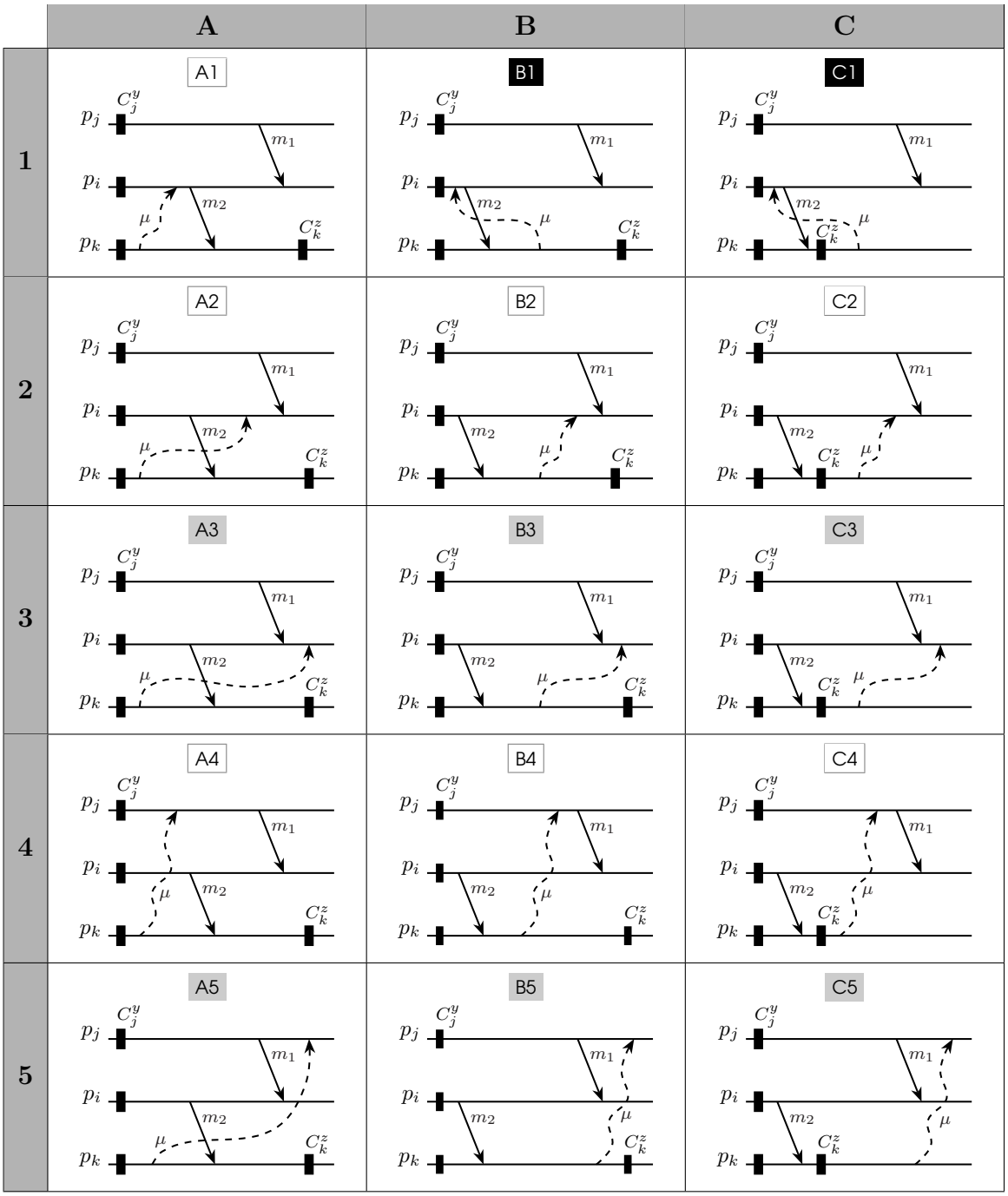


Figura 5.4: Escenarios posibles respecto a cómo el reloj lógico de p_k puede ser acarreado hasta el proceso p_i.

especiales de estos escenarios.

Antes de introducir nuestro enfoque de retraso de *checkpoints*, caracterizamos a un *z-cycle* causal. Esta caracterización la realizamos a partir de los escenarios C2 y C3 de la Figura 5.4.

5.2.1. Caracterización de *z-cycles* causales

Netzer y Xu [37] definieron la noción de un *z-cycle* como un *z-path* de un *checkpoint* a sí mismo (Definición 10, Sección 2.3). Un *checkpoint* involucrado en un *z-cycle* es un *checkpoint* no útil, de lo contrario, es un *checkpoint* útil. La utilidad de un *checkpoint*, radica en su uso para la formación de *Snapshot* Global Consistente (SGC).

Un *checkpoint* útil, siempre es parte de un SGC (siempre se utiliza en el proceso de recuperación de un sistema); por el contrario, un *checkpoint* no útil, nunca es parte de un SGC (nunca se utiliza en el proceso de recuperación de un sistema). En este sentido, los *checkpoints* no útiles sólo degradan el desempeño de un sistema, y no aportan información al proceso de recuperación cuando falla el sistema. Por lo tanto, su eliminación dentro de los algoritmos de comunicación inducida de *checkpointing* resulta crucial para el desempeño del estos algoritmos.

Para realizar nuestra caracterización de *z-cycles* causales, utilizamos la noción de *cadenas de mensajes causales* y *no causales* introducidos en [39]. Las definiciones formales de estas nociones son las siguientes:

Definición 14. *Una* **cadena de mensajes** *es una secuencia* $\mu = [m_1, m_2 \ldots, m_\ell]$ $(\ell \geq 2)$ *de mensajes, tal que* $\forall k \in \{1, \ldots, \ell - 1\}$*, tenemos:*

$$delivery(p_i, m_k) \in I_i^x \wedge send(m_{k+1}) \in I_i^y, \quad x \leq y$$

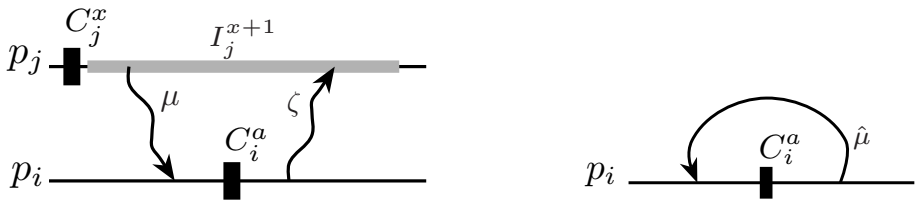
Definición 15. *Una cadena de mensajes* $\mu = [m_1, m_2, \ldots, m_\ell]$ *es causal si:*

$$delivery(p_i, m_k) \rightarrow send(m_{k+1}), \ \forall k = 1, \ldots, \ell - 1$$

en caso contrario, la cadena de mensajes se denomina no-causal.

En la Figura 5.5(a) mostramos la estructura base de un *z-cycle*. Esta estructura es formada por un *checkpoint*, C_i^a y dos cadenas de mensajes, μ y ζ (causales o no causales), que se relacionan por medio de un intervalo de *checkpoint* (I_i^{x+1}); además, si representamos a μ y ζ por una sola cadena de mensajes, $\hat{\mu}$, obtenemos la representación abstracta de un *z-cycle*, ver Figura 5.5(a) y 5.5(b).

Nuestra caracterización de *z-cycle*, por lo tanto, está conformada por un *checkpoint* C_i^a, dos cadenas de mensajes μ y ζ, causales o no causales, y un intervalo de *checkpoints* I_j^{x+1} que relaciona a μ y ζ.



(a) *z-cycle* genérico. (b) Abstracción de un *z-cycle*.

Figura 5.5: Dos diferentes perspectivas de un *z-cycle*.

Una caracterización más amplia y formal de *z-cycles* la realiza Quaglia et al. [39]. Para nuestro análisis, la caracterización anterior de un *z-cycle* es suficiente.

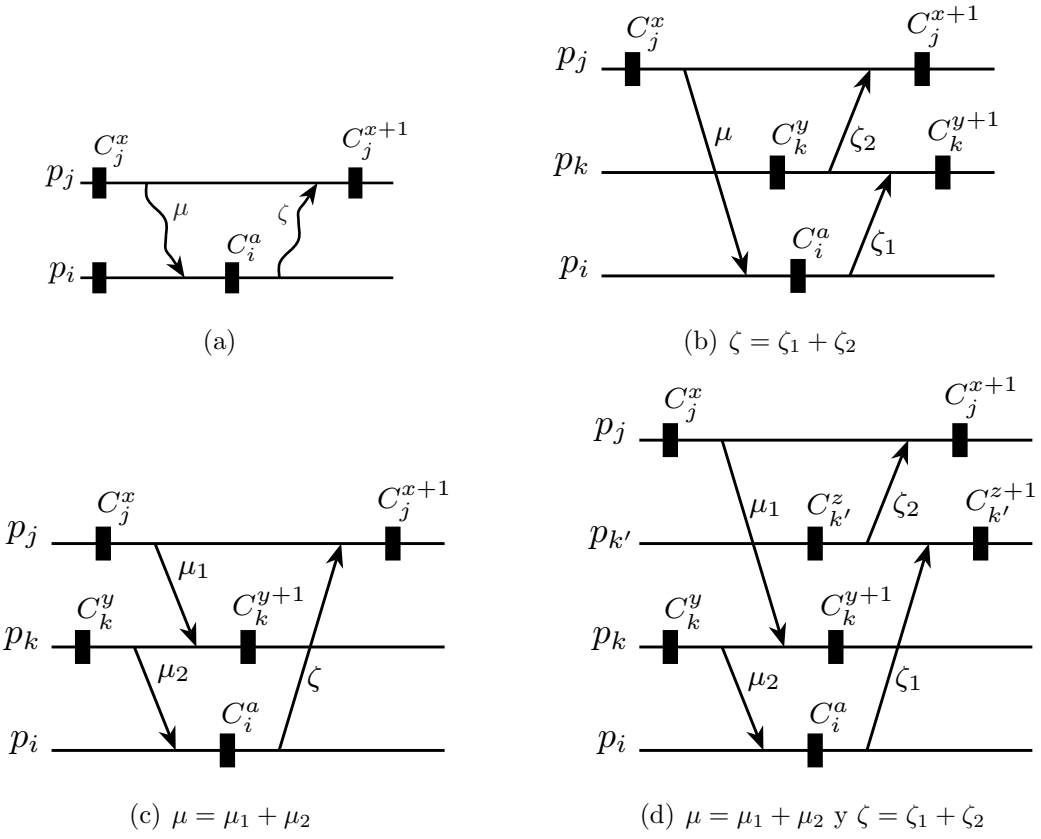
De acuerdo a nuestra caracterización de *z-cycle*, las cadenas de mensajes μ y ζ pueden ser causales y no causales, por lo que tenemos las siguientes cuatro combinaciones:

1. μ **causal -** ζ **causal**. En la Figura 5.6(a) mostramos este caso. Si la cadena de mensajes $\zeta = [z_1, z_2, \ldots z_s]$ $(s \geq 1)$, entonces, el último mensajes de ζ a p_j, informa todo el historial causal posible del *z-cycle*. Además, el proceso p_i, por medio del historial causal, puede conocer la formación del *z-cycle* si $\zeta = [z_1]$ (p_i envía un mensaje a p_j).

2. μ **causal -** ζ **no causal**. En la Figura 5.6(b) mostramos este caso. Aquí, la cadena ζ está formada por dos o más cadenas causales, las cuales se relacionan por medio de uno o más intervalos. Por ejemplo, en la Figura 5.6(b), la cadena ζ está formada por dos cadenas causales, ζ_1 y ζ_2; a su vez, estas están relacionadas por medio del intervalo I_k^{y+1}. En este caso, un algoritmo de *checkpointing* no podría conocer el intervalo donde se forma el *z-cycle*, porque este puede formarse en I_j^{x+1} o I_k^{y+1}, cuando el último de los mensajes de ζ_1 o ζ_2 es entregado (estos son concurrentes). Además, el algoritmo no contaría con historial causal para determinar la formación del *z-cycle* (en este caso, los propios intervalos cortan el historial causal). De esta manera informal, la Figura 5.6(b) muestra la formación de un *z-cycle* por medio de las cadenas causales μ, ζ_1 y ζ_2, y los intervalos I_k^{y+1} y I_j^{x+1}.

3. μ **no causal -** ζ **causal**. En la Figura 5.6(c) mostramos este caso. Este es muy similar al caso anterior. De hecho, si consideramos algunas equivalencias entre cadenas de mensajes, obtenemos el caso de la Figura 5.6(c) a partir de la Figura 5.6(b).

4. μ **no causal -** ζ **no causal**. En la Figura 5.6(d) mostramos este caso. En esta figura μ y ζ están formadas por dos subcadenas causales, $\mu = \mu_1 + \mu_2$ y $\zeta = \zeta_1 + \zeta_2$.



(a)

(b) $\zeta = \zeta_1 + \zeta_2$

(c) $\mu = \mu_1 + \mu_2$

(d) $\mu = \mu_1 + \mu_2$ y $\zeta = \zeta_1 + \zeta_2$

Figura 5.6: Diversos esquemas de un *z-cycle*.

Los intervalos que relacionan a estas cadenas son I_k^{y+1} que relaciona a μ_1 y μ_2, $I_{k'}^{z+1}$ que relaciona a ζ_1 y ζ_2, y finalmente, I_j^{x+1} que relaciona a μ y ζ. Al igual que los dos casos anteriores, la no causalidad de mensajes corta el historial causal del *z-cycle*, impidiendo detectar en cual de estos intervalos se forma el *z-cycle*. La falta de información causal junto con al posibilidad de concurrencia en las recepciones de los últimos mensajes de μ_1, ζ_1 y ζ_2 hace difícil o casi imposible la detección de esta clase de *z-cycle*, de manera que muchos algoritmos de CIC eliminarán esta clase de *z-cycles* con más de un *checkpoint* forzado.

Del análisis respecto a las cadenas de mensajes μ y ζ anterior, podemos concluir lo siguiente:

1. Si ambas cadenas de mensajes son causales entonces podemos rastrear la formación del *z-cycle* en línea (*on-line*).

2. Si ambas cadenas de mensajes son no causales o alguna de ellas es no causal, entonces no podremos rastrear en línea la formación del *z-cycle*. Además, debido a que tenemos

tantos mensajes concurrentes como posibles intervalos de *checkpoints*, será difícil determinar el último mensaje que forma al *z-cycle* de forma única.

z-cycle rastreable

Hasta este punto de nuestro análisis, hemos caracterizado informalmente a dos tipos de *z-cycles*: *rastreables* y *no rastreables*. Los *z-cycles* rastreables los detectamos por medio de su historial causal, mientras que los *z-cycles* no rastreables no los podemos detectar por este medio.

La definición formal del concepto de *z-cycle rastreable* es la siguiente:

Definición 16. *Un z-cycle rastreable es un z-cycle que puede ser rastreado "en linea" (on line), usando para esto, sólo el historial causal transmitido entre procesos. De lo contrario es llamado z-cycle no-rastreable.*

Un *z-cycle* rastreable, Figura 5.7, está formado por un *z-path* con las siguientes características:

1. Un *checkpoint* C_i^a,
2. Una secuencia de mensaje ζ de C_i^a a un intervalo I_j^{x+1} de un proceso p_j, y
3. Una secuencia causal de mensajes μ del intervalo I_j^{x+1} a C_i^a.

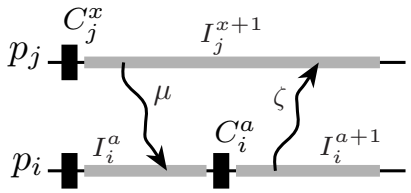


Figura 5.7: Caracterización de *z-cycle* rastreable.

En particular, estamos interesados en los *z-cycles* rastreables que pueden predecir la formación de un *z-cycle*. Estos *z-cycles* rastreables son aquellos con un único mensajes en su cadena de mensajes ζ. Por ejemplo, en la Figura 5.8 mostramos un *z-cycle* rastreable con un sólo mensaje de C_i^a a p_j ($\zeta = [m_\zeta]$).

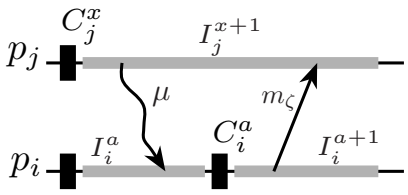


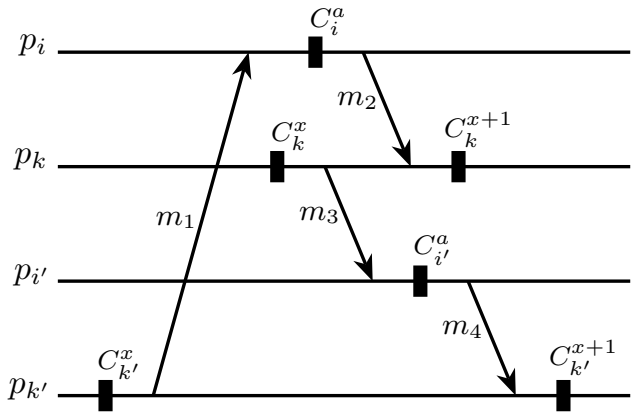
Figura 5.8: *z-cycle* rastreable que detecta la formación de un *z-cycle*.

Para finalizar esta sección, en la Figura 5.9 mostramos un ejemplo de un *z-cycle* no rastreable. Este ejemplo muestra lo difícil que es eliminar una *z-cycle* de forma eficiente[1]. A primera vista el ejemplo de la Figura 5.9(a) parece ser diferente al de la Figura 5.9(c), sin embargo, con un poco de "imaginación" y "tenacidad", podemos observar que las figuras 5.9(a) y 5.9(c) muestran el mismo ejemplo. Si intercambiamos los procesos P_k por $P_{k'}$ y P_i por $P_{i'}$ en el escenario de la Figura 5.9(a), obtenemos el escenario de la Figura 5.9(c), y viceversa. Por otra parte, la Figura 5.9(b) muestra una gráfica de la relación IDR y de *z-paths* entre *checkpoints* (líneas solidas para IDR y líneas punteadas para *z-paths*) del patrón de comunicación y *checkpoints* de la Figura 5.9(a) (y a su vez, de la Figura 5.9(c)). En esta figura podemos observar dos *z-path* ($[m_2, m_3]$ y $[m_4, m_1]$) que forman un *z-cycle* en C_i^a o C_i^a. La concurrencia de los mensajes m_2 y m_4 en los dos escenarios (Figura 5.9(a) y 5.9(c)), junto con los valores de los relojes lógicos en C_k^x y $C_{k'}^x$ determinan si el *z-cycle* se elimina con un *checkpoint* forzado (si los relojes lógicos de C_k^x y $C_{k'}^x$ son diferentes) o con dos *checkpoints* forzados (si los relojes lógicos de C_k^x y $C_{k'}^x$ son iguales).

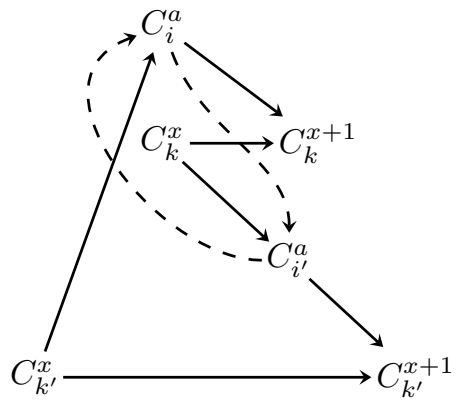
5.3. Enfoque de retraso de *checkpoint*

Hélary et al. [23] establecen un criterio de consistencia para caracterizar *intervalos consistentes* abstractos en un cómputo distribuido. Un intervalo consistente es aquel que no contradice la secuencialidad de los procesos. Por ejemplo, los intervalos formados en la Figura 5.10(a) son inconsistentes porque contradicen la secuencia de intervalos en el proceso P_j, $I_j^y \rightarrow I_j^{y+1}$, $I_j^{y+1} \rightarrow I_i^{x+1}$ y $I_i^{x+1} \rightarrow I_j^y$, por lo tanto $I_j^{y+1} \rightarrow I_j^y$, lo cual es incorrecto. Por otra parte, los intervalos formados en la Figuras 5.10(b) y 5.10(c) son consistentes.

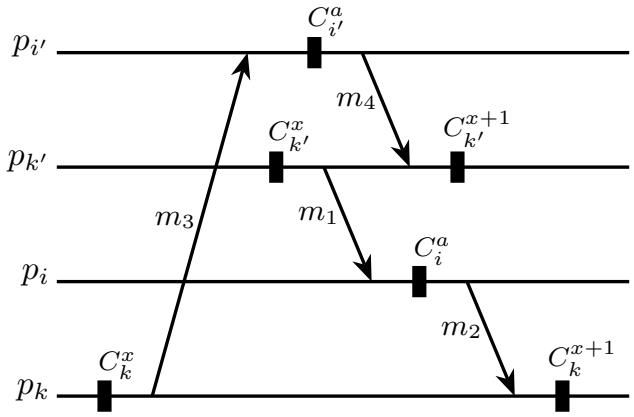
[1]Una forma eficiente de eliminar un z-cycle es aquella en la que utilizaríamos un sólo *checkpoint* forzado por *z-cycle*. En este sentido el número máximo de *checkpoints* forzados estaría determinado por el número de *z-cycles* en el patrón de comunicación y *checkpoints*.



(a) Ejemplo de un *z-cycle* no rastreable.



(b) IDR y *z-paths* entre *checkpoints* de la Figura 5.9(a).



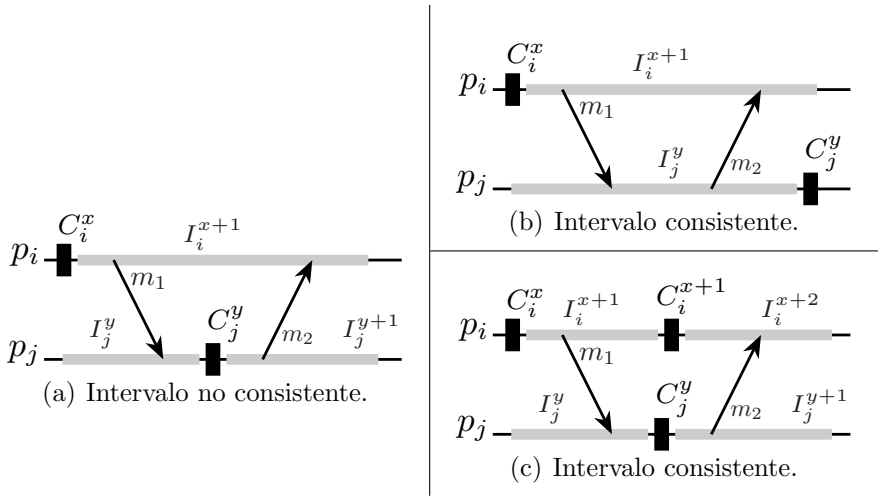
(c) *z-cycle* no rastreable de la Figura 5.9(a), visto desde otra perspectiva.

Figura 5.9: *z-cycle* no rastreable que necesita de al menos dos *checkpoints* forzados para eliminar un *z-cycle.*

El *checkpoint* C_j^y de la Figura 5.10(a) está involucrado en un *z-cycle*, pero el *checkpoint* C_j^y en las figuras 5.10(b) y 5.10(c) no está involucrado en ningún *z-cycle*.

El *z-cycle* de la Figura 5.10(a) corresponde a un *z-cycle* rastreable que predice la formación del *z-cycle* en el proceso p_j. Las figuras 5.10(b) y 5.10(c) muestran dos soluciones para eliminar el *z-cycle* de la Figura 5.10(a). La primera solución (Figura 5.10(b)) implica retrasar el *checkpoint* C_j^y después del mensaje m_2. La segunda solución (Figura 5.10(c)) implica generar un *checkpoint* (C_i^{x+1}) adicional para eliminar el *z-path* que forma al *z-cycle*.

Los esquemas presentados en la Figura 5.10 muestran la relación entre la consistencia de intervalos y la noción de *z-cycle*. La existencia de un *z-cycle* en un patrón de comunicación y *checkpoints* (CCP) genera intervalos inconsistentes, mientras la ausencia de estos, genera



(a) Intervalo no consistente.

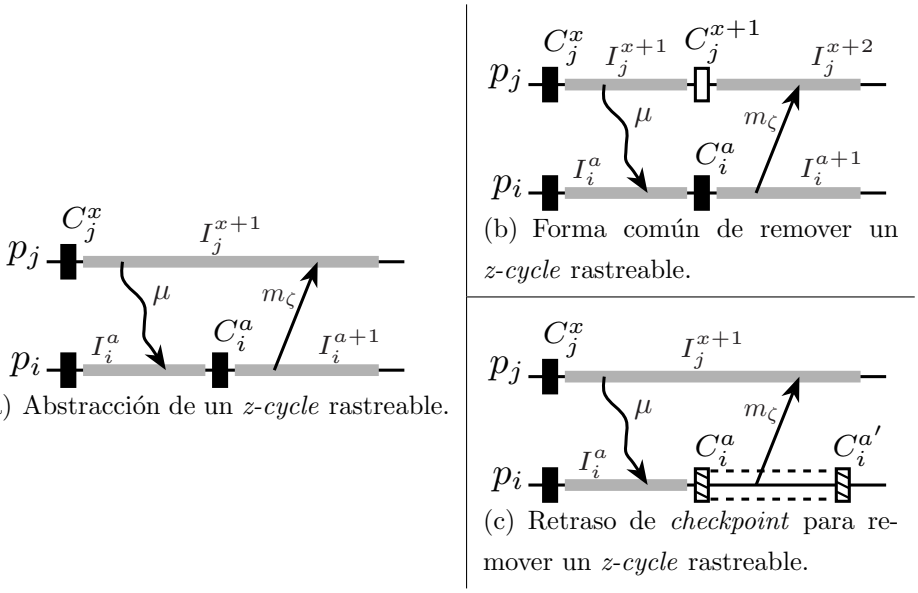
(b) Intervalo consistente.

(c) Intervalo consistente.

Figura 5.10: Esquemas de intervalos consistentes y no consistentes.

intervalos consistentes.

La propiedad Z-Cycle Free (ZCF) [39] asegura que todos los *checkpoints* en un patrón de comunicación y *checkpoints* (CCP) son útiles; no hay *checkpoints* involucrados en un *z-cycle* y todos los *checkpoints* pertenecen a algún *snapshot* global consistente (SGC) [37]. Los algoritmos de *checkpointig* de comunicación inducida (CIC) generalmente eliminan *z-cycles* de un CCP por medio de *checkpoints* forzados; sin embargo muchos de estos *z-cycles* son eliminados indirectamente, porque no pueden ser detectados "al vuelo" (*on-the-fly*).



(a) Abstracción de un *z-cycle* rastreable.

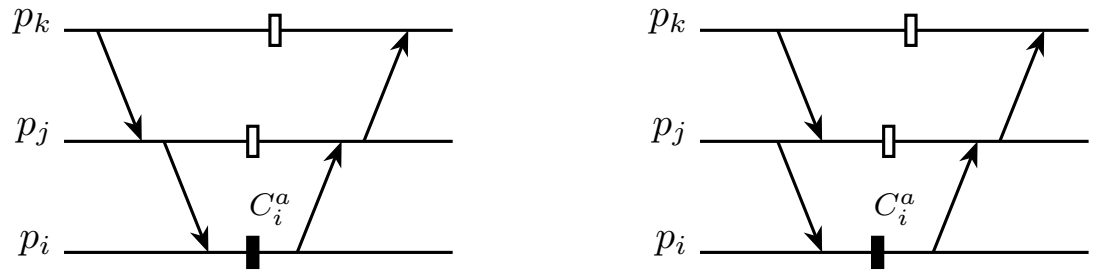
(b) Forma común de remover un *z-cycle* rastreable.

(c) Retraso de *checkpoint* para remover un *z-cycle* rastreable.

Figura 5.11: *z-cycle* rastreable y formas de removerlo.

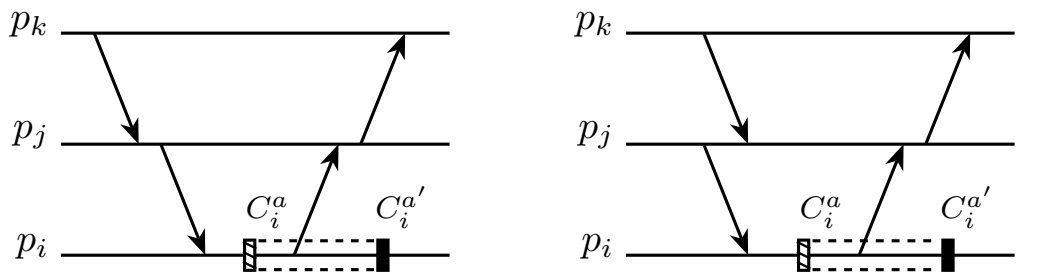
El *Fully-Informed* (FI) algoritmo [22] satisface la propiedad ZFC. Para lograr esto, FI usualmente toma *checkpoints* forzados para eliminar *z-cycles*. En la Figura 5.11(b) mostramos cómo el proceso p_j (en el algoritmo FI) toma el *checkpoint* forzado C_j^{x+1} antes de entregar el mensaje m_ζ. De esta forma, el algoritmo FI elimina el *z-cycle* que involucra a C_i^a. Esta forma de eliminar *z-cycles* rastreables no es la mejor solución porque necesita de un *checkpoint* adicional. No obstante, si podemos retrasar C_i^a hasta después de enviar el mensaje m_ζ (hasta $C_i^{a'}$) entonces eliminamos el *z-cycle* rastreable sin generar un *checkpoint* adicional (Figura 5.11(c)). En otras palabras, con el retraso de *checkpoint*, prevenimos la formación de un *checkpoint* forzado (C_j^{x+1}).

Llamamos *checkpoint retrasado* (*delayed checkpoint*) a la posibilidad de retrasar un *checkpoint* y prevenir la formación de un *checkpoint* forzado. Un *checkpoint* retrasado es un *checkpoint* tentativo y no un *checkpoint* permanente, en este sentido, solo es almacenado en la memoria del proceso. De esta forma un *checkpoint* retrasado no es trasferido al sistema de almacenamiento de *checkpoints* del sistema (*stable storage*).



(a) Efecto cascada en *z-path* causal. (b) Efecto cascada en *z-path* no causal.

Figura 5.12: Ejemplos con *checkpoints* forzados en cascada en algoritmos CIC.



(a) Sin efecto cascada en *z-path* causal. (b) Sin efecto cascada en *z-path* no causal.

Figura 5.13: Ejemplos sin efecto cascada al aplicar el retraso de *checkpoint*.

La ventaja de aplicar el retraso de *checkpoint*, respecto a los algoritmo convencionales

de CIC, la observamos en las Figuras 5.12 y 5.13. En la Figura 5.12 tenemos un efecto en cascada de *checkpoints* forzados cuando un algoritmo CIC intenta eliminar el *z-cycle* que involucra al *checkpoint* C_i^a. Por otro lado, si aplicamos el *checkpoint* retrasado al *checkpoint* C_i^a, que eventualmente formará el *z-cycle* por medio de un *z-path* causal (Figura 5.13(a)) o no-causal (Figura 5.13(b)), eliminamos el *z-cycle* sin generar un *checkpoint* forzado y eliminamos el efecto cascada que se forman en los algoritmos CIC.

5.3.1. Condiciones Seguras para el Retraso de *Checkpoint* (CSRC)

Las condiciones seguras para retraso de *checkpoint* están basadas en el patrón de un *z-cycle* rastreable que predice la formación de un *z-cycle*. Estas condiciones permiten a un proceso detectar y eliminar un *z-cycle* rastreable sin generar un *checkpoint* forzado. La definición formal de estas condiciones es la siguiente:

Definición 17. *Un proceso p_i puede retrasar su último* checkpoint *no-forzado C_i^a hasta después de enviar el mensaje m_ζ a un proceso p_j, si existe un* checkpoint *C_j^x en p_j, tal que las siguientes condiciones se satisfacen:*

$\mathcal{C}1$ $C_j^x \downarrow C_i^a$.

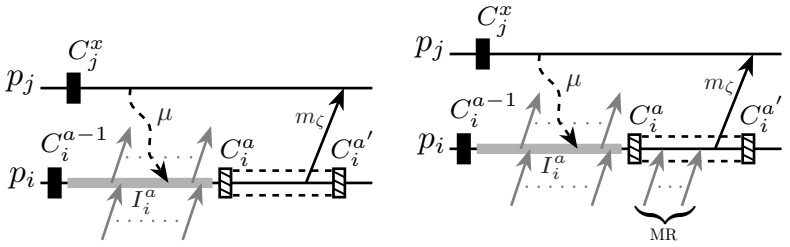
$\mathcal{C}2$ *Si hay un conjunto MR de mensajes recibidos por p_i entre C_i^a y m_ζ, cada mensaje en MR satisface lo siguiente:*

$$\forall m \in MR \; m.lc < lc_i \; \wedge \; \mathscr{L}$$

en donde: lc_i y $m.lc$ son los relojes lógicos del proceso p_i y el emisor de m, respectivamente. Si la entrega del mensaje m satisface el predicado $\mathscr{L}$, esto significa que la entrega de este mensaje no forma un z-cycle en el intervalo de checkpoints*, y por lo tanto, p_i puede realizar el retraso de su* checkpoint *no forzado (ver, Figure 5.14.b). La definición del predicado $\mathscr{L}$ es la siguiente:*

$$\mathscr{L} \equiv \nexists C_k^z \in E_i, \; C_i^{a-1} \rightarrow C_k^z \rightarrow delivery(p_i, m)$$

Note que cuando p_i retrasa su último *checkpoint* no-forzado, C_i^a (ver, Figura 5.14), este elimina el *checkpoint* C_i^a y genera, después de enviar m_ζ, un nuevo *checkpoint* local, $C_i^{a'}$. En este sentido, C_i^a es un *checkpoint* tentativo y únicamente conocido por p_i. Cuando p_i



(a) Caso simple. (b) Caso complejo.

Figura 5.14: Condiciones seguras para el retraso de *checkpoint*.

envía el mensaje m_ζ, este mensaje acarrea toda la información causal del intervalo anterior a C_i^a. Así, p_i nunca informa al sistema de la existencia de C_i^a, por lo que, este *checkpoint* no está presente en el CCP.

Nosotros mostramos *correctness* en nuestras condiciones seguras para retrasar un *checkpoint* por medio del siguiente teorema:

Teorema 4. Las condiciones seguras de retraso de *checkpoint* satisfacen el Teorema 1.

Demostración. La demostración está dividida en dos partes. En la primera parte, mostramos que las condiciones seguras satisfacen el Teorema 1, si no hay mensajes entregados entre C_i^a y m_ζ, Figura 5.14.a. En segundo lugar, extendemos esta demostración para incluir mensajes entregados entre C_i^a y m_ζ, Figura 5.14.b.

Parte I. Suponemos, sin pérdida de generalidad, que el CCP antes del retraso del *checkpoint* C_i^a satisface el Teorema 1. Entonces, cuando p_i retrasa su *checkpoint* C_i^a, este no cambia el estado del CCP, por lo que, cuando p_i envíe el mensaje m_ζ enviará la información causal del intervalo anterior de C_i^a (p_i nunca informó al sistema de la existencia de C_i^a, por lo que este *checkpoint* no se encuentra en el CCP) y genera un nuevo *checkpoint* local $C_i^{a'}$ después de enviar m_ζ. De esta forma, el CCP nunca cambia durante el retraso de C_i^a, y por lo tanto, el CCP sigue satisfaciendo el Teorema 1.

Parte II. En este caso, hay un conjunto de mensajes MR entregados entre C_i^a y m_ζ. De la condición $\mathcal{C}2$, tenemos:

$$\forall m \in MR\ m.lc < lc_i\ \wedge\ \nexists C_k^z \in E_i,\ C_i^{a-1} \rightarrow C_k^z \rightarrow delivery(p_i, m)$$

Nuevamente, suponemos, sin pérdida de generalidad, que el CCP antes de un retraso de *checkpoint* satisface el Teorema 1. Ahora, sea m el primer mensaje en la secuencia ordenada de todos los mensajes de MR, $m \in [m_1, m_2, \ldots, m_\alpha]$. Si p_i retrasa C_i^a hasta después de m, tenemos que $m.lc < lc_i$ y la entrega del mensaje m no forma un *z-cycle*.

$\square_{Teorema\ 4}$

5.4. Especificación y descripción del algoritmo DCFI

En esta sección presentamos el algoritmo DCFI que implementa el enfoque de retraso de *checkpoint* descrito en la sección anterior y presentado en [10]. Este algoritmo hace uso de las estructuras de datos y variables descritas en la Sección 4.1. El algoritmo DCFI está compuesto de tres partes: σ_0, σ_1 y σ_2. En los Cuadros 5.3 y 5.4 mostramos el pseudocódigo de estas tres partes. A continuación realizamos una descripción general de las tres partes con que cuenta nuestro algoritmo DCFI.

σ_0 Inicializa los valores del algoritmo y genera el primer *checkpoint* local de cada proceso. El pseudocódigo para esta parte del algoritmo lo presentamos en el Cuadro 5.3. El reloj lógico lc_i y las estructuras de datos: $ckpt_i[]$, $taken_i[]$ y $greater_i[]$; son inicializadas de acuerdo a lo descrito en la Sección 4.1 (ver lineas 1-5). El procedimiento *taken_checkpoint* (línea 6) genera el primer *checkpoint* del proceso. Este primer *checkpoint* es como un *checkpoint* forzado, en el sentido, de que no puede retrasarse. En el Cuadro 5.5 mostramos el pseudocódigo para el procedimiento *taken_checkpoint*. Este procedimiento difiere del presentado en algoritmo S-FI (ver Cuadro 4.3), en cuanto a que utiliza un parámetro booleano para establecer si el *checkpoint* a generar es forzado o no.

σ_1 En esta parte, p_i procesa el envío de un mensaje. Aquí, p_i usa las CSRC para retrasar su último *checkpoint* no-forzado. Si p_i puede realizar esta acción, el mensaje m es formado a partir de un resguardo de información del intervalo previo (ver líneas 7-20). Para identificar si el último *checkpoint* de p_i puede ser retrasado (línea 7), utilizamos una variable booleana $delay\_ckpt_i$, a arreglo booleano $taken\_before\_A_i[]$ y una variable entera $num\_delay\_ckpt_i$. Particularmente, $taken\_before\_A_i[k]$ captura la condición $\mathcal{C}1$ de la *Definición 17*, mientras que $delay\_ckpt_i$ captura la condición $\mathcal{C}2$. La variable $num\_delay\_ckpt_i$ cuenta el número de retrasos que ha realizado p_i. Sólo permitimos dos retrasos continuos, debido a que en las simulaciones con un número mayor de retrasos, no generan desempeños más significativos. Por otra parte, si p_i no puede retrasar su último *checkpoint*, entonces la variable $delay\_ckpt_i$ se establece

en $false$, m es formado a partir de la información actual del intervalo y, finalmente, se registra el envío de m a p_j ($send\_to_i[j] = true$).

```
   (σ0) Inicialización de procesos p_i.
 1 k, l : 1 ... n, donde n es el número de procesos.
 2 ∀k do ckpt_i[k] := 0; enddo
 3 lc_i := 0;
 4 taken_i[i] := false;
 5 greater_i[i] := false;
   // p_i toma su primer checkpoint. Este checkpoint no puede retrasarse.
 6 taken_checkpoint(false);
   (σ1) Cuando p_i envía un mensaje m a p_j.
 7   if(delay_ckpt_i ∧ ¬taken_before_A_i[j] ∧ num_delay_ckpt_i < 3) then
 8         if(¬rec) then
 9           m := (lc_before_i, ckpt_before_i, greater_before_i, taken_before_i);
10         else
11           m := (lc_before_i, ckpt_before_i, greater_before_i, taken_before_i);
12           ∀k do
13             taken_before_A_i[k] := taken_before_i[k];
14             sent_to[k] := false;
15           enddo
16           ∀k ≠ i do taken_i[k] := true; greater_i[k] := true; enddo
17           rec := false;
18         endif
19         num_delay_ckpt_i := num_delay_ckpt_i + 1;
20   else
21         delay_ckpt_i := false;
22         m := (lc_i, ckpt_i, greater_i, taken_i);
23         sent_to_i[j] := true;
24   endif
25 send(m, Data) to p_j;
```

Cuadro 5.3: Algoritmo DCFI (σ_0 y σ_1).

σ_2 En esta última parte del algoritmo DCFI, procesamos la recepción de mensajes. En σ_2 las estructuras de datos son actualizadas con la información acarreada de otros procesos (ver líneas 35-64). El predicado $\mathscr{L}$ (Definición 17) es evaluado de manera similar a la condición $\mathcal{C}2''$, ambos capturan el mismo patrón. $\mathcal{C}2''$ es evaluada con la información del intervalo de *checkpoint* actual o corriente (linea 32), mientras que $\mathscr{L}$ es evaluada con la información del intervalo previo (línea 27).

(σ_2) *Cuando* p_i *recibe un mensaje* $(m, data)$ *de* p_j. $m := (lc, ckpt, greater, taken)$.

```
26 if (delay_ckpt_i) then
27    if  (m.ckpt[i] = ckpt_before_i[i]  ∧  m.taken[i]) then
28       delay_ckpt_i := false;
29    endif
30 endif
31 if  ((∃k : sent_to_i[k]  ∧  m.greater[k])  ∧  m.lc > lc_i) ∨
32    (m.ckpt[i] = ckpt_i[i]  ∧  m.taken[i]) then
33 taken_checkpoint(false);
34 endif
35 case
36  m.lc > lc_i →
37      lc_i := m.lc;
38      delay_ckpt_i := false;
39      ∀k ≠ i do greater_i[k] := m.greater[k] enddo
40  m.lc = cl_i →
41      delay_ckpt_i := false;
42     ∀k do greater_i[k] := greater_i[k] ∧ m.greater[k]; enddo
43  m.lc < lc_i →
44      if(m.lc = lc_before_i)
45        ∀k do greater_before_i[k] := greater_before_i[k] ∧ m.greater[k]; enddo
46      endif
47 endcase
48 ∀k ≠ i do
49  case
50    m.ckpt[k] > ckpt_i[k] →
51       ckpt_i[k] := m.ckpt[k];
52       taken_i[k] := m.taken[k];
53       if(delay_ckpt_i) then
54          ckpt_before_i[k] := m.ckpt[k];
55          taken_before_i[k] := m.taken[k];
56       endif
57    m.ckpt[k] = ckpt_i[k] →
58       taken_i[k] := taken_i[k] ∨ m.taken[k];
59       if(delay_ckpt_i) then
60          taken_before_i[k] := taken_before_i[k] ∨ m.taken[k];
61       endif
62    m.ckpt[k] < ckpt_i[k] →  skip
63  endcase
64 enddo
65 rec := true;
66 delivery(m);
```

Cuadro 5.4: Algoritmo DCFI (σ_2).

```
   //Cuando p_i toma un checkpoint local o forzado.
67 procedure taken_checkpoint(boolean type)
68   delay_ckpt_i := type;
69   if(delay_ckpt_i) then
70       rec_i := false;
71       num_delay_ckpt_i := 0;
72       lc_before_i := lc_i;
73       ∀k do
74           ckpt_before_i[k] := ckpt_i[k];
75           taken_before_i[k] := taken_i[k];
76           greater_before_i[k] := greater_i[k];
77           taken_before_A[k] := taken_i[k];
78       enddo
79   endif
80   ∀k do sent_to_i[k] := false; enddo
81   ∀ k ≠ i do taken_i[k] := true; greater_i[k] := true; enddo
82   lc_i := lc_i + 1;
83   ckpt_i[i] := ckpt_i[i] + 1;
84 endprocedure
```

Cuadro 5.5: Procedimiento para generar un *checkpoint* en el algoritmo DCFI.

5.5. Simulación del algoritmo DCFI

Comparamos el desempeño de nuestro algoritmo DCFI con los algoritmos de *checkpointing* de comunicación inducida FI [22] y FINE [33]. Estos tres algoritmos (DCFI, FI y FINE) fueron simulados y analizados con el simulador ChkSim [51]. Este simulador fue mencionado en la Sección 4.5.

En la simulación, analizamos el número de *checkpoints* forzados tomados por los tres algoritmos. Analizamos seis escenarios: 1000, 2500, 5000, 7500, 10000 y 50000 mensajes. Cada escenario fue creado con una distribución uniforme de eventos (*send*, *delivery* y internos *checkpoints*) y variando el número de procesos de 10, 20, 30, . . . , 150. Por cada escenario y número de procesos (10, 20, 30,. . . , 150) analizamos 100 patrones de comunicación y *checkpoints* (CCP) generados de forma aleatoria. Por ejemplo, para el escenario de 1000 mensajes simulamos: 100 CCP con 10 procesos, 100 CCP con 20 procesos, . . . , 100 CCP con 150 procesos. Así, por cada escenario (1000, 2500, . . . mensajes) analizamos 1,500 CCP, analizando en total 9,000 CCP por cada algoritmo. Hay que notar que estos 9,000 CCP fueron generados de manera aleatoria por la herramienta CkSim y que estos mismos CCP fueron utilizados por los tres algoritmos (DCFI, FI y FINE) para el análisis

de sus respectivos desempeños.

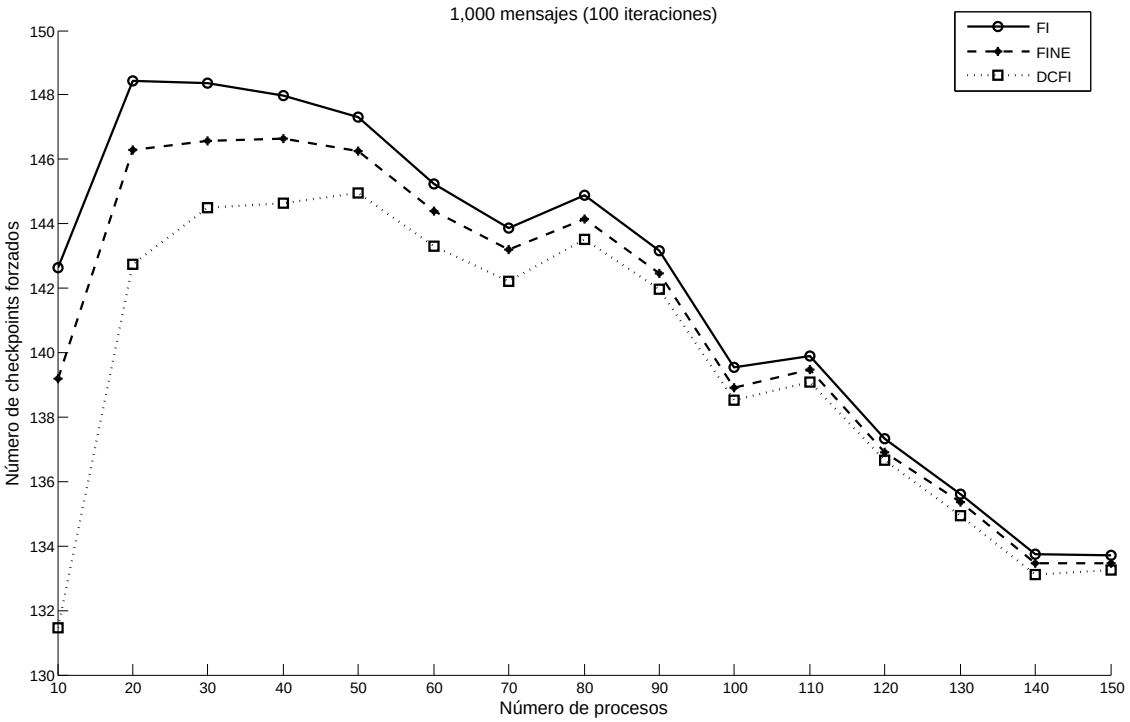
Las gráficas en las Figuras 5.15, 5.16 y 5.17 muestran los resultados del número de *checkpoints* forzados en promedio para los 100 CCP analizados por cada 10 procesos (10, 20,. . . ,150).

En las Figuras 5.15, 5.16 y 5.17 podemos observar que el número de *checkpoints* forzados tomados por el algoritmo DCFI, en los seis escenarios, es menor que el de los algoritmos FI y FINE; mientras que el número de *checkpoints* forzados tomados por FINE es menor que FI. El número de *checkpoints* forzados de DCFI representa en promedio un 3 % menos que FI, mientras que para FINE este representa en promedio un 1.5 % menos con respecto a FI.

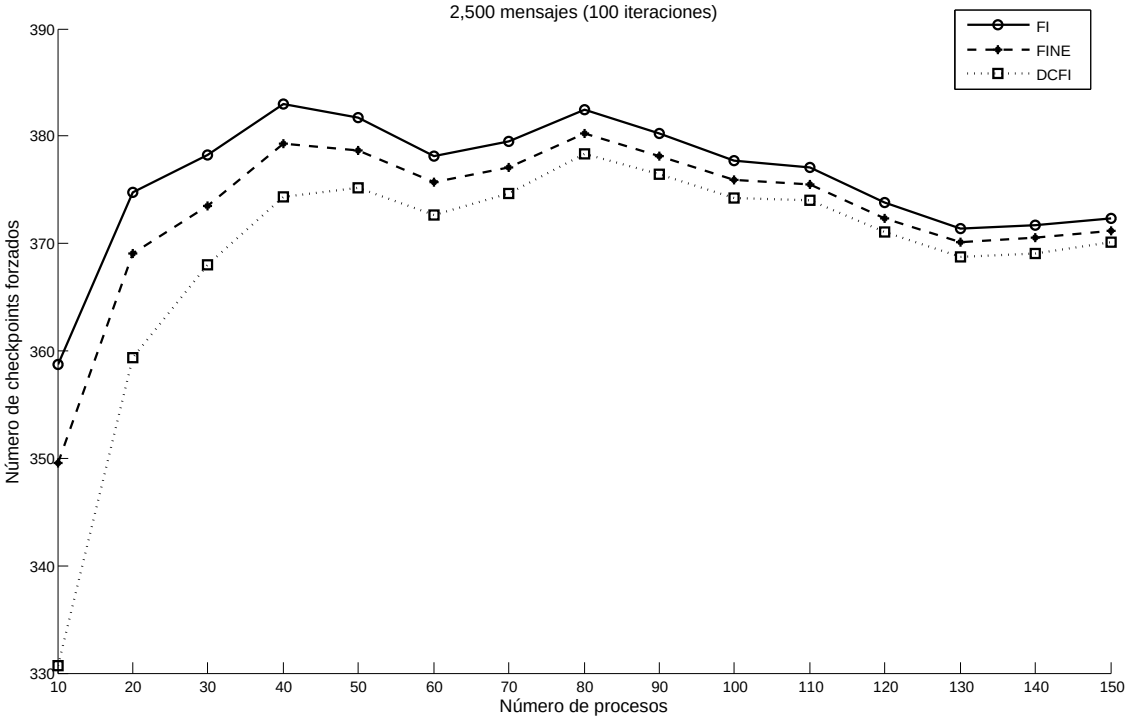
5.6. Conclusiones

En este capítulo presentamos el enfoque de retraso de *checkpoint* no forzado para algoritmos de comunicación inducida. Este enfoque reduce el número de *checkpoints* forzados por medio de la identificación de ciertas condiciones que llamamos *Condiciones Seguras para Retraso de Checkpoint* (CSRC). Además, presentamos el algoritmo DCFI que implementa este enfoque.

Con el objetivo de analizar el desempeño de nuestro algoritmo DCFI, presentamos una simulación del comportamiento del número de *checkpoints* forzados de S-FI y simulamos los algoritmos FI y FINE para comparar el número de *checkpoints* forzados que genera cada algoritmo. Los resultados de estas simulaciones muestran que nuestro algoritmo DCFI toma el menor número de *checkpoints* forzados, un 3 % menos en promedio que FI y un 1.5 % menos en promedio que FINE. En este sentido, DCFI es más eficiente que FI y FINE.

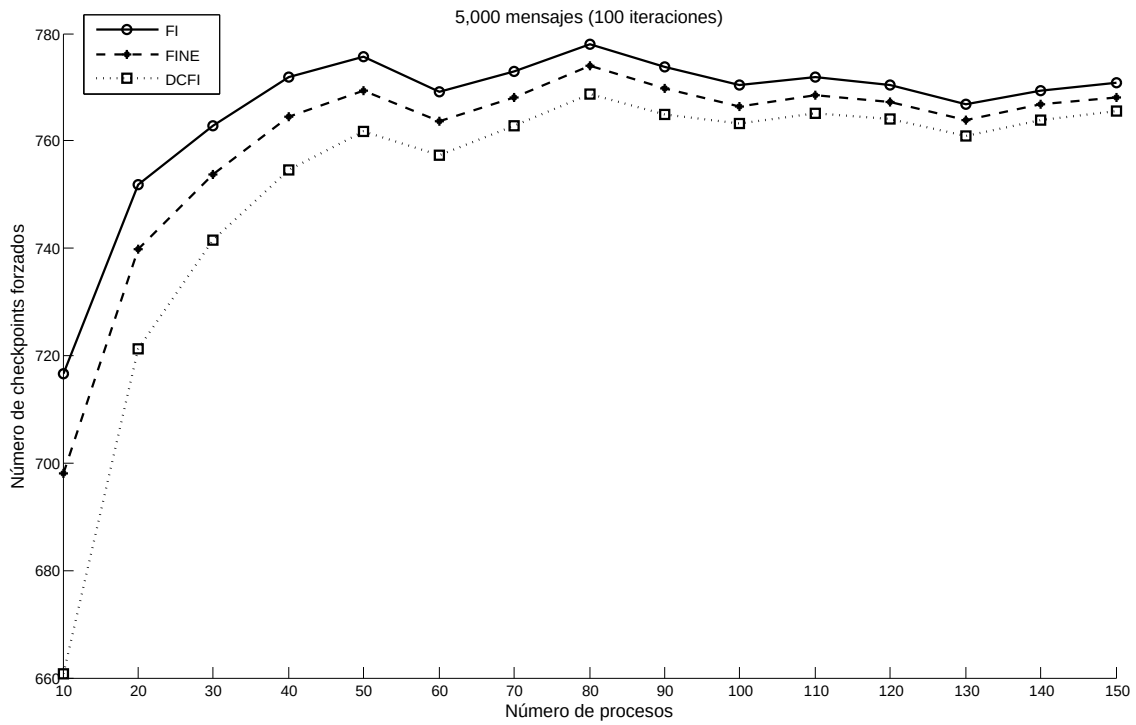


(a) Promedio de *checkpoints* forzados en 100 escenarios con 1,000 mensajes enviados.

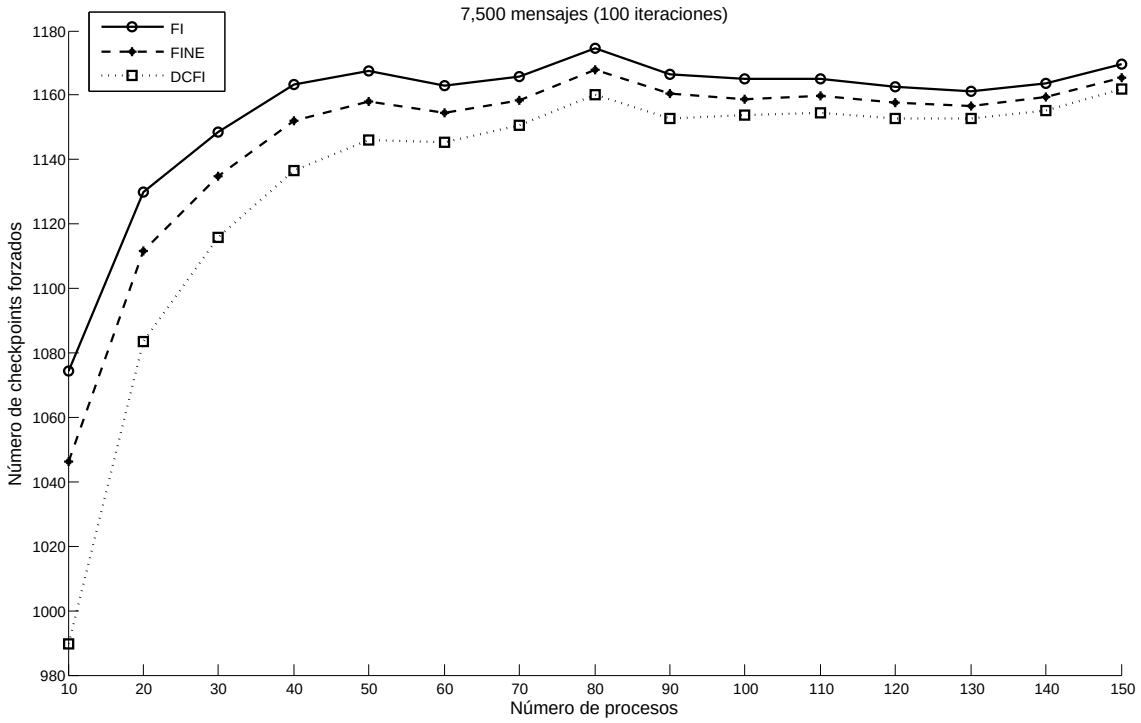


(b) Promedio de *checkpoints* forzados en 100 escenarios con 2,500 mensajes enviados.

Figura 5.15: Resultados de la simulación de DCFI para 1,000 y 2,500 mensajes.

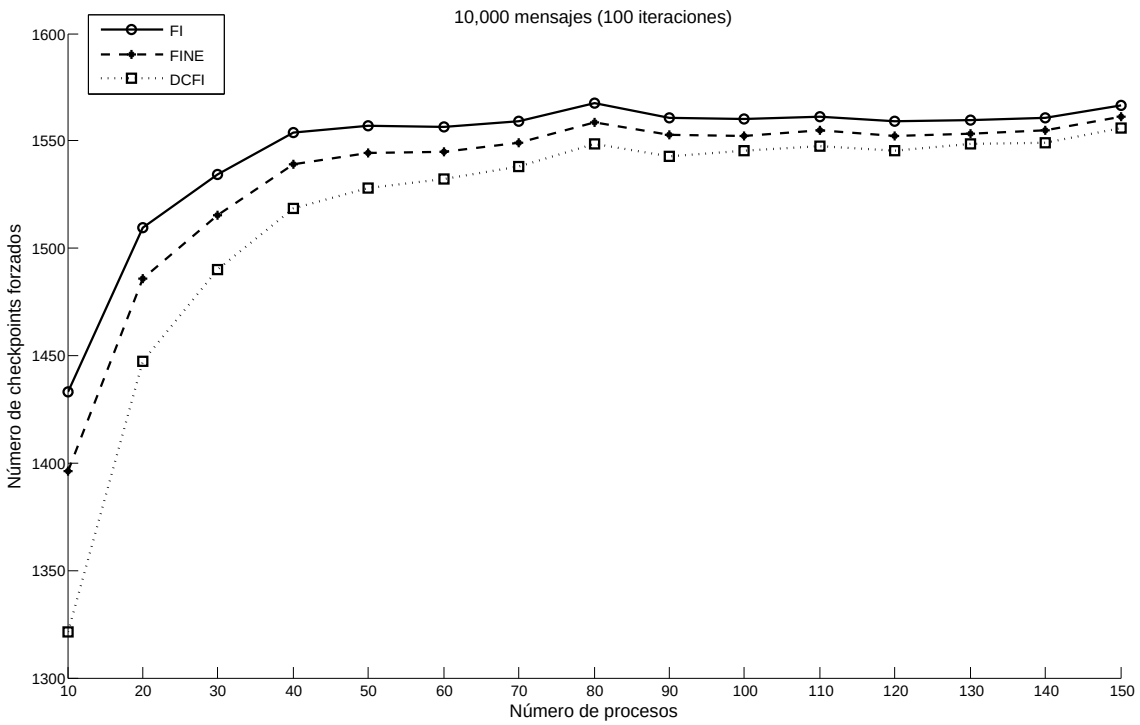


(a) Promedio de *checkpoints* forzados en 100 escenarios con 5,000 mensajes enviados.

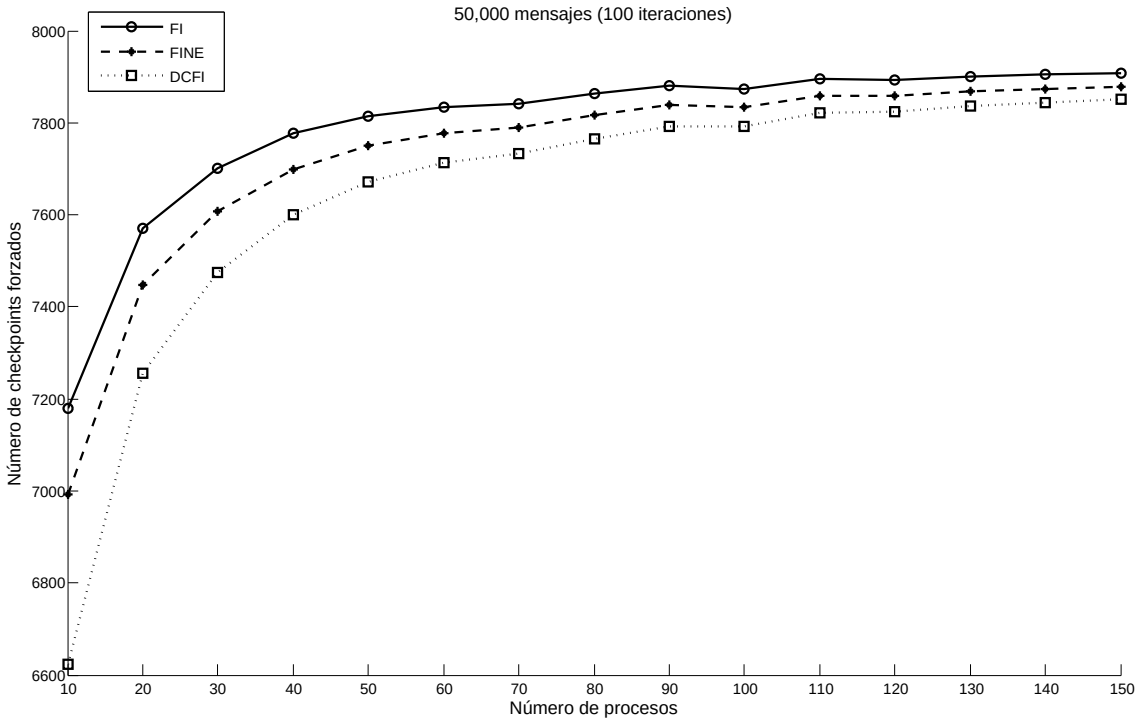


(b) Promedio de *checkpoints* forzados en 100 escenarios con 7,500 mensajes enviados.

Figura 5.16: Resultados de la simulación de DCFI para 5,000 y 7,500 mensajes.



(a) Promedio de *checkpoints* forzados en 100 escenarios con 10,000 mensajes enviados.



(b) Promedio de *checkpoints* forzados en 100 escenarios con 50,000 mensajes enviados.

Figura 5.17: Resultados de la simulación de DCFI para 10,000 y 50,000 mensajes.

Capítulo 6

Algoritmo HSDC para ambientes heterogéneos

En este capítulo introducimos nuestro algoritmo HSDC (*Heterogeneous Scalable Delay Checkpoint*) de comunicación inducida para ambientes heterogéneos. La funcionalidad de este algoritmo radica en la fusión de los algoritmos S-FI y DCFI desarrollados en los capítulos 4 y 5, respectivamente, además de adicionar el soporte para el modelo de ejecución síncrono. De acuerdo a Charron-Bost et al. [15], toda ejecución síncrona puede ser desarrollada por una ejecución asíncrona (con un costo de *overhead*). Sin embargo, en nuestro problema de *checkpointing* para sistemas heterogéneos y de acuerdo a nuestro modelo de sistema (ver Sección 2.1), el modelo de ejecución síncrono únicamente lo tenemos, al interior de los nodos, mientras que el modelo de ejecución asíncrono se tiene tanto al interior de los nodos como entre nodos.

En este sentido, para solucionar nuestro problema de *checkpointing* en ambientes heterogéneos, nosotros optamos por:

1. Fusionar nuestros algoritmos de *checkpointing* asíncronos S-FI y DCFI desarrollados. Estos dos algoritmos son complementarios. S-FI ataca el problema de *overhead* en el sistema y DCFI ataca el problema del número de *checkpoints* forzados en los algoritmos de comunicación inducida.

2. Manejar los nodos de forma que podamos detectar el tipo de dependencia (inter-nodo o inter-proceso) generada por el envío de mensajes entre procesos del sistema.

3. En nodos que procesan un modelo de ejecución síncrona utilizamos la memoria compartida, si el envío de un mensaje genera dependencia inter-proceso (mensajes entre

procesos del mismo nodo); y paso de mensajes, si el mensaje genera dependencia inter-nodo (mensajes entre procesos de nodos diferentes).

4. En nodos que procesan un modelo de ejecución asíncrona utilizamos el paso de mensajes, si el envío de un mensaje genera tanto dependencia inter-proceso como inter-nodo.

Las dependencias inter-nodo pueden ser manejadas por nuestros algoritmos asíncronos S-FI y DCFI debido a la naturaleza asíncrona de las dependencias inter-nodo, mientras que las dependencias inter-proceso pueden ser manejadas por un algoritmo síncrono, como los desarrollados en [24, 1, 53, 6, 5]. De esta forma, no generamos un alto costo de *overhead* en los mensajes internos del nodo y utilizamos las ventajas que nos ofrecen las ejecuciones síncronas (memoria compartida y reloj global para sincronizar los procesos de un nodo). En resumen, un nodo con modelo de ejecución síncrono necesita determinar el tipo de dependencia que generará el envío de un mensaje, para que los procesos de este nodo determinen el mecanismo de comunicación que utilizarán: paso de mensajes para dependencias inter-nodo (mensajes dirigidos a un procesos que no se encuentra en el nodo) o memoria compartida para dependencias inter-proceso (mensajes dirigidos a procesos del mismo nodo).

Por otra parte, para la generación de un *snapshot* global consistente (SGC) del sistema heterogéneo, necesitamos un conjunto de *checkpoints* por cada nodo. Estos *checkpoints*, que forman un SGC del sistema, no deben estar involucrados en un *z-cycle*. Por la parte de los nodos con modelo de ejecución síncrona, la naturaleza síncrona de los nodos permite generar *checkpoints* sin que dos o más de ellos estén relacionados por un *z-path*; pero, por parte de los nodos con modelo de ejecución asíncrona, los procesos generan *checkpoints* con posibilidades de que dos o más estén relacionados por un *z-path*.

Los *checkpoints* generados por nuestros algoritmos S-FI y DCFI (con modelo de ejecución asíncrono) son todos útiles (pueden ser parte de un SCG); sin embargo, no cualquier conjunto de *checkpoints* de un nodo es parte de un SGC del sistema heterogéneo, porque algunos de ellos pueden estar relacionados con un *z-path*. De esta forma, existe la posibilidad de que los tiempos lógicos de dos o más nodos con dependencia inter-nodo no puedan formar un SGC del sistema heterogéneo.

Para resolver el problema que establece la dependencia inter-nodo respecto a la formación de SGC, en la siguiente sección introducimos la noción de agrupación de *checkpoints*

en un nodo. Esta agrupación de *checkpoints*, en cada nodo del sistema, tiene como objetivos:

1. Generar conjuntos de *checkpoints* (uno por cada proceso en el nodo) en donde dos o más *checkpoints* no estén relacionados por un *z-path*.

2. Permitir y dar flexibilidad a la formación de múltiples conjuntos de *checkpoints* en cada nodo, de tal forma que a partir de estos conjunto generados en cada nodo, podamos generar un conjunto global de *checkpoints* que forme un SGC del sistema heterogéneo.

6.1. Principios de agrupación de *checkpoints*

En esta sección introducimos la *Relación de Dependencia Inmediata ZigZag* (ZIDR) para relacionar los *checkpoints* de nodos con modelo de ejecución asíncrona. Además, describimos un método que usa la relación ZIDR para agrupar los *checkpoints* de un nodo en tiempos lógicos. Cada tiempo lógico agrupa a un conjunto de *checkpoints* que no están relacionados por la relación ZIDR dentro del nodo. De esta forma, un tiempo lógico en un nodo con n procesos está formado con uno o a lo más n *checkpoints* (un *checkpoint* por cada proceso en el nodo).

La agrupación de *checkpoints* es aplicada tanto a nodos con modelo de ejecución asíncrono como síncrono. Aunque en principio los *checkpoints* de un nodo con modelo de ejecución síncrono están sincronizados (por un reloj global en el nodo), por lo que pertenecen o se agrupan en un mismo tiempo lógico, algunos *checkpoints* podrían estar retrasados ligeramente debido a un retraso de *checkpoint* (algoritmo DCFI, capítulo 5). Sin embargo, el retraso de *checkpoint* no desplazaría al *checkpoint* del nodo síncrono hacia otro tiempo lógico en el nodo (ver, CSRC Definición 17).

6.1.1. Relación Z-Dependencia Inmediata

La *Relación de Dependencia Inmediata ZigZag* establece una dependencia lógica y caracteriza la noción de *z-path* definida por Netzer y Xu [37] (Definición 9, Sección 2.3). La definición formal de la relación ZIDR es la siguiente:

Definición 18. *Dos checkpoints $C_i^a, C_k^b \in E$, tienen una relación ZIDR, denotada por $C_i^a \rightsquigarrow C_k^b$, si una de las siguientes restricciones se satisface.*

1. *Si* $C_i^a \downarrow C_k^b$, *entonces* $C_i^a \rightsquigarrow C_k^b$.

2. *Si* $\exists\, C_j^x, C_j^{x+1} \in E_i,\ C_j^x \downarrow C_j^{x+1}$ *tal que* $C_i^a \downarrow C_j^{x+1} \wedge C_j^x \rightsquigarrow C_k^b$, *entonces* $C_i^a \rightsquigarrow C_k^b$.

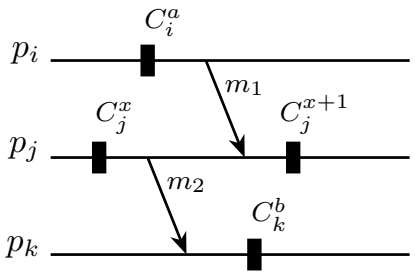


Figura 6.1: Esquema base de la relación $ZIDR$.

El objetivo de la relación $ZIDR$ es capturar la noción de *z-path* entre *checkpoints*. La restricción 1 de la Definición 18 captura los *z-paths* causales entre *checkpoints*, mientras que la restricción 2 captura los *z-paths* no causales. El uso de la relación IDR en la definición de ZIDR asegura una representación compacta de la causalidad entre *checkpoints* del sistema. En la Figura 6.1 mostramos el esquema base para la captura de un *z-path* no causal, por medio de ZIDR, $C_i^a \rightsquigarrow C_k^b$. En esta figura, los *checkpoints* C_i^a, C_j^x, C_j^{x+1} y C_k^b cumplen con la restricción 2 de la definición ZIDR, y por lo tanto, C_i^a y C_k^b tienen una relación ZIDR.

Por otra parte, la definición de un *z-cycle* por medio de la relación ZIDR es la siguiente:

Definición 19. *Un* z-cycle *es un ZIDR de un checkpoint* C_i^x *a sí mismo:* $C_i^x \rightsquigarrow C_i^x$.

6.1.2. Método de agrupación de *checkpoints*

El Cuadro 6.1 describe un método para agrupar los *checkpoints* de un nodo en conjuntos ZIDR relacionados. Cada conjunto representa un tiempo lógico, y los tiempos lógicos representan conjuntos de *checkpoints* ZIDR relacionados por al menos un *checkpoint*. De modo que, todos los *checkpoints* que forman a un conjunto no están ZIDR relacionados. Si un tiempo lógico contiene a un *checkpoint* por cada proceso en el nodo, este es un conjunto factible para ser parte de un *snapshot* global del sistema.

Los conjuntos de *checkpoints* ordenados en tiempos lógicos permiten la construcción de algoritmos de recuperación (*Rollback-Recovery*) eficientes del sistema, eliminar información no útil para la recuperación de sistema (*Garbage Collection*[1]), evaluar condiciones

[1]Un algoritmo *Garbage Collection* elimina toda la información que no es útil para la recuperación del sistema de caso de una falla.

generales del sistema, etcétera.

En la Figura 6.2 mostramos un ejemplo de aplicación del método de agrupación de *checkpoints* descrito en el Cuadro 6.1. El ejemplo utiliza el patrón de comunicación y *checkpoints* (CCP) de la Figura 6.2(a). Sin pérdida de generalidad, nosotros consideramos que el sistema tiene conocimiento de cada *checkpoint* de forma ordenada; es decir, el sistema tiene conocimiento de los *checkpoints* en el siguiente orden: C_i^1, C_j^1, C_k^1, después C_i^2, C_j^2, C_k^2, después C_i^3, C_j^3, C_k^3, y por último C_i^4, C_j^4, C_k^4. Este orden es sólo una de muchas combinaciones, sin embargo, si consideramos otra combinación, el resultado final de la agrupación de *checkpoints* será siempre el mismo.

En la Figura 6.2(c) podemos observar la agrupación de los *checkpoints* C_i^1, C_j^1 y C_k^1 en el tiempo lógico t_0. Cuando C_i^2 es conocido por el sistema, el método genera el tiempo lógico t_1 y agrega a este nuevo *checkpoint*, además, analiza los *checkpoints* del tiempo lógico anterior (t_0) para reorganizar los dos conjuntos de *checkpoints*. Para este caso, C_j^1 y C_k^1 pueden pasar el tiempo lógico t_1 porque no están ZIDR relacionados con C_i^2 (Figura 6.2(d)). Después, el sistema conoce C_j^2 y genera el tiempo lógico t_2 para agregar este *checkpoint*, sin embargo, en este caso los *checkpoints* C_i^2 y C_k^1 no son trasladados al t_2; porque C_k^1 tiene una relación ZIDR con C_j^2, y porque la adicción de C_i^2 a t_2 no completa la formación de un *snapshot* global del nodo (Figura 6.2(e)).

Las Figuras 6.2(f), 6.2(g), 6.2(h), 6.2(i), 6.2(j), 6.2(k) y 6.2(l), representan gráficamente el comportamiento de los *checkpoints* en los tiempos lógicos cuando el sistema tiene conocimiento de los *checkpoints* $C_k^2, C_i^3, C_j^3, C_k^3, C_i^4, C_j^4$ y C_k^4, respectivamente.

6.2. Agrupación de *checkpoints* en el algoritmo HSDC

Para finalizar esta sección, mostramos cómo la agrupación de *checkpoints* en cada nodo es utilizada para la construcción de un *snapshot* global consistente del sistema con modelos de ejecución síncrono y asíncrono.

En la Figura 6.3 mostramos el patrón de comunicación y *checkpoints* de nuestro ejemplo heterogéneo. El sistema esta formado por tres nodos. Los procesos p_1, p_2 y p_3 forman el nodo A con modelo de ejecución asíncrono. El proceso p_4 forma el nodo B con modelo de ejecución síncrono[2]. Y los procesos p_5, p_6 y p_7 forman el nodo C con modelo de ejecución

[2]Un sólo proceso en un nodo es la abstracción de un procesador ejecutando un conjunto de procesos, por lo que, al no haber concurrencia entre procesos del mismo nodo, este cómputo puede ser modelado tanto por un modelo de ejecución síncrono como asíncrono.

Cuadro 6.1: Método para la agrupación de *checkpoints* usando ZIDR.

1. Ordenar los *checkpoints* (de la ejecución del sistema) de izquierda a derecha por medio de la relación ZIDR.

a) Inicializar los conjuntos TL y S.

- $TL = \{\}$. TL es un conjunto de tiempos lógicos $\{t_0, t_1, \ldots, t_\ell\}$, cada tiempo lógico $t_i \in TL$ contiene un conjunto de *checkpoints*.
- $S = \{\}$. S es el conjunto de pares ordenados (a, b) tal que a y b son *checkpoints* y están relacionados por la relación ZIDR ($a \rightsquigarrow b$).

b) Para cada *checkpoint* C_j^x identificado durante la ejecución del sistema, realizar:

b.1. Identificar y anexar a S todos los pares de *checkpoints* (a, b) que cumplan la relación ZIDR, $a \rightsquigarrow b$.

I. Si $b = C_j^x$, $a \rightsquigarrow C_j^x$ para algún *checkpoint* $a \in t_a$ y $t_a \in TL$.

II. Si $a \rightsquigarrow b$, debido a la identificación de C_j^x, con $a \in t_a$, $b \in t_b$ y $t_a, t_b \in TL$.

b.2. Definir el tiempo lógico al que pertenecerá el *checkpoint* C_j^x y anexarlo.

I. Si $TL = \{\}$ (conjunto de tiempos lógicos vacío), entonces creamos t_0 y anexamos t_0 a TL.

II. Si C_j^x es un *checkpoint* inicial, entonces anexamos C_j^x a t_0.

III. Si existe dos tiempos lógicos $t_i, t_s \in TL$ tal que $T(t_i) = T(t_s) + 1$ y un *checkpoint* $a \in t_s$ tal que $a \rightsquigarrow C_j^x$, entonces anexamos C_j^x al conjunto t_i. $T(t_k)$ es una función que obtiene el índice de un tiempo lógico $t_k \in TL$, es decir, $T(t_k) = k$.

IV. Si existe un conjunto finito $A = \{a_0, a_1, \ldots, a_s\}$ de *checkpoints*, tal que $\forall a \in A,\ a \rightsquigarrow C_j^x$ y todo $a \in A$ pertenece a algún conjunto en TL, entonces, buscamos el tiempo lógico máximo, t_z, de entre todos los $a \in A$ y anexamos C_j^x al conjunto t_{z+1} si este existe, de lo contrario creamos el conjunto t_{z+1} y anexamos C_j^x.

2. Reordenar los *checkpoints* de los $t_a \in TL$ que puedan ser trasladados a un conjunto de tiempos lógicos mayor(t_{a+1}) y formen un *snapshot* global consistente.

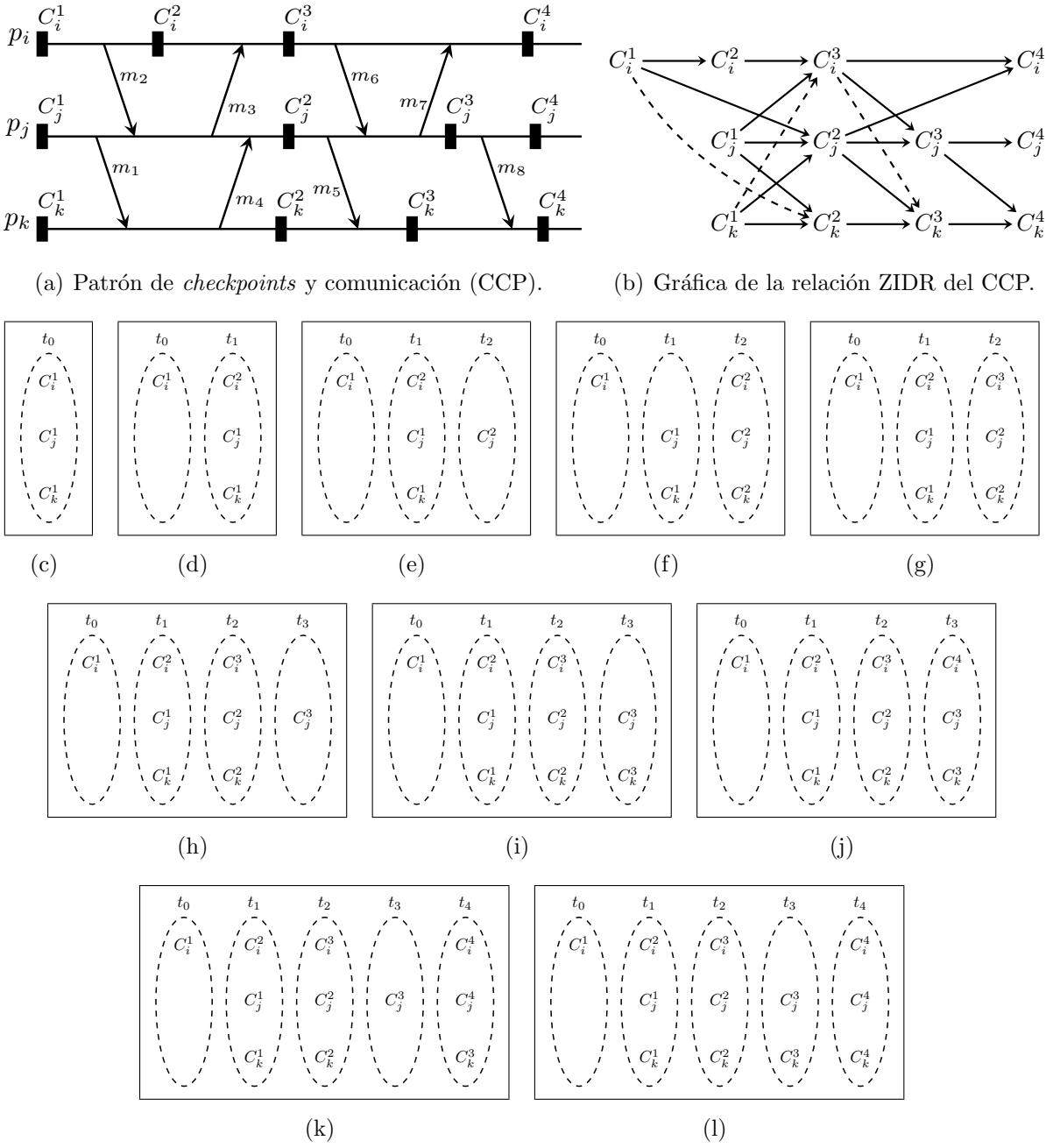


Figura 6.2: Ejemplo de conjunto de *checkpoints* con $ZIDR$.

síncrono.

La agrupación de los *checkpoints* en tiempos lógicos del nodo A se describió en el punto anterior, la Figura 6.2 muestra una representación gráfica de la agrupación de *checkpoints*, conforme los procesos del nodo A desarrollan su cómputo. Por otra parte, la naturaleza de ejecución síncrona de los procesos en los nodos B y C, permite generar tiempos lógicos de manera simple (todos los procesos de un nodo se sincronizan para generar un *checkpoint*

por proceso); por lo que, en cada tiempo lógico, se tiene a un *checkpoint* por cada proceso en el nodo.

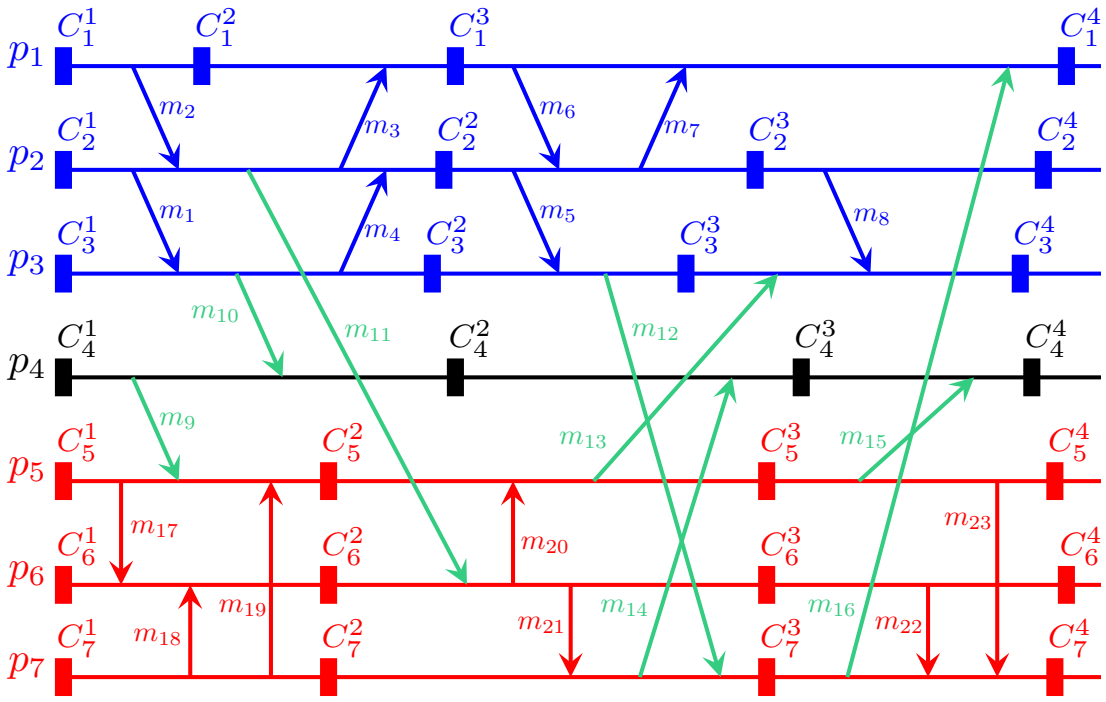


Figura 6.3: Ejemplo de patrón de comunicación y *checkpoints* heterogéneo.

Las dependencias inter-nodo (mensajes entre procesos de diferentes nodos) establecen relaciones causales entre tiempos lógicos de diferentes nodos. Por ejemplo, sea t_A el tiempo lógico formado por los *checkpoints* C_1^3, C_2^2 y C_3^2 del nodo A (Figura 6.3, ó Figura 6.2(g) con $i = 1$, $j = 2$ y $k = 3$), y t_C el tiempo lógico formado por C_5^3, C_6^3 y C_7^3 del nodo C (procesos p_5, p_6 y p_7), ver Figura 6.3; entonces, el mensaje m_{12} enviado por el proceso p_3 al proceso p_7, genera una dependencia inter-nodo entre los tiempos lógicos t_A y t_C.

Por último, los tiempos lógicos en cada nodo tienen la desventaja de formar dependencias causales y no causales. Al igual que los eventos (de cada proceso en el sistema), los tiempos lógicos entre nodos pueden formar *z-path* entre sí. Sin embargo, debido a que cada *checkpoint* del sistema cumple el Teorema 1 (ver Sección 2.3); los tiempos lógicos de cada nodo pueden reestructurase con otros *checkpoints* del mismo nodo, de tal forma, que todos los nodos pueden reestructurar un tiempo lógico, sin que dos de ellos estén relacionados por la relación *z-path*. Finalmente, la unión de estos tiempos lógicos reestructurados y sin ninguna relación *z-path* entre un par de ellos, formarán un *snapshot* global del sistema heterogéneo.

6.3. Especificación del algoritmo HSDC

En esta sección presentamos el algoritmo heterogéneo para modelos de ejecución síncrona y asíncrona. Este algoritmo hace uso de los algoritmos S-FI y DCFI descritos en los capítulos 4 y 5, respectivamente. El algoritmo heterogéneo está compuesto de cuatro partes: η_0, ρ_0, ρ_1 y ρ_2. En los Cuadros 6.2, 6.3 y 6.4 mostramos el pseudocódigo de estas cuatro partes. A continuación realizamos una descripción general de pseudocódigo de estas cuatro partes que forman a nuestro algoritmo heterogéneo.

η_o Inicializa los nodos del sistema. En esta parte, se establecen las características de nodo, como: modelo de ejecución y número de procesos. Inicializa los procesos que conforman al nodo, y, si el nodo tiene un modelo de ejecución síncrono, sincroniza a los procesos del nodo para tomar un *checkpoint* por proceso. El pseudocódigo para esta parte del algoritmo lo presentamos en el Cuadro 6.2.

ρ_0 Inicializa los valores de las variables y estructuras del algoritmo. Si el proceso pertenece a un nodo con modelo de ejecución asíncrona, genera el primer *checkpoint* local del proceso. El pseudocódigo para esta parte del algoritmo lo presentamos en el Cuadro 6.3. Las estructuras de datos y reloj lógico son inicializados de acuerdo a la parte ω_o de S-FI y σ_0 de DCFI. El procedimiento *taken_checkpoint* (línea 23) genera el primer *checkpoint* del proceso. Este primer *checkpoint* es un *checkpoint* forzado, en el sentido de que no puede retrasarse. En el Cuadro 6.5 mostramos el pseudocódigo para el procedimiento *taken_checkpoint*. Este procedimiento es similar al presentado en algoritmo DCFI (ver Cuadro 5.5).

ρ_1 En esta parte p_i procesa el envío de un mensaje (Cuadro 6.3), es similar a ω_1 y σ_1 de S-FI y DCFI, respectivamente. Aquí, p_i deduce el mecanismo de comunicación a utilizar por medio de las variables is_PS y *id_nodo* (línea 26). Cuando p_i va ha enviar un mensaje por paso de mensajes, este puede retrasar su último *checkpoint* no forzado. En caso de retrasar el último *checkpoint*, p_i construye un conjunto de tuplas (similar a como lo hace ω_1 de S-FI) a partir de un resguardo de información del intervalo previo (líneas 30-43). Para hacer esto, utilizamos el procedimiento *getTupla* definido en la Cuadro 6.5. La identificación del último *checkpoint* de p_i es igual que en DCFI. Por otra parte, si p_i no puede retrasar su último *checkpoint*, entonces la variable $delay\_ckpt_i$ se establece en $false$, y calcular las tuplas a enviar como en

S-FI y a partir de la información actual del intervalo (líneas 44-46). Finalmente, se registra el envío de m a p_j ($send\_to_i[j] = true$).

ρ_2 En esta última parte del algoritmo heterogéneo, procesamos la recepción de mensajes. Las estructuras de datos son actualizadas con la información acarreada en los mensajes (ver líneas 60-93). El predicado $\mathscr{L}$ (*Definición 17*) es evaluado de manera similar que en el algoritmo DCFI pero con las estructuras dinámicas (tuplas) de S-FI. En general este parte del algoritmo es una combinación de la parte ω_2 de S-FI y σ_2 de DCFI.

Cuadro 6.2: Algoritmo HSDC heterogéneo (η_0).

```
   (η0) Inicialización del nodo n_i.
1  procedure Init_Nodo(isSyn, NumProcesos, idNodo, N)
2    k : 1...NumProcesos; //NumProcesos es el número total de procesos en el nodo.
3    ∀k do
4      nodo_i[k] := Init_Proceso_i(idNodo, isSyn, N);
5    enddo
6    if (isSyn) then
7      //Sincronizamos los procesos del nodo y generamos un checkpoint por cada
8      //proceso en el nodo. Este es un checkpoint inicial para todos los procesos.
9      nodo_i[1].taken_checkpoint(false);
10   endif
11 endprocedure
```

6.4. Conclusiones

En este capítulo mostramos nuestro algoritmo HSDC para ambientes heterogéneos con modelos de ejecución síncrono y asíncrono. Este algoritmo está formado, en su gran mayoría, por los algoritmos S-FI y DCFI desarrollados en los capítulos 4 y 5, respectivamente. En este sentido, las ventajas y características descritas anteriormente de los algoritmos S-FI y DCFI son heredadas por nuestro algoritmo HSDC, por lo que HSDC es un algoritmo de comunicación inducida (como S-FI y CDFI) y todos los *checkpoint* generados por este son útiles.

La relación ZIDR que definimos en este capítulo, relaciona a dos *checkpoints* con un *z-path* por medio de la relación IDR. A su vez, la relación ZIDR también nos proporciona un mecanismo para agrupar *checkpoints* sin un *z-path*. Por lo que podemos agrupar *checkpoints* en cada nodo (por medio de este mecanismo) y llegar a formar *snapshot* globales consistentes (SGC) del sistema heterogéneo.

Cuadro 6.3: Algoritmo HSDC heterogéneo (ρ_0 y ρ_1).

(ρ_0) *Inicialización del proceso* p_i.

```
12 procedure Init_Proceso_i(idNodo, is_Syn, N)
13   k, l : 1 ... N; //N es el número total de procesos en el sistema.
14   ∀k do lc_ckpt_i[k] := 0; enddo
15   ∀k, l do  T_i[k, l] := true; enddo
16   idr_ckpt_i[i] := true;
17   greater_i[i] := false;
18   lc_i := 0;
19   id_nodo := idNodo; //id_nodo - es el identificador de nodo al que pertenece el proceso.
20   is_PS := is_Syn; //is_PS - define si el proceso pertenece a un no síncrono o asíncrono.
21   //p_i toma su primer checkpoint. Este checkpoint no puede retrasarse.
22   if(¬is_PS)then
23     taken_checkpoint(false);
24   endif
25 endprocedure
```

(ρ_1) *Cuando* p_i *envía un mensaje* m *a* p_j.

```
26 if(is_PS ∧ id_nodo = getidNodo(j)) //getidNodo(j) obtiene el identificador del nodo del proceso j.
27 //Comunicar a p_i y p_j por memoria compartida
28 ⋮
29 else //Comunicar a p_i y p_j por paso de parámetros
30   if(delay_ckpt_i ∧ idr_ckpt_before_A_i[j]  ∧ num_delay_ckpt_i < 3) then
31         if(¬rec) then
32           m := getTuplas(T_i, lc_ckpt_before_i, idr_ckpt_before_i, greater_before_i);
33         else
34           m := getTuplas(T_i, lc_ckpt_before_i, idr_ckpt_before_i, greater_before_i);
35           ∀k do
36             idr_ckpt_before_A_i[k] := idr_ckpt_before_i[k];
37             sent_to[k] := false;
38           enddo
39           ∀k ≠ i do idr_ckpt[k] := false;  greater_i[k] := true; enddo
40           rec := false;
41         endif
42         num_delay_ckpt_i := num_delay_ckpt_i + 1;
43   else
44         delay_ckpt_i := false;
45         m := getTuplas(T_i, lc_ckpt_i, idr_ckpt_i, greater_i);
46         sent_to_i[j] := true;
47   endif
48 send(m, Data) to p_j;
49 endif
```

La importancia, en este sentido, de la relación ZIDR radica en su flexibilidad para formar conjuntos de *checkpoints* que pueden formar SGC al interior de cada nodo. De tal forma que, nuestro algoritmo heterogéneo, asegura que cada *checkpoint* generado por un

proceso en el sistema es parte de algún SGC y la relación ZIDR, permite a cada nodo, agrupar *checkpoints* para formar varios SGC al interior del nodo y poder conformar un SGC del sistema heterogéneo.

Por ultimo, la simulación del algoritmo HSDC no se realizó debido a que este es una fusión de los algoritmos S-FI y DCFI, ambos analizados y simulados con la herramienta ChkSim en los capítulos 4 y 5, respectivamente. Además, dado que los algoritmos S-FI y DCFI son complementarios (ambos atacan problemas diferentes de los algoritmos CIC) consideramos que estos seguirán teniendo en mismo comportamiento (ventajas) cuando se fusionan en HSDC. Por tal motivo, la realización de la simulación de HSDC con la herramienta ChkSim (limitada a ejecuciones asíncronas) nos arrojaría resultados muy similares a los obtenidos en S-FI y DCFI. Por todo esto, y debido a la naturaleza heterogénea de HSDC, necesitamos utilizar una herramienta más sofisticada con soporte para los modelos de ejecución síncrono y asíncrono. Lamentablemente, desconocemos (hasta este momento) de la existencia de una herramienta como estas características.

Cuadro 6.4: Algoritmo HSDC heterogéneo (ρ_2).

(ρ_2) *When* p_i *receives the message* $m := (\psi, Data)$ *from* p_j.

50 **if** $(delay\_ckpt_i)$ **then**
51 **if** $(\exists z \in \psi, z.id = i : lc\_ckpt\_before_i[i] = z.lc\_ckpt \ \wedge \ \neg z.idr\_ckpt)$ **then**
52 $delay\_ckpt_i := false$;
53 **endif**
54 **endif**
55 $max\_lc\_ckpt := \mathbf{max}(\psi)$;
56 **if** $\big[\big(\exists k : sent\_to_i[k] \ \wedge \ (\exists y \in \psi, \ y.id = k : y.greater \ \vee \ \nexists y \in \psi, \ y.id = k)\big) \wedge$
57 $max\_lc\_ckpt > lc_i\big] \vee \big[\exists z \in \psi, z.id = i : lc\_ckpt_i[i] = z.lc\_ckpt \wedge \neg z.idr\_ckpt\big]$ **then**
58 take_checkpoint(false);
59 **endif**
60 **case**
61 $max\_lc\_ckpt > lc_i \rightarrow$
62 $lc_i := max\_lc\_ckpt$;
63 $delay\_ckpt_i := false$;
64 $\forall k \neq i$ **do** $greater_i[k] := true$; **enddo**
65 $\forall \ell \in \psi, \ell.id \neq i$ **do** $greater_i[\ell.id] := \ell.greater$; **enddo**
66 $max\_lc\_ckpt = cl_i \rightarrow$
67 $delay\_ckpt_i := false$;
68 $\forall \ell \in \psi$ **do** $greater_i[\ell.id] := greater_i[\ell.id] \wedge \ell.greater$; **enddo**
69 $max\_lc\_ckpt < lc_i \rightarrow$
70 **if**$(max\_lc\_ckpt = lc\_before_i)$ **then**
71 $\forall \ell \in \psi$ **do** $greater\_before_i[\ell.id] := greater\_before_i[\ell.id] \wedge \ell.greater$; **enddo**
72 **endif**
73 **endcase**
74 $\forall w \in \psi$ **do**
75 **case**
76 $w.lc\_ckpt > lc\_ckpt_i[w.id] \rightarrow$
77 $lc\_ckpt_i[w.id] := w.lc\_ckpt$;
78 $idr\_ckpt_i[w.id] := w.idr\_ckpt$;
79 **if**$(delay\_ckpt_i)$ **then**
80 $lc\_ckpt\_before_i[w.id] := w.lc\_ckpt$;
81 $idr\_ckpt\_before_i[w.id] := w.idr\_ckpt$;
82 **endif**
83 $\forall \ l \neq i$ **do** $T_i[l, w.id] := false$; **enddo**
84 **if** $(max\_lc\_ckpt \neq w.lc\_ckpt) \vee (lc_i > w.lc\_ckpt)$ **then** $T_i[j, w.id] := true$; **endif**
85 $w.cl\_ckpt = cl\_ckpt_i[w.id] \rightarrow$
86 $idr\_ckpt_i[w.id] := idr\_ckpt_i[w.id] \wedge w.idr\_ckpt$;
87 **if**$(delay\_ckpt_i)$ **then**
88 $idr\_ckpt\_before_i[w.id] := idr\_ckpt\_before_i[w.id] \vee w.idr\_ckpt$;
89 **endif**
90 **if** $(max\_lc\_ckpt \neq w.lc\_ckpt) \vee (lc_i > w.lc\_ckpt)$ **then** $T_i[j, w.id] := true$; **endif**
91 $w.cl\_ckpt < cl\_ckpt_i[w.id] \rightarrow$ **skip**
92 **endcase**
93 **enddo**
94 $rec := true$;
95 **delivery**(m);

Cuadro 6.5: Procedimientos y funciones usados en el algoritmo HSDC.

```
// Cuando p_i toma un checkpoint local o forzado.
96 procedure taken_checkpoint(boolean type)
97   if (is_nodo_sincrono) then
98     // Sincronizar a todos los procesos del nodo y tomar un checkpoint
99     // por proceso de manera simultanea.
100  endif
101  delay_ckpt_i := type;
102  if(delay_ckpt_i) then
103    rec_i := false;
104    num_delay_ckpt_i := 0;
105    lc_before_i := lc_i;
106    ∀k do
107      lc_ckpt_before_i[k] := lc_ckpt_i[k];
108      idr_ckpt_before_i[k] := idr_ckpt_i[k];
109      greater_before_i[k] := greater_i[k]
110      idr_ckpt_before_A[k] := idr_ckpt_i[k];
111    enddo
112  endif
113  ∀k do sent_to_i[k] := false; enddo
114  ∀ k ≠ i do
115    idr_ckpt_i[k] := false;
116    greater_i[k] := true;
117    T_i[k, i] := false;
118  enddo
119  lc_i := lc_i + 1;
120  lc_ckpt_i[i] := lc_i;
121 endprocedure
```

```
// Función getTuplas() construye tuplas a partir de estructuras.
122 function getTuplas(T[][], lc_ckpt[], idr_ckpt[], greater[])
123 ψ ← ∅;
124 ∀k do
125    if [(¬T[j, k] ∨ ¬idr_ckpt[k]) ∧ (lc_ckpt[k] > 0)] then
126       ψ ← ψ ∪ (k, lc_ckpt[k], idr_ckpt[k], greater[k]);
127    endif
128 enddo
129 s := 32; //s es el #-bits para representar a reloj lógico (lc_ckpt).
130 if size(ψ) > (n)(s + 2)/(2s + 2) then //size(ψ) regresa la cardinalidad de ψ.
131   ψ ← ∅;
132   ∀k do ψ ← ψ ∪ (−, lc_ckpt[k], idr_ckpt[k], greater[k]); enddo
133 endif
134 return(ψ);
135 endfunction
```

```
136 function max(α) // Obtiene el máximo reloj lógico en α.
137   max := 0;
138   ∀x ∈ α do if x.lc_ckpt > max then max := x.lc_ckpt; endif enddo
139   return(max); endfunction
```

Capítulo 7

Conclusiones y trabajos a futuro

7.1. Conclusiones

En esta tesis analizamos el problema de *checkpointing* en sistemas heterogéneos y propusimos un algoritmo de *checkpointing* de comunicación inducida (CIC) para resolver el problema. Nuestro algoritmo HSDC tiene la ventaja de un *overhead* de mensajes pequeño y dinámico, lo cual hace a nuestro algoritmo HSDC escalable. Además, el número de *checkpoints* forzados que genera es menor en comparación con los algoritmos FI [22] y FINE [33] (con modelos asíncronos).

En esta investigación desarrollamos un algoritmo de la clase CIC porque estos no inhiben el cómputo del sistema, eliminan el efecto dominó, los *checkpoints* son generados de manera independiente (asíncronamente) por cada proceso y todos estos *checkpoints* eventualmente pueden formar un *snapshot* global consistente, y porque estos algoritmos no transmiten mensajes adicionales de control. La no adición de mensajes de control, significa que el patrón de intercambio de mensajes entre procesos está determinado por el cómputo del sistema y no por el algoritmo de *checkpointing*, lo cual tiene la ventaja de que el cómputo de la aplicación determina la interrelación entre procesos y el número de mensajes intercambiados.

Para trasmitir la información de control, los algoritmos CIC adicionan información en los mensajes (*piggyback*) que envía un proceso. A pesar de que este *overhead* es bajo, es una de las desventajas de estos algoritmos junto con el número total de *checkpoints* generados (almacenamiento).

Para disminuir el *overhead* de los mensajes en nuestro algoritmo, utilizamos la relación IDR [38] para caracterizar las relaciones entre *checkpoints* del sistema, lo que nos

permitió tener una representación compacta de las relaciones causales entre *checkpoints*. Esta representación compacta la rastreamos, en el sistema, por medio de una adaptación (a nuestro problema) del mecanismo de eventos relevantes (en nuestro caso *checkpoints*) introducido por Anceaume et al. [4]. Finalmente, la adaptación de rastreo en *checkpoints* nos proporcionó los medios para disminuir el *overhead* de mensajes y evaluar la condición de *checkpoints* forzados de nuestro algoritmo.

El algoritmo S-FI presentado en el capítulo 4 fue desarrollado con el objetivo de disminuir el *overhead* de mensajes de nuestro algoritmo HSDC. S-FI es un algoritmo en línea y captura todas las nociones e ideas descritas anteriormente. El algoritmo fue simulado y comparado con los algoritmos FI [22] y FINE [33], dos algoritmos CIC importantes en la literatura. Los resultados de la simulación muestran que el *overhead* en FI y FINE presenta un crecimiento lineal constante respecto al número de procesos. El *overhead* para S-FI es dinámico y presenta un crecimiento por debajo de lo lineal. Esto se debe a que el *overhead* en S-FI no es directamente proporcional al número de procesos, depende de la densidad de mensajes y la relación IDR entre *checkpoints*. Por lo tanto, el algoritmo S-FI soporta un número mayor de procesos que FI y FINE. Las verificaciones formales que presentamos en el apéndice validan nuestros resultados junto con nuestra simulación. Las conclusiones de la sección 4.6 describen más a detalle esta parte de nuestra investigación.

El algoritmo S-FI fue publicado en [9]. En esta publicación se presenta gran parte del materia del capítulo 4 junto con las demostraciones de los teoremas 2 y 3 presentadas en el apéndice A.1 y A.1, respectivamente.

Por otra parte, para atacar el problema de la cantidad *checkpoints* forzados y disminuirlos, introducimos la noción de *retraso de checkpoints*. Este enfoque detecta las condiciones cuando se forma una clase particular de *z-cycle*, que nosotros llamamos *z-cycle* rastreable. La detección y eliminación de un *z-cycle* rastreable se realiza durante la ejecución del cómputo del sistema y sin generar ningún *checkpoint* forzado. Para detectar esta clase de patrón de *z-cycle* definimos las *Condiciones Seguras para el Retraso de Checkpoint* (CSRC), las cuales nos permiten detectar las condiciones cuando tenemos un *z-cycle* rastreable y en consecuencia, eliminar este *z-cycle* con el enfoque de retraso de *checkpoint*. El algoritmo DCFI presentado en el capítulo 5 implementa el enfoque de retraso de *checkpoints*. Los resultados de la simulación del algoritmo DCFI junto con los algoritmos FI y FINE, muestran que DCFI genera en promedio un 3 % menos de *checkpoints* forzados que FI y en promedio un 1.5 % menos que FINE.

El algoritmo DCFI fue publica en [10]. El material presentado en el capítulo 5 dio forma

al material publicado, entre este material tenemos la definición de *z-cycle* rastreable, el enfoque de retraso de *checkpoints*, las condiciones seguras para el retraso de *checkpoints*, y finamente, la especificación del algoritmo DCFI junto con la simulación y análisis de este.

La fusión de los algoritmos S-FI y DCFI forma, en gran parte, a nuestro algoritmo heterogéneo. La parte síncrona del algoritmo no la desarrollamos, en su lugar, nosotros optamos por algún algoritmo de *checkpointing* con modelo de ejecución síncrona que pueda sincronizar los procesos de un nodo y generar un *snapshot* global (un conjunto de *checkpoints*, uno por cada proceso) al interior del nodo. Utilizamos esta solución debido a que el problema de generar un *snapshot* global consistente al interior de un nodo con modelo de ejecución síncrona es relativamente simple. Pero, para el problema de generar un *snapshot* global consistente del sistema heterogéneo no lo es, porque las dependencias inter-nodo establecen relaciones causales entre *checkpoints* de procesos en diferentes nodos, y a su vez, estas relaciones causales pueden generar *z-paths* entre *checkpoints*. Por su parte, las dependencias inter-proceso en nodos con modelos de ejecución asíncrono también presentan las mismas características.

En este sentido, nosotros definimos la relación ZIDR para caracterizar la noción de *z-path*. Por medio de esta relación construimos un grafo de *checkpoints* que nos permite agrupar *checkpoints* en tiempos lógicos, de tal forma, que el conjunto de *checkpoints* de un tiempo lógico puede formar o complementarse con *checkpoints* de los tiempos lógicos adyacentes para crear un *snapshot* global consistente al interior de cada nodo. De esta forma, un *snapshot* global consistente del sistema heterogéneo puede construirse a partir de los conjuntos de *checkpoints* que cada nodo pueda agrupar.

En conclusión, nuestro algoritmo de *checkpointing* asegura que todos los *checkpoints* generados son útiles; y la agrupación flexible de *checkpoints* en cada nodo, presentada para ambientes heterogéneos, permite que cada nodo genere conjuntos de *checkpoints* que formarán un *snapshot* global consistente del sistema heterogéneo.

7.2. Trabajo a futuro

7.2.1. Algoritmo de *checkpointing* híbrido

En el algoritmo S-FI (Capítulo 4, Sección 4.2 y 4.2.1), la propagación de la información de control entre procesos, permite a un proceso (en un tiempo determinado) deducir cierta información. En particular, por medio del arreglo booleano $greater_i[]$, el proceso p_i puede determinar si el proceso p_k (valor en $greater_i[k]$) conoce el valor del reloj lógico más grande visto por el p_i. Por otra parte, el arreglo booleano de dependencias inmediatas (IDR) entre *checkpoints* $idr\_ckpt_i[]$ establece subconjuntos de *checkpoints* que son parte de un *snapshot* global consistente. Esta característica puede ser utilizada para generar un algoritmo de *checkpointing* híbrido (coordinado y comunicación inducida, ver Sección 3.1.1).

El algoritmo híbrido funcionaría de la siguiente forma:

- Utilizaría un predicado que definiría cuándo el algoritmo se comportaría como un algoritmo coordinado o comunicación-inducida.
- El predicado estaría definido en relación al porcentaje de *checkpoints* con IDR en un proceso. Por ejemplo, si un proceso detecta que tiene un 90 % de *checkpoints* relacionados por medio de la relación IDR, entonces sólo necesita un 10 % de *checkpoints* para formar un *snapshot* global consistente.
- Durante el cómputo algunos procesos ejecutarían el algoritmo coordinado mientras que otros el algoritmo de comunicación inducida.
- Cuando un proceso tomara el rol de un algoritmo coordinado, este proceso se comportaría como un iniciador de un *snapshot* global consistente (SGC) de los algoritmos coordinados, excepto que sólo coordinaría un SGC con aquellos procesos sin *checkpoints* con IDR al proceso. El proceso bloquearía su cómputo, solicitaría a los procesos sin *checkpoints* con IDR que tomaran un *checkpoint* forzado antes de la entrega de la solicitud, e inmediatamente después enviarán un mensaje de notificación.
- Finalmente, el proceso que realiza la solicitud, determinaría si fue posible establecer el *snapshot* global consistente o no.

Las principales ventajas de este algoritmo híbrido serían: disminución del *overhead* de almacenamiento, al generar *checkpoints* que forman un SGC; y la coordinación de una cantidad menor de procesos que un algoritmo coordinado convencional.

7.2.2. Aplicación de la estrategia *lazy indexing*

La estrategia de indexado lento o perezoso (*lazy indexing*) fue introducido por Vieira et al. [52]. Esta estrategia define el incremento del reloj lógico de un proceso. La forma clásica de incremento, cada vez que un proceso toma un *checkpoint*, es monotónico, pero en la estrategia de indexado lento el incremento no siempre se genera. Particularmente, si un proceso p_i recibe mensajes con relojes lógicos menores que el suyo, p_i puede deducir que el número de secuencia de los relojes lógicos tiene un incremento estricto, por lo que no necesita incrementar su reloj lógico en el siguiente *checkpoint* que tome.

La estrategia de indexado lento ha sido aplicada a diversos algoritmos de *checkpointing* (incluido FI and FINE) y los resultados muestran mejoras en la reducción del número de *checkpoints* forzados. En este sentido, proponemos como trabajo a futura la aplicación de la estrategia *lazy indexing* a nuestros algoritmos DCFI y HSDC.

7.2.3. Algoritmo *rollback recovery* (recuperación hacia atrás)

Los algoritmos de *checkpointing* necesitan de un algoritmo *rollback recovery* para proporcionar un sistema tolerante a fallas. En nuestro caso, únicamente desarrollamos el algoritmo de *checkpointing*. Una forma simple de generar el algoritmo de *rollback recovery* para nuestro algoritmo de *checkpointig* es agrupar los *checkpoints* con el mismo valor de reloj lógico de cada proceso participante en el cómputo. Sin embargo, debido a que el reloj lógico de cada proceso no tiene un crecimiento monotónico (ver, Apéndice A1), no siempre tendremos *checkpoints* con el mismo valor de reloj lógico en un SGC.

Por otra parte, la relación ZIDR y la agrupación de *checkpoints* en tiempos lógicos (en cada nodo) podría ser una herramienta o mecanismo fiable y seguro para realizar un algoritmo de *rollback recovery* para nuestro algoritmo de *checkpointing*. En particular, la agrupación de *checkpoints* en tiempos lógicos, permite y provee flexibilidad a cada nodo, para formar *snapshot* globales que sean parte de un *snapshot* global consistente del sistema.

Bibliografía

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM. ISBN 1-58113-839-3.

[2] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8.

[3] L. Alvisi, E. Elnozahy, S. Rao, S.A. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 242–249, 1999. doi: 10.1109/FTCS.1999.781058.

[4] Emmanuelle Anceaume, Jean-Michel Hélary, and Michel Raynal. Tracking immediate predecessors in distributed computations. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 210–219, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7.

[5] Roberto Baldoni, Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Consistent state restoration in shared memory systems. In *Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference (APDC '97)*, APDC '97, pages 330–337, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7876-3.

[6] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Net-*

working, Storage and Analysis, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8.

[7] B. Bhargava and S.R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. *Proc. Seventh IEEE Symp. Reliable Distributed Systems*, pages 3–12, 1988.

[8] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, January 2010. ISSN 1058-9244.

[9] Alberto Calixto Simón, Saul E. Pomares Hernández, Jose R. Perez Cruz, Pilar Gomez Gil, and Khalil Drira. A scalable communication-induced checkpointing algoritm for distributed systems. *IEICE Transactions on Information and Systems*, E96-D(4): 886–896, April 2013.

[10] Alberto Calixto Simón, Saul E. Hernández Pomares, and Jose R. Perez Cruz. A delayed checkpoint approach for communication-induced checkpointing in autonomic computing. In *Proceedings of the 22th IEEE WETICE conference 2013, AROSA Track*, WETICE 2013, 2013.

[11] Guohong Cao and Mukesh Singhal. On coordinated checkpointing in distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 9(12):1213–1225, December 1998. ISSN 1045-9219.

[12] Guohong Cao and Mukesh Singhal. Mutable checkpoints: A new checkpointing approach for mobile computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(2): 157–172, February 2001. ISSN 1045-9219.

[13] Jiannong Cao, Yifeng Chen, Kang Zhang, and Yanxiang He. Checkpointing in hybrid distributed systems. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:136, 2004. ISSN 1087-4089.

[14] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Compututer Systems*, 3(1):63–75, February 1985. ISSN 0734-2071.

[15] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, February 1996. ISSN 0178-2770.

[16] George Coulouris, Tim Kindberg, and Jean Dollimore. *Distributed Systems: Concepts and Design*. Addison Wesley, 4 edition, May 2005.

[17] Robert Cypher and Eric Leu. The semantics of blocking and nonblocking send and receive primitives. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 729–735, Washington, DC, USA, 1994. IEEE Computer Society. ISBN 0-8186-5602-6.

[18] Jack J. Dongarra and Bronis R. De Supinski. Special issue on multiphysics simulations: Challenges and opportunities. *The International Journal of High Performance Computing Applications*, 27(1), February 2013.

[19] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, September 2002. ISSN 0360-0300.

[20] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *In Proceedings of 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[21] SunilKumar Gupta and Parveen Kumar. Review of some checkpointing algorithms for distributed and mobile systems. In *Advances in Network Security and Applications*, volume 196 of *Communications in Computer and Information Science*, pages 167–177. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22539-0.

[22] J.-M. Hélary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. *Distributed Computing*, 13(1):29–43, January 2000. ISSN 0178-2770.

[23] J. M. Hélary, A. Mostefaoui, and M. Raynal. Interval consistency of asynchronous distributed computations. *Journal of Computer and System Sciences*, 64(2):329–349, March 2002. ISSN 0022-0000.

[24] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000. ISSN 0256-2499.

[25] R. Kalla, B. Sinharoy, and Joel M. Tendler. Ibm power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, 2004. ISSN 0272-1732.

[26] Ashfaq A. Khokhar, Viktor K. Prasanna, Muhammad E. Shaaban, and Cho-Li Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, June 1993. ISSN 0018-9162.

[27] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March 2005. ISSN 0272-1732.

[28] József Kovács, Peter Kacsuk, Radoslaw Januszewski, and Gracjan Jankowski. Application and middleware transparent checkpointing with tckpt on clustergrids. *Future Generation Computer Systems*, 26(3):498–503, March 2010. ISSN 0167-739X.

[29] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Hardcover. Cambridge University Press, 1 edition, May 19 2008.

[30] Surender Kumar, R.K. Chauhan, and Parveen Kumar. Design and performance analysis of coordinated checkpointing algorithms for distributed mobile systems. *International Journal of Distributed and Parallel Systems (IJDPS)*, 1(1):61–80, September 2010.

[31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. ISSN 0001-0782.

[32] John C. Linford and Adrian Sandu. Scalable heterogeneous parallelism for atmospheric modeling and simulation. *The Journal of Supercomputing*, 56(3):300–327, June 2011. ISSN 0920-8542.

[33] Yi Luo and D. Manivannan. FINE: A fully informed and efficient communication-induced checkpointing protocol for distributed systems. *Journal Parallel and Distributed Computing*, 69(2):153–167, February 2009. ISSN 0743-7315.

[34] Nancy A. Lynch. *Distributed Algoritms*. Morgan Kaufmann, 1 edition, March 1996.

[35] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, July 1999. ISSN 1045-9219.

[36] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

[37] Robert H. B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995. ISSN 1045-9219.

[38] Saul E. Pomares Hernández, Jean Fanchon, and Khalil Drira. The immediate dependency relation: an optimal way to ensure causal group communication. In *Annual review of scalable computing, Editions world scientific, series on scalable computing*, pages 61–79, 2004.

[39] Francesco Quaglia, Roberto Baldoni, and Bruno Ciciani. On the no-z-cycle property in distributed executions. *Journal of Computer and System Sciences*, 61(3):400–427, December 2000. ISSN 0022-0000.

[40] Srivatsan Raman, Robert Vernon, James Thompson, Michael Tyka, Ruslan Sadreyev, Jimin Pei, David Kim, Elizabeth Kellogg, Frank DiMaio, Oliver Lange, Lisa Kinch, Will Sheffler, Bong-Hyun Kim, Rhiju Das, Nick V. Grishin, and David Baker. Structure prediction for CASP8 with all-atom refinement using Rosetta. *Proteins*, 77:89–99, 2009.

[41] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogeneous archtitectures. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, FTCS '97, pages 58–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7831-3.

[42] B. Randell. System structure for software fault tolerance. *SIGPLAN Notices - International Conference on Reliable Software*, 10(6):437–449, April 1975. ISSN 0362-1340.

[43] Gabriel Rodríguez, María J. Martín, Patricia González, and Juan Touriño. Controller/precompiler for portable checkpointing. *IEICE - Transactions on Information and Systems*, E89-D(2):408–417, February 2006. ISSN 0916-8532.

[44] D.L. Russell. State restoration in systems of communicating processes. *Software Engineering, IEEE Transactions on*, SE-6(2):183–194, 1980. ISSN 0098-5589.

[45] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983. ISSN 0734-2071.

[46] Kuo-Feng Ssu, W. Kent Fuchs, and Hewijin C. Jiau. Process recovery in heterogeneous systems. *IEEE Transactions on Computers*, 52(2):126–138, February 2003. ISSN 0018-9340.

[47] Tongchit Tantikul and D. Manivannan. *A Communication-Induced Checkpointing and Asynchronous Recovery Algorithm for Multithreaded Distributed Systems*, pages 284–292. Springer Berlin eidelberg, 2005.

[48] Jichiang Tsai and Jenn-Wei Lin. On the fully-informed communication-induced checkpointing protocol. In *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, PRDC '05, pages 151–158, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2492-3.

[49] Yuichi Tsujita, Toshiyuki Imamura, Nobuhiro Yamagishi, and Hiroshi Takemiya. Flexible message passing interface for a heterogenous computing environment. In Minyi Guo and LaurenceTianruo Yang, editors, *New Horizons of Parallel and Distributed Computing*, pages 3–19. Springer US, 2005.

[50] K. Tsuruoka, A. Kaneko, and Y. Nishihara. Dynamic recovery schemes for distributed process. *Proc. IEEE Second Symp. Reliability in Distributed Software and Database Systems*, pages 124–130, 1981.

[51] Gustavo M. D. Vieira and Luiz E. Buzato. Chksim: A distributed checkpointing simulator. Technical report, Institute of Computing, University of Campinas, Campinas, Brasil, December 2005. URL `http://www.ic.unicamp.br/~gdvieira/chksim/`.

[52] Gustavo M. D. Vieira, I. C. Garcia, and L.E. Buzato. Systematic analysis of index-based checkpointing algorithms using simulation. In *Proc. of IX Brazilian Symp. on Fault-Tolerant Comput.*, 2001.

[53] John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *Proceedings of the Third international conference on Distributed Computing and Internet Technology*, ICDCIT'06, pages 221–234, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-68379-8, 978-3-540-68379-7.

[54] Yi-Min Wang, Andy Lowry, and W. Kent Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, 50(4):223–230, May 1994. ISSN 0020-0190.

Apéndice A

Demostraciones

A.1. Demostración del Teorema 2

Teorema 2. La condición $\mathscr{D}$ es equivalente a la condición $\mathscr{C}2''$.

Demostración. Dividimos la demostración en dos partes. En la primera parte demostramos que FI_b es equivalente a SFI_b; y en la segunda parte, demostramos que FI_c es equivalente a SFI_c. No demostramos que FI_a es equivalente a SFI_a porque ambas administran y modifican a los arreglos $sen\_to_i[]$ y $greater_i[]$ en la misma forma.

Parte I. Para demostrar que FI_b es equivalente a SFI_b, nosotros formulamos y probamos el siguiente Lema:

Lema 1. El x-ésimo valor del reloj lógico $lc^x_{i_{(FI)}}$ y $lc^x_{i_{(SFI)}}$ de un proceso p_i en FI_b y SFI_b, respectivamente, son iguales. En otras palabras:

$$\forall i \in P: \ lc^x_{i_{(FI)}} = lc^x_{i_{(SFI)}}$$

Prueba. Por inducción, tenemos:

- *Caso base* ($k = 2$). Al inicio, estas variables son inicializadas a 1. Para el segundo valor de $lc^x_{i_{(FI)}}$ y $lc^x_{i_{(SFI)}}$, tenemos dos casos: el primer caso, es cuando el proceso p_i toma un *checkpoint* y actualiza su lc_i; y el segundo, cuando p_i recibe un mensaje m y actualiza su lc_i con la información acarreada en m.

  - *p_i toma un checkpoint.* p_i actualiza su lc_i como sigue:

$$lc^2_{i_{(FI)}} := lc^1_{i_{(FI)}} + 1, \quad lc^2_{i_{(SFI)}} := lc^1_{i_{(SFI)}} + 1,$$
$$lc\_ckpt^2_i[i] := lc^2_{i_{(SFI)}} = 2$$

Por lo tanto, $lc^2_{i_{(FI)}} = lc^2_{i_{(SFI)}} = 2$.

- *p_i recibe m de p_j inmediatamente después de su primer checkpoint y $m.lc = 2$.* En este caso, en FI p_i actualiza $lc^2_{i_{(FI)}}$ en la siguiente forma:

$$if\ m.lc_{(FI)} > lc^1_{i_{(FI)}}\ then\ lc^2_{i_{(FI)}} := m.lc_{(FI)}$$

Por lo tanto, $lc^2_{i_{(FI)}}$ es actualizado con el reloj lógico más grande visto por p_i y p_j.

En S-FI, $lc^2_{i_{(SFI)}}$ es actualizado también con el reloj lógico más grande visto por p_i y p_j, con la diferencia que el reloj lógico más grande visto por p_j es extraído del vector $lc\_ckpt[]$, incluido en m. $lc^2_{i_{(SFI)}}$ y $lc\_ckpt_i[]$ son actualizados en la siguiente forma:

$$if\ \mathbf{max}(m.lc\_ckpt[]) > lc^1_{i_{(SFI)}}\ then$$
$$lc^2_{i_{(SFI)}} := \mathbf{max}(m.lc\_ckpt[])$$
$$\forall\ l \neq i : if\ m.lc\_ckpt[l] > lc\_ckpt_i[l]\ then$$
$$lc\_ckpt_i[l] := m.lc\_ckpt[l]$$

Por lo tanto, $lc^2_{i_{(FI)}} = lc^2_{i_{(SFI)}} = 2$, porque tanto en FI como en S-FI, cada proceso actualiza localmente su reloj lógico en la misma forma.

- *Paso Inductivo.* Suponemos ahora que el resultado se cumple para: $k > 2$, por consiguiente: $lc^k_{i_{(FI)}} = lc^k_{i_{(SFI)}}$
- *Hipótesis Inductiva.* Por demostrar que también se cumple para $k+1$. Esta parte de la prueba la dividimos en dos casos. El primer caso, es cuando el proceso p_i toma un *checkpoint* y actualiza su reloj lógico lc_i. El segundo, cuando p_i recibe un mensaje m y actualiza su reloj lógico lc_i con la información acarreada en m.

  - *p_i toma un checkpoint.* Por lo tanto, p_i actualiza su reloj lógico de la siguiente forma:

$$lc^{k+1}_{i_{(FI)}} := lc^k_{i_{(FI)}} + 1$$
$$lc^{k+1}_{i_{(SFI)}} := lc^k_{i_{(SFI)}} + 1$$
$$lc\_ckpt^{k+1}_i[i] := lc^{k+1}_{i_{(SFI)}}$$

Por lo tanto, $lc^{k+1}_{i_{(FI)}} = lc^{k+1}_{i_{(SFI)}}$.

- *p_i recibe un mensaje m de p_j.* Note que en el algoritmo FI, el $lc_{j_{(FI)}}$ $(j \neq i)$ incluido en el mensaje m $(m.lc_{(FI)})$ corresponde al reloj más grande visto por p_j. En este caso, $lc^{k+1}_{i_{(FI)}}$ es actualizado en la siguiente forma:

$$if\ m.lc_{(FI)} > lc^{k}_{i_{(FI)}}\ then\ lc^{k+1}_{i_{(FI)}} := m.lc_{(FI)}$$

Por lo tanto, $lc^{k+1}_{i_{(FI)}}$ es actualizado con el mayor reloj lógico visto por p_i y p_j.
En el algoritmo S-FI, $lc_{j_{(SFI)}}$ es también el mayor reloj lógico visto por p_j, pero en este caso, este es incluido en el vector $lc\_ckpt_j[]$ en m, $(lc_{j_{(SFI)}} \in m.lc\_ckpt[])$. El reloj lógico $lc^{k+1}_{i_{(SFI)}}$ y el vector $lc\_ckpt_i[]$ son actualizados en la siguiente forma:

$$\begin{aligned} &if\ \mathbf{max}(m.lc\_ckpt[]) > lc^{k}_{i_{(SFI)}}\ then \\ &\qquad lc^{k+1}_{i_{(SFI)}} := \mathbf{max}(m.lc\_ckpt[]) \\ &\forall\ l \neq i : if\ m.lc\_ckpt[l] > lc\_ckpt_i[l]\ then \\ &\qquad lc\_ckpt_i[l] := m.lc\_ckpt[l] \end{aligned}$$

Por lo tanto, $lc^{k+1}_{i_{(SFI)}}$ es actualizado también con el mayor $lc_{(SFI)}$ visto por p_i y p_j; además el vector $lc\_ckpt_i[]$ es actualizado también con el mayor $lc_{(SFI)}$.

Por lo tanto, concluimos que: $lc^{k+1}_{i_{(FI)}} = lc^{k+1}_{i_{(SFI)}}$. $\square_{Lema\ 1}$

Parte II. Ahora demostraremos que FI_c es equivalente a SFI_c. Dividimos la demostración en dos partes. En la primera, probamos que $lc\_ckpt_i[i]$ tiene un similar comportamiento a $ckpt_i[i]$ durante un intervalo de *checkpoint*. En la segunda parte, mostramos que la identificación de *checkpoints* con relación de dependencia inmediata, también nos permite detectar el mismo patrón que el arreglo $taken_i[]$ del algoritmo FI.

- *Parte II.A.* Como el reloj lógico $ckpt_i[i]$ tiene un comportamiento creciente estricto, nosotros probamos que el reloj lógico $lc\_ckpt_i[i]$ también tiene esta propiedad. Con este objetivo, formulamos y demostramos el siguiente Lema:

  Lema 2. El comportamiento del reloj lógico $lc\_ckpt_i[i]$ en un proceso p_i tiene un crecimiento estricto, como:

$$\forall i \in P :\ lc\_ckpt^1_i[i] < \ldots < lc\_ckpt^{x-1}_i[i] < lc\_ckpt^x_i[i],$$

  donde x representa al x-ésimo *checkpoint* de p_i.

Demostración. Esta parte, la demostramos por prueba directa. Note que $lc_{i_{(FI)}}$ tiene un comportamiento creciente estricto [22]. Del Lema 1, tenemos que: $lc^x_{i_{(SFI)}} = lc^x_{i_{(FI)}}$; por lo tanto, el reloj lógico $lc_{i_{(SFI)}}$ tiene el mismo comportamiento que $lc_{i_{(FI)}}$ (creciente estricto). Puesto que $lc\_ckpt^x_i[i]$ toma el valor de $lc^x_{i_{(SFI)}}$, cada vez que un proceso p_i toma un *checkpoint*; tenemos entonces, que el reloj lógico $lc\_ckpt_i[i]$ también tiene un comportamiento creciente estricto. $\square_{Lema\ 2}$

Ahora, por el Lema 2 y con el conocimiento que el reloj lógico $lc\_ckpt_i[i]$ es actualizado únicamente cuando p_i toma un *checkpoint* local, podemos enunciar que el reloj lógico $lc\_ckpt^k_i[i]$ es constante durante un intervalo de *checkpoint*.

- *Parte II.B.* En el algoritmo FI, la estructura $taken_i[j] = true$ en p_i, indica que en el *z-path* causal que se forma del último *checkpoint* C^y_j, conocido por p_i, a su siguiente *checkpoint* C^{x+1}_i, hay por lo menos un *checkpoint*. Específicamente, estamos interesados, cuando $taken_i[j] = true$ y $j = i$. En este caso, el *z-path* causal de C^x_i a C^{x+1}_i incluye un *checkpoint* (por ejemplo, C^z_k del proceso p_k). Nosotros formulamos y demostramos el siguiente Lema, con el objetivo de mostrar, que el algoritmo S-FI detecta este patrón por medio de identificar las relaciones de dependencias inmediatas entre *checkpoints*.

Lema 3. Para una mensaje m enviado por p_j y recibido por p_i, $i \neq j$

$$\textbf{if } (m.idr\_ckpt[i] = false) \textbf{ then } \exists C^z_k \in E_i, k \neq i : C^x_i \rightarrow C^z_k \rightarrow C^{x+1}_i$$

Para el Lema 3, damos un bosquejo de la demostración. De la definición 3, tenemos que dos *checkpoints* C^x_i y C^{x+1}_i tiene una relación IDR, si $\nexists C^z_k : C^x_i \rightarrow C^z_k \rightarrow C^{x+1}_i$.

Durante el intercambio de mensajes entre C^x_i y C^{x+1}_i, en S-FI, el valor de $idr\_ckpt_i[i] = true$ se propaga entre cada par de consecutivos mensajes si y solo si no hay un *checkpoint* C^z_k. Esto se logra, debido a que en la recepción de un mensaje, el vector $idr\_ckpt_i[]$ se actualiza con la última información IDR (ver actualización de $idr\_ckpt_i[]$ al recibir un mensaje, página 40). De otra manera, cuando un *checkpoint* local es generado por algún proceso p_k, el historial IDR de p_k con respecto a p_i se reinicializa, $idr\_ckpt_k[i] = false$ (ver página 40).

Por lo tanto, $FI_c \equiv SFI_c$. $\square$

A.2. Demostración del Teorema 3

Teorema 3. La condición $\mathscr{D}'$ es equivalente a la condición $\mathscr{D}$.

Demostración. Dividimos la demostración en dos partes. Primero, demostramos que la condición $K3(m,k)$ implica $K(m,k)$, esto asegura el *rastreo de checkpoints* predecesores inmediatos, sin requerir que toda la información de control sea acarreada en cada mensaje. En segundo lugar, demostramos que $SFI_a \wedge SFI_b$ es equivalente a $SFI'_a \wedge SFI'_b$ y SFI_c es equivalente a SFI'_c.

Parte I. Para demostrar que $K3(m,k)$ implica $K(m,k)$ enunciamos y demostramos el siguiente teorema:

Teorema 5. Sea $K3(m,k) \equiv ((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k] = 1)) \vee (send(m).lc\_ckpt_i[k] = 0)$

Tenemos: $K3(m,k) \Rightarrow K(m,k)$.

donde la condición abstracta $K(m,k)$ fue definida por Anceaume et al. [4] de la siguiente forma:

$$\begin{aligned} K(m,k) \equiv\ & (send(m).VC_i[k] = 0) \vee \\ & (send(m).VC_i[k] < pred(receive(m)).VC_j[k]) \vee \\ & ((send(m).VC_i[k] = pred(receive(m)).VC_j[k]) \wedge (send(m).IP_i[k] = 1)) \end{aligned}$$

donde $VC_i[]$ es un vector de relojes lógicos y $IP_i[]$ es un arreglo booleano.

Al considerar lo siguiente:

- La información del vector $VC_i[]$ es actualizado de igual forma que el vector $ckpt_i[]$ (Sección 4.1, página 37) del algoritmo FI.
- El arreglo $IP_i[]$ es igual que el arreglo $idr\_ckpt_i[]$ (Sección 4.2, página 40) de nuestro algoritmo S-FI.
- El remplazo del vector $ckpt_i[]$ en S-FI por $lc\_ckpt_i[]$, sin alterar los resultados, como se demostró en el Teorema 2). Específicamente, en este Teorema demostramos que los relojes lógicos del vector $lc\_ckpt_i[]$ presentan un incremento estricto al igual que los relojes lógicos del vector $ckpt_i[]$ y en consecuencia, como los relojes lógicos del vector $VC_i[]$ (ver, Lema 1 y Lema 2).

Tomando en consideración estos comentarios, presentamos la demostración del Teorema 5 como sigue.

Demostración. Iniciamos mostrando que la matriz T_i proporciona un significado correcto a el conocimiento de p_i.

$$(send(m).T_i[j,k] = 1) \Rightarrow \quad ((send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]) \wedge (max(send(m).lc\_ckpt_i[]) > send(m).lc\_ckpt_i[k]))$$

Lema 4. Sea $\mathcal{IT}(e,j,k)$ la siguiente propiedad:

$$(e.T_i[j,k] = 1) \Rightarrow A0 \vee ((A1 \vee A2 \vee A3) \wedge B0),$$

donde :

$$\begin{aligned}
A0 &\equiv (j{=}i),\\
A1 &\equiv (j{=}k),\\
A2 &\equiv (e.lc_ckpt_i[k]{=}0),\\
A3 &\equiv \exists\ m'\ from\ p_j\ to\ p_i, \forall z \in m'.\psi, k = z.id:\ ((receive(m') \rightarrow e) \vee\\
&\quad (receive(m') = e))\ \wedge\ (send(m').lc_ckpt_j[k] = z.lc_ckpt = e.lc_ckpt_i[k]),\\
B0 &\equiv (max(e.lc_ckpt_i[]) > e.lc_ckpt_i[k]).
\end{aligned}$$

$\forall i, \forall e \in H_i, \forall j, \forall k : \mathcal{IT}(e,j,k)$ es válida.

Demostración. La demostración es por inducción en $\hat{E}$. Nosotros consideramos únicamente los eventos e tales que $e.T_i[j,k] = 1$. Cuando $e.T_i[j,k] = 0$, la propiedad $\mathcal{IT}(e,j,k)$ es trivialmente válida.

- *Caso base.* Sea e el primer evento de p_i. Tenemos que $e.T_i[j,k] = 1$ únicamente en los siguientes casos:

  - e es el primer *checkpoint* de p_i. De la actualización del vector $lc\_ckpt_i[]$ (ver, Sección 4.2) y, T0 y T1 (actualización de la matriz $T_i[][]$, Sección 4.2.1); tenemos que $max(e.lc\_ckpt_i[]) = (e.lc\_ckpt_i[i]) = 1$ y:
    - $j = i, \forall k : (T_i[j,k] = 1) \Rightarrow A0$.
    - $\forall j \neq i, \forall k \neq i : (T_i[j,k] = 1) \Rightarrow A2 \wedge B0$.
    - $\forall j \neq i, k = i : T_i[j,k] = 0$.
  - e es la recepción de un mensaje m enviado por p_j inmediatamente después que p_i realiza su primer *checkpoint*. De T2 (actualización de $T_i[][]$, Sección 4.2.1) y del manejo de $lc\_ckpt_i[]$, tenemos:

- $j = i, \forall k : (T_i[j,k] = 1) \Rightarrow A0$.
- $\forall x \neq i, \forall y, \forall z \in m.\psi, k = z.id : (e.T_i[x,y] = 1) \Rightarrow \big[(x = j) \wedge (y = k) \wedge (max(e.lc\_ckpt_i[]) > z.lc\_ckpt)\big] \vee \big[(y \neq i) \wedge (y \neq k) \wedge (lc\_ckpt_i[k] = 0)\big]$.
  La primera alternativa es válida; m satisface $A3$ y $B0$, $receive(m) = e \wedge send(m).lc\_ckpt_j[k] = z.lc\_ckpt = e.lc\_ckpt_i[k] < max(e.lc\_ckpt_i[])$. La segunda alternativa es también válida; $e.lc\_ckpt_i[k] = 0$ satisface $A2$ y $B0$.

De este modo, en cada caso $\mathcal{IT}(e,j,k)$ es válida.

- *Paso inductivo.* Sea $e \in H_i$. Nosotros suponemos que $\forall e' \in \{e' \mid e' \rightarrow e\}, \forall j, \forall k$: $\mathcal{IT}(e',j,k)$ es válida.

- *Hipótesis Inductiva.* Ahora, demostraremos que $\forall j, \forall k$, la propiedad $\mathcal{IT}(e,j,k)$ es válida. Procedemos por análisis de casos sobre el tipo de evento.

  - e es un *checkpoint*. En este caso, p_i reinicia la i-ésima columna de T_i (ver T1), $\forall \neq i : T_i[j,i] := 0$.
  - e es un evento de envío. No hay actualización en la matriz T_i (ver T1). Por lo tanto, $\mathcal{IT}(e,j,k)$ es válida.
  - e es la recepción de un mensaje m de p_j. p_i únicamente actualiza la fila j de T_i (ver T2); entonces: $x = j, \forall z \in m.\psi, z.id = k : (e.T_i[x,k] = 1) \Rightarrow ((x = j) \wedge (max(e.lc\_ckpt_i[]) > z.lc\_ckpt)$. De este modo, e satisface $A3$ y $B0$. $receive(m) = e \wedge send(m).lc\_ckpt_j[k] = z.lc\_ckpt = e.lc\_ckpt_i[k] < max(e.lc\_ckpt_i[])$.

  De este modo, en cada caso, $\mathcal{IT}(e,j,k)$ es válida. $\square_{Lema\ 4}$

Ahora, sea m un mensaje enviado por p_i a p_j ($e = send(m)$) y $send(m).T_i[j,k] = 1$. Del Lema 4, tenemos tres casos (note que $j \neq i$ y e no puede ser un evento de recepción):

- De $A1$, $j = k$. Así, de las propiedades de vector de reloj, tenemos: $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.

- De $A2$ y $send(m).lc\_ckpt_i[k] = 0$, tenemos que: $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.

- De $A3$, tenemos: $send(m').lc\_ckpt_j[k] = e.lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$. De ahí que: $send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k]$.

Por lo tanto, $(sent(m).T_i[j,k] = 1) \Rightarrow (send(m).lc\_ckpt_i[k] \leq pred(receive(m)).lc\_ckpt_j[k])$.

De esto, tenemos que:

$$\begin{aligned} K3(m,k) &\equiv ((send(m).T_i[j,k] = 1) \wedge (send(m).idr\_ckpt_i[k]{=}1)) \vee (send(m).lc\_ckpt_i[k] = 0) \\ &\Rightarrow ((send(m).lc\_ckpt_i[k] {\leq} pred(receive(m)).lc\_ckpt_j[k]) \wedge (send(m).idr\_ckpt_i[k]{=}1)) \\ &\quad \vee (send(m).lc\_ckpt_i[k]{=}0) \\ &\Rightarrow ((send(m).VC_i[k] {\leq} pred(receive(m)).VC_j[k]) \wedge (send(m).IP_i[k]{=}1)) \\ &\quad \vee (send(m).VC_i[k]{=}0) \\ &\equiv K(m,k) \end{aligned}$$

$\square_{Teorema\ 5}$

Parte II.a. Con el objetivo de demostrar que $SFI'_a \wedge SFI'_b$ es equivalente a $SFI_a \wedge SFI_b$, nosotros demostramos por prueba directa que: $SFI_a \wedge SFI_b \Rightarrow SFI'_a \wedge SFI'_b$; donde:

$$\begin{aligned} SFI_a &\equiv (\exists k : sent\_to_i[k] \wedge m.greater[k]) \\ SFI_b &\equiv (max(m.lc\_ckpt[]) > lc_i) \\ SFI'_a &\equiv (\exists k : sent\_to_i[k] \wedge ((\exists y \in m.\psi, y.id = k : \\ &\qquad y.greater) \vee (\nexists y \in m.\psi, y.id = k))\big) \\ SFI'_b &\equiv (max(m.\psi) > lc_i) \end{aligned}$$

Nótese que el valor de $sent\_to_i[k]$ es igual para $\mathscr{D}$ y $\mathscr{D}'$ (ver Secciones 4.2 y 4.2.1), y $max(m.lc\_ckpt[]) = max(lc\_ckpt_j[]) = max(m.\psi)$ (ver Sección 4.2 y Lema 4). Además, $max(lc\_ckpt_j[])$ esta siempre incluido en $m.\psi$ (ver Lema 4). Ahora, sea m enviado por p_j a p_i y $sent\_to_i[k] = true$. Tenemos dos casos para analizar:

- $\exists y \in m.\psi, y.id = k$. En este caso, $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$ se cumple; porque la información que se utilizan para evaluar ambas conjunciones es la misma.
- $\nexists y \in m.\psi, y.id = k$. En este caso, del Teorema 5, tenemos dos casos:
  - $send(m).lc\_ckpt_j[k] = 0$. Del manejo de $greater_j[]$ (ver, Sección 4.1), tenemos: $lc_j \geq send(m).lc\_ckpt_j[j] \geq 1 > send(m).lc\_ckpt_j[k] = 0 = lc_k$; De modo que, $greater_j[k]$ es $true$. Por lo tanto, $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$ se cumple.
  - $(send(m).T_j[i,k]{=}1) \wedge (send(m).idr\_ckpt_j[k]{=}1)$. Sea $e = send(m)$, del Lema 4, tenemos (nótese que $j \neq i$ y e no es un evento de recepción):
    - De $A1$, $k = i$. De modo que: $send(m).lc\_ckpt_j[k] \leq pred(receive(m)).lc\_ckpt_i[k]$, $max(e.lc\_ckpt_j[]) > e.lc\_ckpt_j[k]$ y $(send(m).idr\_ckpt_j[k] = 1)$. Sea $e.lc\_ckpt_j[s] =$

$max(e.lc\_ckpt_j[]) = lc_j$, entonces hay una secuencia de mensajes causales $[m_1 \downarrow m_2 \downarrow \ldots \downarrow m_\ell]$ de p_s a p_j. Así que, tenemos dos casos:

- ◇ Hay una secuencia de mensajes causales de p_s a p_i y otra de p_i a p_j. En este caso, $lc_i = lc_j = max(e.lc\_ckpt_j[])$. Por lo que, $SFI_b = SFI'_b = false$.
- ◇ No hay un secuencia de mensajes de p_s a p_i. Por lo que, $lc_j > lc_i$, entonces $greater_j[k] = true$.

Por lo que podemos concluir que: $SFI_a \wedge SFI_b \Rightarrow SFI'_a \wedge SFI'_b$ se cumple.

- ○ De $A2$, $send(m).lc\_ckpt_j[k] = 0$. Este caso ya fue analizado.
- ○ De $A3$, $\exists m'$ de p_i a p_j, $\forall z \in m'.\psi, \ldots$

  Sea $e.lc\_ckpt_j[s] = max(e.lc\_ckpt_j[]) = lc_j$ y como en el caso $k = i$, tenemos:
  - ◇ Hay una secuencia de mensajes causales de p_s a p_k y otro de p_k a p_j. De modo que, $lc_k = lc_i = lc_j = max(e.lc\_ckpt_j[])$. Por lo tanto, $SFI_b = SFI'_b = false$.
  - ◇ No hay secuencia de mensajes causales de p_s a p_k. De modo que $lc_j > lc_k$. Por lo tanto, $greater_j[k] = true$.

  Por lo tanto, en todos los casos $(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b)$ se cumple.

Por lo que podemos concluir en esta parte de la demostración que:

$$(SFI_a \wedge SFI_b) \Rightarrow (SFI'_a \wedge SFI'_b) \textit{ se cumple.}$$

□

Parte II.b. Finalmente, con el objetivo de probar que SFI'_c es equivalente a SFI_c, probamos por demostración directa que: $SFI_c \Rightarrow SFI'_c$; donde:

$$\begin{aligned} SFI_c &\equiv lc\_ckpt_i[i] = m.lc\_ckpt[i] \wedge \neg m.idr\_ckpt[i] \\ SFI'_c &\equiv (\exists z \in m.\psi, z.id = i : lc\_ckpt_i[i] = z.lc\_ckpt \wedge \\ &\qquad \neg z.idr\_ckpt) \end{aligned}$$

Demostración. En esta demostración tenemos dos casos por analizar:

- $\exists z \in m.\psi, z.id = i$. En este caso $SFI_c \Rightarrow SFI'_c$ se cumple. La información parcial en $m.\psi$ es la misma que p_j enviaría, en caso de enviar toda la información.
- $\nexists z \in m.\psi, z.id = i$. Por lo que, SFI'_c es siempre falso (para este caso). Sea $e = send(m)$, del Teorema 5 tenemos dos casos:

- $send(m).lc\_ckpt_j[k] = 0$. Si $k = i$ entonces tenemos: $\nexists e' \in E$ tal que $e' \in H_i \wedge e' \rightarrow e$. Por lo que, $SFI_c = false$ y como SFI'_c es falso. Por lo tanto, $SFI_c \Rightarrow SFI'_c$ se cumple.
- $(send(m).T_j[i,k] = 1) \wedge (send(m).idr\_ckpt_j[k] = 1)$. Aquí, si $k = i$ tenemos que $\exists e' \in E$ tal que $e' \in H_i \wedge e' \downarrow e$. Por lo que, $SFI_c = false$ $(e.idr\_ckpt_j[i] = m.idr\_ckpt[i] = true)$ y $SFI'_c = false$. Por lo tanto, $SFI_c \Rightarrow SFI'_c$ se cumple.

$\square_{Teorema\ 3}$

Apéndice B

Algoritmos S-FI y DCFI

En este apartado presentamos los códigos fuente de los algoritmos S-FI (capítulo 4) y DCFI (capítulo 5). Para obtener los códigos fuente en formato digital y los archivos de configuración para reproducir las simulaciones de esta investigación, puede solicitarme vía correo electrónico a acalixtomx@gmail.com.

B.1. SFI.java

```
1  package org.sagui.chksim.algorithm.qs;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import org.sagui.chksim.QuasiSynchronousAlgorithm;
6
7  public class SFI extends QuasiSynchronousAlgorithm{
8   private boolean[][] T;
9   private boolean[] idr_ckpt;
10  private boolean[]greater;
11  private boolean[] sent_to;
12  private int[] lc_ckpt;
13  private int lc;
14
15 //-------------------------------------------------------------------
16  protected void doInit(){
17   lc_ckpt = new int[getProcessNumber()];
18   T = new boolean[getProcessNumber()][getProcessNumber()];
19   idr_ckpt = new boolean[getProcessNumber()];
20   greater = new boolean[getProcessNumber()];
21   sent_to = new boolean[getProcessNumber()];
22
23    for(int k = 0; k < getProcessNumber(); k++) {
24    lc_ckpt[k] = 0;
```

```
25      for(int l=0;l<getProcessNumber();l++)
26          T[k][l]= true;
27      }
28
29      idr_ckpt[this.getProcess()] = true;
30      greater[this.getProcess()] = false;
31      lc=0;
32      this.takeBasicCheckpoint();
33    }
34  //-------------------------------------------------------------------
35    protected void doTakeBasicCheckpoint(){
36     doCheckpoint();
37    }
38  //-------------------------------------------------------------------
39    protected void doTakeForcedCheckpoint(){
40     doCheckpoint();
41     this.takeForcedCheckpoint();
42    }
43  //-------------------------------------------------------------------
44    protected void doCheckpoint(){
45     int i = getProcess();
46
47     for(int k = 0; k < getProcessNumber(); k++){
48         sent_to[k]=false;
49         if(k != i){
50        idr_ckpt[k] = false;
51        greater[k] = true;
52        T[k][i]=false;
53         }
54     }
55     lc = lc + 1;
56     lc_ckpt[i]= lc;
57     this.setsrtVC(String.valueOf(lc_ckpt[i]));
58    }
59  //-------------------------------------------------------------------
60     protected Serializable doSendMessage(int j) {
61       boolean type = true;
62       Data U;
63
64       sent_to[j] = true;
65       U = new Data();
66       //t - define si se envía el arreglo completo o sólo parte.
67       float t=((getProcessNumber()*(Integer.SIZE+2))/(2*Integer.SIZE+2));
68
69       for(int k = 0; k < getProcessNumber(); k++) {
70           if ((!T[j][k]||!idr_ckpt[k]) && lc_ckpt[k]>0){
71                U.addData(k,lc_ckpt[k],idr_ckpt[k],greater[k]);
72       }
73       if(U.getsize() > t){
74          type = false;
75          break;
76       }
77       }
78       if(type){
```

```
79          this.addMsgOverHead(U.getsize()*(Integer.SIZE*2 + 2));
80        }else{
81          U.addData(lc_ckpt, idr_ckpt, greater);
82              this.addMsgOverHead(U.getsize()*(Integer.SIZE + 2));
83        }
84        return U;
85    }
86
87  //-----------------------------------------------------------------------
88     protected void doReceiveMessage(int j, Serializable message) {
89        Data U = (Data) message;
90        int m_lc;
91        int i;
92        int i_U;
93
94        m_lc= 0;
95        for(int l=0; l < U.getsize(); l++){
96        if(U.getckpt(l) > m_lc)
97           m_lc = U.getckpt(l);
98        }
99        i = getProcess();
100
101       if(((i_U = U.getIndex(i))!=-1) && (lc_ckpt[i] == U.getckpt(i_U))
             && !U.getidr(i_U)){
102           doTakeForcedCheckpoint();
103       }else{
104             int k;
105             int k_U;
106             boolean chkban = false;
107             boolean k_greater;
108
109             for(k=0;!chkban&&k < getProcessNumber(); k++){
110                 if(sent_to[k]){
111                    k_greater = true;
112                    if(((k_U=U.getIndex(k))!=-1) && (U.getidr(k_U)))
113                         k_greater = U.getgreater(k_U)
114                    chkban = ((k_greater) && (m_cl > lc));
115                    }
116             }
117             if(chkban){
118                doTakeForcedCheckpoint();
119             }
120       }
121
122       for(int y = 0; y < U.getsize(); y++){
123           int y_id;
124
125       y_id = U.getid(y);
126           if(lc_ckpt[y_id] < U.getckpt(y)){
127              lc_ckpt[y_id] = U.getckpt(y);
128              idr_ckpt[y_id] = U.getidr(y);
129              for(int l = 0; l < getProcessNumber(); l++){
130                  if(l!=i)
131                     T[l][y_id]=false;
```

```
132                 }
133                 if(m_cl != lc_ckpt[y_id] || lc > lc_ckpt[y_id] )
134                    T[j][y_id]=true;
135              }else{
136                    if(lc_ckpt[y_id]==U.getckpt(y)){
137               idr_ckpt[y_id]=(idr_ckpt[y_id]&& U.getidr(y));
138                       if(m_cl!=lc_ckpt[y_id]||lc > lc_ckpt[y_id])
139                          T[j][y_id]=true;
140                    }
141               }
142          }
143          if(m_cl > lc){
144             lc=m_cl;
145             greater[i]=false;
146             for(int l=0; l < this.getProcessNumber(); l++){
147                 int l_U;
148                 if(l!=i){
149                          if((l_U=U.getIndex(l))!=-1)
150                              greater[l]=U.getgreater(l_U);
151                              else
152                              greater[l]=true;
153                 }
154             }
155          }else{
156            if(m_cl == lc){
157                   for(int l=0; l < U.getsize(); l++){
158                       int l_id;
159                       l_id = U.getid(l);
160                       greater[l_id]=(greater[l_id] && U.getgreater(l));
161                   }
162               }
163           }
164     }
165
166 //*********************************************************************
167  public class Data implements Serializable{
168   private ArrayList data;
169 //---------------------------------------------------------------------
170   public Data(){
171    data = new ArrayList();
172   }
173 //---------------------------------------------------------------------
174   public void addData(int ckpt[], boolean idr[], boolean greater[]){
175    AData d;
176    data.clear();
177    for(int i = 0; i< ckpt.length;i++){
178        d=new AData(ckpt[i],idr[i],greater[i]);
179        data.add(d);
180    }
181   }
182 //---------------------------------------------------------------------
183   public void addData(int id, int ckpt, boolean idr, boolean greater){
184    BData d = new BData(id,ckpt,idr,greater);
185    data.add(d);
```

```
186   }
187 //-------------------------------------------------------------------
188   public int getsize(){
189    return data.size();
190   }
191 //-------------------------------------------------------------------
192   public int getid(int i){
193    BData d;
194    Object obj = data.get(i);
195    if(obj instanceof BData){
196       d = (BData) obj;
197       return(d.getid());
198    }
199     else
200         if(obj instanceof AData){
201            return(i);
202         }
203    return -1;
204   }
205 //-------------------------------------------------------------------
206   public int getckpt(int i){
207    AData d = (AData)data.get(i);
208     return d.getckpt();
209   }
210 //-------------------------------------------------------------------
211   public boolean  getidr(int i){
212    AData d = (AData)data.get(i);
213    return d.getidr();
214   }
215 //-------------------------------------------------------------------
216   public boolean  getgreater(int i){
217    AData d = (AData)data.get(i);
218    return d.getgreater();
219   }
220 //-------------------------------------------------------------------
221   public int getIndex(int id){
222    for(int i=0;i<getsize();i++)
223        if(id == getid(i))
224           return(i);
225    return(-1);
226   }
227 //*******************************************************************
228     public class AData{
229      private int ckpt;
230      public boolean idr;
231      public boolean greater;
232
233      public AData(int ckpt, boolean idr, boolean greater){
234       this.ckpt= ckpt;
235       this.idr = idr;
236       this.greater=greater;
237      }
238   //-------------------------------------------------------------------
239      public int getckpt(){
```

```
240       return ckpt;
241      }
242   //-------------------------------------------------------------------
243      public boolean getidr(){
244       return idr;
245      }
246   //-------------------------------------------------------------------
247      public boolean getgreater(){
248       return greater;
249      }
250     }
251 //*******************************************************************
252     public class BData extends AData{
253      private int id;
254 
255      public BData(int id,int ckpt, boolean idr, boolean greater){
256       super(ckpt,idr,greater);
257       this.id=id;
258      }
259      public int getid(){
260       return id;
261      }
262     }
263   }
264  }
```

Algoritmo B.1: Código Java para el Algoritmo S-FI.

B.2. DCFI.java

```
1  package org.sagui.chksim.algorithm.qs;
2
3  import java.io.Serializable;
4  import org.sagui.chksim.Event;
5  import org.sagui.chksim.QuasiSynchronousAlgorithm;
6
7  public class DCFI extends QuasiSynchronousAlgorithm {
8
9   protected boolean sent_to[];
10  protected boolean taken[];
11  protected boolean greater[];
12  protected int ckpt[];
13  protected int lc;
14
15  protected boolean taken_before[];
16  protected int ckpt_before[];
17  protected boolean greater_before[];
18  protected int lc_before;
19
20  protected boolean taken_before_A[];
21
22
23  protected boolean delay_ckpt;
24  protected boolean rec;
25  protected int  num_delay_ckpt;
26 //-----------------------------------------------------------------------
27  protected void doInit() {
28
29    sent_to = new boolean[getProcessNumber()];
30    taken = new boolean[getProcessNumber()];
31    greater = new boolean[getProcessNumber()];
32    ckpt = new int[getProcessNumber()];
33
34    taken_before = new boolean[getProcessNumber()];
35    greater_before = new boolean[getProcessNumber()];
36    ckpt_before = new int[getProcessNumber()];
37
38    taken_before_A = new boolean[getProcessNumber()];
39
40    for(int k = 0; k < getProcessNumber(); k++)
41         ckpt[k] = 0;
42    lc=0;
43    taken[getProcess()] = false;
44    greater[getProcess()] = false;
45    takeBasicCheckpoint(false);
46    }
47 //-----------------------------------------------------------------------
48 protected void actualizardata (boolean type){
49  int i = getProcess();
50
51  delay_ckpt = type;
```

```
52  if(delay_ckpt){
53     rec = false;
54     num_delay_ckpt = 0;
55     lc_before = lc;
56     for (int k = 0; k < getProcessNumber(); k++){
57          sent_to_before[k] = sent_to[k];
58          ckpt_before[k]=ckpt[k];
59          taken_before[k]= taken[k];
60          greater_before[k]=greater[k];
61          taken_before_A[k]= taken[k];
62     }
63  }
64  for(int k = 0; k < getProcessNumber(); k++){
65      sent_to[k] = false;
66      if(k!=i){
67         taken[k]=true;
68         greater[k]=true;
69      }
70  }
71  lc++;
72  ckpt[i]++;
73 }
74 //-------------------------------------------------------------------
75  protected void doTakeBasicCheckpoint(){
76   actualizardata(true);
77  }
78 //-------------------------------------------------------------------
79  protected void doTakeBasicCheckpoint(boolean type){
80   actualizardata(type);
81  }
82 //-------------------------------------------------------------------
83  protected void doTakeForcedCheckpoint(){
84   actualizardata(false);
85   this.takeForcedCheckpoint(); // p_i toma un checkpoint forzado.
86  }
87 //-------------------------------------------------------------------
88 protected Serializable doSendMessage(int destination){
89  Serializable m;
90  int i = this.getProcess();
91
92  if(delay_ckpt && !taken_before_A[destination] && num_delay_ckpt < 3){
93     if(!rec){
94        m=new ControlData(lc_before,ckpt_before,greater_before,
              taken_before);
95     }
96      else{
97       m=new ControlData(lc_before,ckpt_before,greater_before,
             taken_before);
98       for(int k = 0; k < getProcessNumber(); k++){
99                taken_before_A[k] = taken_before[k];
100               sent_to[k]= false;
101               if(k!=i){
102                  taken[k]=true;
103                  greater[k]=true;
```

```
104                 }
105             }
106             rec = false;
107         }
108        num_delay_ckpt++;
109     }
110      else{
111           delay_ckpt = false;
112           m = new ControlData(lc, ckpt,greater, taken);
113           sent_to[destination]=true;
114     }
115     this.addMsgOverHead((this.getProcessNumber()+1)*Integer.SIZE+(this.
        getProcessNumber()*2));
116     return m;
117     }
118    //-----------------------------------------------------------------------
119     protected void doReceiveMessage(int sender, Serializable message){
120      int i;
121      ControlData m = (ControlData) message;
122      i = getProcess();
123
124      if(delay_ckpt){
125         if(m.ckpt[i] == ckpt_before[i] && m.taken[i])
126            delay_ckpt=false;
127      }
128      if(m.ckpt[i]==ckpt[i] && m.taken[i]){
129         doTakeForcedCheckpoint();
130         this.addMsgNumber(1);
131      }
132       else{
133           boolean chkban = false;
134           int k;
135
136           for(k = 0; !chkban && k < getProcessNumber(); k++)
137               chkban = (sent_to[k] && m.greater[k] && m.lc > lc);
138           if(chkban){
139              doTakeForcedCheckpoint();
140           }
141      }
142      if(m.lc > lc){
143         i=getProcess();
144         delay_ckpt = false;
145         lc = m.lc;
146         greater[i] = false;
147         for(int k=0;k<this.getProcessNumber();k++){
148             if(k!=i)
149                greater[k] = m.greater[k];
150         }
151      }
152       else{
153           if(m.lc==lc){
154              delay_ckpt = false;
155              for(int k=0;k<this.getProcessNumber();k++)
156                  greater[k]= greater[k] && m.greater[k];
```

```
157         }
158          else{
159               if(m.lc==lc_before){
160                  for(int k=0;k<this.getProcessNumber();k++)
161                       greater_before[k]= greater_before[k] && m.greater[k
                             ];
162               }
163          }
164     }
165    i=getProcess();
166    for(int k=0;k<this.getProcessNumber();k++){
167        if(k!=i){
168           if(m.ckpt[k]>ckpt[k]){
169              ckpt[k]=m.ckpt[k];
170              taken[k]=m.taken[k];
171              if(delay_ckpt){
172                 ckpt_before[k]= m.ckpt[k];
173                 taken_before[k]= m.taken[k];
174              }
175           }
176            else{
177               if(m.ckpt[k]==ckpt[k]){
178                  taken[k]=taken[k] || m.taken[k];
179                  if(delay_ckpt)
180                     taken_before[k]= taken_before[k] || m.taken[k];
181               }
182            }
183        }
184    }
185    rec=true;
186   }
187  //*****************************************************************
188   public class ControlData implements Serializable{
189    public int lc;
190    public int[] ckpt;
191    public boolean[] greater;
192    public boolean[] taken;
193
194    public ControlData(int lc, int[] ckpt, boolean[] greater,boolean[]
          taken){
195      this.lc = lc;
196      this.ckpt = (int[]) ckpt.clone();
197      this.greater = (boolean[]) greater.clone();
198      this.taken = (boolean[]) taken.clone();
199    }
200   }
201  //*****************************************************************
202  }
```

Algoritmo B.2: Código Java para el Algoritmo DCFI.