TESIS DEFENDIDA POR

Luis Daniel Rodríguez Altamirano

Y APROBADA POR EL SIGUIENTE COMITÉ

Dr. José Alberto Fernández Zepeda

Director del Comité

Dr. Carlos Alberto Brizuela Rodríguez

Miembro del Comité

Dra. Josefina Rodríguez Jacobo

Miembro del Comité

Dra. Carmen Paniagua Chávez

Miembro del Comité

Dra. Ana Isabel Martínez García

Coordinador del programa de posgrado en Ciencias de la computación

Dr. David Hilario Covarrubias Rosales

Director de Estudios de Posgrado

12 de Febrero de 2010

CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

PROGRAMA DE POSGRADO EN CIENCIAS EN CIENCIAS DE LA COMPUTACIÓN

ANÁLISIS DE CONTENCIÓN DE FALLAS EN UN ALGORITMO AUTO-ESTABILIZANTE DE ELECCIÓN DE LÍDER

TESIS

que para cubrir parcialmente los requisitos necesarios para obtener el grado de

MAESTRO EN CIENCIAS

Presenta:

LUIS DANIEL RODRÍGUEZ ALTAMIRANO

Ensenada, Baja California, México, Febrero de 2010

RESUMEN de la tesis de **LUIS DANIEL RODRÍGUEZ ALTAMIRANO**, presentada como requisito parcial para la obtención del grado de MAESTRO EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN . Ensenada, Baja California, Febrero de 2010.

ANÁLISIS DE CONTENCIÓN DE FALLAS EN UN ALGORITMO AUTO-ESTABILIZANTE DE ELECCIÓN DE LÍDER

Resumen aprobado por:

Dr. José Alberto Fernández Zepeda

Director de Tesis

La auto-estabilización es un paradigma de tolerancia a fallas en sistemas distribuidos. El objetivo de la auto-estabilización es mantener a un sistema libre de errores. Los algoritmos auto-estabilizantes buscan que un sistema se recupere por si solo de fallas transitorias que lo afecten en un número finito de pasos sin la intervención de agente externo alguno.

Los algoritmos de elección de líder tienen el propósito de elegir a un nodo de un conjunto de nodos candidatos, en ocasiones indistinguibles entre sí. La idea de elegir un nodo como líder es que tenga privilegios sobre los demás nodos.

Los algoritmos auto-estabilizantes se ayudan de la contención de fallas para recuperarse en menor tiempo de las fallas que lo afecten. La contención de fallas busca aislar las fallas para que no se propaguen y evitar que contaminen otras partes del sistema.

La presente investigación busca agregar contención de fallas a un algoritmo auto-estabilizante de elección de líder.

Se presenta un nuevo algoritmo anónimo auto-estabilizante de elección de líder con contención de fallas en grafos de árbol. Se demuestra que el algoritmo es correcto y que termina en $O(n^4)$ pasos, empezando en cualquier estado inicial arbitrario. El algoritmo no puede elegir un nodo hoja como líder. El algoritmo de elección de líder con contención de fallas confina el efecto de cualquier falla simple a los vecinos inmediatos del nodo que falla, con un tiempo esperado de recuperación de $O(1)$. El aspecto más importante del algoritmo es que la *brecha de falla*, definida como el intervalo más pequeño después del cual el sistema está listo para manejar la siguiente falla con la misma eficiencia, no depende del tamaño de la red.

Palabras Clave: Auto-estabilización, elección de líder, fallas, contención.

ABSTRACT of the thesis that **LUIS DANIEL RODRÍGUEZ ALTAMIRANO** presents in partial fulfillment of the requirements for the degree of MASTER IN SCIENCES in COMPUTER SCIENCES . Ensenada, Baja California, February 2010.

ANALYSIS OF CONTAINMENT OF FAULTS IN ALGORITHM SELT-STABILIZING OF LEADER ELECTION

Self-stabilization is a relatively new paradigm for fault tolerance in distributed systems. The goal of self-stabilization is to maintain a system free of errors. A self-stabilizing algorithm helps a system to recover from transient faults in a finite number of steps, without involving any external agent.

There are many graph problems that have been solved by self-stabilizing algorithms. This thesis deals with one of them, the leader election problem, whose definition is as follows. Given a graph G and set of nodes C of G, the idea is to elect a unique node from the set C.

A fault containment technique helps self-stabilizing algorithms to recover faster from transient faults. This technique avoids that the effect of a fault propagates to healthy nodes of the graph. The objective of this thesis is to incorporate a fault containment technique to a self-stabilizing leader election algorithm.

Our algorithm is restricted to run on anonymous trees. We proved the correctness of the algorithm and show that it terminates in $O(n^4)$ time, starting in any arbitrary state. Once the tree is in a legal state and a simple fault occurs, the proposed algorithm takes the tree to a legal state in constant time. The most significant aspect of the algorithm is that the fault –gap is independent of the network size.

Keywords: self-stabilization, leader election, faults containment.

Keywords: self-stabilization, leader election, faults, containment

A mis padres ya que sin ellos no hubiera sido posible para mi el llegar hasta aquí.

A mi esposa Cynthia Denisse y a mi hijo Iker por todo su apoyo y comprensión.

Agradecimientos

A mi director de tesis el Dr. José Alberto Fernández Zepeda por sus comentarios atinados y su paciencia al tratar los detalles de esta tesis.

A Daniel Fajardo por su ayuda en el desarrollo del simulador y sus consejos para que esta tesis resultara mejor.

A los miembros de mi comité de tesis por sus comentarios para la mejora de esta tesis.

A mis compañeros de Cicese por todos los momentos gratos que pasamos y por compartir sus conocimientos.

A todos los investigadores y personal del departamento de ciencias de la computación por su enseñanza académica.

Al Consejo Nacional de Ciencia y Tecnología por su apoyo económico.

Contenido

Página

Contenido (continuación)

Lista de Figuras

Lista de Figuras (continuación)

Lista de Figuras (continuación)

Lista de Figuras (continuación)

Lista de Tablas

Capítulo I

INTRODUCCIÓN

La computadora se ha convertido en una herramienta indispensable en muchas áreas, para el trabajo, el entretenimiento, como medio de comunicación y/o expresión, entre otros. Actualmente son innumerables las actividades que necesitan de una computadora para llevarse a cabo y día a día la computadora se va haciendo más indispensable para otra actividades. Con la creación y el auge de las redes, ahora hay muchos sistemas computacionales que están interconectados y debido a esta alta conectividad, la falla de algún componente de la red puede impactar en algún otro componente cercano ocasionando que las fallas se esparzan. Por tal razón nace la necesidad de crear sistemas tolerantes a fallas que garanticen un funcionamiento correcto.

Un sistema distribuido es una colección de computadoras separadas físicamente y conectadas entre sí mediante una red de comunicación. Cada máquina posee sus componentes de hardware y software que el usuario percibe como un solo sistema, es decir, el usuario normalmente no necesita saber qué elementos están en qué máquina. El usuario accede a los recursos remotos de la misma forma en que accede a los recursos locales. Por esta razón, los sistemas distribuidos deben ser muy confiables, ya que si un componente del sistema se descompone, otro componente debe ser capaz de suplir su función.

El paradigma de auto-estabilización busca que esto sea posible. La auto-estabilización es un área relativamente nueva, aunque Djikstra la introdujo a las ciencias de la computación hace ya varios años (Dijkstra, 1974). La auto-estabilización empezó

a cobrar importancia hasta la década de los ochenta. Algunos investigadores consideran a dicho artículo el trabajo más brillante de Dijkstra (Schneider, 1993).

La auto-estabilización es muy utilizada en los sistemas distribuidos. Dos trabajos amplios y fundamentales referentes a auto-estabilización son (Dolev, 2000) y (Schneider, 1993). La auto-estabilización es una forma de concebir los sistemas tolerantes a fallas. Un algoritmo auto-estabilizante es un algoritmo que permite a un sistema recuperarse automáticamente, en un número finito de pasos, de cualquier falla transitoria que lo afecte sin intervención de algún agente externo.

En los algoritmos auto-estabilizantes, uno de los parámetros más importantes y difíciles de calcular es el tiempo de convergencia. El tiempo de convergencia es el tiempo que le toma a un algoritmo auto-estabilizante alcanzar un funcionamiento correcto sin importar cómo empezó a funcionar. En general, los algoritmos auto-estabilizantes son más lentos que los algoritmos no auto-estabilizantes, y como ya se mencionó anteriormente, su tiempo de convergencia es más difícil de evaluar; para algunos algoritmos, incluso no se han determinado sus tiempos de convergencia con presición, por lo que solamente se dan cotas que algunas veces ni siquiera son ajustadas.

Se han diseñado una gran cantidad de algoritmos auto-estabilizantes para muchas aplicaciones. El algoritmo de (Baala *et al.*, 2003) muestra la construcción de un árbol de esparcimiento en redes inalámbricas ad hoc. (Beauquier *et al.*, 2007a) desarrolla un algoritmo auto-estabilizante para redes de sensores, el objetivo del algoritmo es contar el número de agentes en la red. Los autores suponen dos tipos de comunicaciones; con antena fija y con interacciones entre un par de ellas. (Bein *et al.*, 2007) presentan un algoritmo para el ruteo local en redes ad hoc. (Blair y Manne, 2003) presentan un algoritmo de auto-estabilización eficiente para redes con topología de árbol. (Chaudhuri y Thompson, 2005) presentan un algoritmo de coloreo radial para árboles dirigidos.

(Chen *et al.*, 2005) presentan un algoritmo de enrutamiento en internet. (Mathieu, 2008) presenta un algoritmo auto-estabilizante para sistemas basados en preferencias, la idea de los autores es investigar y comparar la velocidad y calidad del proceso de convergencia con respecto a otros modelos. (Kosowski y Kuszner, 2006) exponen un algoritmo de coloreo de grafos. (Karaata, 2004) presenta un algoritmo auto-estabilizante para un cluster de anillo para minizar el costo de compartir recursos. (Huang *et al.*, 2005) muestran un algoritmo de coloreo en redes planares anónimas.

Un grupo de algoritmos auto-estabilizantes muy estudiados son los algoritmos de elección de líder; estos algoritmos se utilizan como subrutinas para resolver problemas más complejos. Existen algoritmos de elección de líder que corren sobre topologías de anillo, árbol, entre otras. Cada algoritmo aprovecha las características propias de la topología. El problema de elección de líder consiste en elegir un nodo líder a partir de un conjunto de candidatos. Algunos investigadores consideran que el problema de elección de líder más difícil de resolver es cuando se calcula en un sistema anónimo. Un sistema anónimo es un sistema que no permite distinguir entre los nodos del sistema.

Se ha desarrollado una gran cantidad de algoritmos auto-estabilizantes de elección de líder para varias topologías, entre ellos se pueden mencionar: algoritmos auto-estabilizantes de elección de líder para anillos (Huang, 1993), (Itkis *et al.*, 1995), (Beauquier *et al.*, 2007b), árboles (Antonoiu y Srimani, 1996), (Xu y Srimani, 2005), (Alvarado-Magaña y Fernández-Zepeda, 2007), estrellas enraizadas (Shi *et al.*, 2005), hipercubos (Shi y Srimani, 2004) y otras (Beauquier *et al.*, 1999). En algunos se incluye además un análisis para el espacio de memoria que necesitan estos algoritmos como el que presenta (Beauquier *et al.*, 1999). En el Capitulo III se hace una descripción más profunda de los algoritmos de elección de líder.

Una característica en los sistemas distribuidos, como ya se mencionó anteriormente,

es la tolerancia a fallas. También se mencionó que los algoritmos auto-estabilizantes son útiles en el diseño de algoritmos que toleran fallas transitorias. Las fallas transitorias aparecen por un corto periodo de tiempo y no causan daño permanente al sistema, son fallas de caracter lógico. Un ejemplo de una falla transitoria es cuando existe mucho ruido eléctrico en la cercanía del sistema y algún bit de memoria puede cambiar de 0 a 1 o viceversa, sin que la memoria en sí se dañe. Un efecto que en algunos algoritmos auto-estabilizantes ocurre es que los errores producidos por una falla se propagan y contaminan nodos sanos dentro de la red; esto lleva al algoritmo a reconfigurar todos los nodos que se vieron afectados o "contaminados" por el efecto de la falla. Este tipo de comportamiento se debe evitar y una forma de lograrlo es por medio de un algoritmo de contención de fallas. No todas las fallas son iguales, en el Capítulo IV se expone una clasificación de algunas técnicas de contención de fallas.

Algunos algoritmos auto-estabilizantes de elección de líder pueden incorporar alguna técnica de contención de fallas. Un problema de que los algoritmos auto-estabilizantes de elección de líder no incorporen un mecanismo de contención de fallas es que una falla transitoria simple puede llegar a contaminar una gran cantidad de nodos. Por lo que una sola falla, puede ocasionar que el algoritmo tenga que elegir un nuevo líder y/o reconfigurar una gran parte del sistema, o incluso el sistema entero.

Dos de los algoritmos auto-estabilizantes que incorporan contención de fallas son (Dasgupta *et al.*, 2007) y (Ghosh y Gupta, 1996). Estos algoritmos incorporan la contención de fallas como medida preventiva para que las fallas transitorias ocurridas en el sistema no se esparzan a otros nodos. Incluso algunos investigadores, (Dasgupta *et al.*, 2007) proponen un algoritmo auto-estabilizante con contención probabilística de fallas. El problema de este algoritmo es que puede no funcionar adecuadamente en grafos completos o en grafos cuyo grado sea muy cercano a n, donde n es el número

de nodos de la red. El grado de un nodo es igual al número de vecinos de dicho nodo. Además, una falla que ocurre a menos de tres nodos de distancia de donde ocurre otra falla puede ocasionar que el sistema requiera un tiempo más grande en su recuperación.

En esta investigación se busca incorporar la contención de fallas a algoritmos de elección de líder. Con la incorporación de la contención de fallas en el algoritmo, se busca que una falla transitoria simple se corrija rápidamente y a lo más contamine un número constante de nodos. De esta manera el sistema estará más tiempo disponible para trabajar ya que pasará un tiempo menor corrigiendo los errores que le afecten. El algoritmo debe tardar un tiempo relativamente menor en corregir una falla ya que no necesita reconfigurar todo el sistema.

I.1 Contribuciones de esta tesis

El objetivo de esta tesis es analizar algunos algoritmos auto-estabilizantes de elección de líder y ver la factibilidad de agregarles contención de fallas. Se analizó el paradigma de auto-estabilización, los algoritmos de elección de líder y algunas técnicas de contención de fallas. Finalmente, se eligió trabajar con el algoritmo de Xu y Srimani buscando agregarle contención de fallas.

El algoritmo de (Xu y Srimani, 2005) es un algoritmo auto-estabilizante de elección de líder, este algoritmo elige un líder en un árbol anónimo en un tiempo de $O(n^4)$. Una razón por la que se eligió este algoritmo es que es un algoritmo muy simple que consta solamente de tres reglas sencillas. Una vez que el árbol elige un líder y una falla lo afecta, el algoritmo tiene que reconfigurar grandes partes del árbol o incluso el árbol entero para corregir los efectos de la falla. Dado este comportamiento, la contención de fallas puede mejorar notablemente el desempeño del algoritmo. Los detalles de este

algoritmo se ven en el Capítulo III.

Una vez seleccionado el algoritmo para añadirle contención de fallas, se empezaron a realizar modificaciones a dicho algoritmo. Así se logró obtener un algoritmo auto-estabilizante de elección de líder con contención de fallas para grafos de árbol.

El algoritmo propuesto en esta tesis presenta mejorías con respecto al algoritmo de Xu y Srimani ya que experimentalmente se muestra que elige un líder en un número menor de pasos que el algoritmo de Xu y Srimani aunque ambos siguen corriendo en el mismo orden de complejidad temporal (la comparación se puede ver en el Capítulo VII). Además, una vez que elige un líder y una falla lo afecta, el algoritmo corrige inmediatamente al nodo que falla; tal corrección se realiza en un tiempo constante. Esto representa una gran ventaja sobre el algoritmo de Xu y Srimani ya que evita que la falla se propague a otros nodos que no han fallado. Además le permite al sistema permanecer disponible un tiempo mucho mayor, ya que pasa un menor tiempo corrigiendo las fallas que lo afectan.

Adicionalmente se diseñó un simulador que ejecuta tanto el algoritmo diseñado como el algoritmo de Xu y Srimani. El simulador tiene varios fines, uno de ellos es encontrar posibles errores que se hayan pasado por alto en el diseño del algoritmo, calcular el número de pasos promedio que requiere cada algoritmo para elegir un líder, y calcular parámetros importantes de desempeño como el número de pasos de la brecha de falla.

Además, el simulador permite reconstruir el árbol generado y la secuencia de pasos que realiza el algoritmo, así se puede identificar y reconstruir de una forma más rápida errores en el algoritmo para posteriormente enfocarse en analizarlos y buscarles solución en caso de haber.

I.2 Panorama general del documento

El Capítulo II da una visión más profunda del paradigma de auto-estabilización. El Capítulo III describe el problema de elección de líder y muestra algunos algoritmos auto-estabilizantes de elección de líder que resultan interesantes, adicionalmente se describe el funcionamiento de los algoritmos mostrados, y se incluye en algunos casos el análisis del tiempo que les lleva elegir un líder. El Capítulo IV proporciona los conceptos básicos relacionados con la contención de fallas y expone algunas técnicas de contención de fallas en algoritmos auto-estabilizantes. El Capítulo V muestra la descripción del algoritmo propuesto y su pseudo-código. El Capitulo VI incluye las demostraciones de convergencia y cerradura. El Capítulo VII muestra los resultados generados por el simulador desarrollado para probar el funcionamiento del algoritmo propuesto y se comparan con los resultados obtenidos por el algoritmo de Xu y Srimani, algoritmo tomado como base para agregarle la contención de fallas. El Capítulo VIII muestra las conclusiones derivadas de esta investigación, expone las aportaciones y da una idea acerca del trabajo futuro dentro de esta línea.

Capítulo II

AUTO-ESTABILIZACIÓN

La auto-estabilización fue introducida por (Dijkstra, 1974) a las ciencias de la computación hace ya varios años. Un algoritmo auto-estabilizante tiene la propiedad de alcanzar un estado legítimo en un número finito de pasos. Un estado legítimo representa un funcionamiento correcto del algoritmo; un estado ilegítimo representa un funcionamiento incorrecto.

El estado local de un nodo en el sistema; es el conjunto de valores de sus variables internas. Un estado global lo definen los estados locales de todos los nodos del sistema. De esta manera, un estado local legítimo representa un funcionamiento correcto del nodo. Un estado local ilegítimo es un funcionamiento incorrecto del nodo. Se dice que un funcionamiento del sistema es correcto cuando el sistema está en un estado global legítimo. En caso contrario se dice que el sitema es incorrecto.

En la Figura 1 se muestra un grafo de anillo de tres nodos. Cada nodo tiene una variable binaria a. La idea es elegir un nodo del grafo como líder. El nodo líder debe tener su variable $a = 1$ y los nodos restantes deben tener $a = 0$. El estado local de un nodo está definido por su variable a, así, cada nodo puede tener dos estados locales, cuando a vale 0 y cuando vale 1. Los estados globales están definidos por las variables de todos los nodos del grafo. Existen 8 estados globales definidos como se muestra en la Tabla I. De los 8 estados globales, sólo 3 son válidos.

Dentro de los algoritmos auto-estabilizantes, se cumple lo siguiente:

1. En cada estado global ilegítimo se pueden presentar uno o más nodos privilegiados.

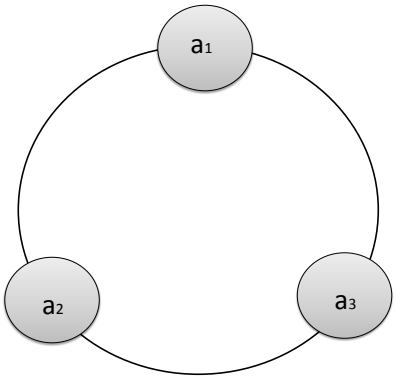

Figura 1. Grafo de anillo de tres nodos.

2. En cada estado global legítimo, cada movimiento válido posible de alguna variable lleva al sistema a otro estado global legítimo.

Tabla I. Representación de los estados globales del grafo. Los renglones en negrita representan a los estados globales válidos.

a_1	a_2	a_3
0	0	0
0	**0**	**1**
0	**1**	**0**
0	1	1
1	**0**	**0**
1	0	1
1	1	0
1	1	1

Los algoritmos auto-estabilizantes buscan alcanzar un funcionamiento correcto sin

importar cómo empezaron a funcionar, además el paradigma de la auto-estabilización busca que un sistema por si solo se recupere de las fallas que lo afecten en un tiempo finito. (Schneider, 1993) propone la siguiente definición para sistemas auto-estabilizantes.

Definición 1 *Dado un sistema S donde las condiciones del estado global válido las define un predicado P, S es auto-estabilizante con respecto a P si satisface las siguientes dos propiedades.*

1. Iniciando desde un estado global arbitrario, se garantiza que S alcanza un estado global donde P es verdadero, en un número finito de pasos.
2. Una vez que P es verdadero, P no se puede convertir en falso.

Lo que Schneider hizo es definir las propiedades de convergencia y cerradura.

1. **Convergencia:** Esta propiedad establece que, sin importar cómo empezó a funcionar, el algoritmo debe alcanzar un estado legítimo en un tiempo finito.
2. **Cerradura:** Una vez alcanzado un estado global legítimo, el algoritmo deberá permanecer en un estado global legítimo a menos que una falla lo afecte.

Existen algunos parámetros de rendimiento para la auto-estabilización. (Ghosh y Gupta, 1996) mencionan los siguientes:

- **Tiempo de estabilización:** Es el tiempo necesario para alcanzar un estado global legítimo partiendo de un estado gloabal inicial arbitrario, en el peor caso.
- **Espacio de estabilización:** Es el espacio máximo utilizado por cualquier nodo en un estado legítimo (es diferente al espacio utilizado en un espacio ilegítimo).

Existen diferentes métodos para demostrar las propiedades de cerradura y convergencia y el tiempo que lleva alcanzar la convergencia en los algoritmos auto-estabilizantes. Algunas de ellas son:

- **Funciones variantes:** Para poder demostrar la convergencia se verifica que la función variante crece o decrece monotónicamente mientras el sistema no haya alcanzado un estado global legítimo. Si la función no cumple lo anterior significa que el sistema no se estabiliza.

- **Escalones de convergencia:** Demuestra las propiedades de convergencia y cerradura. Para realizar la demostración se definen una serie de condiciones que debe cumplir el estado global legítimo. Las condiciones se van demostrando una a una. La demostración de la última condición demuestra la propiedad de convergencia.

- **Grafo de dependencia:** Define un grafo con base en las variables locales y las variables de los vecinos de cada nodo del grafo. Se determinan los componentes completamente conectados del grafo y se demuestra la convergencia de cada uno de esos componentes. Posteriormente se repite el proceso en un nuevo grafo. Los nodos del nuevo grafo representan a cada uno de los componentes completamente conectados del grafo inicial.

- **Acoplamiento:** Define el sistema como una cadena de Markov. Posteriormente se definen dos estados iniciales diferentes de la misma cadena de Markov. Se define una funcion de distancia entre las dos cadenas. En cada paso cambia el mismo bit de cada cadena. Se demuestra que la distancia entre las dos cadenas va decreciendo en forma monotónica con cada paso hasta alcanzar un máximo o

un mínimo. Una vez demostrado lo anterior se busca una β que es una tasa de reducción de la distancia entre las cadenas. En caso de que la función crezca y decrezca se debe encontrar una α; α define la probabilidad de cambio de la cadena de Markov. Una vez definida la probabilidad de cambio se demuestra que en promedio la distancia entre las dos cadenas decrece a cada paso hasta alcanzar un máximo o un mínimo. En ambos casos el máximo o mínimo encontrado representa alcanzar un estado global legítimo.

- **Teoría de control:** Esta técnica es relativamente nueva en ciencias de la computación y tiene sus bases en la teoría de control de circuitos. La idea es modelar un sistema auto-estabilizante como un circuito de control para demostrar la convergencia. Un circuito de control básico consta de 4 partes: una entrada, un controlador, una salida y una señal de retroalimentación. La entrada es un estado global inicial arbitrario del sistema. El controlador o controladores son los guardias de las acciones del algoritmo. La salida es el estado global legítimo del sistema y la señal de retroalimentación es una señal de compenzación que ayuda al controlador a hacer su trabajo, en este caso se puede ver como un guardia compuesto.

II.1 Modelo de computación de un algoritmo auto-estabilizante

El siguiente modelo de computación es el más utilizado en los algoritmos auto-estabilizantes.

Sea $G = (V, E)$ la topología de un sistema distribuido, donde $V = \{0, 1, ..., n-1\}$

representa el conjunto de nodos, y cada arista $(i, j) \in E$ representa el enlace bidireccional entre los nodos i y j. La topología define la forma en la que los nodos se conectan y pueden comunicarse entre sí dentro de una red. Se utiliza la notación $N(i)$ para representar los vecinos de i; asi $(i, j) \in E \iff j \in N(i)$. Los nodos se comunican directamente sólo con sus vecinos inmediatos. Cada nodo i ejecuta un programa que consiste de una o más acciones vigiladas de la forma $g \rightarrow A$, donde g es un predicado con la participación de las variables de i y las de los vecinos inmediatos. Un predicado es una condición que debe cumplirse para poder ejecutar alguna parte de código. A es una acción que actualiza una o más variables de i. Una ejecución del sistema es una secuencia finita o infinita de estados globales que satisfacen dos propiedades:

- Si s y s' son dos estados consecutivos, entonces existe un nodo i tal que i tiene un guardia habilitado en s y la ejecución de la acción correspondiente resulta en el estado s'.
- Si la secuencia es finita, entonces en el último estado de la secuencia, ningún nodo tiene un guardia habilitado.

II.2 Estructura de un algoritmo auto-estabilizante

En esta sección se describe la estructura de un algoritmo auto-estabilizante y las partes que lo componen.

El algoritmo de (Xu y Srimani, 2005) es un algoritmo auto-estabilizante de elección de líder para grafos de árbol. Las partes que componen a un algoritmo auto-estabilizante se describen a continuación y hacen referencia a la Figura 2. El algoritmo de la Figura 2 se utiliza únicamente para explicar las partes que componen un algoritmo

$\triangleright$ R1

1 if $(\exists j, k \in N(_i),\ j \neq k \wedge x_j \geq x_i \wedge x_k \geq x_i)$

2 then if $x_i \neq \text{máx}_{j \in N_i}(x_j) + 1$ then $x_i \leftarrow \text{máx}_{j \in N_i}(x_j) +$

$\triangleright$ R2

4 if $(\exists! j \in N_i \mid x_j \geq x_i)$

5 then if $x_i \neq x_j - 1$ then $x_i = x_j - 1$;

$\triangleright$ R3

7 if $(\nexists j \in N_i \mid x_j > x_i)$

8 then if $P_i \neq i$ then $P_i \leftarrow i$;

Figura 2. Algoritmo de Xu y Srimani (2005).

auto-estabilizante, la descripción del funcionamiento del algoritmo se ve a detalle en el Capítulo III.

El algoritmo consiste únicamente de tres reglas. Las reglas se llaman R1, R2 y R3 y las antecede un triangulo antes de las líneas 1, 4 y 7, respectivamente.

Cada una de las reglas del algoritmo tiene un guardia o predicado que busca satisfacer. El guardia es una condición que el algoritmo debe cumplir para poder ejecutar alguna acción. Un algoritmo auto-estabilizante puede tener más de un guardia en cada regla, este tipo de guardia es un guardia compuesto.

Los guardias de la regla R1 están dados en las líneas 1, 2 y 3, a continuación se describen con el mismo orden.

- $\exists j, k \in N(i) \mid j \neq k \wedge x_j \geq x_i \wedge x_k \geq x_i$
- $x_i \neq max_{j \in N(i)} + 1$

- $P_i \neq i$

El guardia de la línea 1 junto con el de la 2 y el de la 1 junto con el de la 3 forman un par de guardias compuestos, cada uno vigilando una acción.

- $x_i \leftarrow max_{j \in N(i)} + 1$. Vigilado por los guardias de las líneas 1 y 2.
- $P_i \leftarrow i$. Vigilado por los guardias de las líneas 1 y 3.

Los guardias de la regla R2 están dados en las líneas 4, 5 y 6 y se reescriben a continuación.

- $\exists! j \in N(i) | x_j \geq x_i$
- $x_i \neq x_j - 1$
- $P_i \neq j$

Los tres guardias mencionados resultan en un par de guardias compuestos, cada uno vigilando una acción.

- $x_i \leftarrow x_j - 1$. Vigilado por los guardias de las líneas 4 y 5.
- $P_i \leftarrow j$. Vigilado por los guardias de las líneas 4 y 6.

Los guardias de la regla R3 están dados en las líneas 7 y 8 y se reescriben a continuación.

- $\nexists j \in N(i) | x_j \geq x_i$
- $P_i \neq i$

Los dos guardias mencionados resultan en un guardia compuesto, vigilando la siguiente acción.

- $x_i \leftarrow i$

Si algún nodo i satisface el guardia de alguna regla, entonces el nodo i es candidato bajo esa regla; a dicho nodo se le conoce como nodo candidato o nodo habilitado. En un instante, puede haber más de un nodo candidato. Los nodos pueden ser candidatos por cualquiera de las tres reglas. Si a un nodo cadidato lo elige el demonio central, entonces se convierte en un nodo privilegiado. El demonio central o calendarizador es una abstracción utilizada para elegir un nodo del conjunto de nodos candidatos para permitirle ejecutar una acción. Sólo los nodos privilegiados pueden ejecutar alguna acción. De esta forma un nodo candidato no puede ejecutar una acción a menos que haya sido elegido por el demonio central. Las acciones son una fracción de código que pueden ejecutar los nodos que satisfagan las condiciones de los guardias que vigilan dicha acción.

En el algoritmo de Xu y Srimani si un nodo candidato i ejecuta una acción, es posible que alcance un estado legítimo ocasionando que sus vecinos que antes se encontraban en un estado legítimo, dejen de estarlo. Esta es una característica que se debe tener en cuenta al momento de diseñar algoritmos auto-estabilizantes. Razones como ésta es por las que los algoritmos auto-estabilizantes tienen un orden de tiempo mayor al de los algoritmos que no son auto-estabilizantes y por las cuales es más dificil encontrar su tiempo de convergencia. De hecho existen algoritmos auto-estabilizantes propuestos por algunos autores que no mencionan cotas para el tiempo de convergencia.

Capítulo III

ELECCIÓN DE LÍDER

Un algoritmo básico en el cómputo distribuido es el de elección de líder, ya que otros algoritmos frecuentemente lo utilizan como subrutina para resolver problemas más complejos. El problema de elección de líder consiste en elegir como líder a un elemento cualquiera de un conjunto de candidatos. Este problema tiene una gran cantidad de variaciones y su dificultad depende de las características propias del candidato. Una de las variaciones del problema más difícil de resolver es aquella en la que cada candidato es indistinguible de los demás candidatos, y cuando la comunicación entre ellos está restringida de forma tal que cada candidato pueda comunicarse sólo con sus vecinos inmediatos. Dado un grafo, un líder es un nodo con características únicas que lo distinguen del resto de los nodos.

En redes punto a punto, la capacidad de comunicación de un nodo lo define la topología de red. Los algoritmos de elección de líder varían dependiendo de las características de la topología de red (anillo, árbol, hipercubo u otras). A continuación, en la Sección III.1, se discuten algunos algoritmos de elección de líder en anillos y en la Sección III.2 los algoritmos de elección de líder en grafos de árbol.

III.1 Elección de líder en anillos

En los grafos de anillos, un nodo puede comunicarse directamente sólo con dos nodos. Con base en los estados de estos nodos vecinos y en el propio, el nodo debe tomar una desición y ejecutar alguna acción. Al final, el algoritmo debe ser capaz de elegir

un líder. Existen muchos algoritmos de elección de líder en anillos, a continuación se detallan algunos de estos algoritmos.

III.1.1 Algoritmo de elección de líder de Huang

El algoritmo de (Huang, 1993) permite elegir un líder en un anillo uniforme de tamaño primo. El anillo es uniforme si todos los nodos son lógicamente equivalentes, es decir, son idénticos, no existe una característica que los haga diferenciarse de los demás. Usando el algoritmo de Huang como una primera fase, todo los protocolos auto-estabilizantes para anillos uniformes se pueden implantar como una segunda fase. Lo anterior da como resultado un algoritmo auto-estabilizante de exclusión mútua de dos fases para anillos uniformes.

Sea n el tamaño del anillo. El protocolo de exclusión mutua de Huang utiliza $3n$ estados para cada nodo, mejorando el trabajo previo de (Burns y Pachl, 1989) que utiliza aproximadamente $O(n^2/ln\ n)$ estados por nodo.

Cada nodo tiene una variable *label* que puede tomar valores del conjunto $\{0, ..., (n-1)\}$. La variable *label* inicialmente tiene un valor arbitrario. El algoritmo ajusta esta variable de tal forma que el sistema eventualmente alcanza una configuración en la cual todas las etiquetas son distintas; una vez alcanzada esta configuración, el algoritmo termina. Cuando el algoritmo termina, uno y sólo uno de los nodos debe tener su variable *label* en 0 y asume el rol de líder. Los autores demuestran la convergencia del algoritmo pero no proporcionan una cota en el número de pasos que lleva alcanzarla.

Características del protocolo de elección de líder de Huang

El protocolo propuesto por Huang corre sobre topologías de anillo de tamaño primo. Utiliza $3n$ estados por nodo y considera los nodos del sistema como uniformes. Tiene la

ventaja que al estar diseñado por fases, se simplifica su diseño. La primera fase genera las etiquetas del proceso de elección de líder y requiere $O(n)$ estados. La segunda fase ejecuta el protocolo de exclusión mutua de Dijkstra que requiere tres estados.

III.1.2 Elección de líder de Itkis

(Itkis *et al.*, 1995) realiza un análisis del uso de espacio en algoritmos de elección de líder sobre topologías de anillos, considera el caso de la elección de líder sobre un anillo sin identificadores, por medio de protocolos determinísticos y auto-estabilizantes. El algoritmo de Itkis resuelve el problema de elección de líder en anillos anónimos en $O(n^2)$ pasos.

Descripción del protocolo de Itkis

Se considera un anillo uniforme de tamaño primo. Todos los nodos del anillo son idénticos y no tienen identificadores. Un demonio central elige un nodo habilitado para realizar un movimiento. La meta es elegir un líder único de forma determinística.

El primer componente del estado de cada nodo es una etiqueta binaria. Un segmento es una secuencia máxima continua de etiquetas alternadas. Por ejemplo, 010101 es un segmento de longitud 6. A los nodos de los extremos de cada segmento se les llama nodos principales. Si un nodo se encuentra dentro de los nodos principales, sólo cambia su valor.

En uno de los nodos principales se genera un protocolo para medir la longitud del segmento de forma asíncrona. El protocolo de medición primero marca a cada nodo principal. De esta forma, un nodo principal se marca a si mismo y posteriormente envía un token al otro nodo principal de dicho segmento para poder ubicarlo. Cuando el otro nodo principal recibe el token, éste cede el token al nodo vecino que no le pasó el token.

El vecino cambia de dirección el token y lo regresa al nodo principal que originó dicho token. El proceso se repite iniciando en el otro nodo principal. De esta manera el token marca los nodos principales y se realiza la medición del segmento.

Una vez que se marcan los nodos principales, el segmento entra en modo de intercambio para fusionarse con el otro segmento. En el modo de intercambio, todos los nodos dentro del segmento marcado por los nodos principales cambian sus etiquetas para formar un segmento más grande. El mismo proceso se repite hasta alcanzar la configuración deseada de un único segmento de etiquetas alternas. Los segmentos se decrementan en cada proceso de intercambio. Una vez que sólo hay un único segmento en el anillo, uno de los nodos principales se elige como líder.

III.1.3 Elección de líder en anillos bajo un calendario arbitrario de Beauquier

(Beauquier *et al.*, 2007b) presenta un protocolo auto-estabilizante aleatorio de elección de líder y un protocolo auto-estabilizante aleatorio de token circular bajo un calendarizador arbitrario para anillos anónimos y unidireccionales de cualquier tamaño. Estos protocolos utilizan una cantidad de espacio óptima. Los autores además proponen una técnica nueva de administración del calendarizador a través de la composición del protocolo auto-estabilizante.

Se descompone el protocolo de elección de líder en dos sub-protocolos, el cual llaman composición *crossover*. Un punto interesante con la composición cross-over es que los protocolos que lo componen tienen las mismas propiedades en términos de igualdad. De esta manera, si por alguna razón, las elecciones bajo un calendarizador desigual se hacen de manera equitativa, entonces los cálculos bajo este mismo calendarizador

desigual, son también equitativos.

El modelo y la estrategia de Beauquier

Para analizar un protocolo, el calendarizador se debe considerar un adversario: el protocolo debe trabajar apropiadamente a pesar de un mal comportamiento del calendarizador. La interacción entre el calendarizador y el protocolo se puede ver como un juego. En este juego, la meta del calendarizador es prevenir que el protocolo haga su trabajo.

Dada una configuración inicial, el calendarizador selecciona un nodo del conjunto de nodos habilitados por alguna regla del algoritmo. Posteriormente, el nodo seleccionado ejecuta la acción de la regla correspondiente. Una vez realizados los pasos anteriores, el calendarizador repite este proceso. De esta forma, el calendarizador proporciona la secuencia de pasos o de nodos que han de ejecutar alguna regla del algoritmo.

Elección de líder bajo un calendarizador arbitrario

El objetivo es asignar un cierto color a cada nodo para diferenciar al nodo líder del resto de los nodos. Así, el objetivo del color es congelar al líder cuando éste es único, pero también asegurar la circulación del líder cuando el anillo contiene varios líderes. Cuando un nodo es líder y está privilegiado, escoge aleatoriamente un color. Este color debe difundirse a cada nodo del anillo. En cada paso del algoritmo el nodo que tiene el privilegio puede pasarlo o no a algún nodo vecino.

Debido a que el color se elige aleatoriamente, cuando hay varios líderes en el anillo, un líder podría eventualmente privilegiarse cuando su vecino izquierdo no tenga el mismo color que él. En este caso, el líder cede su privilegio de líder a su vecino derecho. El nodo que recibe el privilegio de líder "supone" que varios líderes coexisten en el

anillo y cede su privilegio de líder que circula de nodo en nodo en el anillo hasta llegar al siguiente líder. De esta forma, el privilegio de líder se mueve hasta que sólo exista un líder. En todos los casos, el privilegio de líder realiza a lo más n movimientos a otros nodos. Una vez que el anillo se estabiliza, hay un líder congelado y un privilegio que circula en el anillo.

III.2 Algoritmos de elección de líder en grafos de árbol

El problema de elección de líder no es exclusivo de los grafos de anillo, así como existen algoritmos de elección de líder para anillos, también los hay para grafos de árbol. En los grafos de árbol, un nodo puede tener en la mayoría de los casos mayores posibilidades de comunicación al tener más vecinos, pero esta característica introduce otras técnicas para la elección del líder. A continuación se detallan algunos algoritmos de elección de líder para grafos de árbol.

III.2.1 Algoritmo auto-estabilizante de elección de líder de Antonoiu y Srimani

En el algoritmo de (Antonoiu y Srimani, 1996) un estado global legítimo es un estado global donde un nodo del árbol se elige líder por medio de la distinción de algunas características únicas. El algoritmo no puede elegir una hoja del árbol como líder, únicamente puede elegir un nodo interno arbitrario como líder.

En este algoritmo (Antonoiu y Srimani, 1996) cada nodo en el árbol tiene una variable *nivel* en el intervalo de $0, 1, ..., C$, donde C es un entero positivo mayor o igual al

número de nodos en el árbol. Las variables *nivel* inicialmente tienen valores arbitrarios; el algoritmo cambia las variables *nivel* de tal forma que el sistema alcanza en tiempo finito, un estado donde existe sólo un nodo arbitrario interno cuya variable *nivel* tiene un valor que es estrictamente mayor que todos los demás nodos. Ese nodo interno se trata como líder. Cuando existen múltiples nodos privilegiados, un subconjunto arbitrario de estos nodos (al menos uno en cada paso) puede ejecutar una acción mientras que los nodos activos no sean adyacentes. El algoritmo no supone ningún orden o regla específica en la cual se elige al conjunto de nodos habilitados en cada paso.

En cualquier estado ilegítimo, al menos un nodo de T está privilegiado. En cualquier estado legítimo, hay un nodo que tiene en su variable *nivel* un valor que es estrictamente máximo, es líder y su valor es menor que n, además cada nodo restante a través de su apuntador sabe en qué dirección se encuentra el líder.

Para un árbol T de n nodos, iniciando en un estado ilegítimo, no se puede construir una secuencia infinita de movimientos de los nodos, por lo tanto, un estado estable se alcanza siempre en tiempo finito.

III.2.2 Algoritmo de elección de líder de Xu y Srimani

(Xu y Srimani, 2005) proponen un algoritmo auto-establizante de elección de líder para grafos de árbol con nodos uniformes. El protocolo de Xu y Srimani es correcto y termina en $O(n^4)$ iniciando desde cualquier estado arbitrario.

Este algoritmo, ver Figura 3, corre sobre un grafo de árbol anónimo empezando desde cualquier estado inicial. Un árbol anónimo es un árbol en el cual los nodos que lo forman carecen de identificadores, es decir, no existe una manera de identificar a los nodos del árbol. Se utiliza n para denotar el número de nodos en el árbol T. Cuando el

algoritmo termina, uno y sólo un nodo se selecciona para ser el líder. Todos los demás nodos deben tener un apuntador seleccionando un nodo vecino. Los apuntadores indican en qué dirección se encuentra el nodo líder.

El algoritmo de Xu y Srimani

Definición 2 *Cada nodo i mantiene una variable entera x_i, y un apuntador en la variable P_i. El estado local de un nodo i lo definen los valores x_i, P_i y el estado global lo define la unión de los estados locales de todos los nodos en el árbol.*

A continuación se detalla el comportamiento de las reglas del algoritmo de Xu y Srimani, ver Figura 3.

- **Regla R1:** Si un nodo i tiene más de un vecino, con valores de x mayores o iguales a la variable x_i, el nodo i se privilegia condicionalmente por la regla R1. Si el nodo i ejecuta la regla R1, x_i toma un valor igual al mayor de sus vecinos más uno y el apuntador P_i apunta hacia sí mismo.

- **Regla R2:** Si un nodo i tiene sólo un vecino j con valor de x_j mayor o igual que x_i, el nodo i se privilegia condicionalmente por la regla R2. Si el nodo i ejecuta la regla R2, x_i toma un valor de una unidad menor que x_j y P_i apunta hacia j.

- **Regla R3:** Si un nodo i tiene en su variable x_i un valor que es estrictamente mayor que cualquiera de sus vecinos y el apuntador del nodo i no está apuntando hacia sí mismo, el nodo i se privilegia condicionalmente por la regla R3, si el nodo i ejecuta la regla R3, el apuntador P_i toma el valor de i.

Un estado global es legítimo cuando existe un nodo i que pertenece al conjunto de nodos cuyo valor x es máximo y su P_i es i, la diferencia de los valores de x entre el

$\triangleright$ R1

1 **if** $(\exists j, k \in N(_i),\ j \neq k \wedge x_j \geq x_i \wedge x_k \geq x_i)$

2 **then if** $x_i \neq \text{máx}_{j \in N_i}(x_j) + 1$ **then** $x_i \leftarrow \text{máx}_{j \in N_i}(x_j) +$

$\triangleright$ R2

4 **if** $(\exists! j \in N_i \mid x_j \geq x_i)$

5 **then if** $x_i \neq x_j - 1$ **then** $x_i = x_j - 1$;

$\triangleright$ R3

7 **if** $(\nexists j \in N_i \mid x_j > x_i)$

8 **then if** $P_i \neq i$ **then** $P_i \leftarrow i$;

Figura 3. Algoritmo de Xu y Srimani (2005).

nodo i y un nodo j es la distancia o número de aristas entre el líder y el nodo j, y cada nodo distinto de i siempre tiene su apuntador seleccionando al líder local.

Xu y Srimani demuestran que el algoritmo termina en un estado legítimo por medio de una demostración por contradicción. En ella demuestran que si el algoritmo termina en un estado global que no es legítimo, debe existir una regla en el algoritmo que lo hace cambiar de estado hasta que eventualmente llega a un estado legítimo. De esta forma, realmente no existe forma de que el algoritmo termine en un estado global ilegítimo. Así cuando el algoritmo termina, un nodo se ha elegido como líder.

Xu y Srimani también demuestran que su algoritmo de elección de líder termina en $O(n^4)$ pasos en el peor caso, empezando desde cualquier estado inicial. Además los autores suponen un calendarizador adversario.

Se puede observar que: La elección de un líder en un árbol no está determinada por la configuración inicial del sistema, éste depende de las elecciones del adversario

malicioso.

Posteriormente, con base en el algoritmo de Xu y Srimani, (Alvarado-Magaña y Fernández-Zepeda, 2007) realizan algunas modificaciones a dicho algoritmo y calculan su tiempo promedio de convergencia, además proponen una adecuación del algoritmo para obtener una versión distribuida de él.

III.2.3 Tiempo de ejecución promedio del algoritmo autoestabilizante de elección de líder de Alvarado y Fernández

El algoritmo de (Alvarado-Magaña y Fernández-Zepeda, 2007) se basa en el algoritmo de Xu y Srimani que encuentra un líder en un grafo de árbol. Los autores realizan dos análisis del tiempo de ejecución del algoritmo. El primer análisis es de forma secuencial, de manera que se privilegia un nodo a la vez y su peor tiempo de ejecución es $O(n^2)$. El segundo análisis permite privilegiar más de un nodo en cada paso y su tiempo de ejecución es menor que el secuencial, siendo $O(n)$ el peor caso. El tiempo depende del tipo de árbol sobre el cual se ejecute el algoritmo. Los algoritmos presentados por Alvarado-Magaña y Fernández-Zepeda suponen un demonio equiprobable en su versión secuencial. El demonio equiprobable puede privilegiar a cualquier nodo candidato con la misma probabilidad. En la versión distribuida del algoritmo, se elimina la necesidad del demonio central y se demuestra que el algoritmo reduce aún más sus tiempos de ejecución para grafos de árbol específicos.

Análisis secuencial del algoritmo de Alvarado y Fernández

En el análisis secuencial se divide al algoritmo en dos fases. La fase 1 consta del análisis del número de pasos necesarios para que todos los nodos del árbol sean estables con respecto a la regla R1, y la segunda fase muestra el análisis de los pasos restantes del algoritmo.

Al inicio del algoritmo se puede ver que las hojas ya son estables con respecto a la regla R1, ya que el algoritmo empieza a estabilizar el árbol de las hojas a la raíz por medio de R1. Una vez que el grafo es estable con respecto a R1, se empieza a estabilizar con respecto a R2.

Básicamente la regla R1 define el máximo global, (aquel nodo que posteriormente será el líder del grafo), la regla R2 genera en los nodos restantes del grafo, el camino hacia el líder, y la regla R3 es útil en caso de que el líder global no esté apuntando a sí mismo.

Análisis distribuido del algoritmo de Alvarado y Fernández

En esta parte se permite al algoritmo privilegiar más de un nodo candidato en cada paso. Para lograr esto, se le agregan a las reglas R1, R2 y R3 unas condiciones extra para no privilegiar nodos vecinos durante la ejecución del algoritmo. De esta manera se elimina al demonio del algoritmo y se facilita el análisis. El nuevo conjunto de reglas se llaman R1+, R2+ y R3+ y resultan en una considerable reducción en el tiempo de ejecución promedio del algoritmo. Cα denota el conjunto de nodos candidatos que satisfacen las condiciones de la regla Rα, donde $\alpha \in \{1, 2, 3\}$.

La ejecución de la regla R1+ tiene mayor prioridad que la regla R2+. Esto ayuda a acelerar la estabilización con respecto a R1+. Las reglas modificadas son las siguientes:

- **Regla R1+:** Si el nodo i tiene al menos dos vecinos, j y k, tal que $x_j \geq x_i \wedge x_k \geq x_i$ y $b_i = 1$ y $(b_l = 0)$ para todos los nodos $l \in C1 \cup N(i)$, entonces el nodo i ejecuta las siguientes operaciones:

  $x_i := max\ \{x_j\}_{j \in N(i)} + 1$

  $P_i := i$

- **Regla R2+:** Si el nodo i tiene exactamente un vecino j tal que $x_j \geq x_i$ y $x_i :\neq x_j - 1$ OR $P_i :\neq j$ y $(N(i) \cup C1 = 0)$ y $(j \in C2)$, entonces el nodo i ejecuta las siguientes operaciones: $x_i := x_j - 1$

  $P_i := j$

- **Rule R3+:** Permanece sin cambios con respecto a R3.

El análisis del tiempo de ejecución del algoritmo se realiza de forma similar al secuencial. Como ya se mencionó, el algoritmo para ciertas topologías comunes de árbol presenta un rendimiento aún mayor.

Capítulo IV

CONTENCIÓN DE FALLAS

Para esta investigación, una falla es una manifestación equívoca de un nivel lógico. Las fallas, groso modo, se pueden clasificar en dos tipos: permanentes y transitorias. Las fallas permanentes son aquellas fallas derivadas de un daño físico en un componente del sistema. Las fallas transitorias aparecen esporádicamente sin causar un daño permanente en el sistema. Las fallas permanentes se solucionan reemplazando al componente que se dañó y son más fáciles de identificar que las fallas transitorias. Las fallas transitorias son más difíciles de detectar que las fallas permanentes, pero se puede lograr que el sistema las corrija por sí mismo sin la ayuda de algún agente externo.

Las técnicas de contención de fallas son muy utilizadas en algoritmos auto-estabilizantes para recuperarse rápidamente de fallas transitorias que los afecten. El objetivo de la contención de fallas es identificar las fallas que afecten al sistema y aislarlas para evitar que contaminen otras partes del sistema, o que los efectos de esas fallas se vean reflejadas en partes "sanas" del sistema. Una vez que se identifican las fallas, se ejecuta alguna acción de corrección para eliminar esas fallas; de esta manera, algunas partes del sistema que no fallaron ni siquiera se dan cuenta que tal falla existió.

Existen algunos parámetros de rendimiento para la contención de fallas.

- **Tiempo de contención:** Es el tiempo necesario, en el peor de los casos, para detectar una falla simple que afecta al sistema.
- **Grado de contaminación** Es el número de nodos, que en el peor de los casos,

cambiaron su estado durante la recuperación de una falla simple.

- **Brecha de falla:** Es el tiempo, que en el peor de los casos, le lleva al sistema recuperarse de una falla simple que lo afecte una vez que éste había alcanzado un estado legal. Otra forma de ver la brecha de falla es como el intervalo que pasa entre el instante que una falla afecta al sistema y el instante en que el sistema está disponible para manejar la siguiente falla con la misma eficiencia.

Existen cuatro técnicas principales de contención de fallas. En las secciones siguientes se da una descripción de algunos algoritmos que utilizan dichas técnicas.

IV.1 Algoritmo probabilístico de contención de fallas

(Dasgupta *et al.*, 2007) proponen un algoritmo probabilístico de contención de fallas. El algoritmo que proponen confina el efecto de una falla transitoria simple a los nodos vecinos inmediatos del nodo que falla, con un tiempo esperado de recuperación de $O(\Delta^3)$, donde Δ es el grado máximo de un nodo en el sistema. El aspecto más importante del algoritmo es la *brecha de falla*. La brecha de falla depende de Δ, y es independiente del tamaño de la red. De esta forma se puede observar que mientras más pequeña sea la brecha de falla, más tiempo está disponible el sistema, ya que pasa menos tiempo resolviendo las fallas que lo afectan.

Cabe señalar que si ocurre una falla mientras se está corrigiendo la falla anterior, al algoritmo le cuesta un tiempo mucho mayor el poder realizar las correcciones necesarias y la brecha de falla se degrada.

La contención de fallas utilizada no identifica plenamente el nodo que falla, sólo

encuentra una inconsistencia local. Una inconsistencia local es un estado local ilegítimo detectado por el nodo que falla y los nodos vecinos del nodo que falla. Una vez detectada la inconsistencia, el algoritmo elige aleatoriamente a cualquiera de esos nodos para que ejecute alguna acción que intente corregir la falla. Si el algoritmo elige al nodo que falló, entonces la recuperación es mucho más rápida que si elige un nodo vecino.

El algoritmo considera el protocolo del bit persistente para demostrar la contención de fallas. En el protocolo del bit persistente, un conjunto de nodos mantienen una réplica del valor de un bit 0 o 1 a través de una red. Sea G un grafo, el valor del bit se guarda en una variable v, así el grafo G está en un estado legítimo si y sólo si en el grafo G el valor de v es igual para todos los nodos. Adicionalmente cada nodo mantiene una variable x que utiliza como prioridad para determinar si un nodo puede o no ejecutar una acción para intentar corregir la falla que lo afecte. Si en algún momento ocurre una falla en un nodo i y el valor de v_i cambia, el nodo detecta que el valor de su bit y el de su vecino j son diferentes, pero no puede determinar quién de los dos tiene un valor erróneo.

Para exponer la contención de fallas del algoritmo se supone que $G \equiv \forall j : v_j = 1$, es decir, Para todo nodo j del grafo G el valor de la variable v de cada nodo j perteneciente a G debe ser igual a 1. En algún momento, una falla simple cambia v_i a 0, y x_i a algún valor desconocido. Si el valor de la variable prioridad del nodo que falló incrementa su valor sobre el de todos sus vecinos, entonces el nodo i cambia de nuevo su valor de v_i y la falla se resuelve en un número de pasos constante. En caso de que la variable prioridad cambie su valor x_i por uno cuyo valor sea el más pequeño en N_i, los vecinos del nodo i son mejores candidatos para cambiar el valor de su variable v antes de que el nodo i ejecute una acción para intentar corregir el error, en este caso al algoritmo le toma un número mayor de pasos ($O(\Delta)$) corregir la falla.

IV.2 Contención de fallas en protocolos auto-estabilizantes distribuidos

(Ghosh *et al.*, 2007) proponen un protocolo auto-estabilizante que construye un árbol de esparcimiento en una red de n nodos. Un árbol de esparcimiento de un grafo G es un subgrafo de G que contiene todos los nodos de G y el mínimo número de aristas de G tal que el subgrafo esté conectado. El algoritmo empieza designando un nodo como la raíz del árbol de esparcimiento que se ha de construir. Al nodo raíz se van agregando nodos, hasta completar todos los nodos de la red. A la raíz se le asigna el valor de 0 y a cada nodo que va agregando al árbol de esparcimiento, le asigna un número secuencial ascendente, además el apuntador del nodo que se agrega apunta al nodo que se agregó en un paso anterior. De esta manera, ningún nodo puede tener un número mayor a n.

Ahora si una falla afecta al árbol de esparcimiento, la falla cambia el valor del nodo y ya no concuerda con el valor que le corresponde. El algoritmo nota dicha inconsistencia e instantáneamente ejecuta alguna acción de corrección. Para determinar qué acción ejecutar, el algoritmo necesita obtener información de los nodos vecinos al nodo que falló y realizar la acción de corrección de la falla. Así el algoritmo corrige una falla simple que lo afecte en un tiempo constante.

La estrategia utilizada para la contención de fallas del algoritmo es asignar diferentes prioridades a los guardias de las acciones. De esta manera, el nodo que falla se le permite ejecutar acciones de recuperación antes que sus vecinos. La asignación de prioridades depende del tipo de inconsistencias originadas por las fallas. Básicamente el algoritmo asigna la prioridad de mayor a menor si el nodo que falla es inconsistente con respecto a todos sus vecinos, con respecto a los nodos hijos o con respecto a su nodo padre. Para cada caso mencionado, el algoritmo ejecuta una acción diferente, de esta forma el

algoritmo detecta con exactitud el nodo que falló y el valor que le corresponde.

IV.3 Auto-estabilización y contención de fallas utilizando calendarizador por prioridad

El modelo utilizado en este calendarizador por prioridad de (Ghosh y He, 2000) es un sistema distribuido, donde cada nodo puede comunicarse sólo con sus vecinos de acuerdo al modelo de memoria compartida. El modelo de memoria compartida supone que existen conjuntos de nodos que comparten una misma memoria, en este modelo cada nodo puede leer los datos que otros nodos escriben. El algoritmo para cada nodo consiste en un conjunto de reglas que está en función de las variables de los nodos vecinos y las de él mismo. Una acción del algoritmo sólo puede modificar el estado local del nodo que ejecuta la acción. La ejecución de las acciones es conforme a una regla de prioridad.

La ventaja de utilizar un calendarizador por prioridad se puede mostrar con el algoritmo de (Chen *et al.*, 1991) el cual contruye un árbol de esparcimiento. Este algoritmo auto-estabilizante corre sobre grafos conectados no dirigidos. Primero se elige un nodo como la raíz del árbol de esparcimiento y se van agregando nodos mediante las aristas que los conectan. Cada nodo mantiene una variable interna que se incrementa a medida que se agregan los nodos. Esta variable denota la distancia entre el nodo y la raíz. La distancia es el número de aristas entre el nodo y la raíz. Este algoritmo no se recupera de una falla simple en tiempo constante.

El algoritmo de Ghosh y He asigna diferentes prioridades a las acciones del algoritmo de Chen Yu y otros. De esa manera se le permite al algoritmo ejecutar las acciones de

corrección en un cierto orden, así la falla se corrige en tiempo constante. La prioridad asignada depende del número de inconsistencias que detecten los nodos que "observan" la falla. A mayor número de inconsistencias detectadas, mayor prioridad tiene el nodo de ejecutar la acción de corrección para eliminar la falla.

IV.4 Tiempo adaptativo de auto-estabilización

Un algoritmo auto-estabilizante con contención de fallas es adaptativo cuando el tiempo de recuperación para cualquier falla transitoria que lo afecte es igual al tiempo óptimo de recuperación para dicha falla (Dolev y Herman, 2007).

(Kutten y Patt-Shamir, 1997) consideran el problema del bit persistente. Ellos presentan un algoritmo que se recupera de f fallas para cualquier valor de $f < n/2$. El algoritmo corre sobre un grafo no dirigido de tamaño n en $O(diam)$, donde $diam$ es el diámetro del grafo.

El algoritmo considera que cada nodo mantiene una réplica fiel del valor inicial de cada nodo en el grafo. Estos valores se utilizan para calcular el estado local de un nodo por una regla de mayoría. Como se puede observar, f menor que n/2, por lo tanto se tienen por lo menos $f+1$ nodos libres de falla, de esta manera siempre se puede aplicar la regla de mayoría.

El algoritmo cuando corrige fallas realiza un broadcast en el grafo y verifica la integridad de la información del grafo. Si se detectan errores, el algoritmo realiza otro broadcast para regresar la información del grafo a su estado inicial. De esta manera el algoritmo alcanza el mismo estado legítimo que alcanzó en un principio. Un broadcast es un paquete de información dirigido a todos los nodos del grafo.

IV.5 Algoritmos súper-estabilizantes

En este capítulo se han mencionado algunas maneras de contener fallas en algoritmos auto-estabilizantes. Existe otro paradigma, además de la contención de fallas, que ayuda a los algoritmos auto-estabilizantes a recuperarse de fallas que lo afecten (Herman, 2000). Uno de estos paradigmas es el súper-estabilizador. Para que un algoritmo sea súper-estabilizante debe cumplir dos condiciones:

- Puede recuperarse de graves fallas transitorias arbitrarias.
- El sistema permanece disponible incluso durante el proceso de recuperaración de las fallas transitorias que lo afectan.

Los algoritmos súper-estabilizantes continúan la tendencia de la auto-estabilización agregando otra forma de manejar las fallas (Dolev y Herman, 1997). Los algoritmos súper-estabilizantes combinan los beneficios de los algoritmos auto-estabilizantes con los beneficios de los algoritmos diseñados para cambios de topología dinámicos. Un algoritmo súper-estabilizante no trata los cambios de topología como fallas transitorias que pueden llevar al sistema a un estado arbitrario. El algoritmo utiliza este evento para que los nodos vecinos se ajusten a la nueva topología sin violar el estado global. Un cambio de topología es un evento detectado por sus nodos vecinos.

Existen algunas métricas para evaluar el desempeño de los algoritmos súper-estabilizantes. Empezando en un estado legítimo y considerando un solo cambio de topología, se definen:

- *Tiempo de súper-estabilización:* número de pasos que le toma al algoritmo alcanzar un estado legítimo.

- *Medida de ajuste:* número máximo de nodos que deben cambiar sus estados locales para alcanzar un estado legítimo.

IV.6 Contención de fallas del algoritmo propuesto

En el algoritmo propuesto en este trabajo de tésis, cada nodo puede leer sólo sus variables y las de sus vecinos inmediatos. Para analizar la contención de fallas del algoritmo propuesto se distinguen dos casos:

- La falla ocurre cuando el algoritmo aún no alcanza un estado legítimo.
- La falla ocurre una vez que el algoritmo alcanzó un estado legítimo.

En el primer caso, al momento de ocurrir la falla, el algoritmo no altera el tiempo que le lleva alcanzar un estado global legítimo. Lo anterior se debe a que el algoritmo, al no alcanzar aún un estado legítimo, retrasa el proceso de auto-estabilización a un estado anterior al que tenía antes de la falla, ya que habrá la misma cantidad de fallas que había un paso antes. La falla sólo se detecta cuando ocurre en un nodo que en ese momento se consideraba estable, si la falla ocurre en cualquier otro nodo, ésta no se detecta.

En el segundo caso, cada nodo con base en información local determina si falló o no; con esa misma información determina la acción que debe ejecutar para intentar corregir la falla. Cada nodo para determinar si falló o no, calcula una *razón de error* (los detalles de la razón de error se discuten en el Capítulo V). Sólo el nodo que determine estar en un estado erróneo es candidato a ejecutar una acción para intentar corregir la falla.

En el peor de los casos el número de pasos que le lleva al algoritmo recuperarse de una falla transitoria simple es constante; el análisis se realiza en el Capítulo V.

La contención de fallas del algoritmo propuesto se asemeja a la contención de fallas utilizada en el algoritmo de (Dasgupta *et al.*, 2007) y al algoritmo de (Ghosh y He, 2000) descritos en las Secciones IV.2 y IV.3, respectivamente. El algoritmo de (Ghosh *et al.*, 2007) asigna prioridades a los guardias de las acciones del algoritmo. La prioridad asignada depende de las inconsistencias detectadas localmente por cada nodo que "observa" la falla y con respecto a qué nodo vecino están dadas las inconsistencias. El algoritmo de (Ghosh *et al.*, 2007) asigna prioridades a las reglas del algoritmo, para contener las fallas con base en el número de inconsistencias locales que los nodos "observan". El algoritmo propuesto en esta tesis calcula una *relación de error* en base a las inconsistencias locales que los nodos "observan", esta razón de error se puede ver como una prioridad. Cuanto mayor sea la razón de error de un nodo, mayor es la prioridad que tiene de ejecutar una acción para intentar corregir la falla.

Capítulo V

ALGORITMO DE CONTENCIÓN DE FALLAS

El algoritmo auto-estabilizante de contención de fallas propuesto en este capítulo está basado en el algoritmo de Xu y Srimani y corre sobre un grafo de árbol anónimo. Como ya se mencionó en el Capítulo III, un algoritmo de elección de líder elige un nodo entre un conjunto de candidatos y al nodo elegido se le asignan algunos atributos únicos que no comparte con ningún otro nodo en el grafo. Los nodos restantes tienen un apuntador local que apunta a un nodo vecino, el cual está sobre la trayectoria que va hacia el nodo líder en el árbol.

Para el algoritmo de Xu y Srimani, el estado local de un nodo i está definido por el par de variables x_i, P_i. Para el algoritmo diseñado en el presente trabajo, el estado local lo definen, además de x_i y P_i, la variable $inconsistencia_i$ que se actualiza en cada paso del algoritmo, justo antes de verificar si existen nodos candidatos bajo alguna de las reglas del algoritmo. La variable $inconsistencia_i$ determina el número de inconsistencias que detecta el nodo i con respecto a los nodos pertenecientes a $N(i)$. El número de inconsistencias es la cantidad de cambios requeridos en los nodos vecinos del nodo i para que el nodo i esté en un estado estable. El valor de la inconsistencia está dada por: $inconsistencia_i = \lceil \frac{número\ de\ inconsistencias\ \times\ 10}{número\ de\ vecinos_i\ +\ 1} \rceil$. De esta forma, mientras más alto sea el valor de *inconsistencia* de un nodo, mayor es la relación de errores detectados con respecto a sus vecinos.

V.1 Algoritmo propuesto

El algoritmo propuesto consiste de tres reglas generales, similares a las de Xu y Srimani con algunas modificaciones que permiten al algoritmo contener una falla transitoria simple. Una vez que el nodo i determina que existen inconsistencias entre él y sus vecinos, determina bajo cuál de las tres reglas del algoritmo él es candidato. Si el nodo i no es candidato bajo alguna regla, $inconsistencia_i$ toma el valor de cero; por lo anterior, sólo los nodos que cumplan con alguna de las reglas de Xu y Srimani y detecten una mayor cantidad de inconsistencias en su vecindario son candidatos bajo dicha regla. Las modificaciones añadidas a las tres reglas, le permiten al algoritmo propuesto tomar decisiones que no toma el algoritmo de Xu y Srimani. Con base en las modificaciones mencionadas, el algoritmo incorpora la contención de fallas.

A continuación se muestra el efecto de las reglas del algoritmo una vez que el árbol alcanza un estado legal y se ve afectado por una falla.

V.1.1 Contención de fallas

El análisis de la contención de fallas del algoritmo propuesto se divide en tres grandes partes:

1. Fallas del nodo líder.

2. Fallas de los nodos hoja.

3. Fallas de nodos internos.

Fallas del nodo líder

En la Figura 4 se muestra un ejemplo de un grafo de árbol en un estado legal. Para mostrar los efectos del algoritmo ante fallas que afecten al nodo líder, la Figura 4 se toma como referencia.

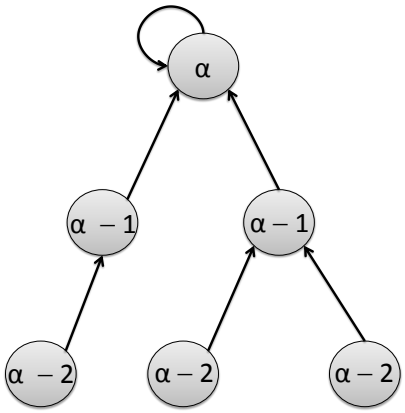

Figura 4. Ejemplo de árbol estable.

Al verse afectado por una falla, el nodo líder puede observar un cambio relativo en sus variables internas x_i, P_i o ambas. Las variables x_i, P_i e $inconsistencia_i$ definen el estado local del nodo, de esta manera, una falla que produzca un cambio en los valores de x_i y P_i conduce al árbol a un estado ilegal. Un cambio en la variable $inconsistencia_i$ no conduce al árbol a un estado ilegal, ya que aunque el valor de esta variable cambie, el algoritmo la vuelve a calcular antes de privilegiar algún nodo, corrigiendo así cualquier error en dicha variable. Las variables x_i y P_i pueden cambiar de diferentes formas: x_i incrementa o decrementa su valor y P_i cambia su valor apuntando a otro nodo, pero siempre dentro de $N(i) \cup i$. Así las fallas que pueden afectar al nodo líder se engloban en siete casos. Los casos se enlistan a continuación:

1. Si como consecuencia de una falla, x_i se incrementa una o más unidades y P_i

permanece sin cambios. Ver Figura 5a.

2. Si como consecuencia de una falla, x_i se incrementa una o más unidades y P_i deja de apuntar a sí mismo. Ver Figura 5b.

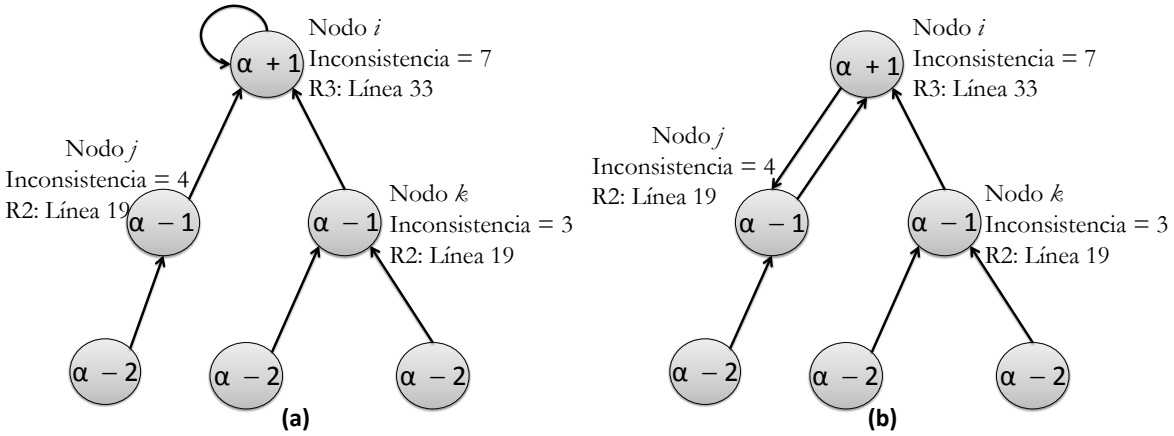

Figura 5. Falla en un nodo líder: a) Caso 1; b) Caso 2.

3. Si como consecuencia de una falla, x_i permanece sin cambio pero P_i deja de apuntar a si mismo. Ver Figura 6

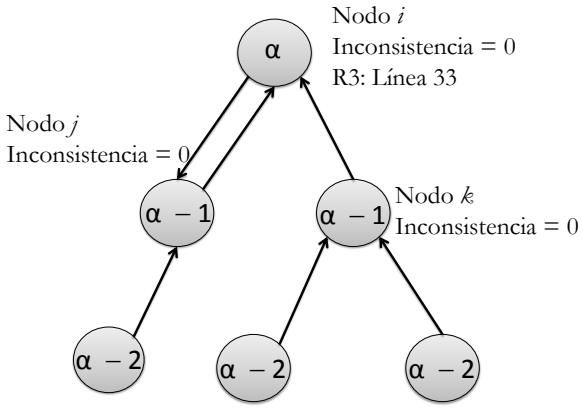

Figura 6. Falla en un nodo líder: Caso 3.

.

4. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i permanece sin cambios. Ver Figura 7a.

5. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i deja de apuntar a sí mismo. Ver Figura 7b.

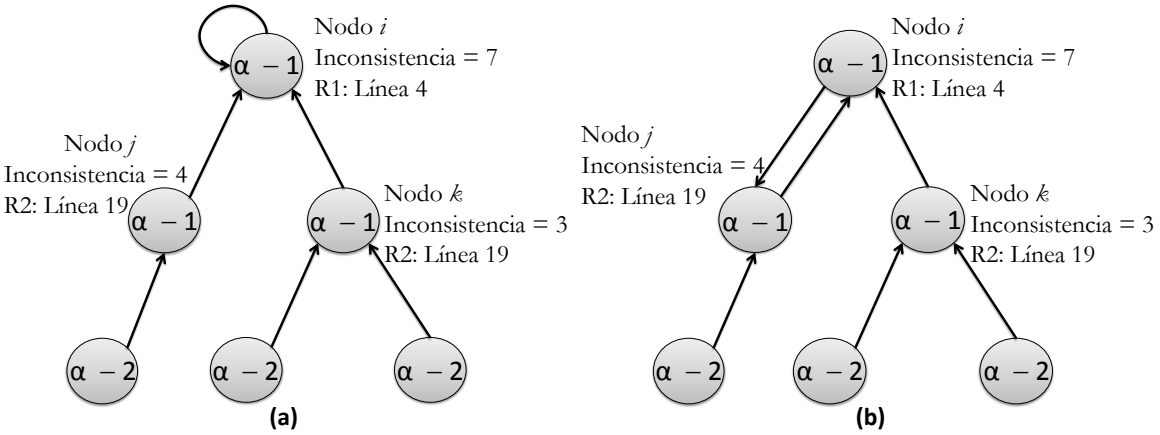

Figura 7. Falla en un nodo líder: a) Caso 4; b) Caso 5.

6. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i permanece sin cambios. Ver Figuras 8a a 11a.

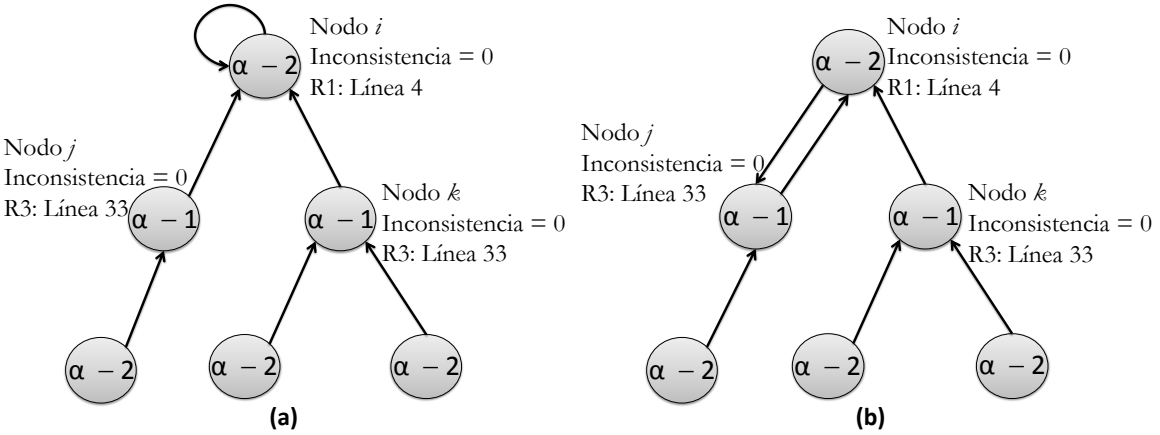

Figura 8. Falla en un nodo líder: a) Caso 6; b) Caso 7.

7. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i deja de apuntar a sí mismo. Ver Figuras 8b a 11b.

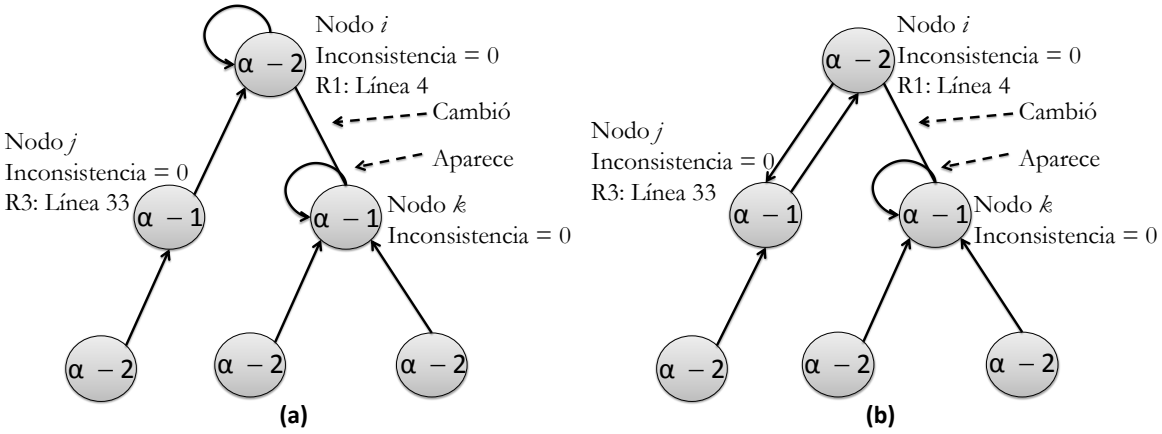

Figura 9. Falla en un nodo líder: a) Caso 6; b) Caso 7. Paso 1.

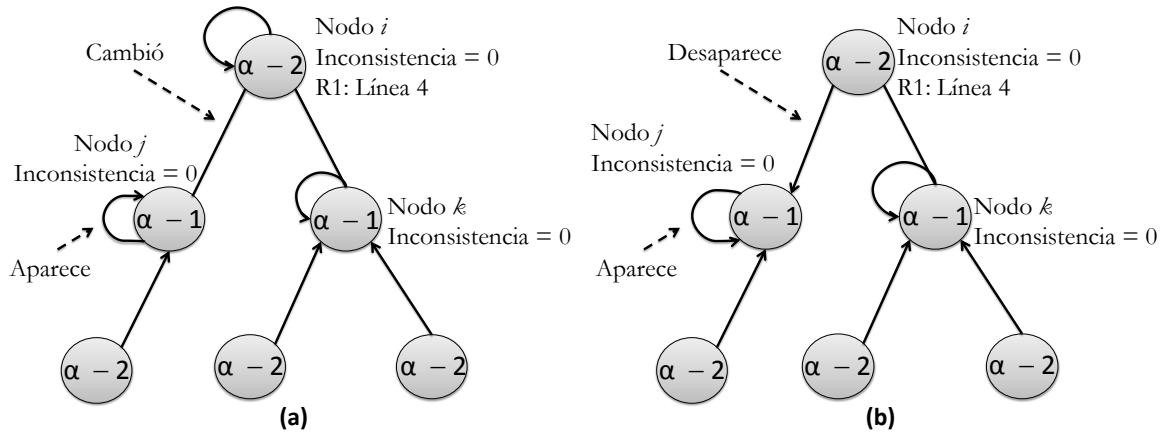

Figura 10. Falla en un nodo líder: a) Caso 6; b) Caso 7.Paso 2.

Las fallas de los casos 6 y 7 se corrigen en un paso si el nodo que falla se elige para efectuar una acción de corrección. En el peor de los casos, la falla se corrige en 5 pasos. El peor caso se da cuando el nodo i es el último de los tres nodos (i, j, k) en ejecutar alguna regla del algoritmo. Los pasos 4 y 5 requeridos por el árbol para recuperarse de la falla, es la ejecución de la regla R2 por los nodos j y k.

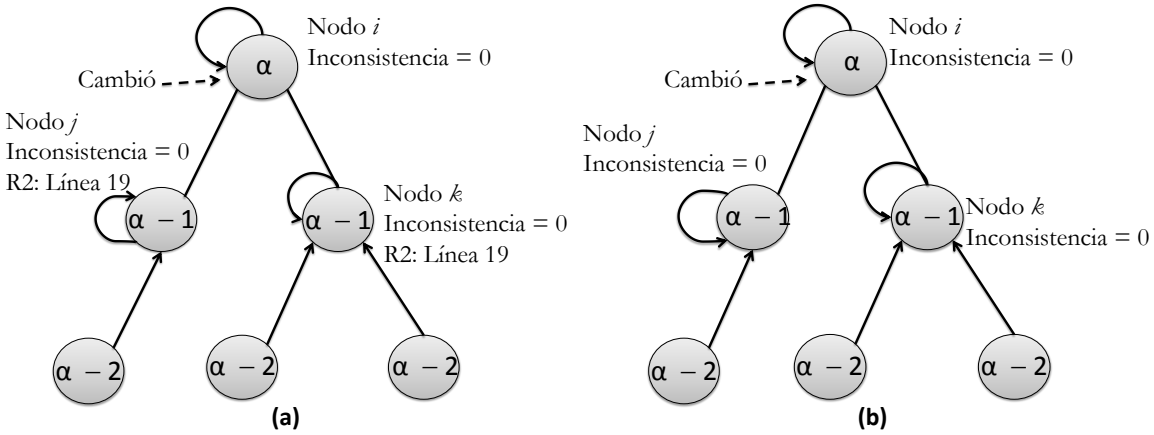

Figura 11. Falla en un nodo líder: a) Caso 6; b) Caso 7. Paso 3.

Los casos expuestos corrigen la falla mediante la ejecución de alguna acción del algoritmo; dicha acción se señala en la figura del caso correspondiente. El nodo privilegiado por alguna de las reglas del algoritmo es aquel cuya variable $inconsistencia_i$ es la mayor de las mostradas en cada figura.

Los casos de las fallas de los nodos líder se resumen en la Tabla II.

Tabla II. Fallas de los nodos líder.

Caso	x_i	P_i	Figura
1	Incrementa una o más unidades	Permanece sin cambios	5a
2	Incrementa una o más unidades	Apunta a otro lado	5b
3	Permanece sin cambios	Apunta a otro lado	6
4	Se decrementa una unidad	Permanece sin cambios	7a
5	Se decrementa una unidad	Apunta a otro lado	7b
6	Se decrementa dos o más unidades	Permanece sin cambios	8a a 11a
7	Se decrementa dos o más unidades	Apunta a otro lado	8b a 11b

La ejecución de alguna regla por parte de un nodo regresa al árbol a la misma configuración que tenía antes de que ocurriera la falla.

Fallas de los nodos hoja

Las fallas de los nodos hojas se dividen en los casos donde el padre del nodo hoja que falla tiene únicamente un hijo y el caso cuando tiene dos a más hijos.

Las Figuras 12a y 12b muestran un fragmento de árbol estable en el que se basa el análisis de los nodos hojas.

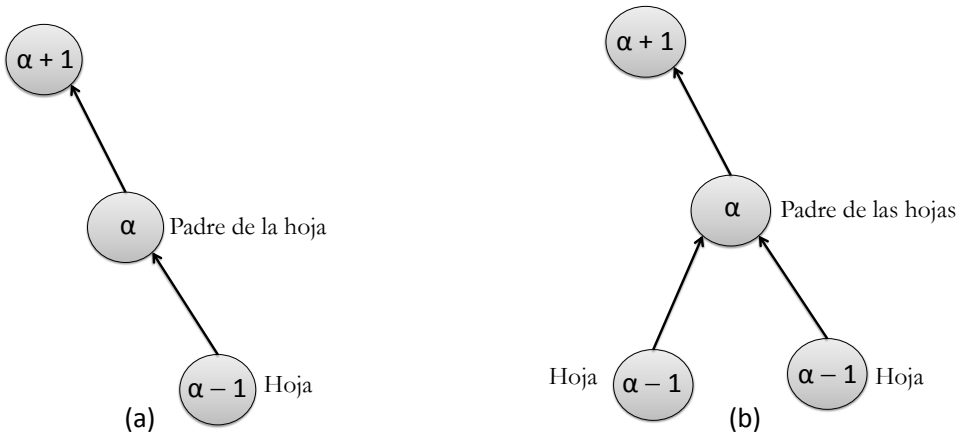

Figura 12. Fallas en nodos hoja: a) Sin hermanos; b) Con hermanos.

Las fallas que pueden afectar los nodos hoja se enlistan a continuación:

1. Si como consecuencia de una falla, x_i se incrementa dos o más unidades y P_i permanece sin cambios. Ver Figuras 13a y 13b.

2. Si como consecuencia de una falla, x_i se incrementa dos o más unidades y P_i deja de apuntar al padre. Ver Figuras 14a y 14b.

3. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i permanece sin cambios. Ver Figuras 15a y 15b.

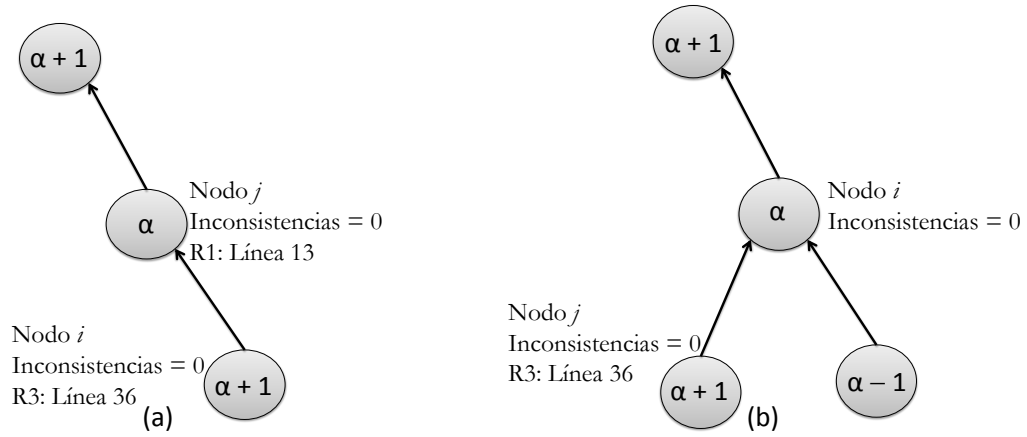

Figura 13. Falla en nodo hoja caso 1: a) Sin hermanos; b) Con hermanos.

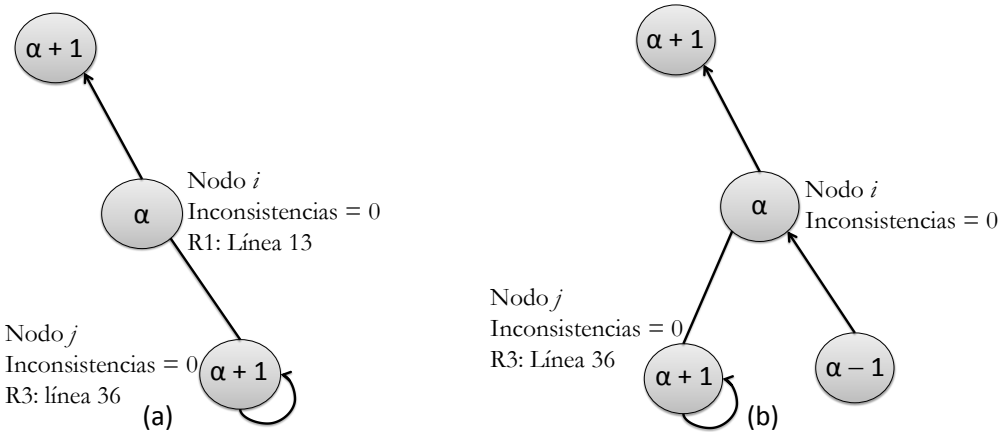

Figura 14. Falla en nodo hoja caso 2: a) Sin hermanos; b) Con hermanos.

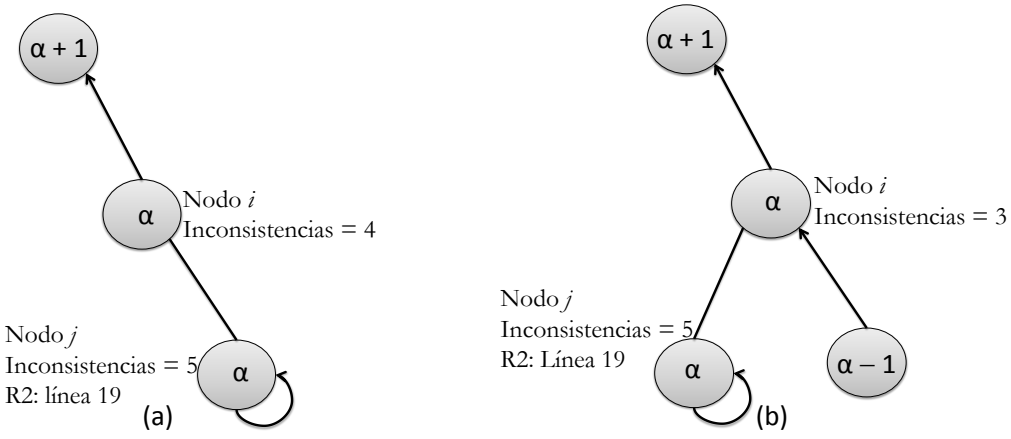

Figura 15. Falla en nodo hoja caso 3: a) Sin hermanos; b) Con hermanos.

4. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i deja de apuntar al padre. Ver Figuras 16a y 16b.

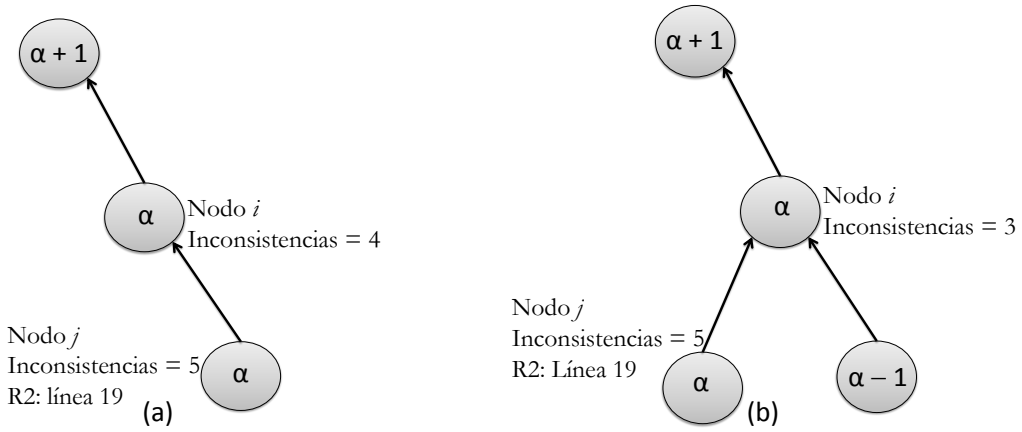

Figura 16. Falla en nodo hoja caso 4: a) Sin hermanos; b) Con hermanos.

5. Si como consecuencia de una falla, x_i permanece sin cambios y y P_i deja de apuntar al padre. Ver Figuras 17a y 17b.

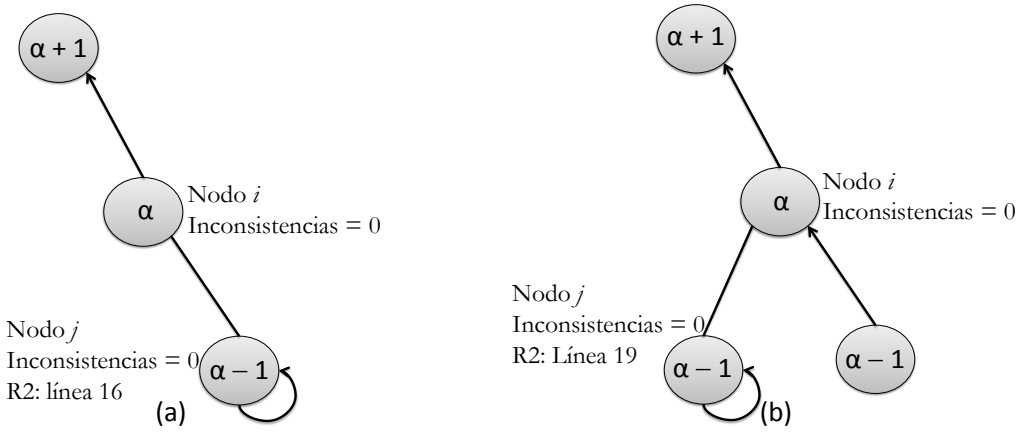

Figura 17. Falla en nodo hoja caso 5: a) Sin hermanos; b) Con hermanos.

6. Si como consecuencia de una falla, x_i se decrementa una o más unidades y P_i permanece sin cambios. Ver Figuras 18a y 18b.

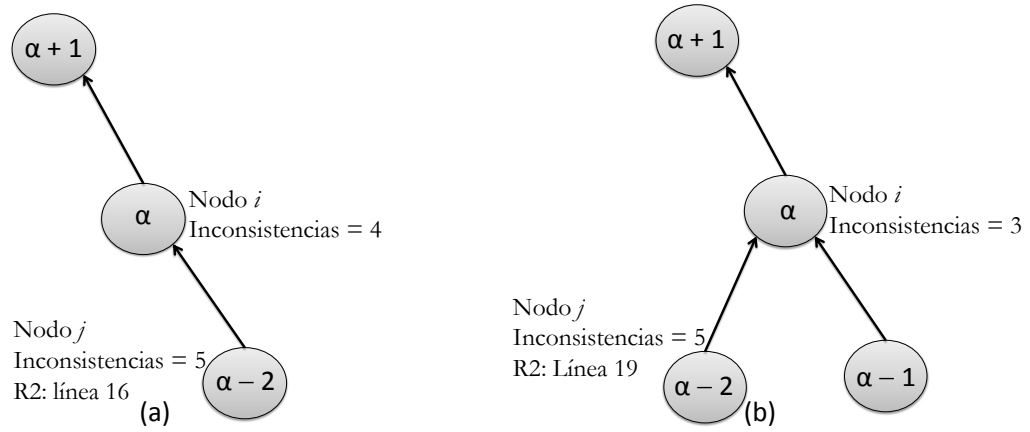

Figura 18. Falla en nodo hoja caso 6: a) Sin hermanos; b) Con hermanos.

7. Si como consecuencia de una falla, x_i se decrementa una o más unidades y P_i deja de apuntar al padre. Ver Figuras 19a y 19b.

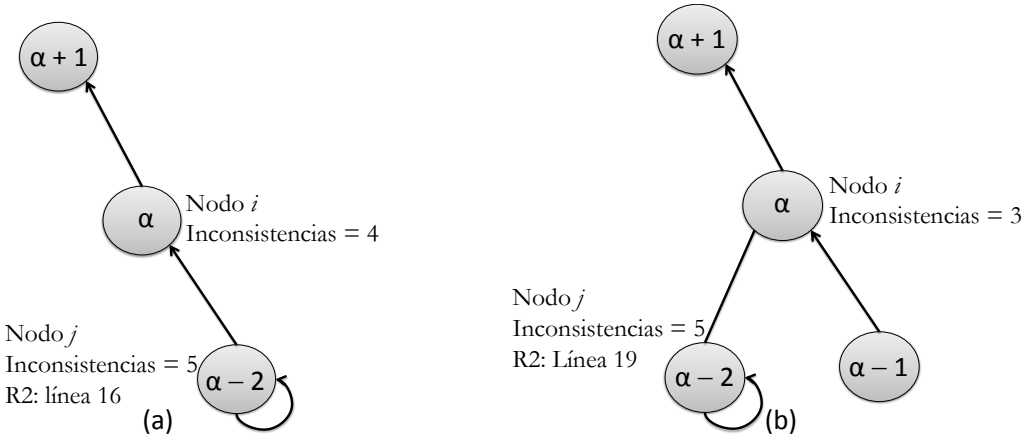

Figura 19. Falla en nodo hoja caso 7: a) Sin hermanos; b) Con hermanos.

Estos casos se resumen en la Tabla III. Los siete casos enlistados se resuelven de la misma forma (línea del algoritmo señalada) si el padre del nodo hoja que falla tiene uno o mas hijos.

Tabla III. Fallas de los nodos hoja.

Caso	x_i	P_i	Figura
1	Se incrementa dos o más unidades	Permanece sin cambios	13
2	Se incrementa dos o más unidades	Apunta a otro lado	14
3	Se incrementa una unidad	Permanece sin cambios	15
4	Se incrementa una unidad	Apunta a otro lado	16
5	Permanece sin cambios	Apunta a otro lado	17
6	Se decrementa una o más unidades	Permanece sin cambios	18
7	Se decrementa una o más unidades	Apunta a otro lado	19

Fallas de los nodos internos

Para el análisis de las fallas que afectan a los nodos internos se divide el análisis en tres partes. Cuando el nodo que falla tiene un sólo hijo, cuando tiene dos hijos y cuando tiene más de dos hijos.

La Figura 20 muestra un fragmento de árbol estable en el que se basa el análisis de los nodos internos.

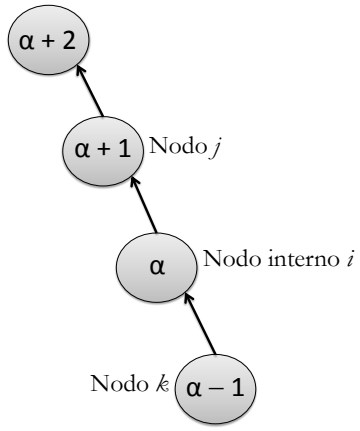

Figura 20. Falla en nodo interno con un solo hijo: ejemplo.

Los casos que pueden generar las fallas de los nodos intermedios son nueve. Los nueve casos se enlistan a continuación:

1. Si como consecuencia de una falla, x_i se incrementa dos unidades o más y P_i permanece sin cambios. Ver Figuras 21 a 24.

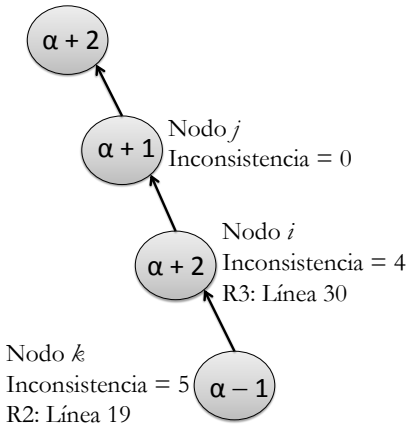

Figura 21. Falla en nodo interno con un solo hijo: caso 1.

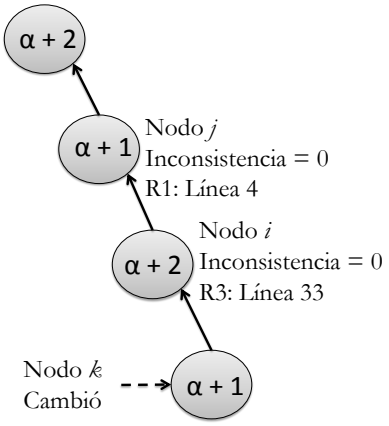

Figura 22. Falla en nodo interno con un solo hijo: caso 1. Paso 1.

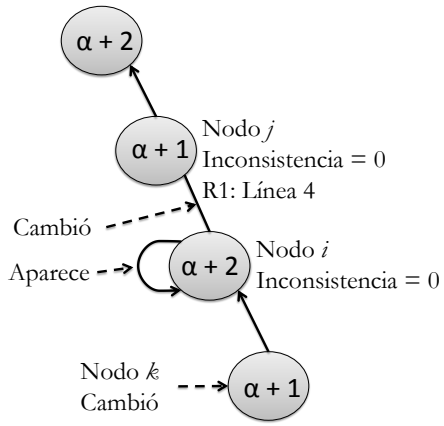

Figura 23. Falla en nodo interno con un solo hijo: caso 1. Paso 2.

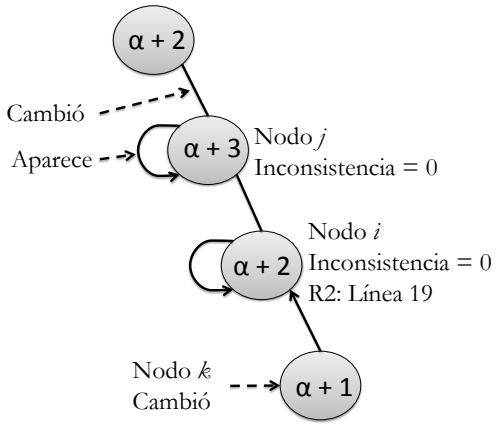

Figura 24. Falla en nodo interno con un solo hijo: caso 1. Paso 3.

2. Si como consecuencia de una falla, x_i se incrementa dos unidades o más y P_i deja de apuntar a sí mismo. Ver Figura 25.

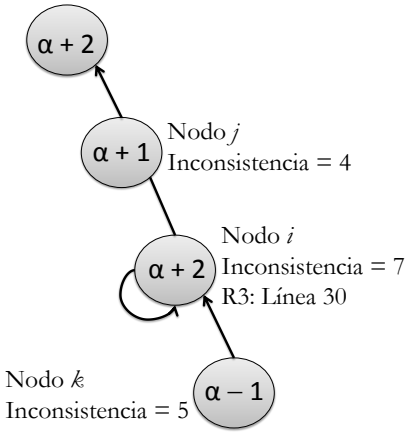

Figura 25. Falla en nodo interno con un solo hijo: caso 2.

3. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i permanece sin cambios. Ver Figura 26.

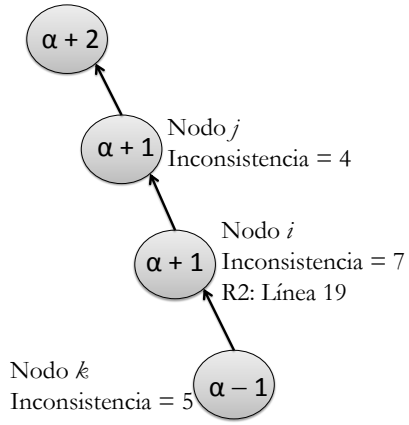

Figura 26. Falla en nodo interno con un solo hijo: caso 3.

4. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i deja de apuntar a si mismo. Ver Figura 27.

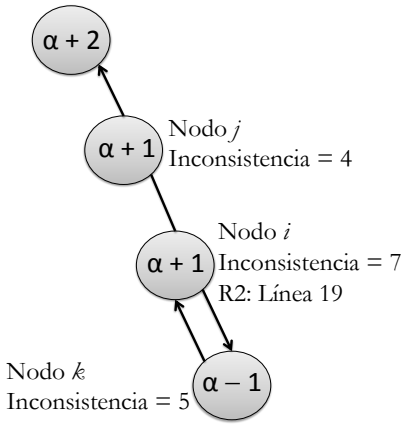

Figura 27. Falla en nodo interno con un solo hijo: caso 4.

5. Si como consecuencia de una falla, x_i permanece intacto pero P_i deja de apuntar a si mismo. Ver Figura 28.

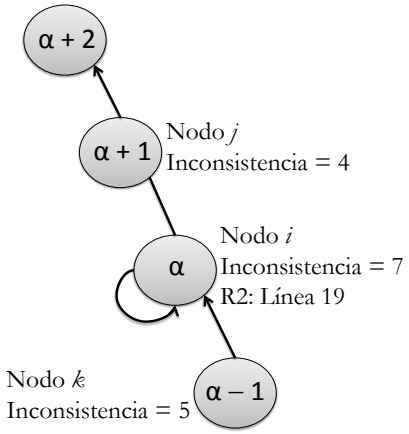

Figura 28. Falla en nodo interno con un solo hijo: caso 5.

6. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i permanece sin cambios. Ver Figura 29.

7. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i deja de apuntar a si mismo. Ver Figura 30.

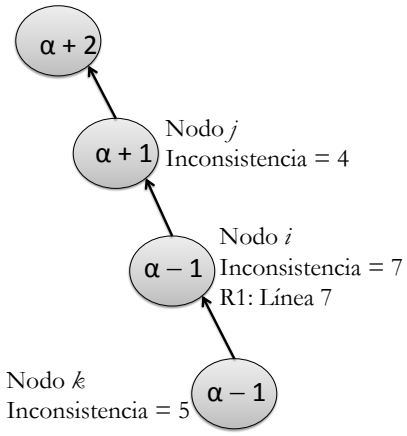

Figura 29. Falla en nodo interno con un solo hijo: caso 6.

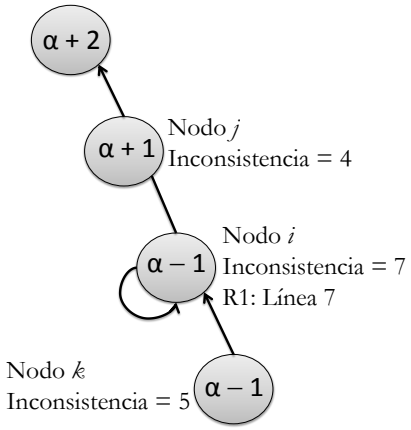

Figura 30. Falla en nodo interno con un solo hijo: caso 7.

8. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i permanece sin cambios. Ver Figura 31.

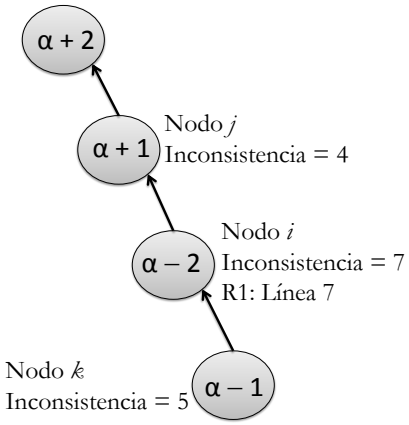

Figura 31. Falla en nodo interno con un solo hijo: caso 8.

9. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i deja de apuntar a si mismo. Ver Figura 32.

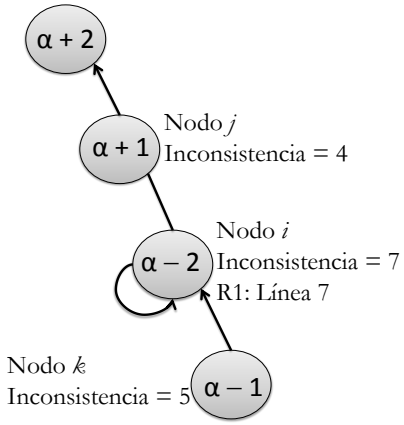

Figura 32. Falla en nodo interno con un solo hijo: caso 9.

Estos casos se resumen en la Tabla IV.

Tabla IV. Fallas de los nodos internos.

Caso	x_i	P_i	Figura
1	Se incrementa dos o más unidades	Permanece sin cambios	21 a 24
2	Se incrementa dos o más unidades	Apunta a otro lado	22
3	Se incrementa una unidad	Permanece sin cambios	23
4	Se incrementa una unidad	Apunta a otro lado	24
5	Permanece sin cambios	Apunta a otro lado	25
6	Se decrementa una unidad	Permanece sin cambios	26
7	Se decrementa una unidad	Apunta a otro lado	27
8	Se decrementa dos o más unidades	Permanece sin cambios	28
9	Se decrementa dos o más unidades	Apunta a otro lado	29

Para los casos donde falla un nodo interno con dos hijos, los casos 1 y 2 toman en el peor de los casos 5 pasos cada uno; los pasos se muestran en las Figuras 33 a 38 y 39 a 43 respectivamente. Si el nodo interno que falla tiene más de dos hijos, el problema es más sencillo de resolver y requiere un sólo paso; este caso se discute más adelante. Los casos 1 y 2 son el peor caso a resolver. Los siete casos restantes necesitan un paso para resolver el efecto de la falla. La falla se resuelve al ejecutar la acción del algoritmo señalada en la Figura que ilustra el caso.

La Figura 33 muestra un fragmento de árbol estable en el que se basa el análisis de los nodos internos.

En los casos de fallas de nodos internos pueden existir varios nodos que tengan el mismo valor en la variable *inconsistencia*, si esto ocurre se elige uno de estos nodos al

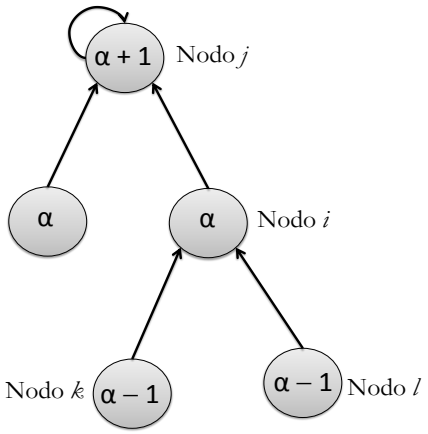

Figura 33. Falla en nodo interno con un dos hijos: ejemplo.

azar para ejecutar alguna acción para intentar corregir la falla. Si se elige el nodo que falla, la falla se soluciona en un solo paso, y la configuración después de la corrección de la falla es exactamente la misma que antes de la falla. En caso contrario, al algoritmo le toma un número mayor de pasos corregir la falla. Los casos de las figuras son el peor caso. La probabilidad de que se elija al nodo que falla para realizar la acción de corrección es de $\frac{1}{3}$ dado que se supone un calendarizador de probabilidad uniforme.

A continuación se muestran los nueve casos de fallas en nodos intermedios con dos hijos.

1. Si como consecuencia de una falla, x_i se incrementa dos unidades o más y P_i permanece sin cambios. Ver Figuras 34 a 38.

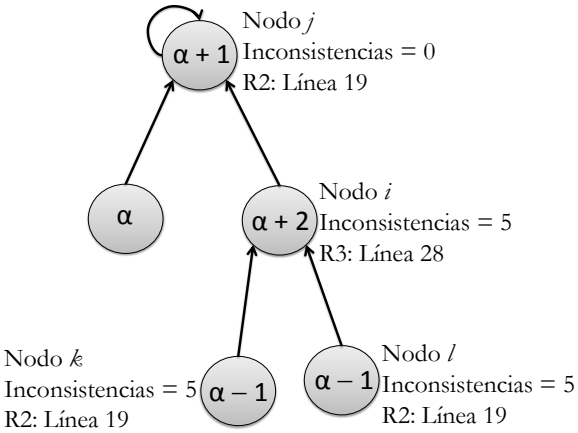

Figura 34. Falla en nodo interno: caso 1, paso 1.

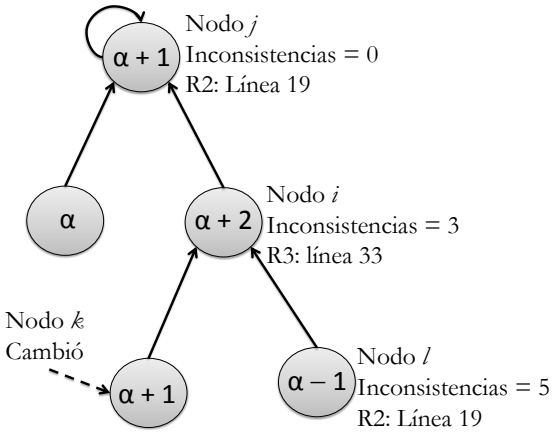

Figura 35. Falla en nodo interno: caso 1, paso 2.

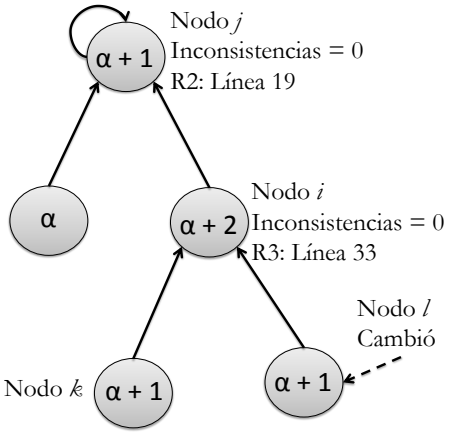

Figura 36. Falla en nodo interno: caso 1, paso 3.

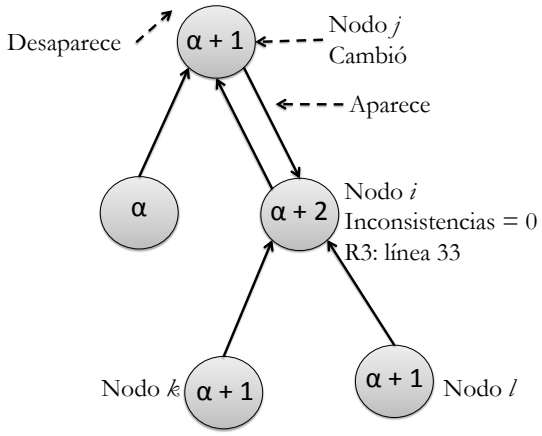

Figura 37. Falla en nodo interno: caso 1, paso 4.

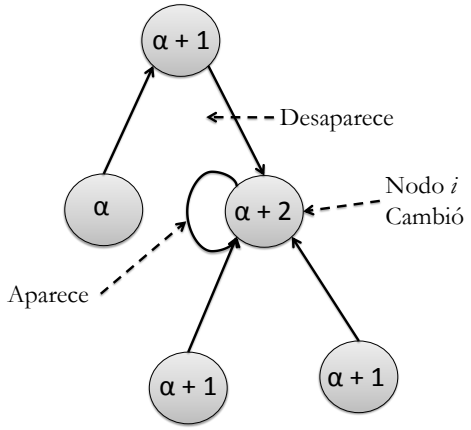

Figura 38. Falla en nodo interno: caso 1, paso 5.

2. Si como consecuencia de una falla, x_i se incrementa dos unidades o más y P_i deja de apuntar al nodo j. Ver Figuras 39 a 43.

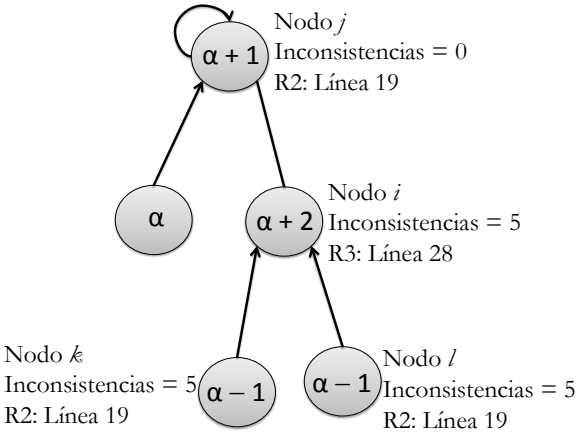

Figura 39. Falla en nodo interno: caso 2, paso 1.

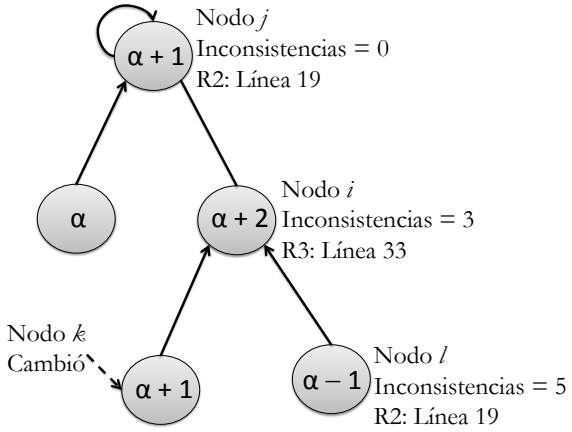

Figura 40. Falla en nodo interno: caso 2, paso 2.

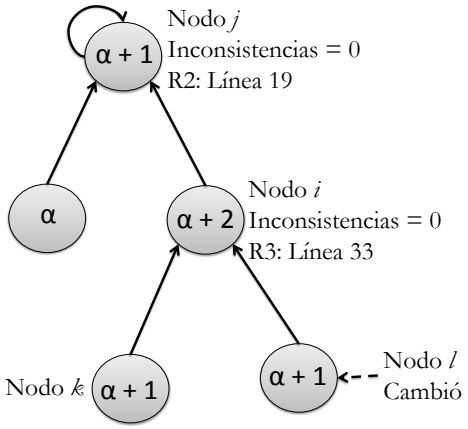

Figura 41. Falla en nodo interno: caso 2, paso 3.

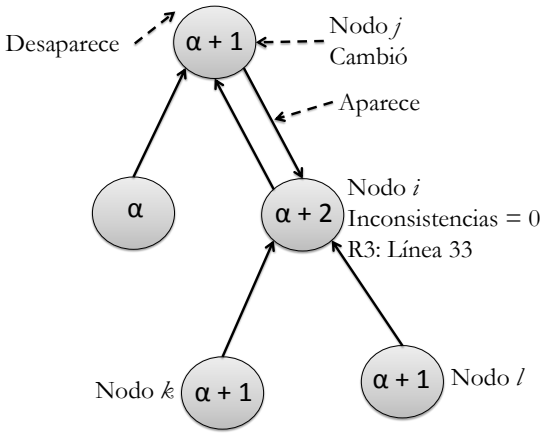

Figura 42. Falla en nodo interno: caso 2, paso 4.

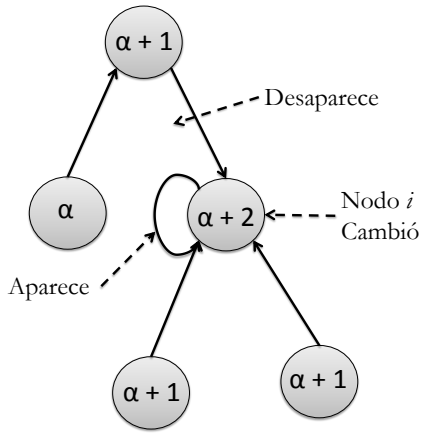

Figura 43. Falla en nodo interno: caso 2, paso 5.

3. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i permanece sin cambios. Ver Figura 44

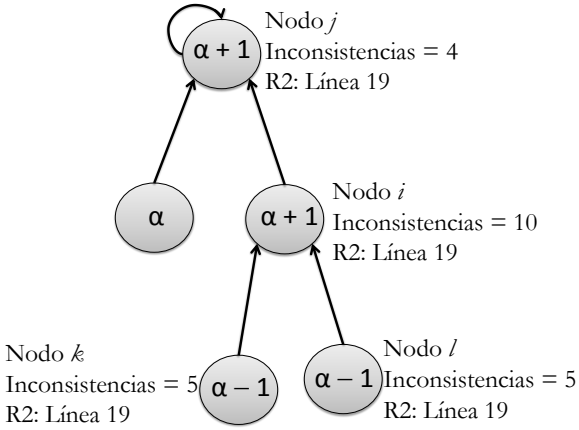

Figura 44. Falla en nodo interno caso 3.

4. Si como consecuencia de una falla, x_i se incrementa una unidad y P_i deja de apuntar al nodo j. Ver Figura 45.

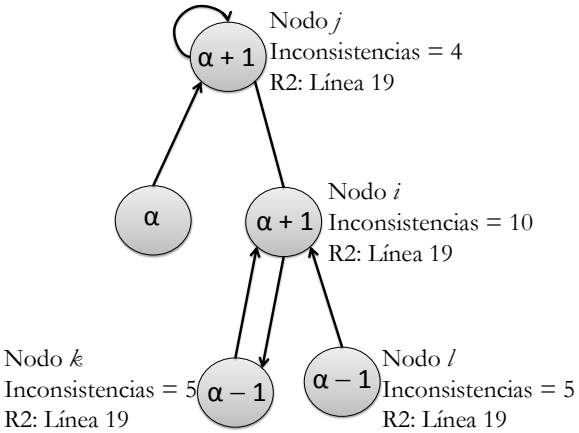

Figura 45. Falla en nodo interno: caso 4.

5. Si como consecuencia de una falla, x_i permanece intacto pero P_i deja de apuntar al nodo j. Ver Figura 46.

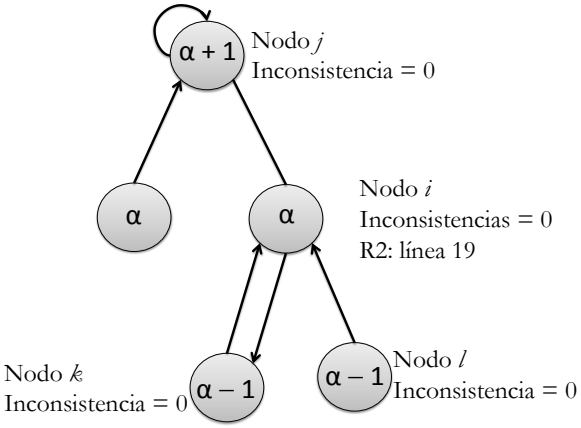

Figura 46. Falla en nodo interno: caso 5.

6. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i permanece sin cambios. Ver Figura 47.

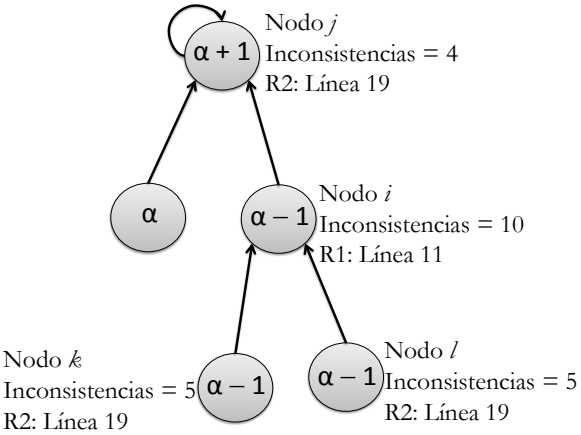

Figura 47. Falla en nodo interno: caso 6.

7. Si como consecuencia de una falla, x_i se decrementa una unidad y P_i deja de apuntar al nodo j. Ver Figura 48.

8. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i permanece sin cambios. Ver Figura 49.

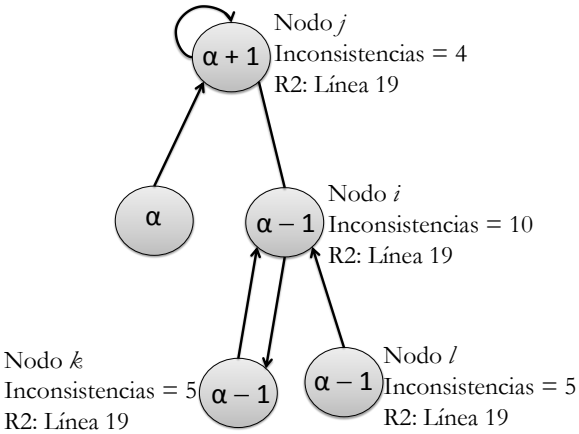

Figura 48. Falla en nodo interno: caso 7.

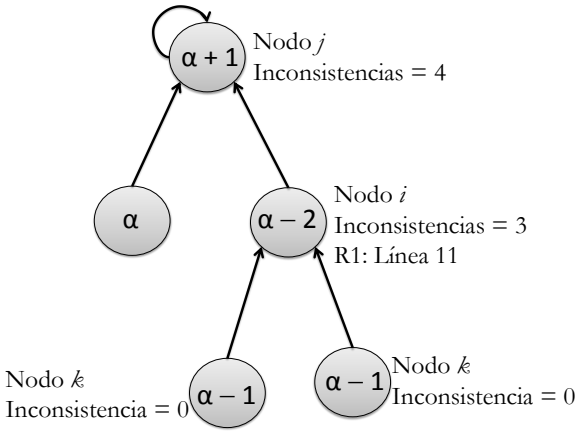

Figura 49. Falla en nodo interno: caso 8.

9. Si como consecuencia de una falla, x_i se decrementa dos o más unidades y P_i deja de apuntar al nodo j. Ver Figura 50.

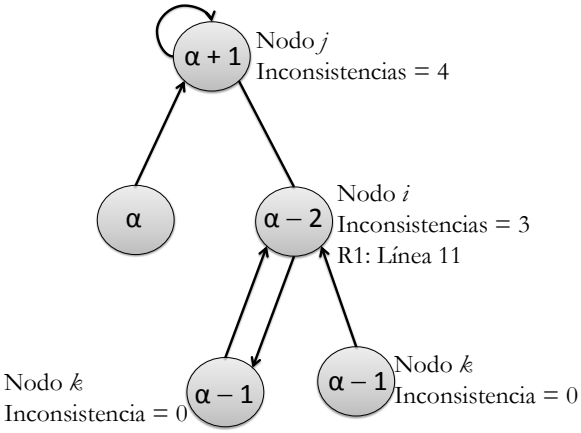

Figura 50. Falla en nodo interno: caso 9.

Los casos de fallas que afectan nodos internos pueden resolverse en más de un paso, pero en tiempo constante. A continuación se muestra que efectivamente el peor caso de corrección de fallas en nodos intermedios es el de los casos 1 y 2. Para mostrarlo se expone el siguiente par de casos:

1. El nodo que falla tiene más de dos hijos. Figura 51

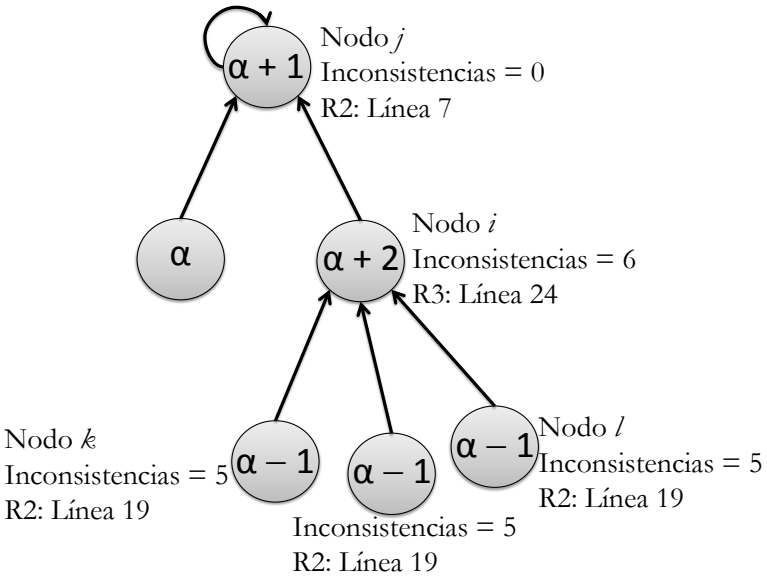

Figura 51. Falla en nodo interno: caso con más de dos hijos.

2. El nodo que falla tiene nietos. Figura 52

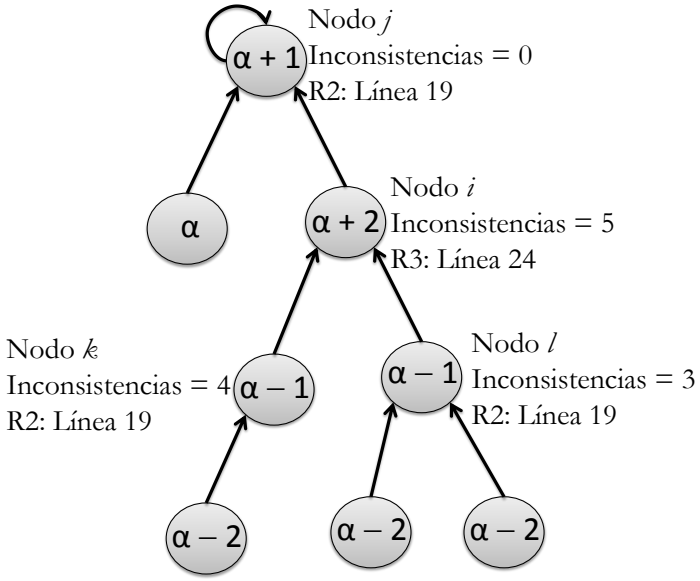

Figura 52. Falla en nodo interno: caso con nietos.

Los dos casos mencionados ocasionan que determinar el nodo que falló sea más fácil, ya que la variable $inconsistencia_i$ con el valor mayor dentro de $N(i)$ determina con exactitud al nodo que falló. De esta manera, el problema se corrige en un paso para todos los casos.

V.2 Pseudo-código del algoritmo auto-estabilizante de elección de líder con contención de fallas

▷ Inconsistencias

1 **if** $\nexists\, inconsistencia_j \in N(i) \mid inconsistencia_j > inconsistencia_i$

2 **then**

▷ R1

3 **if** $(\exists j,k \in N(_i),\ j \neq k \mid x_j \geq x_i \wedge x_k \geq x_i)$

4 **then if** $(\exists j,k \in N(_i),\ j \neq k \mid (x_j = x_k) > x_i \wedge \nexists l \in N(i) \mid x_l > x_{j\ \forall j \in N(i)})$

5 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) + 1$;

6 $P_i \leftarrow i$;

7 **else if** $(\exists j,k \in N(_i),\ j \neq k \mid (x_j = x_k) > x_i\ \wedge\ \exists\ l \in N(i) \mid x_l > x_{j \forall j \in N(i)} \vee \forall j,k \in N(i) \mid |x_j - x_k| = 0 \vee |x_j - x_k| = 2)$

8 **then if** $(x_i = max_{j \in N_i}(x_j) - 1)$

9 **then** $x_i \leftarrow max\ x_j + 1$;

10 $P_i \leftarrow i$;

11 **else** $x_i \leftarrow max\ x_j - 1$;

12 $P_i \leftarrow j$;

13 **else if** $(\exists j \in N(_i) \mid N(_j) = 1)$

14 **then**

15 $P_i = P_i$

16 **else if** $(\forall j, k \;\; \nexists j, k \in N(i) \mid \; |x_j - x_k| \neq 0 \vee |x_j - x_k| \neq 2)$

17 **then** $x_i \leftarrow max\ x_j + 1$;

18 $P_i \leftarrow i$;

▷ R2

19 **if** $(\exists! j \in N_i \mid x_j \geq x_i)$

20 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) - 1$;

21 $P_i \leftarrow j$;

▷ R3

22 **if** $(\nexists j \in N_i \mid x_j > x_i) \wedge P_i \neq i$

23 **then if** $(N(_i) > 1)$

24 **then if** $(\forall j, k \;\; \nexists j, k \in N(i) \mid \; |x_j - x_k| \neq 0 \vee |x_j - x_k| \neq 0 \wedge \exists j, k \in N(i) j \neq k \mid x_j = x_k \wedge \exists j \in N(i) \mid x_j > x_k \forall k \in N(i))$

25 **then if** $(x_i = \max_{j \in N_i}(x_j) - 1)$

26 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) + 1$;

27 $P_i \leftarrow i$;

28 **else** $\max_{j \in N_i}(x_j) - 1$;

29 $P_i \leftarrow j$;

30 **else if** $(\forall j,k \ \nexists j,k \in N(i) \mid \ |x_j - x_k| \neq 0 \vee |x_j - x_k| \neq 2 \wedge \ \nexists j,k \in N(i) j \neq k \mid x_j = x_k)$

31 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) - 1$;

32 $P_i \leftarrow j$;

33 **else if** $(\forall j,k \ \nexists j,k \in N(i) \mid \ |x_j - x_k| \neq 0 \vee |x_j - x_k| \neq 2) \wedge \exists j,k \in N(i) j \neq k \mid x_j = x_k \ \nexists j \in N(i) \mid x_j > x_k \forall k \in N(i)$

34 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) + 1$;

35 $P_i \leftarrow i$;

36 **else if** $(N(_i) = 1)$

37 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) - 1$;

38 $P_i \leftarrow j$;

39 **else if** $(\forall j,k \ \nexists j,k \in N(i) \mid \ |x_j - x_k| \neq 0 \vee |x_j - x_k| \neq 2)$

40 **then** $x_i \leftarrow \max_{j \in N_i}(x_j) + 1$;

41 $P_i \leftarrow i$;

Algoritmo auto-estabilizante de elección de líder con contención de fallas.

Capítulo VI

ANÁLISIS DEL ALGORITMO PROPUESTO

VI.1 El Algoritmo Propuesto Termina en un Estado Legítimo

Recuerde que un estado estable es aquel en el que:

a) Existe sólo un máximo global r.

b) Para cualquier $j \neq r$, la distancia $(r,\ j) = x_r - x_j$.

c) Para cualquier $j \neq r$, su apuntador señala al máximo local.

d) El apuntador de la raíz apunta a sí mismo.

Lema 1 *Si el algoritmo de elección de líder termina, lo hace en un estado legítimo.*

Demostración Por contradicción:

Se supone que el algoritmo termina y queda en un estado global ilegítimo, existen dos casos:

a) Existe más de un nodo i cuyo valor en x_i es igual al máximo global (existe en el árbol más de un líder global).

b) Existe sólo un nodo i tal que x_i es un máximo global; sin embargo, hay un nodo j tal que $x_i - x_j \neq d(i,\ j)$ o hay algún nodo cuyo apuntador no está apuntando a su máximo local. La distancia entre el par de nodos i y j se denota por $d(i,\ j)$.

Para el caso a:

Sean i y k dos nodos tal que $x_i = x_k = max_{j \in V}\ x_j$, existen dos subcasos:

Sub-caso 1: Si i y k son adyacentes y máximos locales ambos son candidatos por la regla R2. Entonces al momento de que uno de ellos ejecute R2 se genera una secuencia de cambios en los nodos adyacentes al nodo que ejecutó R2 y así sucesivamente con los nodos adyacentes a esos nodos.

Sub-caso 2: Los nodos i y k no son adyacentes. Por lo que debe existir una trayectoria única en el árbol T que conecta a i con k. Sean los nodos de esa trayectoria $i, u_1, u_2, ..., u_t, k\ \rangle$, donde $t \geq 1$. En el camino entre los nodos i y el k existe una secuencia descendente y luego ascendente. De esta manera, en algún punto de la trayectoria existe un mínimo local (o dos nodos vecinos que actuan como mínimos locales). Dicho nodo (o nodos) son candidatos bajo la regla R1. Ambos casos ocasionan eventualmente la desaparición de los líderes globales y la aparición de un nuevo y único líder global mediante un reacomodo de los valores de x de los nodos en esa trayectoria.

Para el caso b:

Sea P la trayectoria que une al nodo líder r con j y sea j el único nodo que no satisface $x_j = x_r - d(r,\ j)$, entonces el padre de j sí satisface.

a) Sea k el padre de j. Si $x_j < x_k - 1$, entonces j es candidato por la regla R1.

b) Sea k el padre de j. Si $x_j = x_k$, entonces j es candidato por la regla R2.

c) Sea k el padre de j. Si $x_j > x_k$, entonces j es candidato por la regla R3.

Con esto se demuestra que el algoritmo no puede terminar en un estado ilegítimo. Esto implica que el algoritmo termina en un estado correcto o no termina.

VI.2 Análisis del Tiempo de Ejecución del Algoritmo Propuesto

El análisis mostrado en esta sección se basa en la demostración realizada por (Xu y Srimani, 2006) únicamente se adaptó para el algoritmo propuesto.

Sea T el árbol en estudio. Se define S_0 como el conjunto de nodos que tiene grado 1 en T. Sea S_1 el conjunto de nodos con grado 1 en el árbol $T - S_0$, esto es, en el árbol resultante al quitar las hojas de T. En forma similar se definen los conjuntos $S_2, S3, ..., S_k$. Si T tiene altura k. Ver Figura 53. Se ve a T como una partición donde los bloques de la partición son los conjuntos $S_0, S_1, ..., S_k$; esto es $T = \cup_{i=0}^{k} S_i$.

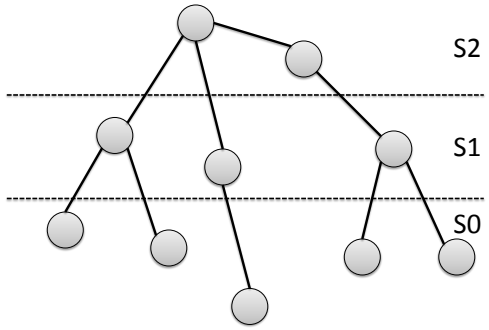

Figura 53. Construcción de los conjuntos $S_0, ..., S_k$.

Se observan las siguientes propiedades de cada conjunto S_i y T_i para un árbol dado. $T_i = \cup_{j=0}^{i} S_j$. Si $j = k$, entonces $T_k = T$.

1. Si $|S_k = 0|$ indica que en S_{k-1} existen un par de nodos. El árbol T_{k-1} es un par de nodos conectados por una arista $h - 1$. Ver Figura 53.

2. Para cualquier árbol T, de altura k. Notar que cualquier árbol (con más de un

nodo) tiene al menos dos nodos hoja, se tiene que $|S_i| \geq 2$, para $0 \leq i < k$. T_i es un árbol para $0 \leq i < k$).

3. $S_i \cap S_j = 0$ para $i \neq j$, es decir, los conjuntos $S_0, ..., S_k$ son mutuamente excluyentes.

4. Para cualquier m, $0 \leq m < k$, y dado un nodo $i \in S_m$, el nodo i tiene a lo más un vecino en T_m (los nodos en S_m son nodos hoja de los subárboles T_m).

Definición 3 *Sea a_i el número máximo de veces que el nodo i puede ejecutar la regla R1 antes de terminar el algoritmo. Sea $A_m = \sum_{i \in S_m} a_i$ (el número máximo de veces que todos los nodos en S_m pueden ejecutar la regla R1).*

Lema 2 *La regla R3 no genera la aparición de nodos privilegiados por la regla R1.*

Demostración En el algoritmo propuesto existen dos acciones posibles para la regla R3.

1. Actualizar el apuntador del nodo privilegiado.

2. Decrementar el valor de x_i y actualizar el apuntador del nodo privilegiado.

La regla R3 privilegia únicamente a los nodos que son líderes locales. Sea el nodo i un líder local. Cuando i no apunta a si mismo, la regla R3 ejecuta el caso 1, no se altera el valor de x y el nodo i sigue siendo máximo local. (No se generan nodos privilegiados por R1).

El caso 2 se produce por alguna falla en el nodo i y la regla R3 decrementa el valor de x_i, por lo tanto esta acción tampoco genera nodos privilegiados por R1.

Lema 3 *Al menos dos vecinos del nodo i deben ejecutar la regla R1 entre dos ejecuciones sucesivas de R1 por el nodo i*

Demostración Después de que un nodo i ejecuta la regla R1, para que el nodo i ejecute R1 de nuevo, al menos dos de sus vecinos deben incrementar sus valores de x, (por medio de la ejecución de reglas R1). La regla R1 es la única regla que incrementa el valor de x de un nodo sobre el valor máximo de los valores de x de sus vecinos.

Corolario 1 *Si un nodo $i \in S_m$ ejecuta la regla R1, para que el nodo i ejecute R1 de nuevo, al menos uno de sus vecinos (podrían ser dos) en $N(i) \cap S_{m-1}$ debe ejecutar R1. Así, para cualquier nodo $i \in S_m$, se tiene que*

$$a_i \leq \sum_{j \in N(i) \cap S_{m-1}} a_j$$

Demostración Por la propiedad 4 de S_m, al menos un vecino de i no está en $N(i) \cap S_m - 1$, está en S_{m+1}. Por lo tanto, al menos un vecino en $N(i) \cap S_{m-1}$ debe ejecutar R1.

Lema 4 *Para dos nodos arbitrarios $i, j \in S_m$, $m > 0$, $(N(i) \cap S_{m-1}) \cap (N(j) \cap S_{m-1}) = 0$, es decir, tienen vecinos mutuamente excluyentes en S_{m-1}*

Demostración Directamente se deriva del hecho de que el subgrafo T_m $(= T_0 - S_0 - S_1 - ... - S_{m-1})$ siempre es un árbol.

Lema 5 *El número total de ejecuciones de la regla R1 para todos los nodos en el conjunto S_m está acotado por debajo por el número total de ejecuciones de la regla R1 de todos los nodos en S_{m-1} más el tamaño del conjunto S_m, para toda $m > 0$, es decir*

$$A_m \leq A_{m-1} + |S_m|$$

Demostración

$$
\begin{aligned}
A_m &= \textstyle\sum_{i \in S_m} a_i && (Por\ la\ definición\ 3)\\
&\leq \textstyle\sum_{i \in S_m} (\sum_{j \in N(i) \cap S_{m-1}} a_j + 1) && (Por\ el\ corolario\ 1)\\
&= \textstyle\sum_{i \in S_m} \sum_{j \in N(i) \cap S_{m-1}} a_j + |S_m| && (Por\ el\ lema\ 5)\\
&\leq \textstyle\sum_{i \in S_m} a_j + |S_m|) && (Simplificando)\\
&= A_{m-1} + |S_m| && (Resolviendo\ la\ sumatoria)
\end{aligned}
$$

Lema 6 *Un nodo hoja en el árbol original T nunca ejecuta la regla R1, es decir, $A_0 = 0$.*

Demostración Para poder ejecutar la regla R1, un nodo debe tener al menos dos vecinos y un nodo hoja en T tiene exactamente un vecino.

Lema 7 *El número total de ejecuciones de la regla R1 hechas por los nodos en S_m, para toda $m > 0$ es a lo más n, es decir,*

$$a_m \leq \sum_{j=1}^{m} |S_j| \leq n$$

Demostración Por la aplicación repetitiva del lema 3,

$$A_m \leq A_{m-1} + |S_m| \leq A_{m-2} + |S_{m-1}| + |S_m| \leq ... \leq \sum_{j=1}^{m} |S_j|$$

ya que $\cup_{j=1}^{m} S_j$ es un subconjunto de T, $\sum_{j=1}^{m} |S_j| \leq |T| = n$

Teorema 1 *El número total de ejecuciones de la regla R1 por todos los nodos de T es $O(n^2)$; más precisamente*

$$\sum_{m=0}^{h} A_m \leq n + \frac{n^2}{2}$$

Demostración Para toda m, $A_m \leq n$. Por lo tanto $\sum_{m=0}^{h} A_m \leq (h+1)n$. Por la propiedad 2 de S_m, se sabe que $h < \frac{n}{2}$. Por lo tanto $\sum_{m=0}^{h} A_m \leq \frac{n^2}{2} + n$

Teorema 2 *El algoritmo de elección de líder termina en $O(n^4)$ pasos en cualquier árbol T con n nodos.*

Demostración Se sabe por el Teorema 1 que, no puede haber más de $\frac{n^2}{2} + n$ ejecuciones de la regla R1 hechas por los nodos en el árbol; ahora se necesita una cota en el número de ejecuciones de las otras reglas del algoritmo.

Se supone que un nodo ejecuta la regla R1. Notar que ninguna de las ejecuciones de las reglas R2 y R3 por algún nodo j puede cambiar el número de vecinos del nodo i que tengan valores de x mayores que x_i; incluso la ejecución de la regla R2 no cambia el valor de los líderes locales; y la regla R3 sólo privilegia a lideres locales, sin embargo, la ejecución de la regla R3 puede comportarse de forma similar a R2 con las implicaciones de ésta (no cambiar el valor de los líderes locales) o cambiar el valor de los líderes locales, decrementando su valor a un valor una unidad menor al valor de x del mayor de sus vecinos, y como se observó en la prueba del Lema 1, lo anterior no ocasiona la aparición de nodos privilegiados por R1.

La regla R3 sirve para corregir errores, ya que la regla genera líderes locales (R1) siempre fija el apuntador hacia sí mismo. En el peor de los casos en un árbol que se asemeje a un arreglo lineal podría haber $O(n)$ máximos locales que potencialmente ejecuten R3.

Si un nodo i está privilegiado por la regla R2, el nodo i tiene exactamente un vecino j tal que $x_j \geq x_i$ y el número de movimientos que el nodo i puede realizar es a lo más uno más de los que puede hacer el nodo j. Si ninguno de los nodos ejecuta la regla R1, todos los nodos que están a una distancia de 1 del máximo local podrán ejecutar la regla R2 a los más un par de veces; los nodos que están a una distancia k del máximo local podrán ejecutar la regla R2 a lo más $k + 1$ veces. Por lo tanto, si ningún nodo

ejecuta la regla R1, se puede tener a lo más $\frac{n(n+1)}{2}$ ejecuciones de la regla R2.

En combinación con el teorema 2, el número total de movimientos para cualquier nodo en el árbol T es $\frac{n(n+1)}{2}(\frac{n^2}{2} + n + 1)$, el cual es $O(n^4)$.

El algoritmo de elección de líder propuesto puede elegir un nodo arbitrario como líder, siempre que el nodo no sea hoja, empezando desde una configuración inicial arbitraria. La elección del nodo líder depende de la configuración inicial y del comportamiento del demonio. Además proporciona un mecanismo de contención de fallas. Tal mecanismo puede corregir en tiempo constante una falla transitoria simple que afecte al sistema.

Lo anterior demuestra que a lo más el algoritmo puede ejecutar $O(n^4)$ reglas, una vez que las haya ejecutado el algoritmo forzosamente debe haber llegado a un estado legítimo. Esto demuestra la convergencia del algoritmo.

Capítulo VII

EXPERIMENTOS Y RESULTADOS

Se realizaron diversos experimentos con grafos de árbol generados de forma aleatoria mediante un simulador en Java. Para conocer las características del simulador ver en Apéndice C. Los detalles y características de los grafos generados se muestran en la Sección VII.1. Los detalles de los experimentos se muestran en la Sección VII.2.

VII.1 Generación de árboles aleatorios

Para evaluar el rendimiento de los algoritmos se generaron grafos de árbol de forma aleatoria. Se generaron árboles aleatorios con tamaños en potencias de 2, desde 2^3 hasta 2^{10}, es decir, desde 8 hasta 1024 nodos. En total fueron 50 experimentos con cada tamaño de árbol.

Para explicar la generación de árboles, primero se establecen algunas suposiciones.

Se supone que el árbol consta de n nodos, un nodo i tiene la misma probabilidad de conectarse a cualquier nodo que forme parte del árbol. Asi, el nodo i se puede conectar a cualquiera de los $i-1$ nodos que ya forman parte del árbol con probabilidad $\frac{1}{i-1}$.

La generación del árbol inicia con un nodo raíz. Primero se genera el nodo raíz, posteriormente se genera un segundo nodo que se conecta a la raíz. Se genera un tercer nodo que se conecta con probabilidad de $\frac{1}{2}$ a cada uno de los nodos en el árbol. El cuarto nodo generado se conecta con probabilidad de $\frac{1}{3}$ a cada uno de los nodos en el árbol. De esta manera, el i-ésimo nodo tiene una probabilidad de $\frac{1}{i-1}$ de conectarse a cualquiera de los $i-1$ nodos del árbol. Los árboles generados por este método son de

diámetro $O(\log\ n)$ con $O(\log\ n)$ ramas. Se puede observar el análisis de este método en (Alvarado, 2006).

Al finalizar la construcción del árbol, cada nodo toma un valor arbitrario de entre 0 y $10n$ y lo asigna a su variable x, posteriormente de su conjunto de vecinos elige uno para apuntar hacia él.

VII.2 Experimentos realizados

Los experimentos consistieron en generar árboles de diferentes tamaños con las características descritas en la sección anterior.

En cada uno de los experimentos se ejecutó el algoritmo de (Xu y Srimani, 2005) y el algoritmo propuesto. Los experimentos se llevaron a cabo en dos fases, primero ambos empezando en un estado inicial arbitario. La segunda fase parte de una configuración estable a la que se le introduce una falla aleatoria.

Al finalizar cada uno de los experimentos, se obtiene en cada uno de ellos el número de pasos necesarios para alcanzar un estado legal y el número de pasos necesario para recuperarse de la falla en cada uno de los algoritmos, ver Figura 54.

Parte de los resultados de los experimentos realizados se muestran en la siguiente sección. De igual manera se muestran las gráficas correspondientes y los experimentos restantes se pueven ver el el apéndice B.

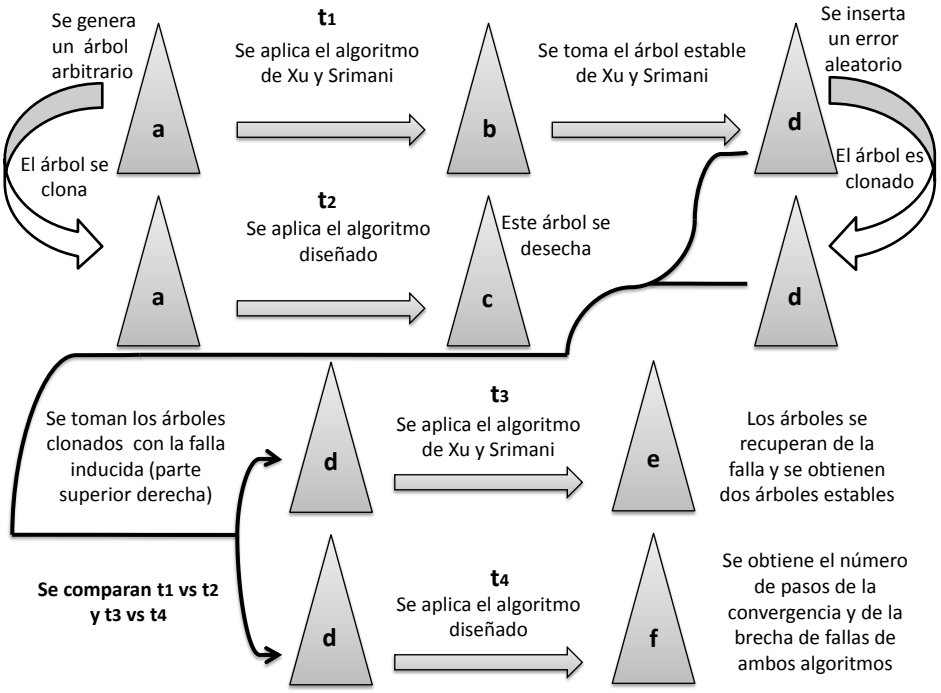

Figura 54. Secuencia general de los experimentos realizados.

VII.3 Resultados experimentales

Se muestra en la Figura 55 el número de pasos que le tomó tanto al algoritmo de Xu y Srimani (t_1) como al algoritmo desarrollado en esta investigación (t_2) alcanzar una configuración legal empezando en un estado arbitrario. Las gráficas muestran un total de 50 experimentos diferentes realizados sobre árboles de 8 nodos.

Posteriormente, en la Figura 56 se muestra el número de pasos (t_3 y t_4) que les tomó a ambos algoritmos de elección de líder recuperarse de una falla transitoria que los afecta una vez que alcanzaron una configuración legal. La falla que se introduce al árbol para probar la contención de fallas se genera de forma aleatoria. Se puede

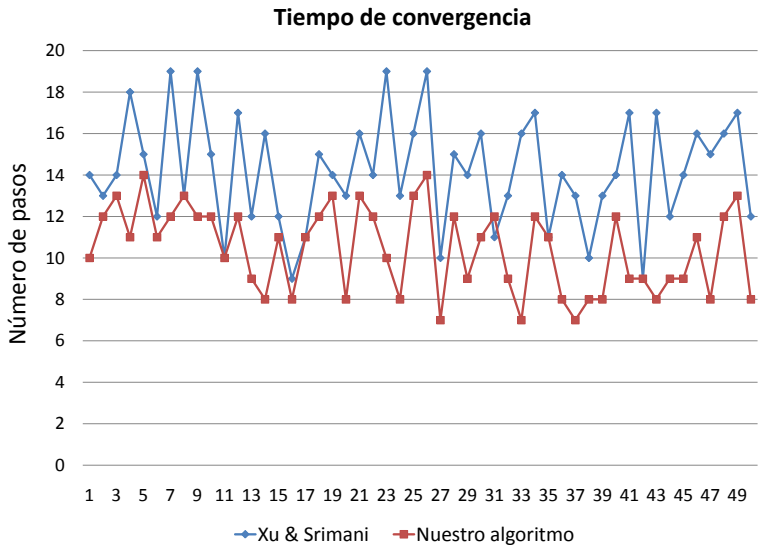

Figura 55. Número de pasos de convergencia para un árbol de 8 nodos.

observar que el algoritmo propuesto resuelve la falla en tiempo constante mientras que al algoritmo de Xu y Srimani le lleva varios pasos más corregir la falla. No se introduce una falla aleatoria para cada algoritmo, en ambos algoritmos la falla es la misma.

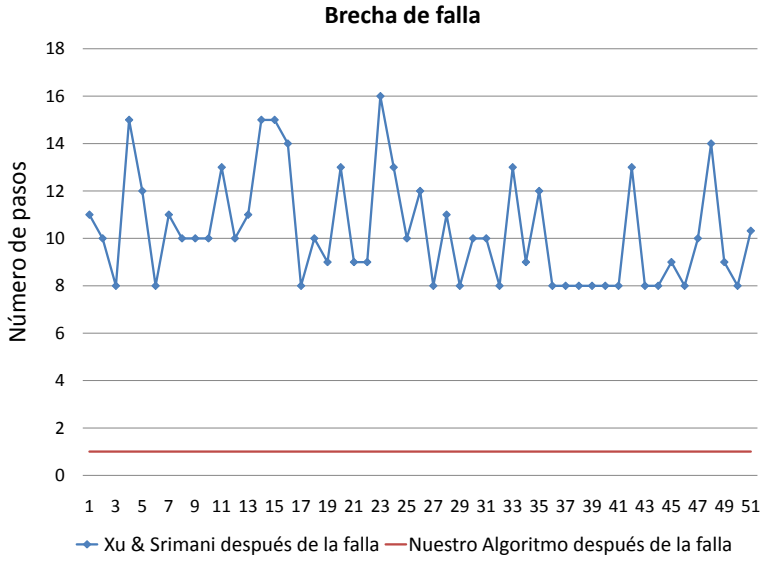

Figura 56. Número de pasos de la brecha de falla para un árbol de 8 nodos.

La Figura 57 muestra el número de pasos (t_1 y t_2) que les tomó a los algoritmos alcanzar un estado legal y la Figura 58 muestra el número de pasos de la brecha de falla (t_3 y t_4) de los algoritmos para los experimentos de 1024 nodos. Las figuras muestran el resultado de los 50 experimentos realizados. Se puede observar que al algoritmo propuesto le lleva en promedio alrededor de 2,000 pasos menos alcanzar un estado legítimo iniciando en un estado arbitrario. Otro punto a señalar es que al algoritmo propuesto nunca le lleva un número mayor de pasos que los que le lleva al algoritmo de Xu y Srimani alcanzar la convergencia.

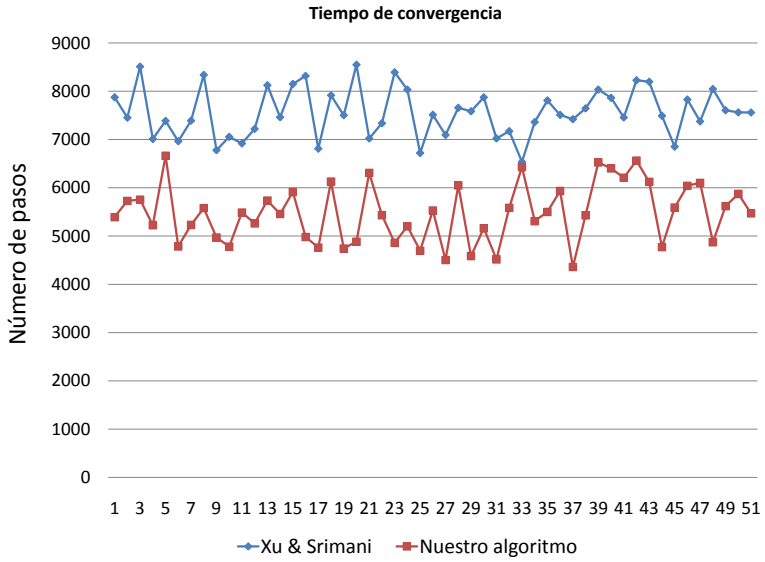

Figura 57. Número de pasos de convergencia para un árbol de 1024 nodos.

La brecha de falla del algoritmo de Xu y Srimani muestra un incremento en su número de pasos mientras que la brecha de falla del algoritmo propuesto permanece independiente del número de nodos simulados.

La Figura 59 muestra el incremento de número de pasos necesarios para la convergencia y brecha de falla del algoritmo de Xu y Srimani conforme se incrementa el número de nodos simulados. En la misma figura se observa que en todos los casos

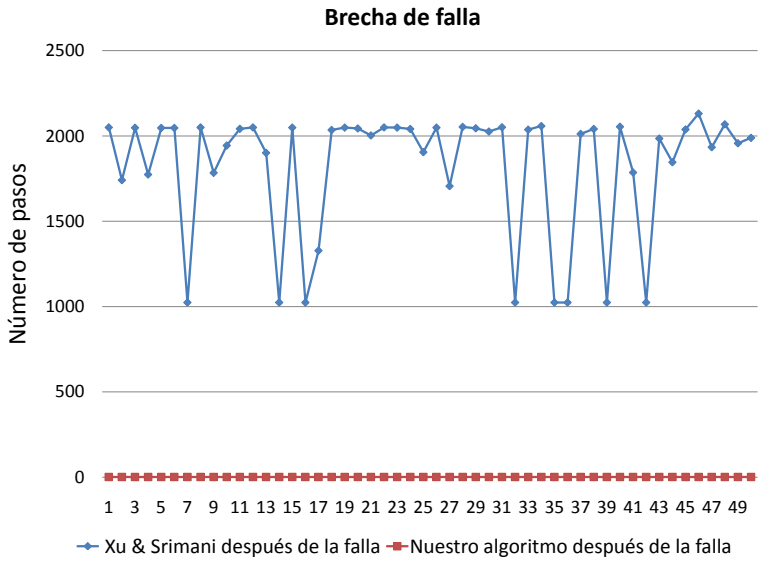

Figura 58. Número de pasos la brecha de falla para un árbol de 1024 nodos.

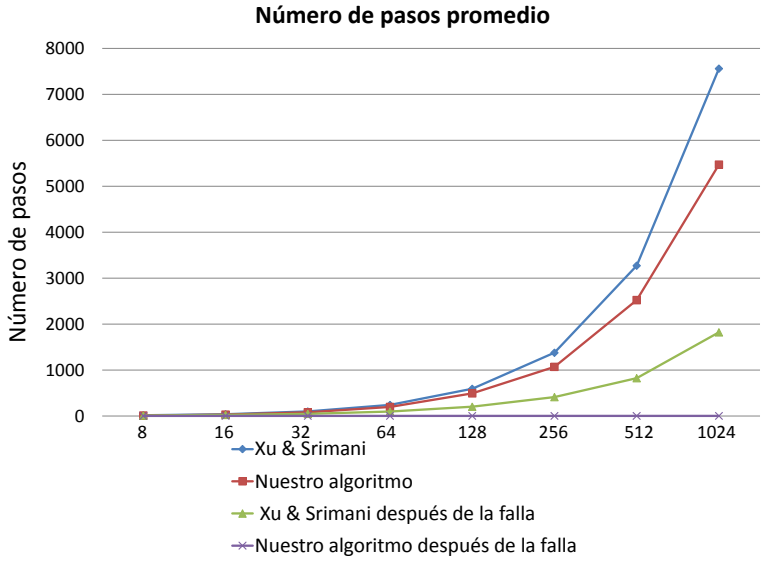

Figura 59. Número de pasos de promedio de los algoritmos.

el número de pasos del algoritmo propuesto es en promedio menor al de Xu y Srimani; además el número de pasos de la brecha de falla del algoritmo propuesto se mantiene sin incremento, por lo tanto, se muestra que la brecha de falla del algoritmo propuesto

es independiente del tamaño de la red mientras que el algoritmo de Xu y Srimani se incrementa conforme se incrementa el número de nodos simulados.

Capítulo VIII

CONCLUSIONES, APORTACIONES Y TRABAJO FUTURO

En el capítulo anterior se exponen experimentos y resultados del algoritmo propuesto en esta investigación.

Las conclusiones del trabajo de investigación se exponen en la Sección VIII.1. Los resultados se exponen en la Sección VIII.2. Posteriormente, en la sección VIII.3 se sugieren algunas líneas de trabajo futuro derivadas de este trabajo de investigación.

VIII.1 Conclusiones

Al inicio de la investigación se planteó la siguiente pregunta de investigación.

- ¿Cómo se puede aplicar la técnica de contención de fallas a algoritmos auto-estabilizantes de elección de líder?

Los objetivos específicos derivados de esta pregunta se enlistan a continuación:

- Hacer un estudio comparativo entre las principales técnicas de contención de fallas en el diseño de algoritmos auto-estabilizantes.
- Hacer un estudio comparativo entre los algoritmos auto-estabilizantes de elección de líder.
- Diseñar una serie de algoritmos auto-estabilizantes para el problema de elección de líder que incluya las técnicas de contención de fallas.

- Hacer simulaciones de los algoritmos diseñados.
- Hacer un estudio comparativo entre ellos.

El primero de los objetivos específicos derivados de la la pregunta de investigación se cumple en el Capítulo IV, donde se analizan algunos algoritmo auto-estabilizantes y se observan las características de estos algoritmos y la forma en que resuelven las fallas que los afectan.

En el Capítulo III se cumple el segundo objetivo específico, donde se analizan y comparan algoritmos auto-estabilizantes de elección de líder en distintas topologías. Se analiza el espacio de memoria utilizado por algunos algoritmos auto-estabilizantes de elección de líder bajo un calendarizador arbitrario. Además se selecciona el algoritmo con el cuál se trabajó en la investigación incorporándole contención de fallas.

El tercer objetivo específico se cumple en el Capítulo V, en este capítulo se expone el algoritmo auto-estabilizante de elección de líder desarrollado que además incluye contención de fallas. El pseudo-código del algoritmo diseñado. Un ejemplo de la ejecución del algoritmo propuesto se muestra en el Apéndice A.

En el Capítulo VII se exponen las características de los experimentos que se efectuaron en esta investigación. Se expone cómo se realizó la generación de grafos necesarios para ejecutar los algoritmos y la manera en que se realizaron los experimentos. Además se muestran las gráficas resultantes de las pruebas realizadas en un simulador que se desarrolló específicamente para el algoritmo de Xu y Srimani y el algoritmo propuesto.

Por último, en el Capítulo VIII se exponen las conclusiones derivadas de la investigación, las aportaciones y se propone trabajo futuro en esta área.

La segunda pregunta de investigación se responde ya que se puede observar que efectivamente se pudo agregar contención de fallas al algoritmo auto-estabilizante de elección de líder.

En el Capítulo VII se muestran los resultados experimentales de las pruebas realizadas a los algoritmos auto-estabilizantes de elección de líder. Se puede observar en la Figura 65 el número de pasos promedio para convergencia y brecha de falla de ambos algoritmos.

La brecha de falla del algoritmo que incorpora la contención de fallas es constante, mientras que la brecha de falla del algoritmo de Xu y Srimani se va incrementando conforme incrementa el número de nodos del grafo sobre el cual se ejecuta el algoritmo.

En la Figura 65 se puede apreciar la diferencia en el número de pasos necesarios para alcanzar un estado legal por los dos algoritmo auto-establizantes de elección de líder. El algoritmo de Xu y Srimani utliza la información de los nodos vecinos para elegir un líder. Con base en la información de los nodos vecinos al nodo i, el nodo i cambia su estado; sin embargo, no aprovecha esa misma información para establecer un conjunto de reglas más específicas que ayuden a contener una falla.

El algoritmo propuesto en esta investigación hace uso de la información señalada. El algoritmo aprovecha la información que un nodo i puede obtener de sus vecinos para elegir un líder y para establecer una relación de error. La relación de error establecida por el nodo i le permite "conocer" qué tan correcto o incorrecto es su estado con respecto a sus vecinos. Con la relación de error establecida, los nodos que tengan una relación de error mayor con respecto a sus vecinos, son los que pueden obtener mejores resultados al momento de efectuar una acción de corrección. Por lo anterior, el algoritmo propuesto se enfoca en darle una prioridad mayor a aquellos nodos cuya relación de error sea mayor. De esta manera, el algoritmo aprovecha mejor la información que tiene a su

alcance y esto a su vez le permite alcanzar un estado legal en un tiempo menor con respecto al algoritmo de Xu y Srimani.

De la misma manera, el establecer la relación de error, le permite al algoritmo identificar con mucha mayor eficiencia al nodo afectado por la falla y poder corregir la falla con un esfuerzo menor utilizando un número constante de pasos. Lo anterior se puede observar en las gráficas mostradas en el Capítulo VII, en la sección VII.3.

VIII.2 Aportaciones

El presente trabajo de investigación muestra que al algoritmo propuesto le toma alcanzar un estado legítimo, en el peor de los casos, el mismo tiempo que le toma al algoritmo original (algoritmo de Xu y Srimani). Además, el agoritmo propuesto puede recuperarse de fallas transitorias simples que lo afecten en un tiempo constante, ya que normalmente el mismo nodo donde sucede la falla ejecuta una acción de corrección que soluciona el problema. Por otro lado, el algoritmo de Xu y Srimani necesita reconfigurar gran parte o incluso todos los nodos del sistema para volver a alcanzar un estado legítmo, lo que constituye un tiempo mucho mayor. Ademas, el tiempo de recuperación ocasionado por una falla transitoria simple que afecte al algoritmo de Xu y Srimani se incrementa en función del número de nodos del sistema; mientras que el algoritmo propuesto resuelve el problema en un tiempo constante; por lo tanto, el sistema no tiene necesidad de dedicar tanto tiempo a corregir dichas fallas y el sistema puede estar un mayor tiempo disponible para ejecutar el trabajo que debe realizar.

Para que esto sea posible, fue necesario establecer una prioridad o relacion de error. La relación de error consiste en que cada nodo que detecta una inconsistencia tiene la capacidad de saber con relación a cuantos nodos pertenecientes a su vecindario existe tal

inconsistencia. De la forma anterior, varios nodos pueden identificar una inconsistencia. La inconsistencia la pueden detectar el nodo donde se origina la falla y todos los nodos vecinos al nodo que falla. Una vez que los nodos saben cuál es su relación de error, aquel nodo que tenga una relación mayor que la relación de error de sus vecinos es el nodo que falló y por lo tanto, es el nodo que ejecuta la acción de corrección.

VIII.3 Trabajo futuro

El trabajo futuro respecto a la incorporación de contención de fallas a algoritmos auto-estabilizantes de elección de líder se presenta en la Sección VIII.3.1. En la Sección VIII.3.2 se proponen una idea para extender el alcance de la contención de fallas y corregir un grupo de fallas contenidas en un radio k que afecten a un sistema.

VIII.3.1 Incorporación de contención de fallas en algoritmos de elección de líder

Los algoritmos presentados a lo largo del presente documento para la elección de líder pueden incorporar técnicas de contención de fallas. Es factible incorporar contención de fallas a algoritmos auto-estabilizantes de elección de líder sin afectar la complejidad del algoritmo, como ya se mostró con el algoritmo propuesto en esta investigación. Sin embargo, para lograr agregar la contención de fallas, al menos en este caso fue necesario incorporar una serie de restricciones al algoritmo. Las restricciones incorporadas tienen el objetivo de ser bastante selectivas con la acción que permiten ejecutar para identificar la falla y realizar la acción justa que habrá de eliminar la falla. Queda como trabajo futuro poder resolver una mayor cantidad de fallas contiguas, es decir, fallas en nodos vecinos.

Agregar contención de fallas a los algoritmos de elección de líder no es una tarea sencilla. Agregar la contención de fallas en la mayoría de los casos implica el uso de más variables. Al agregar una variable al algoritmo se generan nuevas condiciones y se modifican las existentes por lo que es necesario asegurar que al agregar dicha variable todas las condiciones del algoritmo sigan siendo auto-estabilizantes para todos los casos. De ésta manera mostrar que efectivamente el algoritmo con las modificaciones pertinentes sigue siendo auto-estabilizante en definitiva no es una tarea sencilla.

VIII.3.2 Algoritmo de radio k

Se propone un algoritmo que contiene y elimina un grupo de fallas que ocurren dentro de un radio k, donde k es la distancia en número de aristas entre dos nodos cualesquiera del sistema. El algoritmo supone que ocurre una serie de fallas en el sistema en un radio k, tomando como referencia para el radio al nodo más céntrico del grupo de fallas. Para realizar la correción de las fallas, este algoritmo necesita obtener información de un grupo de nodos localizados fuera del radio k (nodos libres de fallas). Con la información de los nodos que están fuera del radio k, un nodo dentro del radio de fallas puede determinar qué valor le corresponde y corregir la falla que lo afecta.

La idea de este algoritmo es que k sea constante con respecto al número de nodos del sistema para que las fallas puedan resolverse en un tiempo k constante.

Referencias

Alvarado, M. J. P. (2006). *Análisis de algoritmos auto-estabilizantes para elección de líder*. Tesis de maestría, Centro de Investigación Científica y de Educación Superior de Ensenada.

Alvarado-Magaña, J. P. y Fernández-Zepeda, J. A. (2007). Average execution time analysis of a self-stabilizing leader election algorithm. En *IPDPS '07: Proceedings of the 21th International Parallel and Distributed Processing Symposium*, páginas 1–7, Long Beach, CA, USA. IEEE Computer Society. ISBN 1-4244-0910-1.

Antonoiu, G. y Srimani, P. K. (1996). A self-stabilizing leader election algorithm for tree graphs. *Journal of Parallel and Distributed Computing*, **34**(2): 227–232.

Baala, H., Flauzac, O., Gaber, J., Bui, M., y El-Ghazawi, T. (2003). A self-stabilizing distributed algorithm for spanning tree construction in wireless ad hoc networks. *Journal of Parallel and Distributed Computing*, **63**(1): 97–104.

Beauquier, J., Gradinariu, M., y Johnen, C. (1999). Memory space requirements for self-stabilizing leader election protocols. En *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, páginas 199–207, New York, NY, USA. ACM. ISBN 1-58113-099-6.

Beauquier, J., Clement, J., Messika, S., Rosaz, L., y Rozoy, B. (2007a). Self-stabilizing counting in mobile sensor networks. En *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, páginas 396–397, New York, NY, USA. ACM. ISBN 978-1-59593-616-5.

Beauquier, J., Gradinariu, M., y Johnen, C. (2007b). Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, **20**(1): 75–93.

Bein, D., Datta, A. K., y Villain, V. (2007). Self-stabilizing local routing in ad hoc networks. *The Computer Journal*, **50**(2): 197–203.

Blair, J. R. S. y Manne, F. (2003). Efficient self-stabilizing algorithms for tree networks. *ICDCS '03: IEEE International Conference on Distributed Computing Systems*, **00**: 20.

Chaudhuri, P. y Thompson, H. (2005). A self-stabilizing algorithm for radio-coloring directed trees. En *Journal of Parallel and Distributed Computing*, páginas 633–637.

Chen, N.-S., Yu, H.-P., y Huang, S.-T. (1991). A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, **39**: 147–151.

Chen, Y., Datta, A. K., y Tixeuil, S. (2005). Stabilizing inter-domain routing in the internet. *Journal of High Speed Networks*, **14**(1): 21–37.

Dasgupta, A., Ghosh, S., y Xiao, X. (2007). Probabilistic fault-containment. En T. Masuzawa y S. Tixeuil, editores, *9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Vol. 4838 de *Lecture Notes in Computer Science*, páginas 189–203. Springer. ISBN 978-3-540-76626-1.

Dijkstra, E. W. (1974). Self stabilizing systems in spite of distributed control. *Communications of the ACM*, **17**(11): 643–644.

Dolev, S. (2000). *Self-Stabilization*. MIT Press.

Dolev, S. y Herman, T. (1997). Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, **1997**(4).

Dolev, S. y Herman, T. (2007). Parallel composition for time-to-fault adaptive stabilization. *Distributed Computing*, **20**(1): 29–38.

Ghosh, S. y Gupta, A. (1996). An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, **59**(5): 281–288.

Ghosh, S. y He, X. (2000). Fault-containing self-stabilization using priority scheduling. *Information Processing Letters*, **73**(3-4): 145–151.

Ghosh, S., Gupta, A., y Herman, T. (2007). Fault-containing self-stabilizing distributed protocols. *Distributed Computing*, **20**(1): 53–73.

Herman, T. (2000). Superstabilizing mutual exclusion. *Distributed Computing*, **13**(1): 1–17.

Huang, S.-T. (1993). Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*, **15**(3): 563–573.

Huang, S.-T., Hung, S.-S., y Tzeng, C.-H. (2005). Self-stabilizing coloration in anonymous planar networks. *Inf. Process. Lett.*, **95**(1): 307–312.

Itkis, G., Lin, C., y Simon, J. (1995). Deterministic, constant space, self-stabilizing leader election on uniform rings. En *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, páginas 288–302, London, UK. Springer-Verlag. ISBN 3-540-60274-7.

Karaata, M. H. (2004). Stabilizing ring clustering. *Journal of Systems Architecture: the EUROMICRO Journal*, **50**(10): 623–634.

Kosowski, A. y Kuszner, L. (2006). Self-stabilizing algorithms for graph coloring with improved performance guarantees. En *Artificial Intelligence and Soft Computing – ICAISC 2006*, Vol. 4029 de *Lecture Notes in Computer Science*, páginas 189–203. Springer Berlin/Heidelberg. ISBN 978-3-540-35748-3.

Kutten, S. y Patt-Shamir, B. (1997). Time-adaptive self stabilization. En *PODC '97: Proceedings of the sixteenth annual ACM Symposium on Principles of Distributed Computing*, páginas 149–158, New York, NY, USA. ACM. ISBN 0-89791-952-1.

Mathieu, F. (2008). Self-stabilization in preference-based systems. *Peer-to-Peer Networking and Applications*, **1**(2): 104–121.

Parc, X. (1979). Thing model view editor an example from a planning system. Reporte técnico, Xerox Parc.

Schneider, M. (1993). Self-stabilization. *ACM Computing Surveys*, **25**(1): 45–67.

Shi, W. y Srimani, P. K. (2004). Leader election in hyper-butterfly graphs. *Network and Parallel Computing*, **3222**(optional): 292–299. optional.

Shi, W., Bouabdallah, A., y Srimani, P. K. (2005). Leader election in oriented star graphs. *Networks*, **45**(3): 169–179.

Xu, Z. y Srimani, P. K. (2005). Self-stabilizing anonymous leader election in a tree. *IPDPS '05: Proceedings of the 19th International Parallel and Distributed Processing Symposium*, **09**: 207a.

Apéndice A

EJEMPLO DE EJECUCIÓN DEL ALGORITMO PROPUESTO

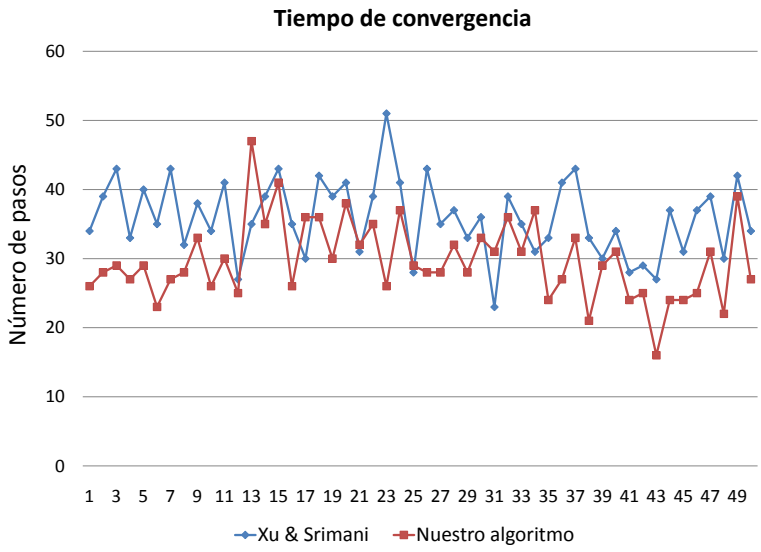

Figura 60. Número de pasos de convergencia para un árbol de 16 nodos.

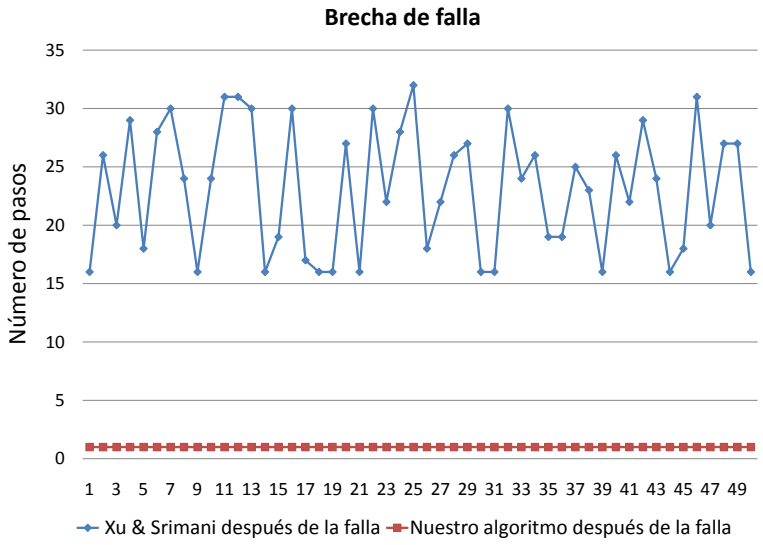

Figura 61. Número de pasos de la brecha de falla para un árbol de 16 nodos.

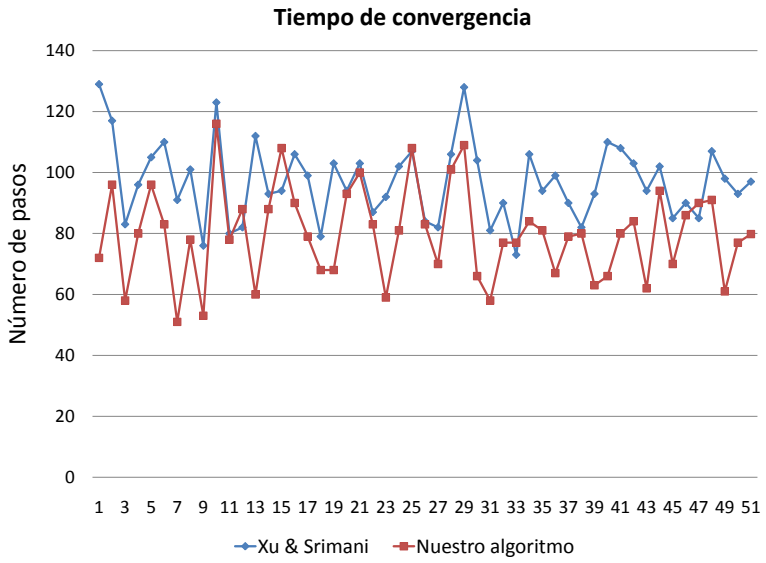

Figura 62. Número de pasos de convergencia para un árbol de 32 nodos.

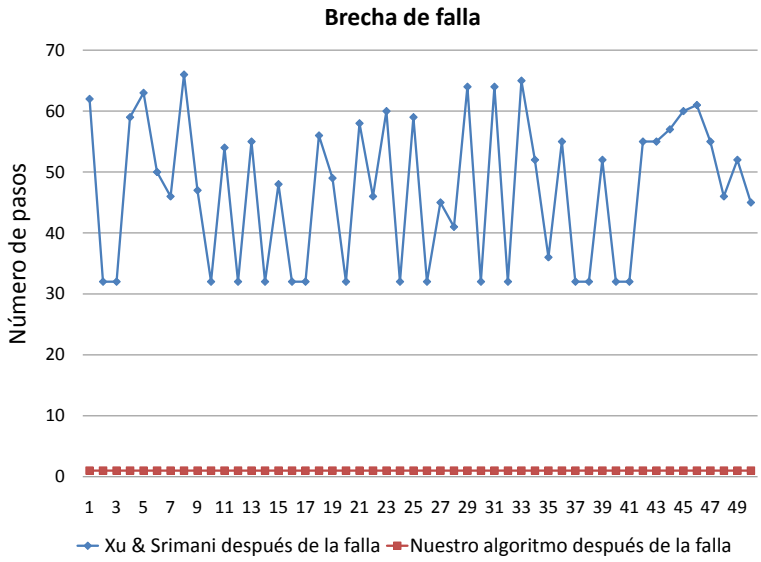

Figura 63. Número de pasos de la brecha de falla para un árbol de 32 nodos.

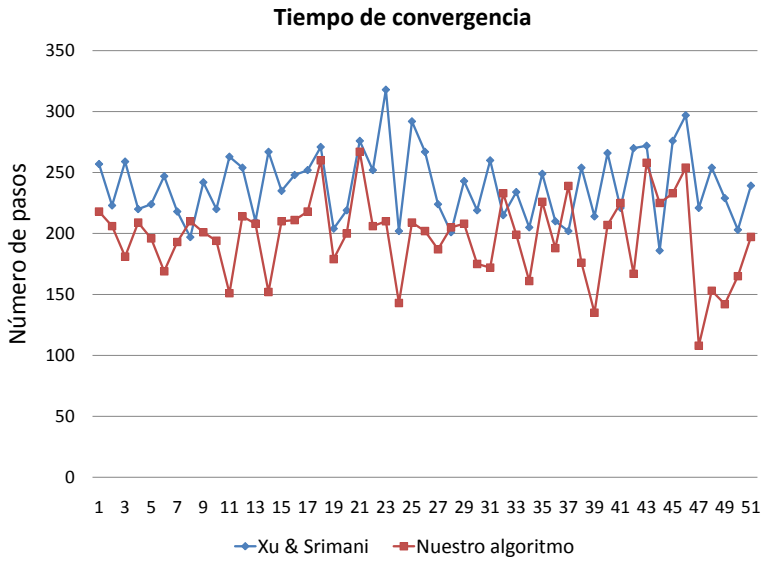

Figura 64. Número de pasos de convergencia para un árbol de 64 nodos.

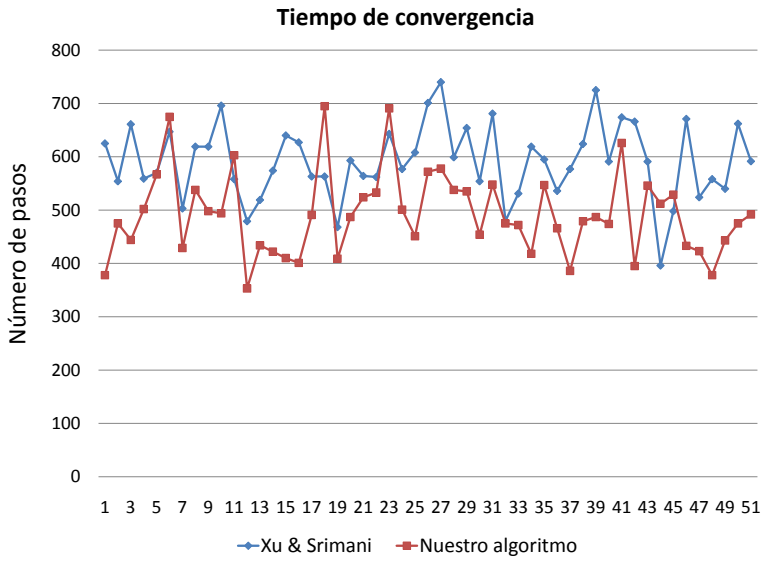

Figura 65. Número de pasos de la brecha de falla para un árbol de 128 nodos.

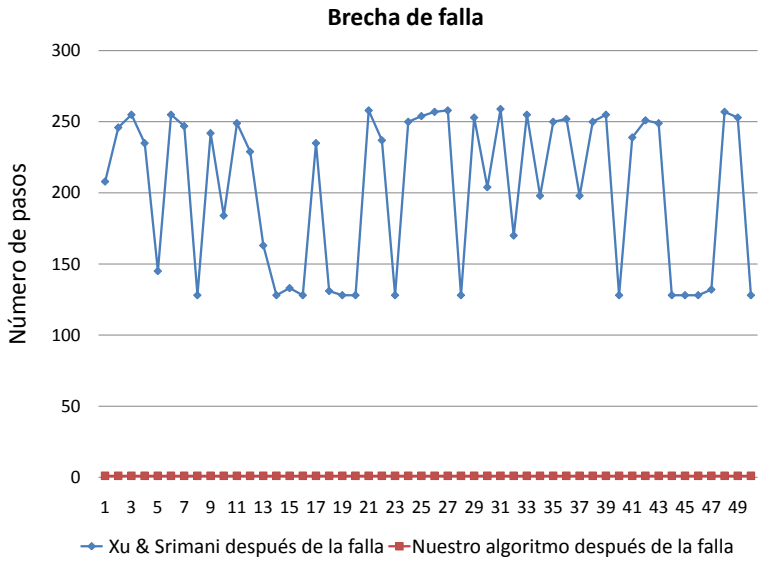

Figura 66. Número de pasos de la brecha de falla para un árbol de 128 nodos.

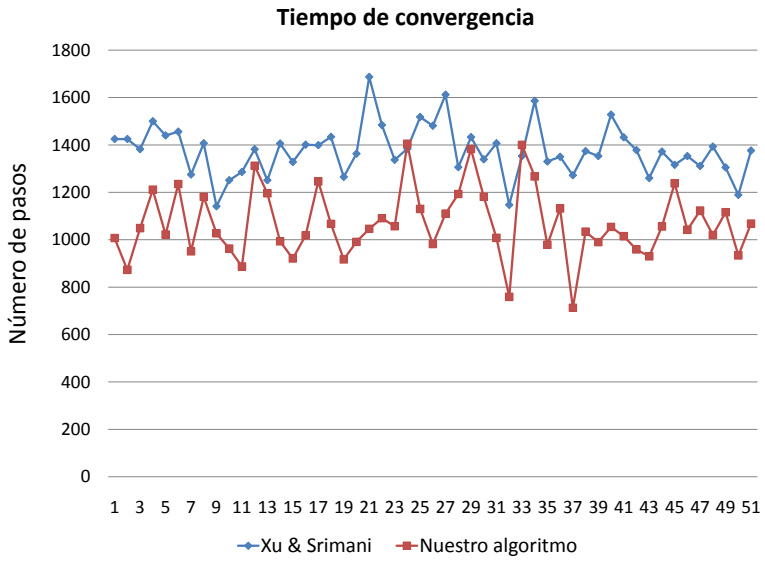

Figura 67. Número de pasos de convergencia para un árbol de 256 nodos.

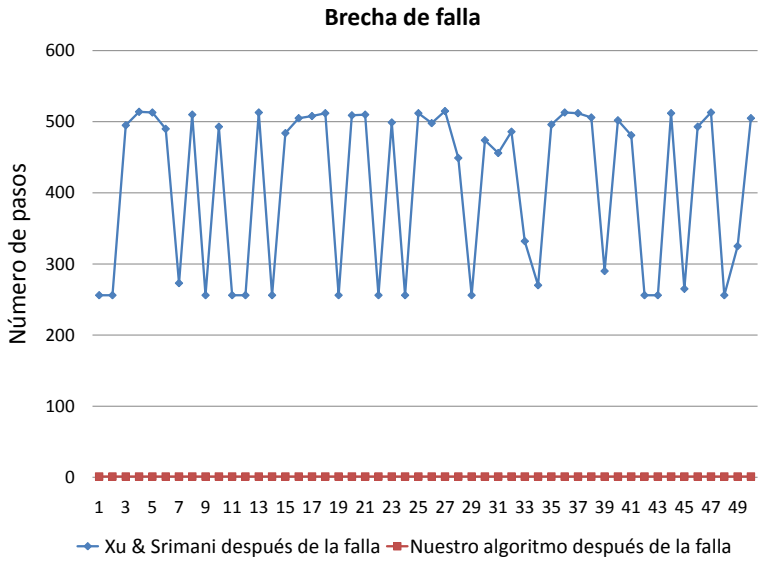

Figura 68. Número de pasos de la brecha de falla para un árbol de 256 nodos.

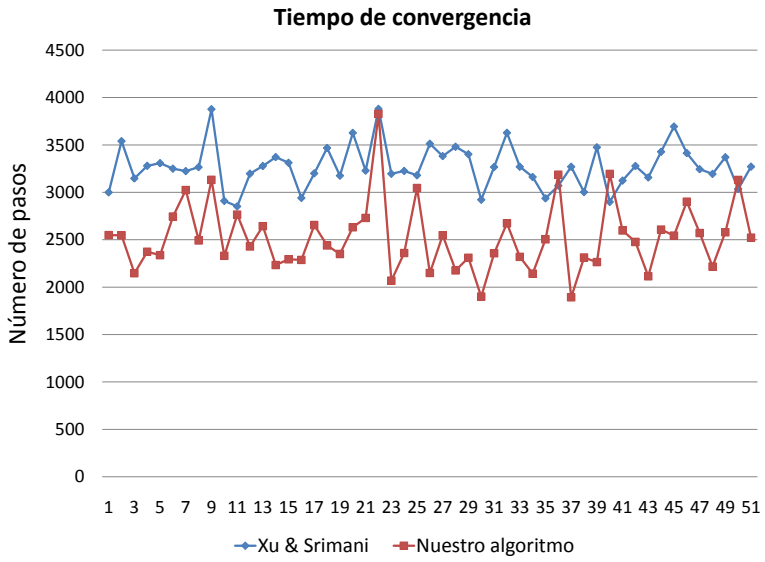

Figura 69. Número de pasos de convergencia para un árbol de 512 nodos.

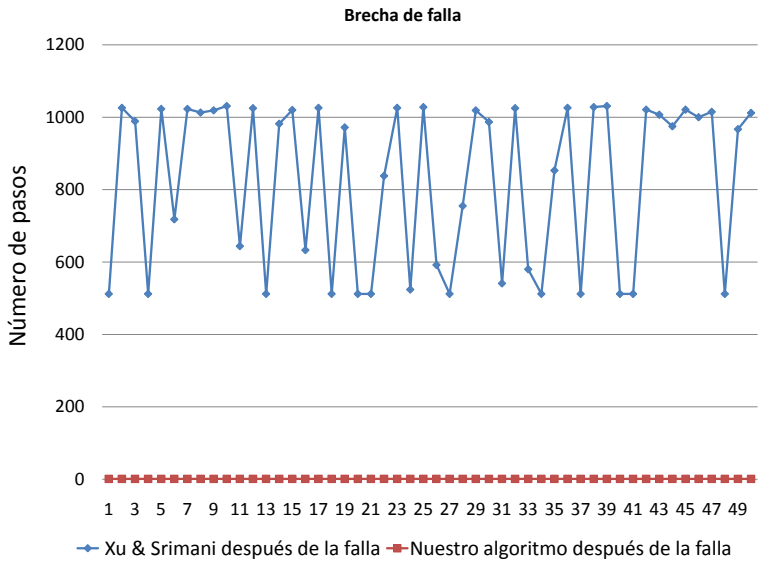

Figura 70. Número de pasos de la brecha de falla para un árbol de 512 nodos.

Apéndice B

GRÁFICAS DE RESULTADOS EXPERIMENTALES

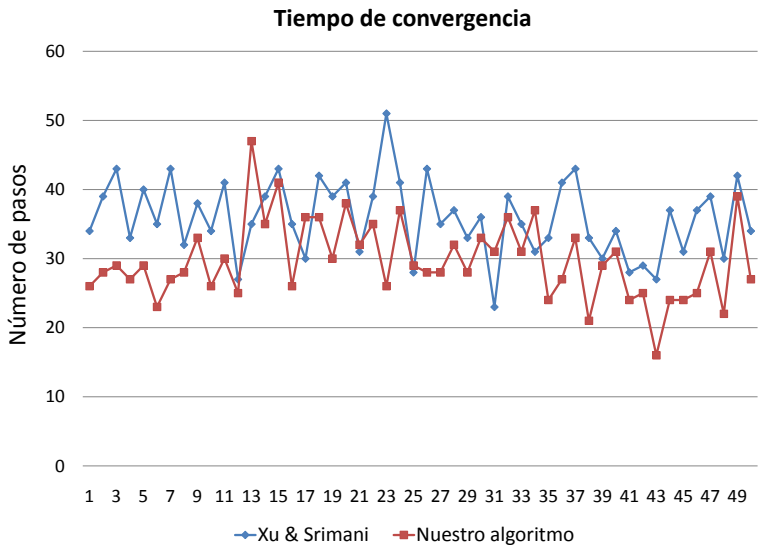

Figura 71. Número de pasos de convergencia para un árbol de 16 nodos.

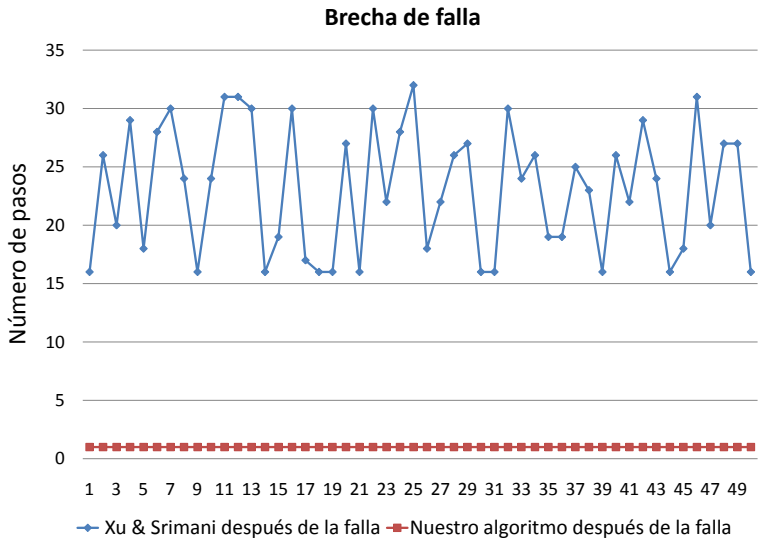

Figura 72. Número de pasos de la brecha de falla para un árbol de 16 nodos.

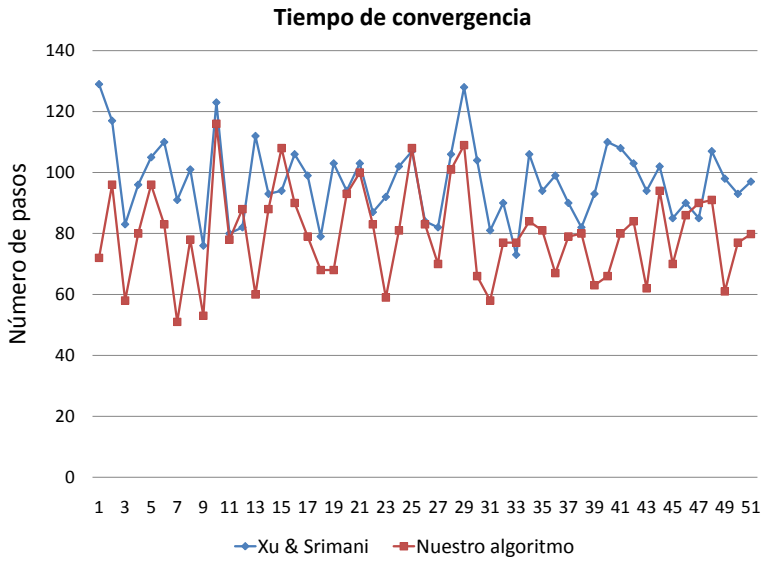

Figura 73. Número de pasos de convergencia para un árbol de 32 nodos.

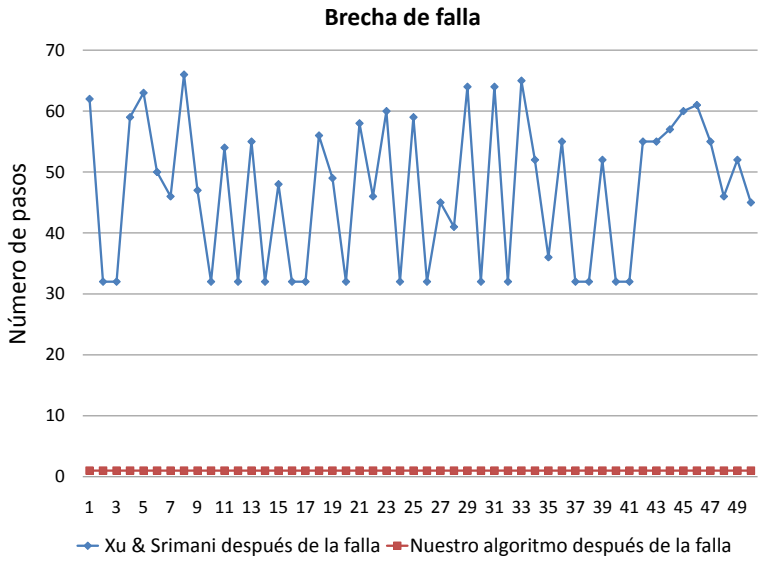

Figura 74. Número de pasos de la brecha de falla para un árbol de 32 nodos.

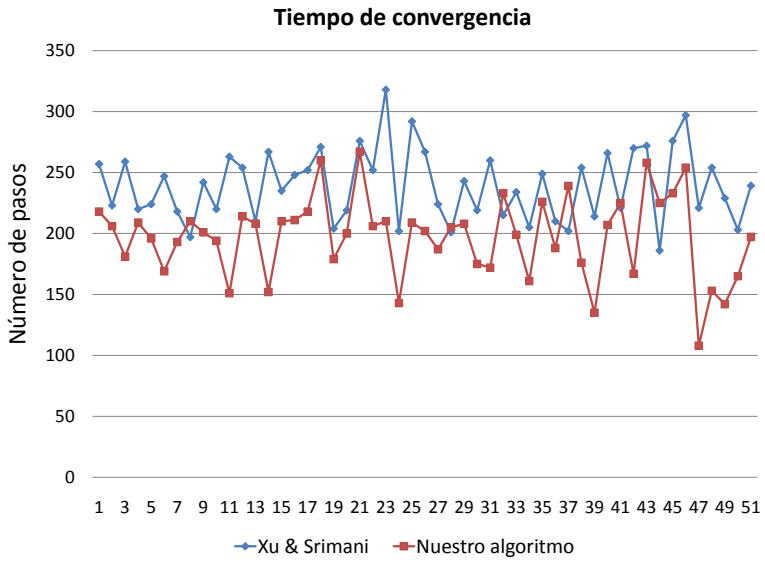

Figura 75. Número de pasos de convergencia para un árbol de 64 nodos.

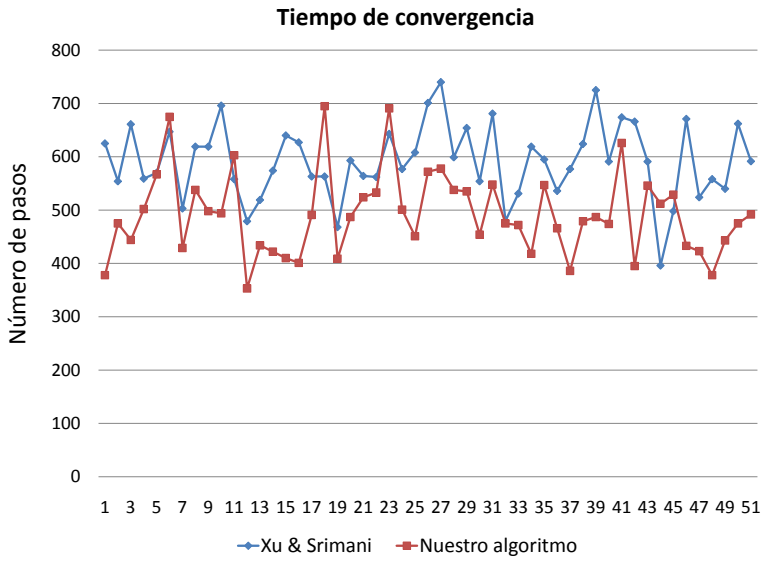

Figura 76. Número de pasos de la brecha de falla para un árbol de 128 nodos.

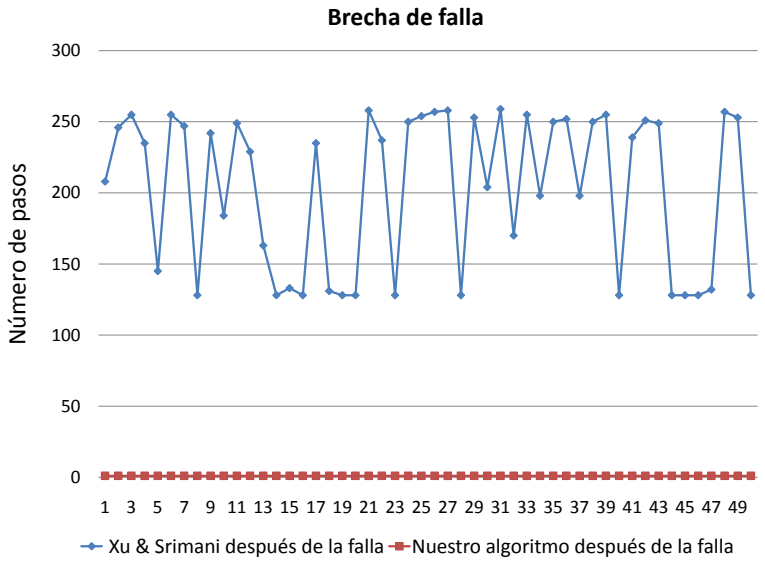

Figura 77. Número de pasos de la brecha de falla para un árbol de 128 nodos.

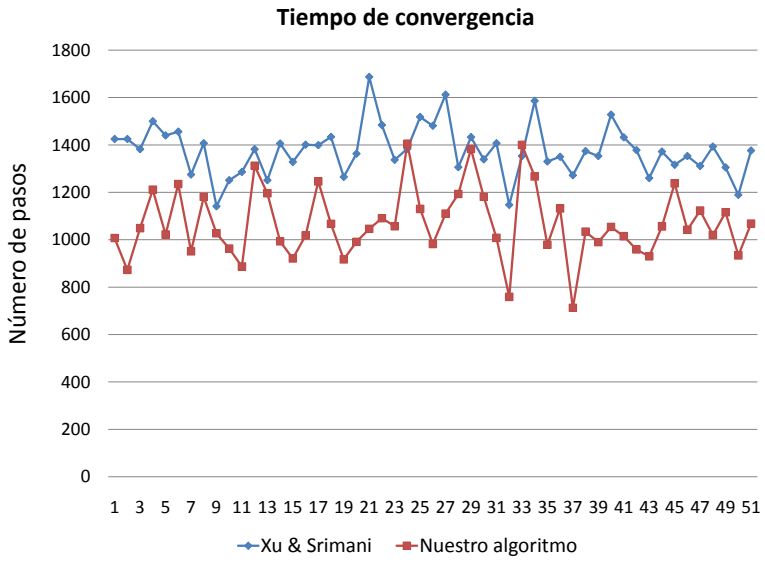

Figura 78. Número de pasos de convergencia para un árbol de 256 nodos.

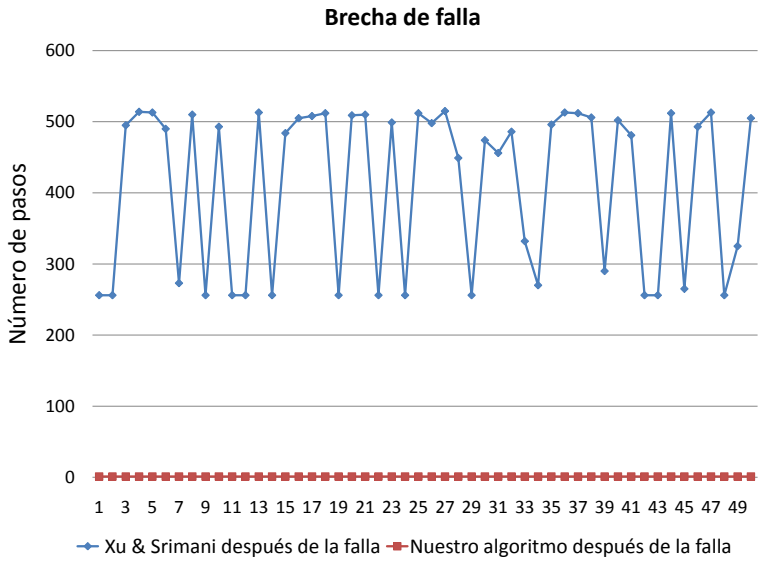

Figura 79. Número de pasos de la brecha de falla para un árbol de 256 nodos.

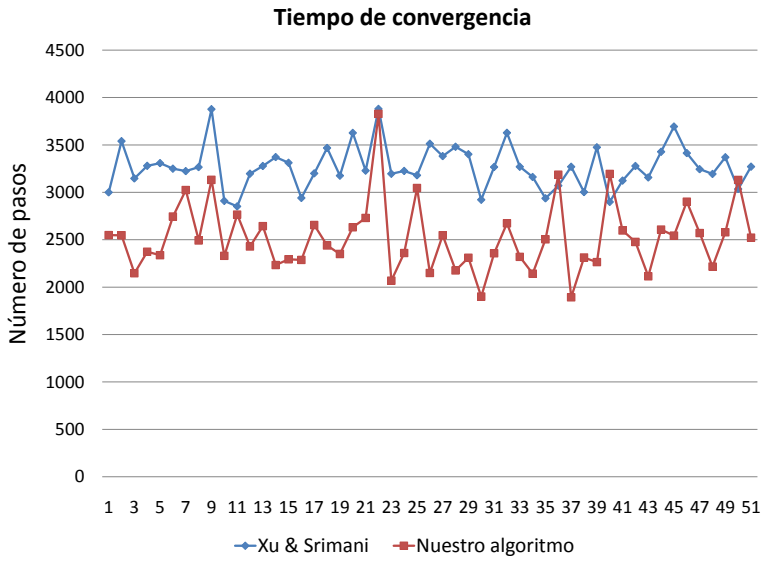

Figura 80. Número de pasos de convergencia para un árbol de 512 nodos.

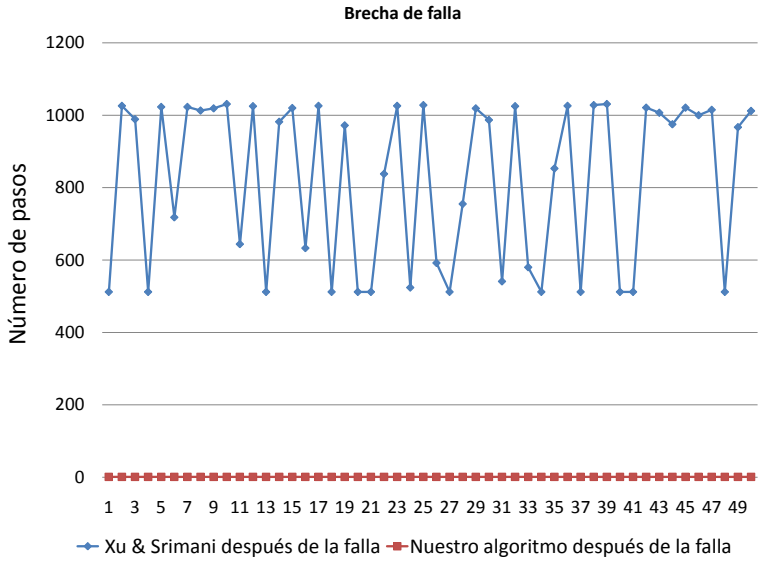

Figura 81. Número de pasos de la brecha de falla para un árbol de 512 nodos.

Apéndice C

SIMULADOR

Se desarroll ì un software que simula el algoritmo propuesto en esta tesis y lo compara con el algoritmo de Xu y Srimani en arboles aleatorios de altura O(log n). Los arboles ì ì se construyeron con base en uno de los criterios planteados en Alvarado (2006); los detalles se dicha generación de arboles se discute en la Secci ìn VII.1.

El simulador presenta de forma gráfica los resultados y adicionalmente genera un reporte con la información necesaria para analizar el desempeño del algoritmo en diferentes etapas de su ejecución; desde la configuraci ìn inicial del sistema hasta la configuración estable del mismo junto con los pasos realizados por cada nodo en la ejecución.

Adicionalmente el simulador permite la ejecución por lotes de dos o más experimentos de entradas diferentes. Al final se construye un reporte que resume el tiempo promedio del total de experimentos.

BlueJ es un entorno de desarrollo integrado (IDE por sus siglas en inglés) para el lenguaje de programación.

BlueJ ofrece un entorno sofisticado que permite una mayor libertad de interacción y experimentación por parte del usuario y ofrece una visualización que muestra claramente la estructura de clases del proyecto. El IDE BlueJ es gratuito y está disponible para entornos Linux/Unix, Windows y MacOs. www.bluej.org.

El software desarrollado en Bluej se apoya en GraphViz para realizar la representación de los grafos generados. Graphviz es un conjunto de herramientas que

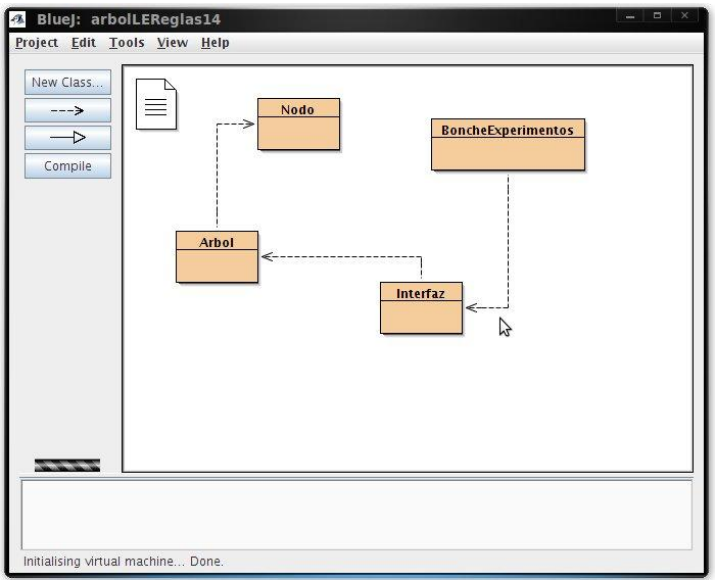

Figura 82. Pantalla principal del simulador desarrollado.

permiten representar información estructural por medio de diagramas gráficos o de redes. Graphviz es un software libre desarrollado por los laboratorios ATT por John Ellson en los Estados Unidos de Norteamérica.

El software se diseñó utilizando el Modelo Vista Controlador Parc (1979) (ver Figura 82) que separa el algoritmo, la interfaz de usuario y la lógica de control en componentes distintos. En la figura se muestran las clases principales del software. La clase N odo incluye el algoritmo que cada nodo del arbol ejecuta, la clase Arbol incluye la lógica de control del simulador e Interf az presenta al usuario las opciones de ejecución y construye los archivos de reporte.

El software se diseñó utilizando una laptop hp de la serie dv2000, con un procesador AMD Turion 64x2 a 1.8 ghz y 2 gb de memoria RAM con sistema operativo Linux a 32 bytes, versión Linux Mint 6. Las pruebas del algoritmo se realizaron en la computadora descrita y en una desktop Dell con un procesador intel a 2.4 ghz y 1 gb de memoria RAM con sistema operativo Windows XP.