Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas

Simulación y Análisis de un Algoritmo de Elección de Líder Tolerante a Fallas en un Sistema Distribuido Empleando Canales ADD

T E S I S

que para optar por el grado de
Maestro en Ciencia e Ingeniería de la Computación

PRESENTA:
Carlos López Rodríguez

DIRECTOR DE TESIS:
Dr. Sergio Rajsbaum Gorodezky
Instituto de Matemáticas

Ciudad de México Diciembre, 2020

Simulación y Análisis de un Algoritmo de Elección de Líder Tolerante a Fallas en un Sistema Distribuido Empleando Canales ADD

por

Carlos López Rodríguez

Ingeniero en Computación

Tesis que para optar por el grado de

Maestro en Ciencia e Ingeniería de la Computación

en el

Posgrado en Ciencia e Ingeniería de la Computación

Universidad Nacional Autónoma de México

Ciudad de México. Diciembre, 2020

A mis padres por todo el apoyo brindado
A mis hermanos por ser mis grandes fuentes de inspiración
A mis profesores por compartir su conocimiento y todas sus valiosas lecciones
A todos mis colegas del Instituto por el apoyo y la motivación

Agradecimientos

Al CONACYT por el apoyo económico que me permitió realizar estos estudios de posgrado.

A la Universidad Nacional Autónoma de México por abrirme las puertas al conocimiento y de nuevas experiencias, permitirme estudiar un posgrado y los años de formación previa.

Al Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) que a través de su Laboratorio Universitario de Cómputo de Alto Rendimiento (LUCAR) me permitió llevar a cabo este trabajo de investigación.

Al Dr. Sergio Rajsbaum Gorodezky y la M.C. Karla Rocío Vargas Godoy por compartir sus conocimientos, por la orientación y la paciencia brindada en la realización de este trabajo. Por motivar en mi la pasión por la investigación y la habilidad para plasmarlo en estas páginas.

Investigación realizada gracias al Programa UNAM-PAPIIT IN106520.

Índice general

1. Introducción **1**

1.1. Antecedentes y contexto del trabajo 1

1.1.1. Sistemas Distribuidos 1

1.1.2. Detectores de Fallas 2

1.1.3. Algoritmos de Elección de Líder 3

1.1.4. Modelo de Comunicación ADD 3

1.2. Motivación Objetivos y Contribución 4

1.3. Organización de la tesis 4

2. Detectores de Fallas **6**

2.1. Propiedades de un Detector de Fallas 10

2.1.1. Integridad y Precisión (*Completeness and Accuracy*) 10

2.1.2. Clasificaciones de los Detectores de Fallas. 10

2.2. Tipos de Detectores de Fallas 13

2.3. El Problema del Consenso 13

2.3.1. El Problema de los Generales Bizantinos 15

2.4. El detector de fallas Omega 17

3. Algoritmos Distribuidos **18**

3.1. Conceptos Fundamentales 19

3.2. Algoritmos de Elección de Líder en Sistemas Distribuidos 22

3.2.1. Definición del Problema . . . 23
3.2.2. Imposibilidad de un Resultado en Sistemas Anónimos . . . 24
3.2.3. Suposiciones y Principios Básicos en Algoritmos de Elección de Líder . . . 25
3.3. Notas y Lecturas Complementarias Recomendadas . . . 26

4. Sistemas de Eventos Discretos **27**
4.1. Eventos . . . 29
4.2. Sistemas Basados en Tiempo y Sistemas Basados en Eventos . . . 29
4.3. Simulación de Eventos Discretos . . . 31
4.3.1. Esquema de Calendarización de Eventos(*Event Scheduling Scheme*) . . . 32

5. Desarrollo **36**
5.1. Modelo de Comunicación ADD . . . 39
5.1.1. Características del Modelo ADD Original . . . 40
5.1.2. Características del Modelo ADD Simplificado . . . 41
5.2. Modelo del Sistema a Simular . . . 42
5.2.1. Modelo y Comportamiento de los Procesos . . . 42
5.2.2. Modelo de la Red . . . 43
5.2.3. Modelo y Comportamiento de los Canales . . . 43
5.2.4. Principio General del Algoritmo . . . 44
5.2.5. Comportamiento Detallado de los Procesos . . . 45
5.3. Algoritmo de Elección Eventual de Líder en el modelo $\diamond ADD$ con Pertenencia Conocida . . . 47
5.4. Construcción del Simulador del Sistema de Procesos Propensos a Fallas Ejecutando el Algoritmo de Elección Eventual de Líder en el Modelo $\diamond ADD$ con Pertenencia Conocida . . . 49
5.4.1. Abstracción de los Procesos y Canales ADD . . . 50
5.4.2. Comunicación de los Procesos . . . 56
5.4.3. Descripción de las Redes en el Simulador . . . 60

5.5. Simulador en su Versión Web 62
5.5.1. Arquitectura del Simulador 62
5.5.2. Estructura y Peticiones del Cliente 63
5.5.3. Vista del Cliente 64
5.5.4. Estructura del Servidor y la Base de Datos 65

6. Resultados **71**
6.1. Infraestructura que Soporta el Simulador 71
6.1.1. Descripción de la Infraestructura que Soporta el Simulador 71
6.2. Descripción de los Experimentos Realizados 72
6.2.1. Descripción de los Experimentos sin Fallas de los Procesos 74
6.2.2. Descripción de los Experimentos con Fallas de Procesos 77

7. Conclusiones **83**

A. Clases Auxiliares Seleccionadas del Simulador en el *back-end* **85**

B. Código de Componentes Seleccionados del *front-end* **96**

Índice de tablas

2-1. Ocho clases de detectores de fallas en Términos de Integridad y Precisión 10

Índice de figuras

2-1. Comportamiento de un Detector de Fallas en un Diagrama de Tiempos 8

2-2. Comunicación confusa entre Generales Bizantinos 16

3-1. Representación de Algunos Grafos de Estudio Común 21

4-1. Ubicación de los Sistemas de Eventos Discretos dentro de la Clasificación General de Sistemas 28

4-2. Esquema de Calendarización de Eventos en una Simulación por Computadora . . 34

5-1. Logo del *framework* SimPy para Simulación de Sistemas de Eventos Discretos . . 36

5-2. Algoritmo de Elección Eventual de Líder en modelo $\diamond ADD$ con Pertenencia Conocida. Código en p_i 47

5-3. Ejemplo de Archivo en Notación GML y Grafo Generado 62

5-4. Arquitectura del Simulador en su Versión Web 66

5-5. Estructura del Proyecto del Cliente del Simulador Empleando Angular y *request* para Iniciar una Simulación 67

5-6. Vistas del Cliente Web. Apartado de Simulación y Búsqueda 68

5-7. Vistas del Cliente Web. Apartado de Configuración, Resultados y Redes 69

5-8. Estructura del Servidor del Simulador Empleando Django y Modelo ER de la DB 70

6-1. Tiempo de Convergencia en Simulaciones de Anillos de Hasta 100 Nodos sin Fallas de Procesos. 73

6-2. Tiempo de Convergencia en Simulaciones de Anillos de Hasta 100 Nodos sin Fallas de Procesos. 75
6-3. Tiempo de Convergencia en Simulaciones de Anillos de Hasta 400 Nodos sin Fallas de Procesos. 76
6-4. Estimación de tiempo de convergencia empleando los modelos obtenidos 77
6-5. Comportamiento de los tiempos de convergencia para redes regulares aleatorias de hasta 50 000 nodos. Tasa de pérdidas del 1 % para distintos valores del Parámetro T 78
6-6. Comportamiento de los tiempos de convergencia para redes regulares aleatorias de hasta 50 000 nodos. Tasa de pérdidas del 1 % para distintos valores del Parámetro T 79
6-7. Comportamiento de los tiempos de convergencia para anillos de hasta 200 procesos. Tasa de pérdidas del 1 % cuando el líder falla en el instante previo a alcanzar la convergencia por primera vez 81
6-8. Comportamiento de los tiempos de convergencia para anillos de hasta 200 procesos. Tasa de pérdidas del 1 % cuando el líder falla en el instante previo a alcanzar la convergencia por primera vez 82

Simulación y Análisis de un Algoritmo de Elección de Líder Tolerante a Fallas en un Sistema Distribuido Empleando Canales ADD

by

Carlos López Rodríguez

Abstract

Failure Detection constitutes the core of reliable Distributed Systems building. Fault tolerant systems, the result of handling Failure Detection, is an even more challenging task. When we put it all together: building correct, efficient, easily scalable and fault tolerant systems we have an extremely complex job at hand. In pursuing such task its common to sacrifice one of those features or make the environment of such system a little more forgiving. What's the reason behind this complexity? Timing issues is the reason distributed systems that leads them to unpredictable behavior, locking, race conditions, etc.

This work presents the necessary background and mechanisms to implement a distributed system simulation. This simulation relies on a system capable of eventual leader election, namely an Ω failure detector, where processes are prone to failing. This system communicates through a weak partially synchronous model of communication known as $\diamond ADD$.

Such simulation relies on a process oriented framework based on Python programming language.

Simulación y Análisis de un Algoritmo de Elección de Líder Tolerante a Fallas en un Sistema Distribuido Empleando Canales ADD

por

Carlos López Rodríguez

Resumen

La Detección de Fallas constituye una parte fundamental de la creación de Sistemas Distribuidos confiables. Mas allá de la detección de fallas está la creación de Sistemas Distribuidos tolerantes a fallas. Este objetivo de construir sistemas correctos, eficientes, de fácil escalabilidad y que adicionalmente sean tolerantes a fallas resulta una tarea sumamente compleja en la que al menos una de esas características debe sacrificarse o el entorno de aplicación debe permitir algún o algunos compromisos para mantener la creación de estos sistemas en el mundo real. Detrás de esta complejidad inherente de los sistemas distribuidos están las variaciones temporales que llevan a los sistemas a comportamientos impredecibles.

En este trabajo se presenta los mecanismos necesarios para realizar la simulación de un sistema distribuido de procesos para la elección eventual de un líder, conocido como Ω, comunicados por un modelo de comunicación débil parcialmente síncrono denominado $\diamond ADD$, donde los procesos son susceptibles a fallas empleando un algoritmo de paso de mensajes de un tamaño reducido.

Esta simulación se logra empleando un *framework* de simulación orientado a procesos basado en el lenguaje Python.

Capítulo 1

Introducción

1.1. Antecedentes y contexto del trabajo

En la actualidad los sistemas de cómputo se encuentran en muchas de las actividades que realizamos cotidianamente y cada vez en más campos con aplicaciones increíblemente diversas. No solamente se han vuelto más rápidas las computadoras que empleamos, se han popularizado a un nivel en que las redes de computadoras se encuentran incluso en dispositivos móviles. Es en este contexto que aparece el concepto de Sistemas Distribuidos.

1.1.1. Sistemas Distribuidos

Por este concepto de Sistema Distribuido entendemos múltiples componentes de software que se encuentran en múltiples dispositivos pero que se comportan como un sistema único. Estos dispositivos se pueden encontrar físicamente cercanos en una red local o físicamente distantes empleando una red de área extensa.

Un sistema distribuido puede ser asíncrono, es decir que no existe un tiempo específico para el tiempo que le toma a un proceso para ejecutar un paso de cómputo, o para un mensaje viajar desde su emisor hasta el receptor.

Adicionalmente un sistema distribuido debe permitir la ejecución confiable y continua de servicios a pesar de la falla de alguno de sus elementos o componentes. Por estas razones la

construcción de sistemas distribuidos que sean tolerantes a fallas es una tarea retadora. Como consecuencia la detección de fallas se vuelve de vital importancia en la creación de estos sistemas.

1.1.2. Detectores de Fallas

Una aproximación para construir sistemas distribuidos tolerantes a fallas es iniciar por la detección de fallas en estos sitemas, es decir, la detección de cambios en el comportamiento normal. Un detector de fallas es ampliamente aceptado como un oráculo, un elemento que proporciona información respecto a los dispositivos que conforman el sistema distribuido y los procesos que se ejecutan en ellos, que puede sospechar de forma inteligente cuando un proceso ha fallado [1].

Estos detectores de fallas son empleados en redes de comunicaciones y clústeres de computadoras, ambos ejemplos de sistemas distribuidos. Sin embargo, no es suficiente con la generación de sospecha cuando un proceso ha fallado, un detector de fallas debe también cumplir con criterios de precisión y restricciones de detección de las fallas en un umbral de tiempo determinado.

Entonces entendemos un detector de fallas como una capa de abstracción que separa lo relacionado con temporizadores y detección de fallas de bajo nivel, de lo que un desarrollador puede ver con una interfaz que simplemente provee de información sobre el estatus operacional de los procesos en el sistema. [2].

En esencia un detector de fallas se define por dos propiedades básicas, integridad (*completeness*) y precisión (*accuracy*). La primera propiedad corresponde a la detección real de fallas, es decir que el detector sospecha de los procesos que han fallado y la segunda propiedad se refiere a la restricción de errores, procesos sospechados de forma incorrecta, que el detector puede tener.

Las estrategias para la detección de fallas basadas en la estimación de tiempos de llegada de *heartbeats* parecen ser las más aceptables además de contar con información disponible en la literatura [3].

1.1.3. Algoritmos de Elección de Líder

Adicionalmente la elección de un líder en el mundo del cómputo distribuido y las redes móviles es de gran importancia. Un líder es el responsable de asegurar la sincronización entre varios dispositivos, puede organizar tareas, actualizar la información de encaminamiento en una red, etc. De ahí la importancia de estos algoritmos en un sistema distribuido [4].

1.1.4. Modelo de Comunicación ADD

Usualmente cuando se abordan problemas de tolerancia y detección de fallas se hacen suposiciones sobre la confiabilidad y características de entrega en tiempo de los canales que soportan la comunicación. Estos modelos de comunicación entre los procesos hacen ciertas suposiciones sobre los canales, estas suposiciones pueden ser que los canales siempre se comportan de forma confiable o que eventualmente[1] se comportan de esa forma. En otras palabras, los modelos de comunicación suponen que los canales pierden a lo más una cantidad finita de mensajes hasta cierto punto en el tiempo a partir del cual los canales que conectan los procesos se comportarán siempre de forma confiable. Otras suposiciones en los modelos citados es la existencia de una cota superior, conocida o no, para el retraso que un mensaje puede tener de un punto a otro o la existencia de un retraso promedio.

Estas suposiciones evidentemente no reflejan lo que ocurre en muchos sistemas, basta con saber que los retrasos o pérdidas de los mensajes pueden ocurrir de forma intermitente y espontánea en la duración de la comunicación. Es bajo esta realidad que se describe el modelo de sincronía parcial basado en enlaces no confiables llamados Canales ADD (*Average Delayed/Dropped*). En [5] se parte de la idea que este modelo de comunicación presenta un modelo más débil de sincronía a los previamente ocupados en sistemas detectores de fallas y tolerantes a fallas.

Este modelo de comunicación resulta de particular interés para realizar el estudio y la descripción del comportamiento de los detectores de fallas mediante la experimentación y la

[1]Se emplea eventualmente como traducción del término en inglés *eventually*, algunos autores consideran que una traducción más fiel al significado original es *finalmente*.

simulación de estos sistemas. Lo anterior ya que existe un número considerablemente menor cuando se compara el trabajo experimental con las contribuciones en *journals* de inclinación puramente teórica.

1.2. Motivación Objetivos y Contribución

A pesar de la gran cantidad de información que se encuentra en la literatura, esta consiste principalmente en trabajos, conferencias y *journals* de inclinación teórica [6].

Esta es la razón que motiva el desarrollo de este trabajo. Por ello buscamos realizar una simulación de un algoritmo de elección de líder que emplea ideas y conceptos de detectores de fallas en sistemas asíncronos volviéndolo tolerante a fallas en la red. Nuestra intención al realizar esta simulación es entender el comportamiento de este algoritmo en redes cercanas a las que podemos encontrar en el mundo real y describiendo su desempeño y proponiendo mejoras al algoritmo actual.

1.3. Organización de la tesis

La introducción de la información se realiza de forma incremental en este trabajo. Iniciamos con una revisión detallada de los conceptos fundamentales que soportan el tema de estudio. En el capítulo 2 se revisan los conceptos de los detectores de fallas, su funcionamiento, sus clasificaciones y las propiedades que lo definen. Se revisa la descripción realizada por Chandra y Toueg inicialmente y consideramos las propiedades de precisión e integridad que dan lugar a 8 clasificaciones de los detectores de fallas.

Una vez que contamos con los conceptos de detección de fallas, el tipo de problemas que podemos resolver con ellos y su funcionamiento, en el capitulo 3 se revisan los conceptos que conforman el modelo empleado por algoritmos distribuidos. Revisamos la definición de lo que entendemos por un sistema distribuido, el modelo de comunicación en estos sistemas y algunas técnicas que existen para su representación y estudio. Para concluir este capitulo se revisa con detalle en que consiste un algoritmo de elección de líder en un sistema distribuido.

El capítulo 4 presenta las ideas que rigen la simulación de eventos discretos. Por ello revisamos los sistemas de eventos discretos formalmente, alternativas que existen para realizar simulaciones con ayuda de la computadora y se introduce la idea de los simuladores de eventos discretos.

El capítulo 5 se presenta el algoritmo original que se simulará, las herramientas que se emplearon, el modelo y la arquitectura del simulador detalladamente.

El capítulo 6 se centra en la descripción de los experimentos realizados y análisis de los resultados obtenidos.

Este trabajo cierra con un capítulo de conclusiones, trabajo futuro y un apéndice donde se presentan los componentes de código centrales del simulador.

Capítulo 2

Detectores de Fallas

Los detectores de fallas son una abstracción fundamental cuando se habla de sistemas distribuidos. Desde la introducción en 1996 por T. D. Chandra y S. Toueg en el artículo *Unreliable Failure Detectors for Reliable Distributed Systems* [7] se reconoce el diseño y la verificación de aplicaciones y sistemas que sean tolerantes a fallas como una tarea desafiante. Al igual que en muchos otros casos cuando una tarea resulta difícil de abordar se introducen algunas restricciones al problema en cuestión o un nuevo paradigma que permite tratarlo. Siguiendo esa idea se introdujeron los conceptos de Consenso (*Consensus*) y de Difusión Atómica (*Atomic Broadcast*).

El consenso, como su nombre lo sugiere, permite llegar a un acuerdo común por las partes involucradas, en este caso procesos. Este acuerdo común siempre[1] se alcanza dependiendo únicamente de las entradas al sistema sin importar las fallas que hayan tenido lugar en la ejecución del sistema.

La Difusión Atómica permite a los procesos difundir los mensajes de forma confiable, permitiendo el acuerdo entre ellos de los mensajes que se entregan y el orden de las entregas.

La solución descrita por Chandra y Toueg se centra en sistemas distribuidos asíncronos. En estos sistemas no existe un límite para el tiempo que le toma a un proceso ejecutar un paso

[1]El término "siempre", se emplea unicamente para enfatizar que el consenso depende exclusivamente de las entradas del sistema. El lector debe entender que existen condiciones en los Sistemas Distribuidos en los que alcanzar este común acuerdo resulta imposible. Tal es el caso de un sistema distribuido Asíncrono donde un proceso falla de forma no anunciada (*FLP Impossibility of Consensus*) [8]

de cómputo o en el tiempo que le toma a un mensaje para ir de su emisor al receptor. En un sistema asíncrono no existen cotas superiores para la velocidad relativa del procesador, tiempos de ejecución o retrasos durante la transmisión de los mensajes, aunque estos tiempos son finitos. Por esta razón podemos decir que un sistema asíncrono no realiza suposiciones de sincronización de ningún tipo. La razón para esto es que existen cargas impredecibles en el sistema que nos llevan a no poder realizar ninguna suposición. También podemos hablar de sistemas síncronos que se caracterizan por tener parámetros estrictos en los tiempos de ejecución, de transmisión y del retardo de los mensajes.

El modelo asíncrono tiene ciertas ventajas cuando se compara con el modelo síncrono en sistemas distribuidos. Una de estas ventajas es que las aplicaciones basadas en el modelo asíncrono permiten una mayor portabilidad pues no existen suposiciones temporales que deban tomarse en cuenta al momento de implementarlas. Ahora bien, a pesar de que esta es una razón muy atractiva para trabajar con este modelo, se ha demostrado que otros problemas de transmisión confiable no pueden resolverse de forma determinista incluso para un sólo proceso debido a estas características temporales sin restricción. La principal causa para la imposibilidad de conocer este resultado es que resulta muy difícil determinar en un sistema asíncrono si un proceso ha fallado o simplemente esta tomando un tiempo muy largo para su ejecución; resultando en una gran dificultad para lidiar con fallas en estos sistemas.

A pesar de estas limitaciones el modelo asíncrono en sistemas distribuidos es ampliamente utilizado, volviendo la tarea de detección de fallas una tarea de vital importancia para estos sistemas.

Intuitivamente entendemos un detector de fallas como una capa de abstracción que separa lo relacionado con temporizadores y detección de fallas de bajo nivel, de lo que un desarrollador puede ver con una interfaz que simplemente provee de información sobre el estatus operacional de los procesos en el sistema. [2].

Para describir formalmente un detector de fallas consideremos un primer modelo donde

tenemos un sistema distribuido asíncrono en donde no existen restricciones de tiempo para los retrasos de los mensajes o el tiempo de ejecución de un paso computacional. Este sistema consiste de un número finito de n procesos, $Q = \{p_1, p_2, ..., p_n\}$. Donde cada par de procesos se encuentra conectado por un canal confiable de comunicación. Un proceso puede fallar únicamente por una interrupción prematura y este proceso se comporta de forma correcta hasta el momento de su posible falla.

Se asume un reloj discreto global y el rango de los pulsos del reloj, denotado Φ es el conjunto de números naturales. Este reloj global es empleado para simplificar la representación y razonamiento y no es accesible a los procesos.

Se dice que un proceso p_i ha fallado en el tiempo t si p_i no realiza ninguna acción después del tiempo t. Esta falla en un proceso es permanente: una vez que el proceso ha fallado este no se recupera. Un proceso se dice que es *correcto* si no ha fallado.

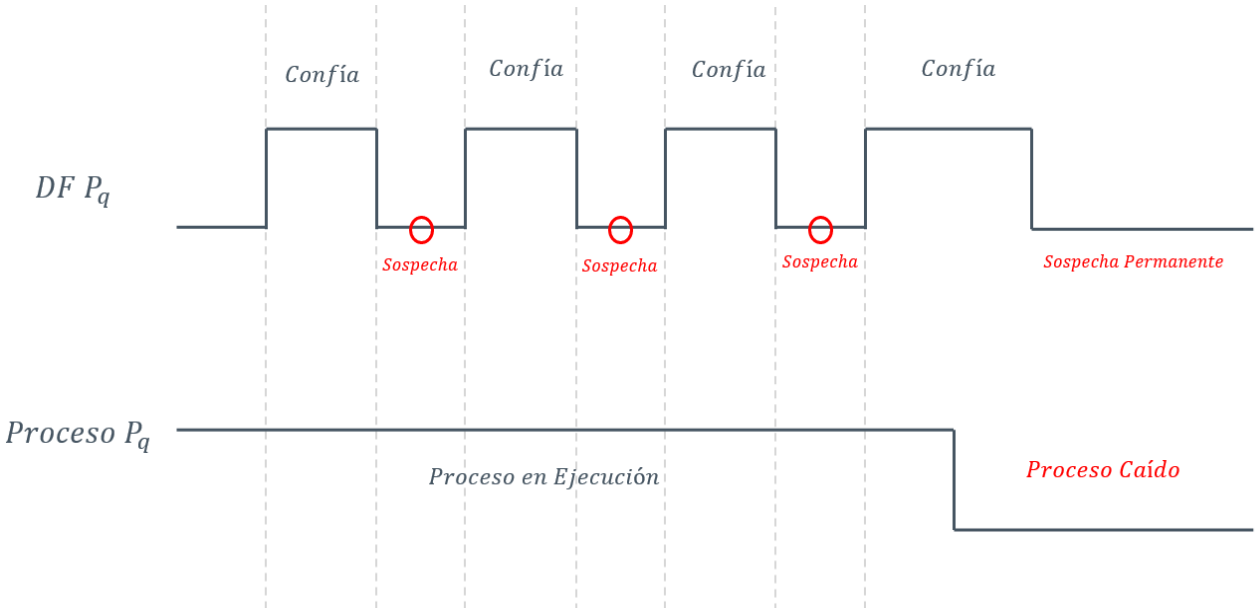


Figura 2-1: Comportamiento de un Detector de Fallas en un Diagrama de Tiempos. El Detector de Fallas puede sospechar falsamente en múltiples ocasiones, pero existe un instante a partir del cual sospechará definitivamente de un proceso que ha fallado

Por último, informalmente, una ejecución es una sucesión infinita de instrucciones en el sistema. Para una ejecución σ , *Fallados*(t, σ) es el conjunto de procesos que han fallado hasta

el tiempo t y *Vivos(t, σ)* es el conjunto de procesos que se encuentran correctos, es decir que no han fallado hasta el tiempo t. Entonces tenemos que *Vivos(t, σ)* = $Q-$ *Fallados(t, σ)*. Del mismo modo tenemos *Fallados(σ)* como los procesos que han fallado en una ejecución σ. En donde si $p \in Fallados(\sigma)$ decimos que p es un proceso que ha fallado en σ y por otro lado si $p \in Vivos(\sigma)$ decimos que un proceso es correcto en σ. Donde únicamente consideramos ejecuciones donde al menos un proceso es correcto.

Un detector de fallas, al que denotaremos como D, es un oráculo distribuido que sugiere patrones de fallas. Cada proceso p_i en el entorno distribuido tiene su propio detector de fallas D_i que monitorea todos los demás procesos y mantiene una lista de procesos de los que p_i sospecha en ese momento. Esta sospecha esta basada en *timeouts* de los otros procesos en p_i.

Por lo tanto un detector de fallas es un vector de la forma $D = \{D_{p_1}, D_{p_2}, ..., D_{p_n}\}$ donde D_i es el módulo detector de fallas en el proceso p_i que informa el conjunto de los procesos de los que en ese momento sospecha que han fallado. Formalmente un detector de fallas se define como la función de tiempo y del conjunto de todas las ejecuciones 2^Q.

$D_p(t, \sigma)$ es el conjunto de procesos que se sospecha que han fallado por el detector de fallas de p en el tiempo t en la ejecución σ. Si $q \in D_p(t, \sigma)$ se dice que el proceso p sospecha de q en el tiempo t en la ejecución σ.

Con esto se establece formalmente lo que es un detector de fallas. Recordemos que un detector de fallas puede equivocarse, esto es, que un proceso correcto puede agregarse a la lista de procesos de los que se sospecha y que posteriormente puede salir de esta lista si el detector de fallas reconoce que fue un error. Esto significa que el detector de fallas puede agregar y remover procesos de esta lista de procesos de los que se sospecha. Los procesos pueden agregarse o removerse de la lista de procesos sospechados por cada uno de los módulos detectores de fallas en cualquier momento, cualquier número de veces. Por último, respecto a esta lista de procesos sospechados, en cualquier momento los módulos detectores de fallas en dos procesos pueden tener una lista de procesos sospechados diferentes.

2.1. Propiedades de un Detector de Fallas

2.1.1. Integridad y Precisión (*Completeness and Accuracy*)

Chandra y Toueg clasificaron los detectores de fallas de acuerdo con sus propiedades de integridad y de precisión. Informalmente la propiedad de integridad (*completeness*) corresponde a la detección real de fallas, es decir que el detector sospecha de los procesos que han fallado y la propiedad de precisión (*accuracy*) se refiere a la restricción de errores que el detector puede tener. En [7] los autores definen dos tipos de integridad y cuatro tipos de precisión, surgiendo así las 8 clasificaciones de detectores de fallas. Adicionalmente en su trabajo Chandra y Toueg introdujeron el concepto de reducibilidad para detectores de fallas y en función de este concepto las ocho clases fueron ordenadas en una jerarquía. En esta jerarquía, algunos detectores de fallas pueden resolver el problema del consenso para cualquier número de procesos que han fallado, algunos otros detectores de fallas necesitan un número mínimo de procesos correctos para resolverlo.

2.1.2. Clasificaciones de los Detectores de Fallas.

Entre los detectores de fallas descritos por Chandra y Toueg encontramos las clasificaciones en términos de precisión e integridad que se muestran en la Tabla 2-1. Entre estas clasificaciones centramos nuestra atención en los detectores Eventualmente Perfectos ($\diamond P$) por ser suficientemente poderosos para resolver problemas como el consenso [7]. Más aún, esta clasificación de detectores de fallas es implementable en sistemas parcialmente síncronos.

	Precisión			
Integridad	**Fuerte**	**Débil**	**Eve. Fuerte**	**Eve. Débil**
Fuerte	Perfecto (P)	Fuerte (S)	E. Perfecto ($\diamond P$)	E. Fuerte ($\diamond S$)
Débil	Quasi Perfecto (Q)	Débil (W)	E.Quasi Perfecto ($\diamond Q$)	E. Débil ($\diamond W$)

Tabla 2-1: Ocho clases de detectores de fallas en Términos de Integridad y Precisión

En nuestra implementación particular trabajamos retomando las ideas de un detector de fallas basado en la propuesta de Toueg y Chen y canales ADD (*Average Delayed/Dropped*)

descritos formalmente en [5]. Ahora retomamos los conceptos de integridad y precisión y damos una definición formal.

Integridad (*Completeness*) Existe un tiempo después del cual todo proceso que ha fallado es permanentemente sospechado por un proceso correcto. De acuerdo con esta definición la propiedad de integridad puede ser de dos tipos:

1. **Integridad Fuerte (*Strong Completeness*)**. Eventualmente cada proceso que falla es permanentemente sospechado por todo proceso correcto. Esto es:

$$\forall \sigma, \forall p \in Fallados(\sigma), \forall q \in Vivos(\sigma), \exists\ t\ para\ el\ cual\ \forall t' \geq t : p \in D_q(t', \sigma)$$

2. **Integridad Débil (*Weak Completeness*)**. Eventualmente cada proceso que falla es permanentemente sospechoso por algún proceso correcto. Esto es:

$$\forall \sigma, \forall p \in Fallados(\sigma), \exists q \in Vivos(\sigma), \exists\ t\ para\ el\ cual\ \forall t' \geq t : p \in D_q(t', \sigma)$$

Por si sola la propiedad de integridad no es de mucha utilidad. Podemos decir que un detector de fallas satisface la propiedad de integridad fuerte al hacer que todos los procesos sospechen permanentemente de todos los demás procesos. Evidentemente este detector de fallas carece de cualquier utilidad, pues no provee ninguna información sobre procesos que realmente han fallado. Por ello es necesario complementar la propiedad de integridad con alguna propiedad de precisión (*accuracy*) que se define a continuación:

Precisión (*Accuracy*). Existe un tiempo después del cual un proceso no es sospechoso nunca por cualquier otro proceso correcto. Nuevamente para esta definición de precisión existen dos tipos:

1. **Precisión Fuerte (*Strong Accuracy*)**. Los procesos correctos nunca son sospechados por ningún otro proceso correcto.

$$\forall \sigma, \forall t, \forall p, q \in Vivos(t, \sigma) : p \notin D_q(t, \sigma)$$

2. **Precisión Débil (*Weak Accuracy*)**. Algún proceso correcto nunca es sospechado por ningún otro proceso correcto. Esto es:

$$\forall\sigma, \exists p \in Vivos(\sigma), \forall t, \forall q \in Vivos(t,\sigma) : p \notin D_q(t,\sigma)$$

Cuando hablamos de precisión y en el caso particular de la precisión fuerte en un sistema con cualquier tipo de aplicación practica, cumplir con esta propiedad es extremadamente difícil, razón por la que existe la propiedad de precisión débil e incluso aún con este tipo de precisión es difícil de satisfacer. La razón de esta dificultad es que en algún momento del tiempo un proceso, incluso un proceso correcto, puede sospechar de forma equivocada de otro proceso correcto para después darse cuenta de que sospechaba de él de forma equivocada.

Nuevamente se relaja la propiedad de precisión para permitir que el detector de fallas sospeche de un proceso correcto en algún instante de la ejecución, pero eventualmente (*eventually*) satisface las propiedades de precisión fuerte y débil. [9]

Precisión Eventual (*Eventual Accuracy*). No se requiere que la propiedad de precisión se satisfaga para cada proceso en todo momento sino únicamente se requiere que la propiedad de precisión sea satisfecha eventualmente. Esta definición da lugar a dos nuevos tipos de precisión eventual:

1. **Precisión Fuerte Eventual (*Eventual Strong Accuracy*)**. Existe un tiempo después del cual todos los procesos correctos no son sospechados por ningún proceso correcto. Esto es:

$$\forall\sigma, \exists t, \forall t' \geq t, \forall p, q \in Vivos(t',\sigma) : p \notin D_q(t,\sigma)$$

2. **Precisión Débil Eventual(*Eventual Weak Accuracy*)**. Existe un tiempo después del cual algún proceso correcto no es sospechado por ningún proceso correcto. Esto es:

$$\forall\sigma, \exists t, \forall t' \geq t, \exists p \in Vivos(\sigma), \forall q \in Vivos(\sigma) : p \notin D_q(t,\sigma)$$

2.2. Tipos de Detectores de Fallas

De acuerdo con el tipo de precisión y de integridad que un detector de fallas satisface, se les puede clasificar en las siguientes categorías:

1. **Detectores de Fallas Perfectos** (*Perfect Failure Detectors*) (P). Satisfacen la propiedad de integridad fuerte y precisión fuerte.

2. **Detectores de Fallas Eventualmente Perfectos** (*Eventually Perfect Failure Detectors*) ($\diamond P$). Satisfacen la propiedad de integridad fuerte y precisión fuerte eventual.

3. **Detectores de Fallas Fuertes** (*Strong Failure Detectors*) (S). Satisfacen la propiedad de integridad fuerte y precisión débil.

4. **Detectores de Fallas Eventualmente Fuertes** (*Eventually Strong Failure Detectors*) ($\diamond S$). Satisfacen la propiedad de integridad fuerte y precisión débil eventual.

5. **Detectores de Fallas Débiles** (*Weak Failure Detectors*) (W). Satisfacen la propiedad de integridad débil y precisión débil.

6. **Detectores de Fallas Eventualmente Débiles** (*Eventually Weak Failure Detectors*) ($\diamond W$). Satisfacen la propiedad de integridad débil y precisión débil eventual.

7. **Detectores de Fallas Casi Perfectos** ($\diamond Q$). Satisfacen la propiedad de integridad débil y precisión fuerte.

8. **Detectores de Fallas Eventualmente Casi Perfectos** ($\diamond Q$). Satisfacen la propiedad de integridad débil y precisión fuerte eventual.

2.3. El Problema del Consenso

En la sección anterior se abordan algunas clasificaciones de detectores de fallas. Ahora se aborda uno de los problemas fundamentales en sistemas distribuidos considerado por muchos investigadores en el área como el problema fundamental del cómputo distribuido: el consenso

distribuido. Es decir, cómo lograr que un conjunto de procesos, ejecutándose en distintos nodos, se pongan de acuerdo con respecto al valor de un dato. Para que cualquier protocolo de coordinación funcione correctamente, éste debe verificar que todos los procesadores correctos se pongan de acuerdo con respecto al valor del dato que intercambian. El problema del consenso consiste en conseguir que, aún en presencia de procesadores erróneos, el acuerdo entre los procesos del sistema se alcance.

También llamado el problema del acuerdo, constituye un requerimiento fundamental para un gran número de aplicaciones. Muchas formas de coordinación en una aplicación requieren que los procesos intercambien información para negociar entre sí y finalmente llegar a un acuerdo común, antes de tomar alguna acción particular. Un ejemplo clásico de consenso es la operación de consolidar (*commit*) en sistemas de bases de datos, donde los procesos deciden colectivamente si se realiza la operación de consolidación (*commit*) o se cancela una transacción (*rollback*) en la que participan.

Al intentar dar una solución al problema del consenso se involucran, en alguna combinación, las siguientes características o propiedades. En función de estas combinaciones es que la solución al problema del consenso puede volverse extremadamente compleja o bien ser imposible de resolver.

1. Modelos de falla. Donde se entiende que dentro de los n procesos, a lo más f procesos pueden ser procesos que han fallado. Ejemplos de modelos de fallas son las fallas bizantinas, *fail-stop*, omisión de envío o de recepción.

2. Comunicación síncrona o asíncrona. Como ya se mencionó en un sistema asíncrono propenso a fallas cuando un proceso decide enviar un mensaje a P_i pero falla, P_i no puede detectar esta falla por ser indistinguible del caso en que la transmisión del mensaje toma un largo periodo de tiempo.

3. Conectividad de la red. Determinando en que forma los procesos pueden comunicarse con todos los demás procesos en el sistema.

4. Identificación del emisor. Determina la información con la que cuentan los procesos sobre

el emisor de cierto mensaje recibido. El conocimiento y definición de esta característica es importante al existir fallas con un comportamiento Bizantino.

5. Confiabilidad del canal de comunicación. Como el nombre lo indica, define si el canal puede propiciar que los mensajes enviados entre procesos pueden ser alterados o no entregados. La suposición que simplifica la resolución del problema del consenso es evidentemente contar con un canal de comunicación entre los procesos que es confiable y las fallas únicamente se genera en los procesos mismos.

6. Mensajes autenticados o mensajes no autenticados. Para esta característica, contar con un mensaje autenticado simplifica la resolución del problema del consenso. Empleando por ejemplo firmas digitales, si un proceso decide falsificar un mensaje o alterar la información que se envía el destinatario puede darse cuenta de estas alteraciones.

2.3.1. El Problema de los Generales Bizantinos

Para entender la complejidad de resolver el problema del consenso comúnmente se emplea el problema de los generales bizantinos planteado originalmente por Lamport, Shostak y Peace en 1982. Este problema está inspirado en las guerras que el Imperio Bizantino enfrentó en la Edad Media, contempla esencialmente lo siguiente:

Cuatro campamentos del ejército atacante, cada uno dirigido por un general, se encuentran acampando alrededor del fuerte de Bizancio. Su ataque es únicamente efectivo si atacan simultáneamente. Por lo tanto, deben llegar a un acuerdo sobre el momento del ataque. La única forma en que pueden comunicarse es enviando mensajeros entre ellos. Los mensajeros modelan los mensajes. El sistema asíncrono es modelado por estos mensajeros que pueden tomar un tiempo ilimitado para viajar entre dos campamentos. Un mensaje perdido es modelado por un mensajero que ha sido capturado por el enemigo. Un proceso bizantino es modelado por un general que es un traidor. Este general traidor intentará corromper el mecanismo que se emplea para alcanzar el acuerdo, proporcionando información falsa a los otros generales. Por ejemplo, un traidor puede informar a un general que ataque tendrá lugar a las 10:00, e informar a los

otros generales que ataquen al mediodía. Incluso podría no enviar mensaje alguno a ningún general. Del mismo modo, puede alterar los mensajes que recibe de otros generales, antes de transmitirlos.

El gráfico que se muestra a continuación describe una posible comunicación confusa entre los generales bizantinos, donde un valor booleano de 1 corresponde a atacar y un 0 a aguardar. Ante semejante comportamiento el reto es determinar si es posible llegar al consenso y de ser posible bajo qué condiciones se llega a un acuerdo.

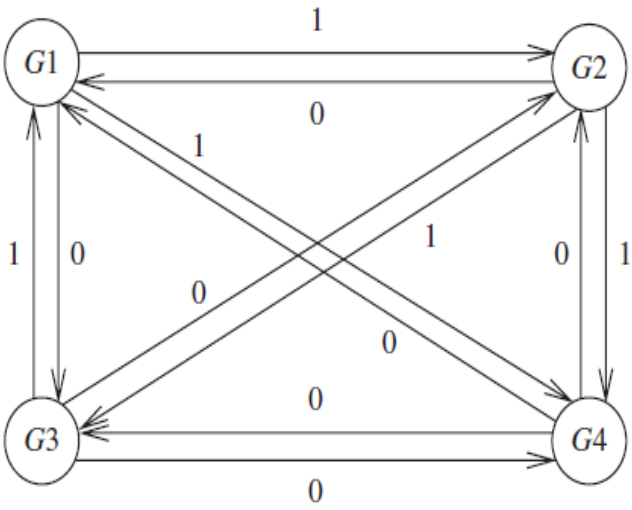


Figura 2-2: Comunicación confusa entre Generales Bizantinos [9]

Formalmente el Problema del Consenso y el Problema de los Generales Bizantinos, aunque similares, difieren en el estado inicial de los procesos y en que para el problema del consenso todos los procesos deben estar de acuerdo con un único valor final. Esto es:

1. **Finalización** Cada proceso correcto debe acabar asignando un valor a su variable de decisión.

2. **Acuerdo**. El valor finalmente decidido por todos los procesos correctos es el mismo.

3. **Integridad** Si todos los procesos correctos han propuesto el mismo valor, entonces cualquier proceso correcto en el estado de decisión ha elegido este valor.

2.4. El detector de fallas Omega

Ahora que una clasificación inicial de detectores de fallas ha sido introducida y que se ha descrito brevemente el problema del consenso y la importancia que tiene dentro del área del cómputo distribuido, se describe el detector de fallas más débil que basta para resolver este problema. Chandra y Toueg en [10] demostraron que el detector de fallas eventualmente débil ($\diamond W$) es la clase de detectores de fallas mínima necesaria para dar solución al problema del consenso. Para demostrar este resultado Chandra *et al.* primero demostraron que Ω es cuando menos igual de fuerte que $\diamond W$, y por lo tanto cualquier detector de fallas D que pueda ser empleado para resolver el consenso es cuando menos tan fuerte como Ω entonces al menos igual de fuerte que $\diamond W$. [11]

La salida del módulo detector de fallas de Ω en el proceso p es un único proceso q, este proceso es considerado en ese instante como correcto: $p\,conf\acute{\imath}a\,en\,q$. Un detector de fallas en Ω satisface la siguiente propiedad:

- Existe un instante en el tiempo después del cual todos los procesos correctos confían siempre en el mismo proceso correcto [12].

Se dice entonces que Ω provee funcionalidad de Elección Futura de Líder (*Eventual Leader Election*).

Al igual que con ($\diamond W$), la salida del módulo de detección de fallas de un detector en Ω en un proceso p puede cambiar con el tiempo, es decir, puede confiar en diferentes procesos en diferentes instantes. Además, en cualquier momento dado, dos procesos p y q pueden confiar en procesos diferentes. Sin embargo, el período durante el cual la salida de Ω es arbitraria es finita.

Con el Detector de Fallas Omega concluye la revisión en este trabajo sobre Detectores de Fallas, sin embargo, el lector debe saber que las clasificaciones de Detectores de Fallas aquí presentada no representa la totalidad de clasificaciones existentes y solo pretende establecer un punto de partida sólido en el estudio de los mismos.

Capítulo 3

Algoritmos Distribuidos

Al hablar de sistemas distribuidos no solo hablamos de la infraestructura que soporta este tipo sistemas, hablamos también de los algoritmos que lidian con problemas que la computación tradicional, en ocasiones secuencial, puede ignorar en muchas situaciones. Hasta este momento se ha presentado un único ejemplo de los retos que el cómputo distribuido enfrenta, concretamente en el capitulo 2. Discutimos el Problema del Consenso ilustrado con el Problema de los Generales Bizantinos.

Resulta evidente que el mundo del cómputo ya no es más un mundo de computadoras y sistemas trabajando individualmente. El mundo en el que vivimos está distribuido, es solo natural observar que los sistemas y el cómputo sea distribuido también.

Desde el trabajo de investigación de Lamport titulado *"Time, clocks, and the ordering of events in a distributed system"* [13] en 1978, el cómputo distribuido pasó de ser un compendio de recetas y trucos a un dominio de la ciencia de la computación, con sus propios conceptos, métodos y aplicaciones. [14]

Por ello la importancia de revisar los conceptos fundamentales que se emplean cuando se habla de algoritmos en el cómputo distribuido, contrastando con el cómputo paralelo o secuencial. Para empezar a discutir sobre este tipo de algoritmos, introducimos algunas definiciones y suposiciones fundamentales que se deben tener presentes cuando estudiamos algoritmos en un sistema distribuido.

3.1. Conceptos Fundamentales

La primera consideración importante es que un sistema distribuido puede ser visto como un grafo, donde los vértices del grafo representan los procesos y las aristas del grafo son la representación de los canales de comunicación. A continuación, se presentan algunas ideas adicionales, así como conceptos que son importantes cuando trabajamos con algoritmos distribuidos como el que abordamos en este trabajo.

Procesos. Un sistema distribuido está compuesto de una colección de unidades de cómputo. Cada una de estas unidades de cómputo se abstrae en la noción de *proceso*. Los procesos cooperan con un objetivo en común, esto implica que intercambian información de alguna manera.

El conjunto de procesos es estático. Se compone de n procesos denotados $\Pi = [p_1, p_2, ..., p_n]$ donde para cada p_i, $1 \leq i \leq n$, representa un proceso diferente. Cada proceso p_i es secuencial, en el sentido de que ejecuta un paso a la vez.

El entero i denota el índice del proceso p_i, que no es más que una forma en que un observador externo puede distinguir los procesos. Casi siempre se asume que cada proceso p_i cuenta con su propia identidad, denotada id_i; entonces cada p_i conoce id_i, siendo usual que $id_i = i$, aunque esto último no siempre es así.

Medio de Comunicación. Los procesos se comunican enviando y recibiendo mensajes a través de canales. Cada canal se asume como confiable, es decir, que no crea, modifica o duplica mensajes. En algunos casos se asume que los canales son "primero que entra, primero que sale"*(FIFO)* lo que significa que los mensajes son recibidos en el orden que fueron enviados. También se asume que los canales son bidireccionales y que tienen una capacidad infinita, en el sentido que pueden contener cualquier número de mensajes de cualquier tamaño. En algunos casos particulares esta suposición de canales bidireccionales puede ser cambiada para asumir que los canales son unidireccionales. Cada proceso p_i tiene un conjunto de vecinos, denotado $neighbors_i$. En este contexto, este conjunto contiene las identidades locales de los canales conectados de p_i a sus procesos vecinos o bien la identidad de estos procesos.

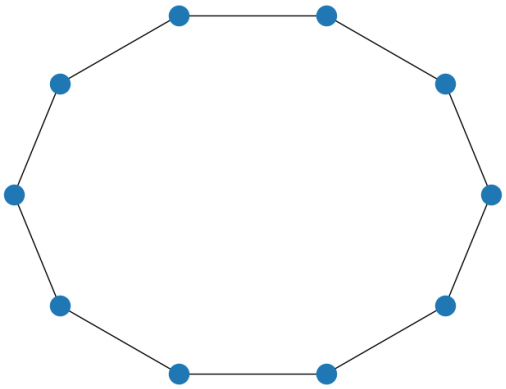
Vista Estructural. Como ya se comentaba un sistema distribuido puede ser visto, estructuralmente hablando, como un grafo no dirigido $G = (\Pi, C)$, donde C denota los canales. De acuerdo con Michel Raynal, existen tres grafos de particular interés:

- *Anillo (Ring)*. Un grafo donde cada proceso tiene exactamente dos vecinos con quienes se puede comunicar directamente, un vecino a la izquierda y un vecino a la derecha.
- *Árbol (Tree)*. Un árbol es un grafo donde se cumplen dos propiedades. Se dice que es acíclico y conectado. La primera propiedad se refiere a que añadir un nuevo canal crearía un ciclo y eliminar un canal lo desconectaría.
- *Completamente Conectado (Fully Connected)*. Un grafo completamente conectado es aquel en el que cada proceso se encuentra directamente conectado a todos los demás procesos. Empleando un término de grafos, se le llama clan o una clique pronunciado /klik/.

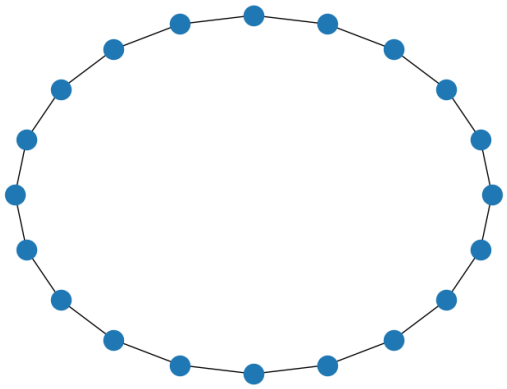
En la Figura 3-1 se ilustran algunos ejemplos de estos grafos.

Algoritmo Distribuido. Un algoritmo distribuido es una colección de n autómatas, uno para cada proceso. Un autómata describe la secuencia de pasos ejecutados por el proceso correspondiente. Adicionalmente a todas las capacidades de una Máquina de Turing, estos autómatas tienen capacidades de realizar operaciones de comunicación, las cuales les permiten enviar un mensaje a través de un canal y recibir un mensaje en cualquier canal. Estas operaciones son enviar() y recibir().

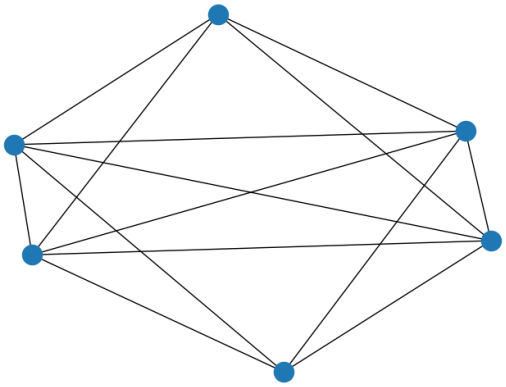
Algoritmo Síncrono Un algoritmo síncrono es un algoritmo diseñado para ser ejecutado en un sistema distribuido síncrono. El progreso de estos sistemas distribuidos está regido por un reloj externo global, los procesos ejecutan de forma colectiva una secuencia de rondas dependiendo del valor global del reloj. Durante una ronda, un proceso manda a lo más un mensaje a cada uno de sus vecinos. La propiedad fundamental de un sistema síncrono es que el mensaje enviado por un proceso durante una ronda r es recibido en el destino o por el destinatario en la misma ronda r. Entonces, cuando un proceso avanza a la ronda $r + 1$, ha recibido y procesado todos los mensajes que fueron enviados durante la ronda r y sabe que lo mismo es cierto para todos los demás procesos.



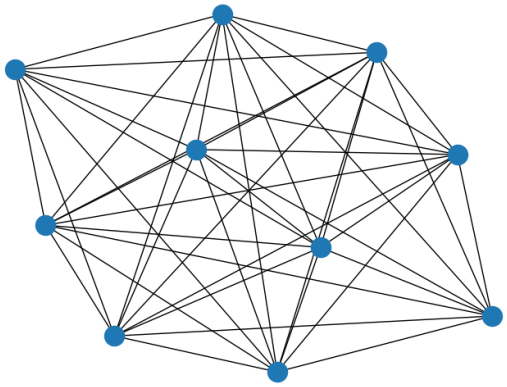
(a) Grafo de un anillo con 10 nodos



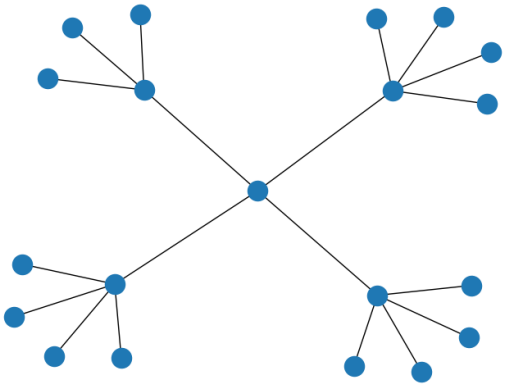
(b) Grafo de un anillo con 20 nodos



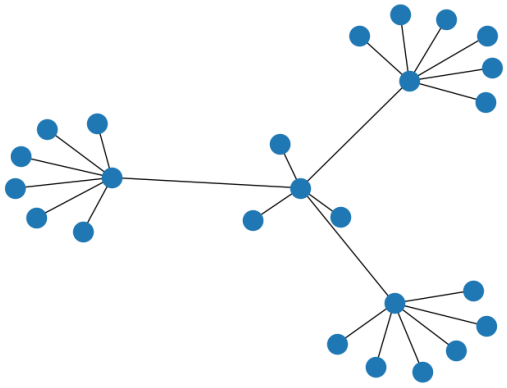
(c) Grafo de un Clique, también llamado clan, con 6 nodos



(d) Grafo de un Clique, también llamado clan, con 10 nodos



(e) Grafo de un árbol con 17 nodos



(f) Grafo de un árbol con 25 nodos

Figura 3-1: Representación de Algunos Grafos de Estudio Común

Diagrama Espacio/Tiempo. Una ejecución distribuida puede representarse gráficamente por lo que se conoce como un diagrama espacio/tiempo, introducidos por Leslie Lamport en [15]. Cada proceso secuencial es representado por una flecha de izquierda a derecha y un mensaje es representado por una flecha de un proceso emisor a un proceso receptor.

Algoritmo Asíncrono. Un algoritmo distribuido asíncrono, es un algoritmo diseñado para ser ejecutado en un sistema distribuido asíncrono. En dicho sistema, no existe la noción de un tiempo externo. Es por esta razón que en ocasiones los sistemas asíncronos son llamados también sistemas libres de tiempo (*time-free systems*). En un algoritmo asíncrono, el progreso de un proceso está asegurado por su propio procesamiento y el mensaje recibido. Cuando un proceso recibe un mensaje, procesa el mensaje y dependiendo del algoritmo local, posiblemente envíe mensajes a sus vecinos. Un proceso procesa un mensaje en cada instante del tiempo. Esto significa que el procesamiento de un mensaje no puede ser interrumpido por la llegada de otro mensaje. Cuando un mensaje llega, es agregado a un buffer de entrada en el proceso receptor. Será procesado después de que todos los mensajes que le preceden en el buffer hayan sido procesados.

Conocimiento inicial de un Proceso. Otra consideración importante cuando se intenta resolver un problema en un sistema sea síncrono o asíncrono, son los parámetros de entrada con los que cuenta un proceso. Estos parámetros de entrada caracterizan a cada uno de los procesos y definen el conocimiento inicial de su entorno.

3.2. Algoritmos de Elección de Líder en Sistemas Distribuidos

Una vez que hemos abordado los conceptos fundamentales y sus definiciones cuando hablamos de algoritmos distribuidos podemos revisar un problema recurrente en el mundo del cómputo distribuido: el problema de elección de un líder.

La elección de un líder consiste la designación que llevan a cabo los procesos de un sistema distribuido de uno de todos ellos. Usualmente, una vez que uno de ellos ha sido elegido líder, este proceso designado como líder es requerido para desempeñar un rol especial con fines de

coordinación o control [16].

Con esto dicho procedemos a definir formalmente el problema de elección de líder en un sistema distribuido.

3.2.1. Definición del Problema

Sea un proceso p_i que posee dos variables locales booleanas $elegido_i$ y $terminado_i$, ambas inicializadas en un valor $falso$. Las variables $elegido_i$ cumplen con la función de que eventualmente una y solo una se vuelve verdadera, mientras que cada una de las variables $terminado_i$ se vuelven verdaderas cuando el proceso p_i correspondiente se percata de que un proceso ha sido elegido.

Formalmente el problema de elección de líder esta definido por las propiedades de *liveness* y de *safety* que se muestran a continuación, donde var_i^τ denota el valor de la variable local var_i en el tiempo τ.

- Propiedad de *safety*

  - $(\forall i : \ elegido_i^\tau \Rightarrow (\forall \tau' \geq \tau : elegido_i^{\tau'})) \wedge (\forall i : terminado_i^\tau \Rightarrow (\forall \tau' \geq \tau : terminado_i^{\tau'}))$.

  Esta primera propiedad establece que las variables booleanas locales $elegido_i$ y $terminado_i$ son estables, es decir, una vez que se vuelven verdaderas, permanecen verdaderas para siempre.

  - $\forall i, j, \tau, \tau' : (i \neq j) \Rightarrow \neg(elegido_i^\tau \wedge elegido_i^{\tau'})$.

  Esta segunda propiedad establece que a lo más un proceso es elegido.

  - $\forall i : terminado_i^\tau \Rightarrow (\exists j, \tau' \leq \tau : elegido_j^{\tau'})$

  Por último, esta tercer propiedad establece que un proceso no puede percatarse que la elección ha terminado mientras no exista un proceso elegido.

- Propiedad de terminación

  - $\exists i, \tau : elegido_i^{\tau'}$

  - $\forall i : \exists \tau : terminado_i^\tau$

Esta propiedad de *liveness* establece que un proceso es eventualmente elegido y este hecho eventualmente es conocido por todos los procesos.

3.2.2. Imposibilidad de un Resultado en Sistemas Anónimos

Cuando hablamos de un sistema distribuido anónimo hablamos de un sistema distribuido en el que los procesos no tienen identidad o identificador. Esto implica que no existe forma alguna de diferenciar un proceso p_i de otro proceso p_j. Más aún, los procesos tienen el mismo número de vecinos, mismo código y estado inicial, de otra forma alguna de estas características habría sido candidata para ser empleada como identidad.

Cuando tenemos un sistema anónimo no existe solución al problema de elección de líder. Se presenta el siguiente teorema y su prueba para dar soporte al anterior enunciado, considerando una red de anillo por razones de simplicidad. [16]

Teorema 3.2.1 *No existe un algoritmo determinista de elección de líder en un anillo (bidireccional/unidireccional) de $n > 1$ procesos.*

Demostración. La demostración, por contradicción. Asumamos que existe un algoritmo distribuido determinista A que resuelve el problema de elección de líder en un anillo anónimo. A esta conformado de n algoritmos deterministas locales $A_1, A_2, ..., A_n$, donde A_i es el algoritmo local ejecutado por p_i. Incluso podemos decir que $A_1 = A_2 = ... = A_n$ por ser un sistema anónimo. Mostramos que A no puede satisfacer las propiedades de seguridad y de terminación que hemos presentado en la definición formal del problema de elección de líder.

Para cualquier i, sea σ_i^0 que denota el estado inicial de p_i, y sea $\Sigma^0 = (\sigma_1^0, ..., \sigma_n^0)$ el estado inicial global. Dado que todos los proceso ejecutan el mismo algoritmo determinista local A_i, por lo tanto existe una ejecución síncrona durante la cual todos los procesos ejecutan el mismo paso de su algoritmo determinista local A_i, y cada proceso p_i pasa del estado σ^0 al estado σ^1. Además dado que $\sigma_1^0, ..., \sigma_n^0$ y A_i es determinista e igual para todos los procesos, podemos decir en consecuencia que el conjunto de procesos pasan de $\Sigma^0 = (\sigma_1^0, ..., \sigma_n^0)$ a $\Sigma^1 = (\sigma_1^1, ..., \sigma_n^1)$ donde $\sigma_1^1 = ... = \sigma_n^1$. Lo importante a notar en este momento es que la ejecución progresa de un

estado simétrico global Σ^0 a otro estado simétrico global Σ^1, donde por simétrico entendemos que todos los procesos se encuentran en el mismo estado local.

Dado que todos los algoritmos A_i son idénticos y deterministas, la ejecución síncrona anterior puede ser continuada y el conjunto de procesos progresa de un estado simétrico global Σ^1 a un estado simétrico global Σ^2 y así sucesivamente. Siguiendo esta idea podemos ver que una ejecución simétrica puede ser extendida siempre de un estado simétrico global Σ a otro estado simétrico global Σ'.

En consecuencia, la ejecución síncrona presentada nunca termina. La razón es que para que la ejecución pueda terminar se requiere que entre en un estado asimétrico global, por la propia definición, dicho estado en el que un líder es elegido es asimétrico. Por lo tanto, el algoritmo A falla en satisfacer la propiedad de terminación para el problema de elección de líder, con lo que se concluye la prueba de este teorema. ■

3.2.3. Suposiciones y Principios Básicos en Algoritmos de Elección de Líder

Por lo presentado en la sección anterior, cuando hablamos de un algoritmo de elección de líder, como el que pretendemos simular, debemos realizar algunas consideraciones previas y seguir algunos principios básicos. Debemos asumir que cada uno de los procesos p_i involucrados en este tipo de algoritmos tiene una identidad id_i y que procesos distintos poseen una identidad diferente, esto es $i \neq j \Rightarrow id_i \neq id_j$. También debemos asumir que estas identidades pueden ser comparadas con ayuda de los operadores $<, =, >$.

Otro principio que hacemos al hablar de la elección de un líder, al igual que muchos otros autores en la literatura, es elegir un proceso como líder cuya identidad se encuentre en un extremo, es decir, aquel proceso que posea la mayor identidad o bien la menor entre todos los procesos del sistema distribuido. Dado que no existen dos procesos que compartan la misma identidad y que las identidades pueden ser comparadas, existe una identidad única en cada uno de los extremos.

3.3. Notas y Lecturas Complementarias Recomendadas

Si el lector desea conocer más sobre algoritmos de elección de líder, incluyendo un análisis detallado de diversos algoritmos, así como contar con un punto de partida y de profundización a los algoritmos distribuidos recomiendo la lectura del libro *Distributed Algorithms for Message-Passing Systems* de Michel Raynal, de donde se tomó una parte importante de este capítulo.

Capítulo 4

Sistemas de Eventos Discretos

En este momento tenemos una visión general de el tipo de problemas que se abordan en los sistemas distribuidos y una propuesta de resolución a algunos de ellos en los Detectores de Fallas, ahora bien, como se mencionó al comienzo de este trabajo nuestro objetivo es realizar una simulación de un algoritmo Detector de Fallas. Para ello nos apoyaremos del concepto de Sistemas de Eventos Discretos. Como primera aproximación a este concepto consideremos la clasificación de sistemas de la Figura 3-1.

Recordando la definición donde un sistema es un conjunto de componentes que interactúan y que se comportan de manera conjunta para realizar una función donde esta función no puede ser realizada por ninguna de las partes individuales que lo conforman. [17]

De acuerdo con la clasificación presentada en la Figura 3-1, un Sistema de Eventos Discretos se encuentra ubicado en el nivel de los Sistemas de Estado Discreto Basado en Eventos y Tiempo. Si un Sistema de Estado Discreto depende tanto del tiempo como de los eventos se dice que se tiene un Sistema Híbrido de Eventos Discretos.

En un momento específico del tiempo, el comportamiento de un sistema puede ser descrito en alguna forma que llamamos estado. El estado del sistema en un tiempo t_0 está definido como la salida $y(t)$ del sistema para toda $t \geq t_0$ y está determinada de forma única por el estado del sistema en t_0 y la entrada del sistema. Decimos entonces que el espacio de estados del sistema es el conjunto de todos los posibles valores que un estado puede tomar.

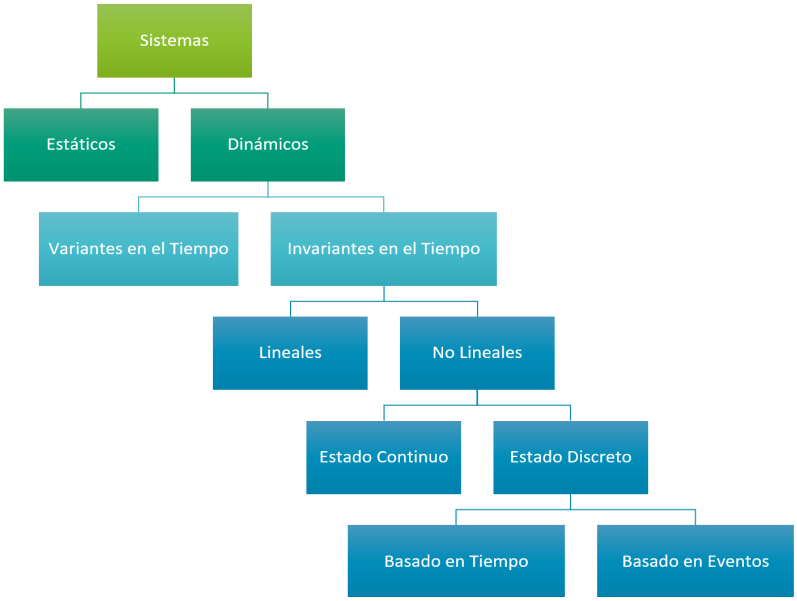


Figura 4-1: Ubicación de los Sistemas de Eventos Discretos dentro de la Clasificación General de Sistemas

De acuerdo con el tipo de estados en el modelo un sistema puede ser clasificado en sistemas de estado continuo o en un sistema de estado discreto. Para los sistemas de estado continuo, la variable de tiempo t permite que el sistema cambie de un estado a otro de forma continua. Estos sistemas reciben el nombre de basados en tiempo (*Time-Driven Systems*).

Por otro lado, en un sistema de estado discreto los estados solo pueden cambiar de un estado discreto a otro. Estas transiciones de estado están sincronizadas por un reloj o pueden ocurrir de forma asíncrona en un momento particular del tiempo a causa de los eventos del sistema. Un evento puede ocurrir instantáneamente y causar una transición de estado. Si las transiciones de estado en un sistema de estado discreto están sincronizadas por un reloj entonces es un Sistema de Estado Discreto Basado en Tiempo (*Discrete-State Time-Driven System*). De lo contrario, si las transiciones de estado ocurren de forma asíncrona en momentos aleatorios de tiempo y son ocasionados por los eventos propios del sistema, nos referimos a un Sistema de Estado Discreto Basado en Eventos (*Discrete-State Event-Driven System*) o bien de forma abreviada un Sistema de Eventos Discretos (SED) llamado en inglés *Discrete Event System (DES)*. [18]

Esta última clasificación de sistemas, los Sistemas de Estado Discreto Basado en Eventos, son

de particular interés para este trabajo pues el algoritmo de Detección de Fallas que pretendemos implementar se ejecuta en una red distribuida que puede ser modelada como un sistema de este tipo. Por ello profundizaremos un poco más en ellos y revisaremos brevemente los conceptos mas relevantes en este tipo de sistemas.

4.1. Eventos

El concepto de lo que es un evento tiene una base intuitiva adecuada y se debe entender que sucede de forma instantánea y como causante de una transición de un estado a otro. Un evento puede ser identificado con una acción especifica, alguien pulsando un botón, por ejemplo. Puede ser visto como una ocurrencia espontánea dictada por la naturaleza, por ejemplo un sistema computacional que falla por alguna razón para la que no tenemos recursos suficientes para diagnosticar. O bien un evento puede ser el resultado de varias condiciones que se cumplen. Usualmente un evento se denota e y en un sistema que puede ser afectado por diferentes tipos de eventos se suele denotar el conjunto de eventos con E donde todos sus elementos son eventos y E es un conjunto discreto.

4.2. Sistemas Basados en Tiempo y Sistemas Basados en Eventos

Cuando hablamos de sistemas de estado continuo el estado generalmente cambia conforme el tiempo cambia. En los modelos de tiempo discreto el reloj es quien regula estos cambios. Con cada pulso del reloj se espera que el estado cambie. Cuando esto se cumple entonces estamos hablando de un Sistema Basado en Tiempo, independientemente de si tratamos con la variable del tiempo t, en el caso continuo o k para el caso discreto, estamos hablando de una variable independiente que aparece como el argumento de todas las funciones de entrada, estado y salida.

En los Sistemas Discretos observamos que el estado cambia únicamente en ciertos momentos del tiempo a través de transiciones instantáneas. Con cada una de esas transiciones podemos asociar un evento. Como mecanismo que regula los eventos que ocurren se consideran dos

posibilidades:

1. En cada pulso del reloj un evento e es seleccionado del conjunto de eventos E. Si ningún evento debe ocurrir, podemos pensar en un evento nulo que pertenece a E, cuya propiedad es que no altera el estado.

2. En varios instantes de tiempo, que no necesariamente son conocidos con anticipación y que no necesariamente coinciden con un pulso del reloj, algún evento e anuncia que esta ocurriendo.

Las diferencias entre estas dos posibilidades son las siguientes:

- En la primera posibilidad, las transiciones de estado están sincronizadas por el reloj. Esto es, ocurre un pulso del reloj y un evento o el evento nulo es seleccionado. el estado cambia, y el proceso se repite. El reloj es el único responsable de cualquier posible transición de estado.

- En la segunda posibilidad cualquier evento $e \in E$ define un proceso diferente a través del cual el instante de tiempo en el que e ocurre es determinado. Las transiciones de estado son el resultado de combinar estos procesos asíncronos y concurrentes. e

Son estas diferencias entre las dos posibilidades las que dan origen a los términos de Sistemas Basados en Tiempo y Sistemas Basados en Eventos. Resulta evidente que el análisis y modelado de los Sistemas Basados en Eventos es generalmente más complicado, dado que existen distintos mecanismos asíncronos que se deben especificar como parte del entendimiento del sistema.

Para entender aun mejor este tipo de sistemas se puede pensar a la noción de una *interrupción* en una computadora. Muchas funciones dentro de una computadora se encuentran sincronizadas por un reloj interno, por lo tanto, son Basadas en Tiempo, adicionalmente los Sistemas Operativos están diseñados para responder a llamadas asíncronas que pueden ocurrir en cualquier momento [19].

4.3. Simulación de Eventos Discretos

Nuestra finalidad es entender mejor un algoritmo de Elección de Líder que retoma la idea de Detección de Fallas en un Sistema Distribuido y conocer su comportamiento. Si bien es posible obtener resultados analíticos para Sistemas de Eventos Discretos [19] estos resultados son extremadamente difíciles de obtener, lo que convierte una simulación en una herramienta muy atractiva para su estudio.

Pensando en una simulación por computadora como el equivalente electrónico de un experimento en un laboratorio tradicional. El único *hardware* involucrado es una computadora, y en lugar de dispositivos físicos conectados entre ellos tenemos un *software* recreando sus interacciones. El factor aleatorio, ruido, presente en un sistema real puede ser reemplazado de la misma forma con *software*, empleando un generador de números aleatorios, por ejemplo. Definitivamente la simulación no representa fielmente la realidad, pero es una excelente opción antes de tener que construir algún sistema caro y complicado.

La simulación de SED es empleada en la manufactura de sistemas de comunicación para medir su desempeño, se emplea en la prueba de protocolos para la competencia de recursos de red, incluso existen simulaciones de tráfico en caminos y carreteras empleando estas técnicas [19]. Todos estos son ejemplos de sistemas complejos altamente estocásticos.

Con la finalidad de realizar la simulación de un SED existen algunas consideraciones previas. La secuencia de eventos debe estar asociada a un reloj. Suponiendo que tenemos un SED con un solo evento $E = \{\alpha\}$. Entonces el conjunto de eventos posibles $\Gamma(X) = \{\alpha\}$ para toda $x \in X$. La secuencia de eventos en este SED es $\vec{e} = \{e_1, e_2, ..., , e_k\}$ donde $e_1 = e_2 = ... = e_k = \alpha$. El Tiempo de vida de un Evento, denotado por v_k, está definido por la longitud del intervalo de tiempo de dos eventos sucesivos. Retomando nuestro SED de un único evento, el tiempo de vida del k_i evento estaría definido por:

$$v_k = t_k - t_{k-1}, k = 1, 2, ..., k, v_k \in \mathbb{R}$$

En el tiempo t_{k-1}, el k-ésimo evento, e_k esta habilitado un tiempo de vida v_k. Es entonces

cuando un temporizador asociado al al evento e_k comienza una cuenta descendente desde v_k. En el tiempo $t_k = t_{k-1}+v_k$, el temporizador llega a 0, e_k debe ocurrir y una función de transición es causada de x_{k-1} hacia x_k. Entonces el mismo proceso debe repetirse para el evento e_{k+1}. Por lo tanto podemos decir que un SED puede ser definido por la secuencia de reloj de los eventos, esto es, $\vec{v} = \{v_1, v_2, ..., v_k\}$. [18]

Desde la perspectiva de implementación en una computadora, una simulación de un SED debe considerar un esquema de calendarización de los eventos (*Event Scheduling Scheme*), esto puede ser un autómata basado en tiempo, en donde se cumple que para cada evento i es alcanzado en un tiempo t_n, su siguiente ejecución esta calendarizada para un tiempo posterior $t_n + v_i$ donde v_i es un tiempo de vida provisto por el SED. Por lo que se puede tener una lista calendarizada de eventos. Este es solamente un ejemplo que muestra como se podría realizar una simulación en una computadora. Un segundo esquema que permite la simulación de SED es el esquema orientado a los procesos (*Process-Oriented Scheme*).

4.3.1. Esquema de Calendarización de Eventos(*Event Scheduling Scheme*)

Entender los dos principales esquemas existentes en la simulación de Sistemas de Eventos Discretos es crucial para lograr el objetivo que nos hemos propuesto. Empezamos a revisar el esquema de calendarización de eventos en primer lugar.

En este esquema existe un espacio contable de eventos denotado ϵ y un espacio contable de estados χ. El tiempo se inicializa en $t = 0$ y se asume que el estado inicial $x_0 \in \epsilon$ es dado. Para cada estado x existe un conjunto de eventos posibles $\Gamma(x) \subset \epsilon$. Esto significa que ese subconjunto de eventos son los únicos que pueden ocurrir partiendo de ese estado. Dado x_0 se tiene $\Gamma(x_0)$. Si $\Gamma(x_0)$ es un evento posible, se asocia con un valor de reloj, este valor representa el tiempo necesario hasta que el evento i ocurra. Inicialmente, el valor de reloj de todos los posibles eventos esta establecido en un valor de vida del evento (*event lifetime*). En una simulación computacional es común que este valor este dado por un generador de números aleatorios (RNG). Los valores de reloj del evento i es denotado por y_i y su tiempo de vida v_i. Entonces, en $t = 0$ el generador de números aleatorios provee tiempos de vida y se establece

$y_i = v_i$ para toda $i \in \Gamma(x_0)$

Dado que el estado actual esta denotado por x, que incluye x_0 podemos observar todos los valores y_i donde $i \in \Gamma(x)$. El evento activado e' es el evento que ocurren enseguida de ese estado, es decir, el evento que tiene el valor de reloj menor. Formalmente se denota en la ecuación 4-1.

$$e' = min_{i \in \Gamma(x)}(y_i) \tag{4-1}$$

En ese momento se puede actualizar el estado basado en el mecanismo de transición del estado. El nuevo estado es x' con la probabilidad $p(x'; x, e')$. El valor de x' es generado por el RNG (*Random Number Generator*) . Si existe un único posible nuevo estado cuando el evento e' ocurre en el estado x, se dice que se tiene una transición de estado determinista, representada por la función de transición de estado $f(x, e')$.

El tiempo que se emplea en el estado x recibe el nombre de tiempo de intervención (*Intervent Time*).

Empleando estos temporizadores, funciones de transición entre posibles estados y los eventos generados se puede mantener una Lista de Eventos Calendarizados (*Scheduled Event List*).

$$L = \{(e_k, t_k)\}, k = 1, 2, ..., m_L \tag{4-2}$$

Donde $m_L \leq m$ es el número de posibles eventos en el estado actual y m es el número de eventos en el conjunto ϵ. Adicionalmente la lista de eventos está siempre ordenada siguiendo el valor de tiempo de calendarización más pequeño primero. Esto implica el primer evento en la lista, e_1, siempre será el evento que se active. [19]

El procedimiento encargado de la simulación es responsable de la repetición continua de los siguientes seis pasos:

Paso 1. Borrar la primer entrada (e_1, t_1) de la Lista de Eventos Calendarizados.

Paso 2. Actualizar el tiempo de la simulación avanzando hasta el tiempo del nuevo evento t_1.

Paso 3. Actualizar el estado de acuerdo a la función de transición de estado, $x' = f(x_1, e_1)$

Paso 4. Borrar de la Lista de Eventos Calendarizados cualquier entrada correspondiente a eventos inviables en el nuevo estado, es decir borrar todos $(e_k, t_k) \in L$ tales que $e_k \notin \Gamma(x')$.

Paso 5. Agregar a la Lista de Eventos Calendarizados cualquier evento viable que no se encuentre previamente calendarizado, incluyendo posiblemente el evento activador que se borró en el paso 1. El tiempo del evento calendarizado para una i está dado por $(tiempo + v_i)$, donde *tiempo* fue establecido en el paso 2 y v_i es el tiempo de vida obtenido del RNG.

Paso 6. Reordenar la Lista de Eventos Calendarizados basado en el valor de tiempo de calendarización más pequeño primero.

Este procedimiento se repite, iniciando en el Paso 1, con la nueva lista ordenada. la Figura 4-2 ilustra este esquema de calendarización de eventos.

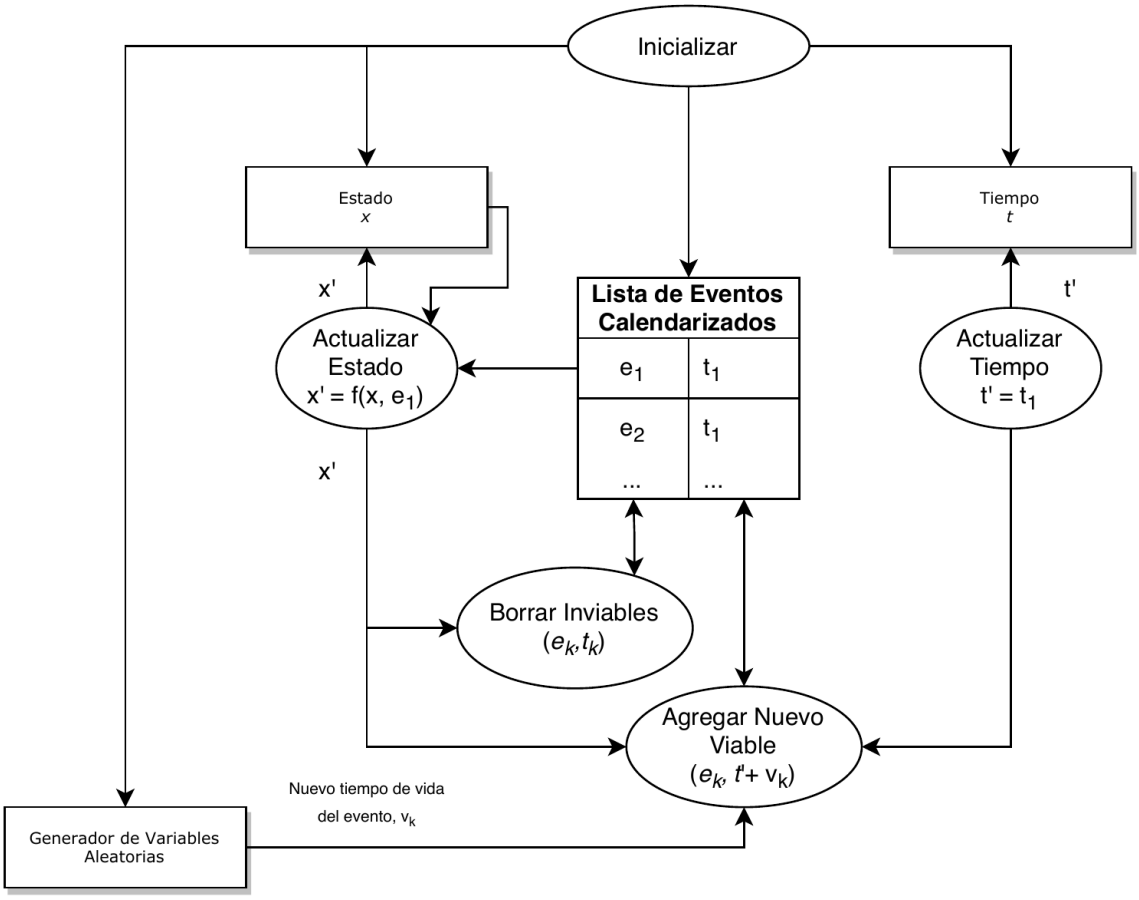


Figura 4-2: Esquema de Calendarización de Eventos en una Simulación por Computadora

En resumen, este esquema de simulación basado en la calendarización de eventos emplea los siguientes componentes al ser implementado en una computadora: [19]

1. Estado: una lista donde todas las variables de estado son almacenadas.
2. Tiempo: una variable donde el tiempo de la simulación es guardado.
3. Lista de Eventos Calendarizados: una lista donde todos los eventos calendarizados son almacenados, incluyendo el tiempo en el que ocurrirán.
4. Registros de Datos: variables o listas donde los datos empleados en las estimaciones son almacenados.
5. Rutina de Inicialización: una rutina encargada de la inicialización de las estructuras de datos (1 - 4) cuando se inicia la simulación.
6. Rutina de Actualización de Tiempo: una rutina que identifica el siguiente evento por ocurrir y avanza el tiempo de la simulación al momento de ocurrencia de ese evento.
7. Rutina de Actualización de Estado: una rutina que actualiza el estado con base en el siguiente evento que está por ocurrir.
8. Rutina Generadora de Variables Aleatorias: una colección de rutinas encargadas de transformar valores generados por la computadora en variables aleatorias de acuerdo a alguna distribución de tiempo de vida.

El simulador que pretendemos simular, si bien puede ser implementado utilizando las ideas de un esquema de calendarización de eventos, resulta conveniente considerar el esquema que aún no hemos revisado. Este esquema como ya se había mencionado es el denominado Orientado a Procesos. El siguiente capitulo presenta los conceptos del Esquema de Simulación Orientado a Procesos y un *framework* de Simulación que implementa estas ideas.

Capítulo 5

Desarrollo

Ahora tenemos un panorama general de los Sistemas de Eventos Discretos y la importancia de la simulación computacional para entender su comportamiento y para su estudio. Surgen entonces dos posibles caminos para realizar una simulación del sistema de interés: construir un simulador de SED desde cero o emplear alguno existente. Cualquiera que sea el esquema de simulación empleado, un esquema de calendarización o un esquema orientado a procesos, existen implementaciones en diversos lenguajes de programación populares (C, C++, Java, Python, etc.) listos para ser adecuados en función de nuestras necesidades. Dado que nuestro interés particular es entender el comportamiento y desempeño de un algoritmo de Detección de Fallas, la implementación de un simulador desde cero escapa el alcance de este trabajo, por ello emplearemos un *framework* de simulación de eventos discretos existente. En este trabajo ese *framework* es SimPy.

Figura 5-1: Logo del *framework* SimPy para Simulación de Sistemas de Eventos Discretos

SimPy es un *framework* de Simulación de Eventos Discretos orientado a procesos basado en el lenguaje de programación Python estándar. El comportamiento de los componentes activos se modela con procesos. [20]

Estos procesos pueden ser vehículos, clientes o en nuestro caso nodos de una red que se comunican entre ellos. El algoritmo presentado en este trabajo y en general al hablar de algoritmos en sistemas distribuidos se emplea el termino proceso para denominar las unidades de cómputo involucradas en el sistema, esta idea no debe confundirse con la noción de un proceso de SimPy. En SimPy todos los procesos viven en un entorno (*environment*) e interactúan con él y con otros procesos a través de eventos.

Estos procesos son descritos por un generador de Python y pueden ser llamados por una función o un método dependiendo de si pertenecen a una clase o no. Durante el tiempo de vida de estos procesos, crean eventos y los ceden para esperar a que sean disparados.

Cuando un proceso cede su ejecución por la ocurrencia de un evento, el proceso queda suspendido. SimPy continua la ejecución de ese proceso cuando un evento que ocurre lo dispara nuevamente. Múltiples procesos pueden esperar el mismo evento, entonces SimPy continua su ejecución en el mismo orden en el que comenzaron la espera de ese evento.

Antes de continuar con la revisión de procesos específicamente para la biblioteca de SimPy se debe entender la abstracción que se pretende obtener al emplearlos. En general un proceso es una secuencia de funciones, en donde estas funciones pueden ser de dos tipos:

1. **Funciones Lógicas** Son acciones instantáneas tomadas por la entidad que dispara la función dentro de su proceso. Ejemplos de estas funciones son verificar una condición, como si un servidor o un canal esta disponible, actualizar una estructura de datos también es un ejemplo de estas funciones.

2. **Funciones de Retraso (*Delay*)** Estas funciones retienen entidades por un periodo de tiempo que puede estar determinado de dos formas:

    a) **Tiempo Especifico** El tiempo del retraso esta fijo, usualmente por un valor obtenido mediante un Generador Aleatorio empleado en la simulación. Un ejemplo

de esto es un servicio que debe completarse en una ventana de tiempo especificada únicamente por un valor superior.

b) **Tiempo No Especifico** El tiempo de retraso depende del estado del sistema. Por ejemplo, el tiempo que un proceso debe esperar en una cola antes de que la entidad pueda emplear el procesamiento de un servidor o un emisor pueda ocupar el canal de comunicación.

Ahora se puede observar que este tipo de simulación orientado a procesos esta particularmente pensada para sistemas de colas, donde los clientes son entidades transitando una red interconectada de servidores y otras colas. Por último antes de volver a nuestro *framework* particular de simulación, revisaremos los componentes de una simulación orientada a procesos.

1. **Entidades**. Son objetos que solicitan un servicio (*componentes* en un sistema de manufactura, *tareas* en una computadora, *paquetes/mensajes* en una red, *vehículos* en una autopista, etc.). Cada una de estas entidades se caracteriza por un proceso que tiene que realizar en el SED al que pertenece.

2. **Atributos**. Es la información que caracteriza una entidad de forma individual. En este aspecto recuerda a los atributos de la instanciación de un objeto en el paradigma de la Programación Orientada a Objetos.

3. **Funciones del Proceso**. Las acciones instantáneas o los retrasos que las entidades experimentan.

4. **Recursos**. Son los objetos que proveen de servicios a las entidades (*un procesador* en una computadora, *el canal de comunicación* en una red, etc.). Retrasos de tiempo experimentados en una entidad pueden ocasionarse por esperar que un recurso particular se encuentre disponible.

5. **Colas**. Son conjuntos de entidades que comparten características en común, usualmente esa característica es que se encuentran esperando el uso de un recurso particular. Una

entidad dentro del sistema siempre se encuentra en una cola a menos que este recibiendo servicio de algún recurso.

Estos componentes son los que describen una simulación orientada a procesos. Regresando a la visión de SimPy de procesos, tenemos algunas consideraciones adicionales. Una de ellas ya se ha mencionado, la existencia de un entorno (*environment*) donde todos los procesos existen, este entorno es requerido para que un proceso pueda crear nuevos eventos.

Entendiendo estos conceptos, recordamos que la intención de ocupar SimPy en nuestra simulación es implementar el modelo de comunicación que el algoritmo de elección de líder que pretendemos simular emplea. Antes de presentar formalmente el algoritmo, debemos conocer el modelo de comunicación que emplea canales ADD.

5.1. Modelo de Comunicación ADD

El modelo de comunicación emplea canales ADD (*Average Delayed/Droped*), este modelo fue descrito por Srikanth Sastry y Scott M. Pike en [5] donde se comparan los modelos clásicos de sincronización parcial que se emplean en las implementaciones de $\diamond P$ por ejemplo en [7, 21, 22, 23, 24, 25, 26] se hacen suposiciones sobre la confiabilidad y características de entrega en tiempo de los canales que soportan la comunicación. Estos modelos de comunicación entre los procesos hacen suposiciones sobre los canales, estas suposiciones pueden ser que los canales siempre se comportan de forma confiable o que eventualmente se comportan de esa forma. En otras palabras, los modelos de comunicación suponen que los canales pierden a lo más una cantidad finita de mensajes hasta cierto punto en el tiempo a partir del cual los canales que conectan los procesos se comportarán siempre de forma confiable. Otras suposiciones en los modelos citados es la existencia de una cota superior, conocida o no, para el retraso que un mensaje puede tener de un punto a otro o la existencia de un retraso promedio.

Estas suposiciones evidentemente no reflejan lo que ocurre en muchos sistemas, basta con saber que los retrasos o pérdidas de los mensajes pueden ocurrir de forma intermitente y es-

pontánea en la duración de computación. Es bajo esta realidad que se describe el modelo de sincronía parcial basado en enlaces no confiables llamados Canales ADD. En [5] se parte de la idea que este modelo de comunicación presenta un modelo más débil de sincronía a los previamente ocupados en sistemas detectores de fallas y tolerantes a fallas.

5.1.1. Características del Modelo ADD Original

Cada canal ADD es un enlace de comunicación unidireccional que conecta dos procesos. Todos los mensajes enviados en un canal ADD pueden ser particionados en dos conjuntos lógicos disjuntos: *privilegiados* y *no-privilegiados*. Esta distinción es meramente un artefacto de modelado que no es conocido por el canal ni por los procesos que están empleando dicho canal. Los mensajes no privilegiados carecen de garantía alguna de confiabilidad y de garantía de entrega en tiempo. En comparación, los canales ADD proveen las siguientes garantías para los mensajes privilegiados:

1. Si un proceso p manda un número infinito de mensajes a un proceso correcto q a través de un canal ADD, entonces un subconjunto infinito de esos mensajes será privilegiado. Todos esos mensajes privilegiados serán entregados con toda certeza a q.

2. Para cada ejecución de un canal ADD, existe una ventana desconocida de tamaño $w \in \mathbb{N}^+$, un retraso desconocido $d \in \mathbb{N}^+$ y una proporción de mensajes $r \in \mathbb{N}^+$, tal que para cada intervalo de envío que contiene la secuencia S de exactamente w mensajes privilegiados:

   a) El retraso promedio de todos los mensajes *privilegiados* en S es a lo más d.

   b) El número promedio de mensajes *no-privilegiados* enviados entre cualquier par de mensajes *privilegiados* en S es a lo más r.

De forma intuitiva lo anterior nos dice que los mensajes privilegiados son entregados de forma confiable y que en promedio, no se entregan demasiado tarde ni demasiado separados uno del otro. Tanto los mensajes privilegiados como los no-privilegiados pueden estar intercalados, pero cualquier intervalo que contiene w mensajes privilegiados está sujeto a las cotas definidas por

d y r, estas cotas restringen el retraso promedio de los mensajes privilegiados y la relación de mensajes no-privilegiados respectivamente. Adicionalmente, a pesar de la existencia de las cotas w, d, y r para cada uno de los canales ADD, estas cotas son desconocidas y pueden variar en cada ejecución.

5.1.2. Características del Modelo ADD Simplificado

Estas propiedades se pueden simplificar preservando para dos procesos p y q lo siguiente:

1. El canal no crea ni duplica mensajes
2. Existe un entero K tal que para cada K mensajes consecutivos enviados de p a q, al menos uno de estos mensajes es entregado a q en tiempo D.

Los otros mensajes pueden perderse o experimentar retrasos arbitrarios. Las constantes K y D no son conocidas ni por p ni por q y cada canal ADD que conecta dos procesos puede tener su propio valor para K y D.

Estas son las propiedades que definen el modelo de comunicación parcialmente síncrono donde los canales pueden reordenar y perder mensajes en la implementación del simulador de este trabajo.

A pesar de que este modelo de comunicación es bastante débil es posible implementar $\diamond P$, un detector de fallas eventualmente perfecto, en una red completamente conectada que emplea canales ADD, donde procesos asíncronos son propensos a fallar permanentemente. [5] Más aún, se ha mostrado que es posible implementar $\diamond P$ en una red arbitraria con canales ADD. [27]. Como se discutió en el capítulo dedicado a los detectores de fallas, recordamos que $\diamond P$ es suficientemente poderoso para resolver el consenso y aún así se puede implementar en un sistema real.

La implementación presentada en [27] de este detector de fallas empleaba mensajes de tamaño acotado. A su vez, dicha implementación surgió como una mejora a la implementación presentada en [28] en la que el tamaño de los mensajes no se encontraba acotado. Sin embargo, el tamaño de los mensajes se encontraba acotado por un comportamiento exponencial en n, el número de los procesos. En [2] se presenta una mejora a estas implementaciones de $\diamond P$ con

mensajes de tamaño $O(nlog(n))$ en una red arbitrariamente conectada.

5.2. Modelo del Sistema a Simular

El algoritmo de elección de líder pasa de $\diamond P$ a Ω para disminuir aún más el tamaño de mensajes. El trabajo presentado en [29] demuestra que Ω es implementable en una red arbitrariamente conectada por canales que eventualmente satisfacen las propiedades ADD donde los procesos son susceptibles a fallar bajo un modelo en el que una vez que se presenta esta falla en los procesos nunca vuelven a ser correctos. El tamaño de los mensajes bajo esta implementación de Ω es de $O(log(n))$, donde los procesos requieren conocer n, el número de procesos. Este es el algoritmo que ejecutan los procesos de la simulación en este trabajo, nombrado ***Algoritmo de Elección Eventual de Líder en el modelo*** $\diamond ADD$ ***con Pertenencia Conocida*** (*Eventual Leader Election in the* $\diamond ADD$ *model with known membership*).

A pesar de que este es el algoritmo presentado en este trabajo, el lector debe saber que en [29] se presenta también la extensión de la idea para diseñar un algoritmo donde no se necesita conocer n, nombrado ***Algoritmo de Elección Eventual de Líder en el modelo*** $\diamond ADD$ ***con Pertenencia No Conocida*** (*Eventual Leader Election in the* $\diamond ADD$ *model with unknown membership*).

En este punto conocemos todo lo necesario para presentar formalmente el algoritmo y el sistema que se pretende simular. A continuación, se presenta lo relacionado al modelo de computación descrito en el simulador implementado.

5.2.1. Modelo y Comportamiento de los Procesos

El sistema consiste un número finito de procesos $\Pi = \{p_1, p_2, ..., p_n\}$. Cada proceso p_i tiene una identidad, sin pérdida de generalidad se considera que esta identidad es el índice i de cada uno de los procesos p_i. Por esta razón empleamos indiferentemente i y p_i para denotar el mismo proceso.

Cada proceso p_i tiene su propio reloj local de solo lectura $clock_i()$ que se asume genera

ticks a un ritmo constante. Estos relojes locales propios de cada proceso no necesitan estar sincronizados para exhibir el mismo instante de tiempo simultáneamente, estos relojes locales son empleados exclusivamente para el manejo de temporizadores. Por último, por simplicidad se asume que el tiempo de procesamiento es cero en cualquiera de los procesos del sistema.

Los procesos que ejecutan el algoritmo de elección de líder en el modelo de canales eventualmente ADD, al igual que en muchos otros algoritmos de elección de líder, eligen al proceso con el identificador más pequeño entre todos los procesos que se encuentran correctos, es decir aquellos procesos que no han fallado.

En el contexto de este trabajo la falla de un proceso se entiende como el cese de toda comunicación de un proceso hacia sus vecinos y cese de todo procesamiento de mensajes enviados de sus vecinos hacia este proceso que ha fallado. Esta falla es irreversible, es decir, una vez que un proceso ha fallado no vuelve a ser correcto durante la ejecución del algoritmo ni del simulador.

5.2.2. Modelo de la Red

La red esta representada por un grafo dirigido $G = (\Pi, E)$, donde una arista $(p_i, p_j) \in E$ significa que existe un canal unidireccional que permite al proceso p_i enviar mensajes a p_j. Esto permite que un canal bidireccional sea representado por dos canales unidireccionales, posiblemente con sus propias suposiciones y características temporales. En consecuencia, cada proceso tiene un conjunto de canales de entrada (*input*) y un conjunto de canales de salida (*output*).

La conectividad de este grafo debe cuando menos ser suficiente para definir un árbol generador (*Spanning Tree*), en otras palabras el grafo debe ser conectado.

5.2.3. Modelo y Comportamiento de los Canales

Como punto de partida se asume que ningún canal dirigido crea, corrompe o duplica mensajes. Los canales satisfacen la propiedad ADD. Esto es, existen dos constantes K y D, desconocidas por los procesos, tales que:

- Para cada K mensajes consecutivos enviados de p_i a p_j, al menos uno de ellos es entregado

a p_j en D unidades de tiempo desde que fue enviado. Los otros mensajes de p_i a p_j pueden ser perdidos o experimentar retrasos finitos arbitrarios.

Cada canal dirigido puede tener sus propios valores para las constantes K y D, sin perdida de generalidad establecemos que los valores de K y D son los mismos para todos los canales.

Por último, los canales no solo son canales ADD son canales $\diamond ADD$, el diamante al igual que en el caso de detectores de fallas, representa un comportamiento que eventualmente se satisface. Por lo tanto, podemos llamarlos canales eventualmente ADD. Lo que significa que existe un instante en el tiempo desconocida a partir del cual los canales satisfacen la propiedad ADD, antes de este instante de tiempo finito los canales se pueden comportar de forma arbitraria.

5.2.4. Principio General del Algoritmo

El algoritmo emplea un único tipo de mensaje denominado *alive*. Este mensaje transporta dos valores: un identificador de proceso y un entero $x \in 2, .., n-1$.

Un mensaje $alive(*, n-1)$ (donde entendemos que " $*$ ", representa el identificador de cualquier proceso) se le llama *mensaje alive de generación* mientras que un mensaje $alive(*, n-k)$, donde $1 < k < n-1$, se le llama *mensaje alive de retransmisión* y el valor $n-k$ es llamado valor de *hopbound*.

Cuando el proceso p_i inicia el algoritmo se propone a si mismo como líder. Envía un mensaje de generación $alive(i, n-1)$ a sus vecinos cada T unidades de tiempo.

Cuando un proceso p_i recibe un mensaje $alive(j, n-k)$ tal que $1 < k < n-1$, se entera que p_j es candidato a líder y que hay un camino con k saltos desde p_j a él. Si $j < i$, p_i toma a p_j como líder y retransmite el mensaje $alive(j, n-(k+1))$ a sus vecinos. En consecuencia un mensaje alive de generación puede generar un número finito de mensajes de retransmisión ($alive(j, n-2)$, $alive(j, n-3)$, $potencialmente\, hasta, alive(j, 2)$)).

Por último, dado que es posible que existan múltiples caminos de p_j hacia p_i, de diferentes longitudes, p_i puede recibir diferentes valores de hopbound de mensajes alive de retransmisión $alive(j, n-k)$ con un líder j. Por esta razón p_i maneja un temporizador para cada valor de $n-k$ de cada líder potencial.

5.2.5. Comportamiento Detallado de los Procesos

Para lograr el comportamiento descrito en la sección anterior los procesos deben contar con cierta información de la red y almacenar cierta información de esta. Cada proceso p_i mantiene las siguientes variables:

- $in\_neighbors_i$ un conjunto constante que contiene las identidades de los procesos tales que existe un canal de p_j a p_i.
- $out\_neighbors_i$ un conjunto constante que contiene las identidades de los procesos tales que existe un canal de p_i a p_j.
- $leader_i$ contiene la identidad del líder elegido.
- $timeout_i[1..n, 1..n]$ es una matriz de valores de *timeout* y $timer_i[1..n, 1..n]$ es una matriz de temporizadores tales que en conjunto, la pareja $(timer_i[j, n-k], timeout_i[j, n-k])$ es utilizada para monitorear las rutas desde p_j hacia p_i cuya distancia es k.
- $hopbound_i[1..n]$ es un arreglo de enteros no negativos donde $hopbound_i[i]$ se inicializa a n y las demás entradas $hopbound_i[j]$ se inicializan a 0. Cuando $j \neq i$, $hopbound_i[j] = n-k \neq 0$ significa que si p_j es actualmente considerado como líder por p_i, la información que recibió en el último mensaje $alive(j, n-1)$ enviado por p_j hacia sus $out\_neighbors$ por un camino de k procesos diferentes antes de ser recibido por p_i. Para mayor claridad agregar que el nombre *hopbound* viene de la idea de una cota máxima de saltos (*Upper bound on the number forwarding*) y en cierto modo es similar a un valor TTL (*time-to-live*).
- $not\_expired_i$ variable local auxiliar.

Para iniciar el algoritmo ejecutado en el proceso p_i, mostrado en la Figura 5-2, no se necesita que los procesos comiencen la ejecución simultáneamente. De hecho, iniciar la ejecución de esta forma sería imposible dado que incluso si sus relojes generan un pulso con la misma frecuencia no se encuentran sincronizados inicialmente. Por lo tanto, se asume que algunos procesos inician independientemente el algoritmo y algunos otros inician cuando reciben un mensaje alive. Cuando lo anterior sucede, los procesos deben ejecutar la rutina de inicialización antes de procesar el mensaje recibido.

En las lineas 1 a 5, del algoritmo 1 en la Figura 5-2, se describe el comportamiento en el que en un principio cada proceso p_i se propone como candidato a ser líder. El proceso p_i asigna n a $hopbound_i[i]$, asigna $timer_i[i, n]$ a $+\infty$ con la intención de que este temporizador nunca expire. Enseguida, para cada $j \neq i$, p_i asigna valores positivos arbitrarios a cada $timeout_i[j, x]$ donde $1 \leq x \leq n$, asigna las entradas $timer_i[j, x]$ a los valores asociados de timeout y asigna 0 a cada $hopbound_i[j]$.

5.3. Algoritmo de Elección Eventual de Líder en el modelo $\diamond ADD$ con Pertenencia Conocida

Algoritmo 1: Algoritmo de Elección Eventual de Líder en el modelo $\diamond ADD$ con Pertenencia Conocida

Resultado: Elección Eventual de Líder

Inicialización

1 $leader_i \leftarrow i$; $hopbound_i[i] \leftarrow n$; $asignar\, timer_i[i, n]\, a\, +\infty$;
2 **foreach** $j \in \{1, .., n\}$ ***and each*** $x \in \{1, .., n\}$ **do**
3 $\quad timeout_i[j, x] \leftarrow cualquier\, entero\, positivo$; $asignar\, timer_i[j, x]\, a\, timeout_i[j, hb]$;
4 $\quad hopbound_i[j] \leftarrow 0$;
5 **end**
6 **foreach** T ***unidades de tiempo del*** $clock_i()$ **do**
7 $\quad$ **if** $hopbound_i[leader_i] > 1$ **then**
8 $\quad\quad$ **foreach** $j \in out\_neighbors_i$ **do**
9 $\quad\quad\quad enviar\, alive(leader_i, hopbound_i[leader_i] - 1)\, a\, p_j$
10 $\quad\quad$ **end**
11 $\quad$ **end**
12 **end**
13 **cuando** $alive(l, hb \leftarrow n - k)$ **tal que** $l \neq i$ **es recibido**
14 **if** $l \leq leader_i$ **then**
15 $\quad leader_i \leq l$;
16 $\quad$ **if** $timer_i[leader_i, hb]\, expirado$ **then**
17 $\quad\quad$ incrementa valor de $timeout_i[leader_i, hb]$
18 $\quad$ **end**
19 $\quad asignar\, timer_i[leader_i, hb]\, a\, timeout_i[leader_i, hb]$;
20 $\quad not\_expired_i \leftarrow \{x | timer_i[leader_i, x] no\, expirado\}$;
21 $\quad hopbound_i[leader_i] \leftarrow max\{x \in not\_expired\}$
22 **end**
23 **cuando** $timer_i[leader_i, hb]$ **expira y** $leader_i \neq i$ **do**
24 **if** $\forall\, 1 \leq x \leq n\, timer_i[leader_i, x]\, expiraron$ **then**
25 $\quad leader_i \leftarrow i$;
26 **else**
27 $\quad not\_expired_i \leftarrow \{x | timer_i[leader_i, x] no\, expirado\}$;
28 $\quad hopbound_i[leader_i] \leftarrow max\{x \in not\_expired\}$
29 **end**

Figura 5-2: Algoritmo de Elección Eventual de Líder en modelo $\diamond ADD$ con Pertenencia Conocida. Código en p_i

Posteriormente en las líneas 6 a 11 se describe el envío de los mensajes alive tanto de generación como de retransmisión. Esto se realiza cada T unidades de su reloj local $clock_i()$, el proceso p_i envía el mensaje $alive(leader_i, hopbound_i[leader_i] - 1)$. Este envío de mensajes sea de generación o de retransmisión, está controlado por el predicado de la línea 8 que indica que el envío del mensaje correspondiente sólo se realiza si $hopbound_i[leader_i] > 1$.

El reenvío de los mensajes surge por el hecho de que si $hopbound_i[leader_i] > 1$ quizás existan procesos que aún no han recibido un mensaje $alive(leader_i, -)$ que fue enviado por $leader_i$ y propagado mediante un camino de procesos, donde cada uno de estos procesos en el camino decrementaba el valor de hopbound. Sea este el caso, p_i debe participar en la retransmisión del mensaje como se indica en la linea 8.

Este envío de mensajes propicia que solamente durante un periodo de estabilización de la red se envíe un gran número de mensajes, puesto que los procesos compiten para establecerse como líderes. Pero a partir del momento en el que la red se estabiliza, converge, solamente el proceso correcto con el menor identificador, p_l envía mensajes $alive(l, n)$ y ningún otro proceso p_j envía mensajes $alive(j, n)$.

El complemento a este envío de mensajes se presenta en las lineas 13 a 22, encargadas de la recepción de los mensajes. Cuando un proceso p_i que tiene un líder que no es el mismo, $leader_i \neq i$ recibe un mensaje de generación o de retransmisión $alive(l, hb)$ realiza la comparación del identificador del líder que el mensaje transporta y en caso de ser mayor al identificador que actualmente considera líder lo desecha, como lo indica la línea 14 en $if(l > leader_i)$. Esto claramente obedece la búsqueda de los procesos del identificador de proceso más pequeño. Si el mensaje transporta un identificador de líder más pequeño al que actualmente el proceso considera como líder, $l \leq leader_i$ entonces p_i debe considerar a l como su nuevo líder y en caso de que el valor del identificador sea el mismo el líder permanece, $l = leader_i$.

Cuando un mensaje que llega lo hace de forma indirecta a través de un camino de $k = n - hb$ diferentes procesos, p_i incrementa el valor de timeout asociado al temporizador $timer_i[leader_i, hb]$ que expiró antes de que el mensaje $alive(l, hb)$ fuera recibido, mostrado en la linea 17. En cualquier caso el temporizador ($timer_i[leader_i, hb]$) se restablece con el valor de timeout previo o el

nuevo timeout incrementado. Con el restablecimiento de este temporizador se inicia una nueva sesión de monitoreo respecto del líder actual y el camino de longitud hb desde $lider_i$ hasta él.

La finalidad del temporizador, $timer_i[l, hb]$ es permitirle a p_i monitorear p_l con respecto a los mensajes retransmitidos $alive(l, hb)$ tales que $hb = n - k$.Finalmente p_i actualiza $hopbound_i[leader_i]$, para este fin p_i debe calcular el valor de $not\_expired_i$ que contiene la longitud de los caminos elementales *elemental path* x para los cuales $timer_i[leader_i, n - x]$ se encuentre aún corriendo.

Por último, el algoritmo considera el caso en el que todos los temporizadores asociados a todos los valores de hopbound que monitorean el líder actual han expirado, nuevamente p_i se propone como candidato de líder.

Con esto tenemos la descripción completa del algoritmo y el sistema a simular. En la siguiente sección mostramos todo lo referente a la simulación y la construcción del simulador.

5.4. Construcción del Simulador del Sistema de Procesos Propensos a Fallas Ejecutando el Algoritmo de Elección Eventual de Líder en el Modelo $\diamond ADD$ con Pertenencia Conocida

La construcción del simulador una vez que contábamos con el entendimiento del sistema que se pretendía simular, el algoritmo que los procesos estarían ejecutando y las capacidades y limitaciones del *framework* se realizó como un proceso incremental.

Iniciamos construyendo los componentes fundamentales: la abstracción los procesos y los canales que los comunicaban cumpliendo la propiedad $\diamond ADD$. Una vez que la comunicación entre procesos estuvo resuelta, incluyendo el paso de mensajes elementales, los retrasos en la comunicación y la pérdida de mensajes obedeciendo las constantes K y D involucradas, se integraron los procesos como nodos de una red y los canales como aristas con ayuda de la biblioteca de NetworkX para poder configurar redes o familias de redes para realizar simulaciones en un

gran número de configuraciones que compartían ciertas características en común. Finalmente, la simulación en conjunto se migró a una arquitectura de servicios que permitía la simulación de configuraciones particulares que se dieran de alta o bien la simulación en masa de familias de redes que compartirían parámetros de simulación.

5.4.1. Abstracción de los Procesos y Canales ADD

Partiendo de la descripción del algoritmo de la Figura 5-2 que muestra un pseudocódigo del algoritmo que cada proceso de la red ejecutará, llevarlo a código en Python es un proceso directo, pero en el que hay que hacer ciertas consideraciones previas de diseño y manejo cuidadoso del lenguaje para no introducir comportamientos no esperados en la simulación. Las clases que se muestras a continuación en el Código 1 (*process.py*) en conjunto con la clase del Código 2 (*add_channel.py*), conforman el núcleo de la simulación por esta razón serán revisadas con mayor detalle que el resto de los componentes del simulador. Estos componentes que no serán revisados en detalle pueden ser encontrados en el apéndice A dedicado al código auxiliar del simulador.

Adicionalmente como fue comentado brevemente al inicio de esta sección, el lector debe saber que, si bien esta es la funcionalidad central del simulador, no muestra la construcción completa del mismo. Con la intención de hacer más amigable su uso, una vez que la funcionalidad básica se encontraba desarrollada, se creó un cliente web para definir configuraciones de la simulación, dar de alta redes sobre las que se simularía el algoritmo y mostrar gráficamente los resultados de la simulación. Este cambio de arquitectura a servicios web expuestos por un *back-end* encargado estrictamente de la simulación, cuyo *core* es revisado en esta sección, y un *front-end* que consume estos servicios y presenta los resultados gráficamente se describen brevemente en la sección 5.5 de este capitulo.

Código 1: Clase Proceso en la Simulación - process.py

```
1  import numpy
2  import simpy
3  import sys
4  import logging
5
```

```
6  from simulador.src.add_channel import ADDChannel
7  from simulador.src.message import Message
8  from simulador.src.metrics import Metrics
9  from simulador.src.simulation_params import SimulationParams
10
11 log = logging.getLogger(__name__)
12
13
14 class Process:
15     def __init__(self, env: simpy.Environment, id_nodo, num_nodos, args, metricas, config):
16         # Entorno de simulacion de SimPy
17         self.env = env
18
19         self.id_nodo = id_nodo
20         self.num_nodos = num_nodos
21         self.lider = id_nodo
22         self.hopbound = [0 for x in range(num_nodos)]
23         self.timeout = [[0 for x in range(num_nodos)] for y in range(num_nodos)]
24         self.timer = [[0 for x in range(num_nodos)] for y in range(num_nodos)]
25         self.set_hopbound()
26         self.set_timeout()
27         self.set_timer()
28
29         # Variables auxiliares
30         self.config = config
31         self.lider_registrado = False
32         self.nuevo_lider_registrado = False
33         self.bandera_deteccion = False
34
35         self.args = args
36         self.metricas = metricas
37
38     def set_hopbound(self):
39         for i in range(len(self.hopbound)):
40             if self.id_nodo != i:
41                 self.hopbound[i] = 0
42             else:
43                 self.hopbound[i] = self.num_nodos
44
45     def set_timer(self):
46         for i in range(len(self.timer)):
47             for j in range(len(self.timer[i])):
48                 self.timer[i][j] = self.timeout[i][j]
49         self.timer[self.id_nodo][self.num_nodos - 1] = sys.maxsize
50
51     def set_timeout(self):
52         for i in range(len(self.timeout)):
53             for j in range(len(self.timeout[i])):
54                 if i != self.id_nodo:
55                     self.timeout[i][j] = numpy.random.randint(1, self.num_nodos)
56
57     def send_alive(self, canal: ADDChannel, params: SimulationParams):
58         while True:
```

```
59              if self.hopbound[self.lider] > 1:
60                  alive = Message(self.lider, self.hopbound[self.lider] - 1)
61                  canal.intenta_entrega(alive)
62              yield self.env.timeout(params.config['param_T'])
63              self.decrementa_timer(self.config['param_T'])
64
65      def on_alive(self, canal: ADDChannel):
66          while True:
67              mensaje_alive = yield canal.recibe()
68              if self.id_nodo != mensaje_alive.lider:
69                  if mensaje_alive.lider <= self.lider:
70                      self.metricas.mensaje_procesado()
71                      self.lider = mensaje_alive.lider
72
73                      if self.check_timer(mensaje_alive.hopbound):
74                          self.incrementa_timeout(self.lider, mensaje_alive.hopbound)
75                      self.timer[self.lider][mensaje_alive.hopbound] = self.timeout[self.
                            lider][mensaje_alive.hopbound]
76                      not_expired = self.valores_no_expirados()
77                      self.hopbound[self.lider] = max(not_expired)
78
79              if self.lider != mensaje_alive.lider:
80                  if self.check_timer(mensaje_alive.hopbound):
81                      if self.todos_expirados():
82                          if self.lider == 0 and (self.env.now > self.args['
                                tiempo_promedio_convergencia']) and not self.bandera_deteccion
                                :
83                              self.metricas.acumula_t_deteccion(self.args['
                                    tiempo_promedio_convergencia'], self.env.now)
84                              self.bandera_deteccion = True
85                          self.lider = self.id_nodo
86                      else:
87                          not_expired = self.valores_no_expirados()
88                          self.hopbound[self.lider] = max(not_expired)
89
90
91              # Instrucciones auxiliares para la mediciõn de tiempos
92              # Instrucciones para verificar si la red ha convergido
93              if self.lider == 0 and not self.lider_registrado:
94                  self.metricas.nodos_lider_encontrado()
95                  self.lider_registrado = True
96                  if self.metricas.get_num_lideres() == self.num_nodos:
97                      self.metricas.set_t_conv(self.env.now)
98                      print("Red convergio en {}".format(self.env.now))
99
100             # Instrucciones para verificar si la red ha reconvergido despues de la falla de
                    un proceso
101             if self.lider == 1 and not self.nuevo_lider_registrado:
102                 self.metricas.nodos_nuevo_lider_encontrado()
103                 self.nuevo_lider_registrado = True
104                 if self.metricas.get_nuevo_num_lideres() == self.num_nodos - 1:
105                     self.metricas.set_t_reconv(self.env.now)
106                     print("Red reconvergio en {}".format(self.env.now))
```

```
107
108     def incrementa_timeout(self, lider_i, hbd):
109         self.timeout[lider_i][hbd] *= 2
110         return self.timeout[hbd]
111
112     def check_timer(self, hopbound):
113         return (self.timer[self.lider][hopbound]) <= 0
114
115     def valores_no_expirados(self):
116         valores_no_expirados = []
117         for x in range(len(self.timer[self.lider])):
118             if self.timer[self.lider][x] > 0:
119                 valores_no_expirados.append(x)
120         return valores_no_expirados
121
122     def todos_expirados(self):
123         for x in range(len(self.timer[self.lider]) - 1):
124             if (self.timer[self.lider][x]) > 0:
125                 return False
126         return True
127
128     def decrementa_timer(self, ticks_decremento):
129         for i in range(len(self.timer)):
130             for j in range(len(self.timer[i])):
131                 if i != self.id_nodo:
132                     self.timer[i][j] -= ticks_decremento
133
134     def info_proceso(self):
135     info = "Proceso: p_{} lider: {}".format(self.id_nodo, self.lider)
136     return info
137
138     def __str__(self):
139         return "p_{}".format(self.id_nodo)
```

Cada objeto que sea instancia de la clase Proceso debe recibir 6 argumentos en su constructor. Estos argumentos son los que rigen el comportamiento de cada proceso y la interacción con el resto de los procesos en la red definidos en el constructor en la linea 15:

- *env* - simpy.Environment. El entorno propio de SimPy que permite el manejo de los tiempos de la simulación y los mecanismos para controlar el progreso en pasos de esta y comunicación entre los procesos.

- *id_nodo* - int. Un entero que define la identidad del proceso que se esta creando. Este número es recibido desde una clase superior *LeaderElectionSimulator* encargada de iniciar un proceso de SimPy por cada proceso que existe en la red.

- *num_nodos* - int. Un entero que le permite al proceso que se está creando conocer el orden de la red en su conjunto. Requerido por el algoritmo en su versión de membresía conocida.

- *args* - dictionary. Un diccionario empleado cuando la simulación se ejecutaba desde una línea de comandos, permitía controlar la salida *verbose* para monitorear la ejecución en curso o bien disminuir la salida *verbose* del simulador. En la versión presentada se emplea únicamente como auxiliar para el cálculo de tiempo de detección de la falla en una simulación.

- *metricas* - simulador.src.Metricas - Objeto que permite recabar mediciones de los tiempos de convergencia, tiempos de reconvergencia, tiempos de detección de falla, número de mensajes enviados, mensajes procesados y mensajes perdidos al finalizar la simulación y persistirlos para su posterior análisis.

- *config* - dictionary. Diccionario con los parámetros de configuración involucrados en la simulación. Incluye el valor del parámetro T para controlar la periodicidad con la que el proceso debe realizar el envío de los mensajes alive.

Continuando la instanciación de un objeto de la clase Process, se ejecutan las lineas 16 - 36 correspondientes a la inicialización descritas en el algoritmo. Se asigna la variable local propia de cada proceso a su propio identificador $self.lider = id\_nodo$ en la línea 22, se declaran las matrices que manejan las sesiones de monitoreo del líder y se inicializan, nuevamente siguiendo el algoritmo con las líneas 22 - 27. Por último, se asignan las variables auxiliares para realizar las mediciones de interés en las líneas 30 -36.

Con lo anterior se cubre la rutina de inicialización del algoritmo original. A continuación el algoritmo describe el comportamiento de los procesos cada T unidades de tiempo en el método *send_alive*() en la linea 57. Este método lo ejecutará el proceso durante la duración completa de la simulación de ahí la linea 58 que indica un ciclo *while* que no termina por cuenta propia, sino por la finalización del entorno creado en la clase *Simulation*, esta clase es donde se define la duración de la simulación y encargada de controlar la simulación y la creación

de los objetos involucrados en ella, su código se presenta en el apéndice A. Continuando el envío de los mensajes de alive se respeta la descripción del algoritmo original, se llama el método $intenta\_entrega()$ que pone en la cola de envío de un objeto de clase $ADDChannel$ el mensaje, el canal en cuestión puede entregar el mensaje con un retraso que cumple la cota D de la simulación o bien perderlo, este comportamiento se detalla en la siguiente sección. Finalizando la tarea del proceso de poner el mensaje a disposición del canal ADD, el proceso debe esperar otras T unidades de tiempo hasta realizar un nuevo envío, esto esta controlado por la línea 62 $yield\, self.env.timeout(params.config['param\_T'])$. Se aprovecha este momento para actualizar el temporizador $timer$ del proceso que esta enviando el mensaje, pues han pasado T unidades de tiempo en el instante que alcanza esta línea, con el método $decrementa\_timer()$ de la línea 63.

El complemento al método $send\_alive()$ se encuentra en el método $on\_alive()$ en la línea 65, este método es llamado por un proceso de SimPy que se encuentra siempre en espera de un mensaje que alguno de los Canales ADD que lo conectan a otros procesos en la red le hagan llegar. Propiamente la línea 67 es la encargada de recibir el mensaje: $mensaje\_alive = yieldcanal.recibe()$. Si ningún canal ADD ha puesto un mensaje a disposición del proceso que se encuentra ejecutando esta línea el método devuelve el control a otro proceso hasta el siguiente instante de tiempo en el que compruebe nuevamente la presencia de un mensaje. En caso de que el proceso haya recibido un mensaje de alguno de sus procesos vecinos ejecuta lo descrito en el algoritmo, empezando en la línea 68, realizando las comparaciones de que el líder que contiene el mensaje de alive sea menor al que considera actualmente como líder y en caso de que no sea así descarta el mensaje. Cuando el líder que trasporta el mensaje alive recibido es menor al que el proceso p_i consideraba en el instante de recepción inicia el procesamiento del mensaje en la línea 70 se invoca un método de un objeto auxiliar para recabar las métricas de que un mensaje ha sido procesado, posteriormente se realiza la actualización de la variable de líder en la línea 71, se comprueban el temporizador que monitorea el líder que ha sido recibido en el mensaje y en caso de que se encuentre expirado se duplica el valor (esta estrategia de aumento del valor del timeout puede ser cambiada para generar menor número de falsos positivos de fallas) que

el *timeout* tenía puesto que ha expirado prematuramente, todo esto en las líneas 73 y 74. A continuación independientemente de si el temporizador expira o no, se debe resetear para dar inicio a una nueva sesión de monitoreo para el líder actual, descrito en la línea 75. Lo ultimo, es el cálculo del hopbound siguiendo la descripción del algoritmo original en la líneas 76 y 77.

La verificación en el caso de que todos los temporizadores se encuentren expirados se muestra en la línea 80 y 81. Cuando esto sucede el proceso debe proponerse nuevamente como líder a el mismo, descrito en la linea 85. Previo a esta línea se encuentran mecanismos ajenos al algoritmo para realizar mediciones de tiempos de detección de fallas, en las simulaciones en las que el líder falla después de un instante en el tiempo conocido, se acumula este valor de tiempo de detección por proceso y se calcula su promedio cuando todos los procesos han detectado la falla del líder que originalmente era candidato obteniendo el tiempo promedio de detección de falla en la red completa. Las líneas siguientes describen instrucciones auxiliares para conocer el tiempo en el que la red converge en las líneas 93 - 98, o si reconverge en las líneas 101 - 106, describen un mecanismo muy sencillo para detectar que esto ha sucedido en la red. Simplemente se compara el número de procesos que consideran el líder de menor identificador contra el número de procesos en la red, cuando estos dos valores son iguales, podemos decir con confianza que la red ha convergido. Ocurre algo similar para conocer que la red ha reconvergido, con la diferencia que se compara el número total de procesos en la red disminuido en uno, por el proceso que ha fallado, contra el número de procesos que consideran como líder al siguiente identificador de proceso menor.

Finalmente los métodos $info\_procesos()$ y $\_\_str\_\_()$ son métodos que facilitaban observar la salida con fines descriptivos para cada proceso.

5.4.2. Comunicación de los Procesos

Ahora continuamos revisando el código asociado a la abstracción del canal ADD. La comunicación entre los procesos se logra implementando la clase Store de SimPy. Esta clase permite la intercomunicación de elementos de la simulación, separando el tiempo de *delay* de los eventos entre los procesos, el envío y recepción de los mensajes, y los procesos mismos. Esta del canal

ADD es una modificación del ejemplo encontrado en la documentación de SimPy para mostrar latencia entre eventos *Event Latency*.

La razón para implementar estos canales de esta forma es porque permite modelar objetos físicos como cables o propagación de RF, etc. Encapsulando para mantener el mecanismo de propagación fuera de los procesos emisores y receptores. Resulta también ideal para simular componentes que se comunican enviando mensajes [30]

Código 2: Clase Canal ADD en la Simulación - add_channel.py

```python
import simpy
import logging

from numpy import random

from simulador.src.metrics import Metrics
from simulador.src.simulation_params import SimulationParams

log = logging.getLogger(__name__)


class ADDChannel:
    def __init__(self, env: simpy.Environment, params: SimulationParams, src, dst, metricas
        : Metrics, args):
        # Entorno de simulaciõn de SimPy
        self.env = env

        self.src = src
        self.dst = dst
        self.param_K = params['param_K']
        self.param_D = params['param_D']
        self.drop_rate = params['drop_rate']

        # Variables auxiliares
        self.ventana_msj = 0
        self.falla = args['falla']
        self.tiempo_falla = params['tiempo_promedio_convergencia']
        self.metricas = metricas

        self.args = args
        self.mensaje_no_mostrado = True
        self.cola_envio = simpy.Store(env)


    def intenta_entrega(self, mensaje):

        if self.ventana_msj == self.param_K:
            self.ventana_msj = 0
            self.metricas.mensaje_enviado_limite()
            self.envia(mensaje, random.randint(1, self.param_D), self.src, self.dst)
```

```
40          elif random.randint(0, 100) > (self.drop_rate * 100.0):
41              self.ventana_msj = 0
42              self.envia(mensaje, random.randint(1, self.param_D), self.src, self.dst)
43          else:
44              self.ventana_msj += 1
45              self.metricas.mensaje_perdido()
46
47      def envia(self, mensaje, retraso, src, dst):
48          self.env.process(self.latencia(mensaje, retraso))
49          if self.args['verbose']:
50              print("Envío -> Proceso {} envia {} a {} con latencia {} en {}"
51                    .format(src, mensaje, dst, retraso, self.env.now))
52
53      def latencia(self, mensaje, retraso):
54          self.metricas.mensaje_enviado()
55          # Codigo para hacer fallar al proceso en caso de que la simulacion lo indique
56          if self.falla == 'True':
57              if mensaje.lider != 0:
58                  yield self.env.timeout(retraso)
59                  self.cola_envio.put(mensaje)
60              else:
61                  if self.env.now < self.tiempo_falla:
62                      yield self.env.timeout(retraso)
63                      self.cola_envio.put(mensaje)
64                  elif self.env.now >= self.tiempo_falla and self.mensaje_no_mostrado and
                        self.src.id_nodo == 0:
65                      print("El lider falla en {}".format(self.env.now))
66                      self.metricas.set_t_falla(self.env.now)
67                      self.mensaje_no_mostrado = False
68          else:
69              yield self.env.timeout(retraso)
70              self.cola_envio.put(mensaje)
71
72      def recibe(self):
73          self.metricas.mensaje_entregado()
74          return self.cola_envio.get()
75
76      def __str__(self):
77          return "(Canal ADD de {} a {}: (param_k: {}, param_d: {}))"\
78              .format(self.src, self.dst, self.param_K, self.param_D)
```

El constructor de la clase *ADDChannel*, línea 12, recibe los siguientes argumentos:

- *env* - simpy.Environment. El entorno propio de SimPy que permite el manejo de los tiempos de la simulación y los mecanismos para controlar el progreso en pasos de la misma y comunicación entre los procesos.
- *params* - dictionary. Un diccionario de python con los parámetros que describen el canal ADD que debe crearse. Contiene el valor para el parámetro K, el parámetro D y la tasa

de pérdidas que el canal tendrá.

- *src, dst* - simulador.src.Process. Referencias a los procesos de origen y destino que el canal interconecta.
- *metricas* - simulador.src.Metricas - Objeto que permite recabar mediciones de los tiempos de convergencia, tiempos de reconvergencia, tiempos de detección de falla, número de mensajes enviados, mensajes procesados y mensajes perdidos al finalizar la simulación y persistirlos para su posterior análisis.
- *args* - dictionary. Un diccionario empleado cuando la simulación se ejecutaba desde una línea de comandos, permitía controlar la salida *verbose* para monitorear la ejecución en curso y disminuir la salida *verbose* del simulador. En la versión presentada se emplea únicamente para como auxiliar para el cálculo de tiempo de detección de la falla en una simulación y bandera que define si un proceso debe fallar.

Continuando la instanciación de un objeto de la clase ADDChannel, se ejecutan las líneas 14 - 31 correspondientes a la inicialización de los parámetros y variables auxiliares. Las variables auxiliares *ventana_msj* en conjunto con *param_K* controlarán el cumplimiento de una de las propiedades de un canal ADD, la entrega de al menos un mensaje de K mensajes consecutivos. $tiempo\_falla$ controla el caso en que la simulación debe hacer que un proceso falle, propiamente indica el instante en el cual los canales asociados al proceso que falla deben cesar permanentemente toda comunicación. Existe una bandera que controla la salida con fines informativos, *mensaje_no_mostrado* y finalmente un objeto simpy.Store propio de *SimPy* que se comporta como un buffer de envío para el canal en cuestión, *cola_envio*.

Ahora pasamos al método que los procesos llaman cada T unidades de tiempo, $intenta\_entrega()$, en la línea 36 aquí se describe el comportamiento y la toma de decisión del canal para determinar si el mensaje alive debe ponerse en el buffer *cola_envio* para que pueda ser recibido en el destinatario. En la linea 36 se pregunta si la *ventana_envio* ha alcanzado el umbral máximo permitido por *param_K*, en caso de que sean iguales el canal debe colocar el mensaje en el buffer de envío para respetar la propiedad ADD. En caso de que la ventana límite de envío

continúe vigente, se intenta colocar el mensaje en el buffer de envío según una probabilidad en función del valor de *drop_rate*, en la línea 40, si el mensaje no se coloca en el buffer, línea 44, entonces el mensaje se pierde y se incrementa la métrica de mensajes perdidos.

Cuando un mensaje se colocará en el buffer de envío primero se le asigna un retraso (*delay*) que va desde 1 hasta el valor definido por $param_D$ propiamente se observa esto en el método *envia*() que internamente hace uso del método *latencia*(). Además de ser responsable de colocar los mensajes en *cola_envio* el método *latencia*() también es el encargado de describir el comportamiento de los envíos, o mejor dicho del cese de envíos, de un proceso cuando se ha alcanzado el tiempo de falla indicado por $tiempo\_falla$ en las líneas 56 - 67. El envío efectivo se realiza con el método *put*() de la instancia *cola_envio* de la clase Store, que crea un evento StorePut para colocar el mensaje en el buffer. El método que complementa el envío de mensajes es el método *recibe*() en la línea 72, que es invocado desde un objeto de la clase Process cuando verifica si ha recibido un mensaje. Simplemente invoca el método *get*() del objeto Store que genera un evento para recuperar un mensaje del buffer.

Con esta revisión a las dos clases que conforman el núcleo de la simulación podemos revisar brevemente el mecanismo que ocupamos para describir las redes empleadas en la simulación con ayuda de NetworkX.

5.4.3. Descripción de las Redes en el Simulador

Al emplear el paquete de Python NetworkX se puede realizar la manipulación, creación y el estudio de estructuras y dinámica de redes complejas. [31]

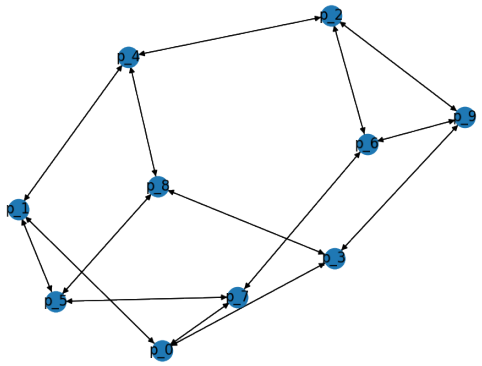
La funcionalidad obtenida al emplear este paquete es la de realizar el parseo de un archivo que contiene la descripción de cualquier grafo que representa una red en notación GML *Graph Modeling Language*. Esta notación es un formato de archivo basado en un ASCII jerárquico que describe grafos. Un ejemplo de esta notación se presenta en la Figura 5-3, donde se describe el grafo de una red aleatoria regular con grado ($d = 3$) y con 10 nodos y la representación gráfica generada.

La intención de emplear archivos que usan esta notación es brindar flexibilidad de con-

figuración de cualquier red o familias de redes que necesitemos. En nuestro simulador como se mostrará en el capítulo siguiente, empleamos dos familias de redes: Anillos o Ciclos y Redes Aleatorias Regulares. El lector debe saber que el simulador se puede extender a cualquier red o familia de redes descritas mediante GML, como en nuestro caso las conocidas redes de Barabasi-Albert que fueron empleadas en un principio antes de decidir por las familias de redes especificadas anteriormente. Adicionalmente de la flexibilidad de configuración que permiten estos archivos, es importante mencionar que el paquete NetworkX provee la funcionalidad para generar redes particulares y familias de redes conocidas. [32] NetworkX provee algunos algoritmos de aplicación común en los grafos como algoritmos para medir distancias, conectividad, etc. [33] Si bien la notación GML es poderosa, existen otros mecanismos que se pueden emplear para especificar una red. Empleando nuevamente la funcionalidad proporcionada del paquete NetworkX se implementó un mecanismo para especificar una red a través de una lista de adyacencias.

```
graph [
node [id 0 label "0"]
node [id 1 label "1"]
node [id 2 label "7"]
node [id 3 label "3"]
node [id 4 label "8"]
node [id 5 label "2"]
node [id 6 label "4"]
node [id 7 label "5"]
node [id 8 label "9"]
node [id 9 label "6"]
edge [source 0 target 1]
edge [source 0 target 2]
edge [source 0 target 3]
edge [source 1 target 7]
edge [source 1 target 6]
edge [source 2 target 7]
edge [source 2 target 9]
edge [source 3 target 4]
edge [source 3 target 8]
edge [source 4 target 7]
edge [source 4 target 6]
edge [source 5 target 6]
edge [source 5 target 8]
edge [source 5 target 9]
edge [source 8 target 9]
]
```

(a) Archivo GML que describe una Red Aleatoria Regular ($n = 10, d = 3$)



(b) Red producida por el archivo GML de (a)

Figura 5-3: Ejemplo de Archivo en Notación GML y Grafo Generado

5.5. Simulador en su Versión Web

5.5.1. Arquitectura del Simulador

Una vez que el funcionamiento del simulador fue validado y las primeras simulaciones empezaban a generar resultados surgió la necesidad de facilitar la configuración y especificación de las redes que el simulador debía simular. Adicionalmente el manejo de archivos de texto como mecanismo para guardar los resultados arrojados por la simulación y el posterior procesamiento de lo datos contenidos en estos archivos pronto se volvió una tarea poco práctica. Por ello con una primera versión estable, la solución original del simulador se construyó la arquitectura mostrada en la Figura 5-4.

Con esta nueva arquitectura, la especificación de la configuración que el simulador debía

ejecutar que previamente se realizaba mediante una línea de comandos fue reemplazada por solicitudes HTTP que un cliente realizaría al servidor encargado de realizar la simulación propiamente. Los archivos de resultados fueron reemplazados por un Sistema Manejador de Base de Datos que permitía estructurar, almacenar y extraer los resultados para su análisis.

Esta sección no pretende mostrar todos los detalles de implementación de la solución completa del simulador y únicamente se redacta porque resultará relevante conocer de donde provienen, como se almacenaban y que información conformaba los resultados que el simulador genera en el capítulo dedicado a su análisis.

Para concluir, mencionar que el consumo de recursos computacionales cuando se pasa de redes de algunos cientos de nodos ejecutando el algoritmo, cada uno con sus variables asociadas (matrices de temporizadores, matrices de *timemouts*, etc.) a configuraciones de red con miles o decenas de miles de nodos, necesarias para describir con certeza el comportamiento de las redes, aumentan en una medida no prevista. Un cálculo rápido en memoria para una red con 50 000 procesos requería al menos 24 GB de memoria RAM para su ejecución en una versión optimizada cuando los procesos no eran propensos a fallar. Esta cantidad de memoria y poder computacional para mantener los tiempos de ejecución dentro de un rango razonable escapaba la capacidad de cómputo a nuestra disposición.

Por esta razón, después de presentar la problemática y evaluar las posibles soluciones para llevar a cabo las simulaciones requeridas por este trabajo, se solicitó ayuda al Laboratorio Universitario de Cómputo de Alto Rendimiento (LUCAR) del Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) quienes proporcionaron los recursos computacionales con los que se realizaron las simulaciones requeridas.

5.5.2. Estructura y Peticiones del Cliente

En la Figura 5.5 se presenta la estructura del simulador en el *front-end* y un ejemplo de petición hacia el *back-end* de forma ilustrativa al lector. Se observa que la petición POST va dirigida a un *endpoint* identificado por la URL */simulador/simulacion/bulk*. Esta URL ejecutará la simulación para todas las redes de anillos que estén plenamente descritas con un archivo

GML, en hasta 40 procesos de forma simultanea. El *back-end* expone también un URL para ejecutar una sola simulación particular, */simulador/simulacion/single*, para una configuración previamente registrada. Por último, para la simulación de redes aleatorias regulares en masa, para todas las redes de esta familia que se encuentren registradas, se expone el servicio en */simulador/simulacion/bulk_aleatorias*.

5.5.3. Vista del Cliente

En las Figuras 5.6 y 5.7 se presentan algunas imágenes que presentan el cliente del simulador como se muestra la aplicación en un explorador web. En la primera figura se muestra el apartado correspondiente a la simulación y búsqueda de resultados para ejecuciones previas.

En la simulación existen dos posibles caminos: la simulación de una configuración particular o la simulación en masa para una familia de redes, en cualquiera de los casos, la simulación requiere cierta información que en el cliente debe capturarse mediante un formulario. En el caso de la simulación para una configuración particular viaja el *id* asociado a la configuración que debe simularse. Esta configuración plenamente identificada por este *id* tiene los campos para los parámetros empleados en la simulación (*param_K*, *param_D*, *param_K* y *drop_rate*). En el caso de la simulación en masa muchas configuraciones para una familia de redes, se solicita mediante el formulario la familia que se simulará y los valores de los parámetros que compartirán las simulaciones. Se comprueba si las configuraciones involucradas en las simulaciones que se realizarán existen previamente en la BD o si deben crearse. Al finalizar la ejecución, se muestra una tabla con los resultados arrojados por la simulación, en el caso de una simulación particular. En el caso de las simulaciones en masa se muestra un gráfico con el comportamiento de tiempos de convergencia para todas las simulaciones que se ejecutaron.

En la Figura 5.6 en (a),(b) y (e) se muestra la ejecución finalizada para una simulación con una configuración particular. Un mensaje de éxito, el grafo involucrado en la simulación, y una tabla con los resultados.

Adicionalmente el apartado de simulación también ofrece los mecanismos para realizar la búsqueda de resultados para ejecuciones previas del simulador. Para este fin se expone un

servicio de consulta mediante un GET a la URL */simulador/busqueda/* donde se deben pasar los criterios de la búsqueda. La Figura 5.6 en (c) y (d) muestran una consulta de ejemplo para anillos y para redes aleatorias donde se ha variado el Parámetro T. El resultado de la consulta regresa un gráfico con el comportamiento de los tiempos de convergencia y modelos matemáticos de mejor ajuste según la tendencia de los datos que el simulador tiene registrados.

Pasando a la Figura 5.7 se muestran los apartados de Redes y de Configuración. Estos apartados permiten el alta, baja, cambios y la consulta de Redes y Configuraciones que se encuentran registradas para llevar a cabo la simulación. Adicionalmente a lo anterior, en el caso de las configuraciones se pueden consultar los resultados ligados a ella y alcanzar gráficos mas detallados con esa información como se muestra en (c) y (d). De igual manera en la Figura 5.7, se muestran los mecanismos para dar de alta una nueva red a través de una lista de adyacencias o subiendo un archivo GML al servidor en (e) y (f).

5.5.4. Estructura del Servidor y la Base de Datos

En la Figura 5.8 se presenta la estructura del simulador en el *back-end* y el diagrama ER (Entidad Relación) que describe la Base de Datos.

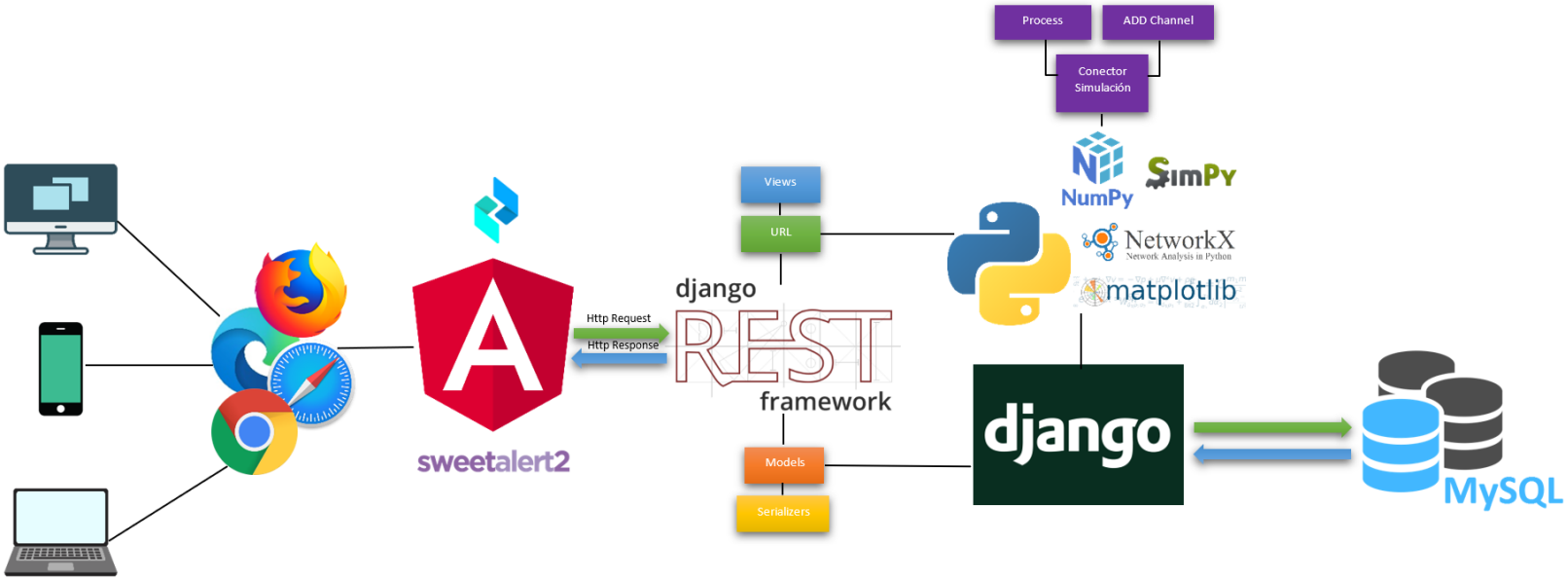
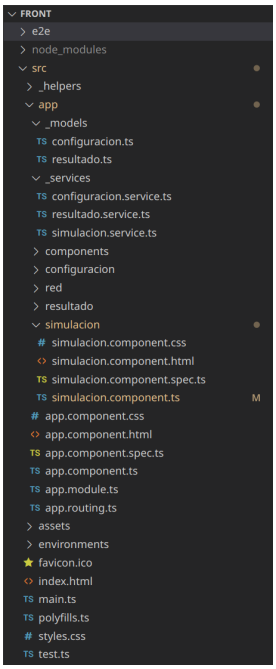
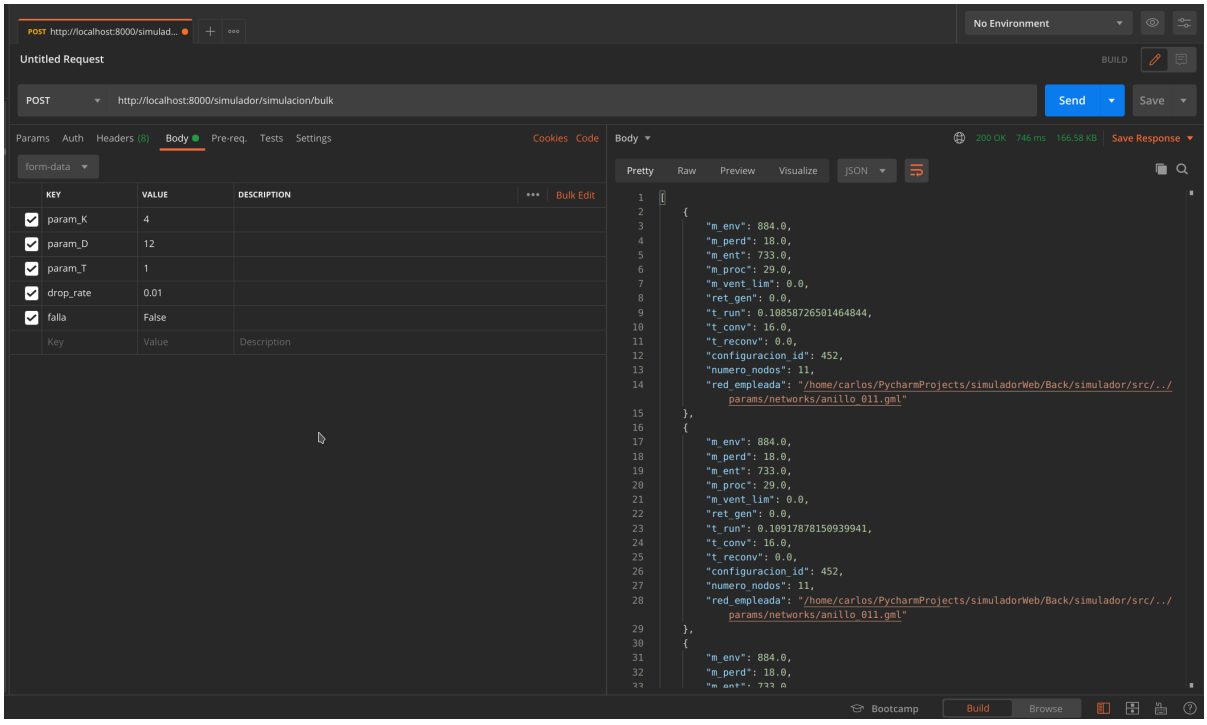


Figura 5-4: Se muestra a la izquierda los clientes que mediante un explorador web acceden a una aplicación construida con Angular, realizan peticiones HTTP hacia un servidor que ejecuta el simulador y recibe estas peticiones con ayuda de los *frameworks* Django y Django REST, este servidor se comunica con una Base de Datos MySQL donde persisten la información con los resultados arrojados por la simulación para su posterior análisis

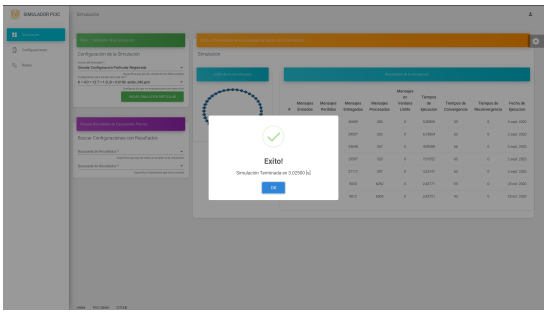


(a) Estructura del Proyecto para el Cliente en el *front-end*.

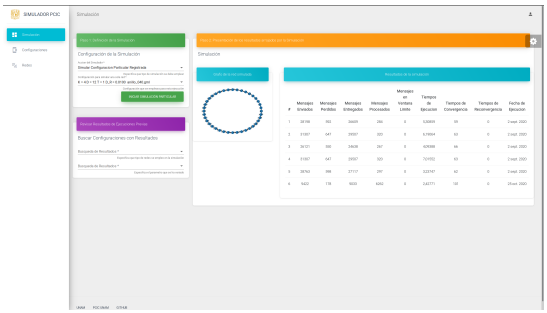


(b) Petición enviada desde el cliente hacia el Servidor para iniciar la Simulación

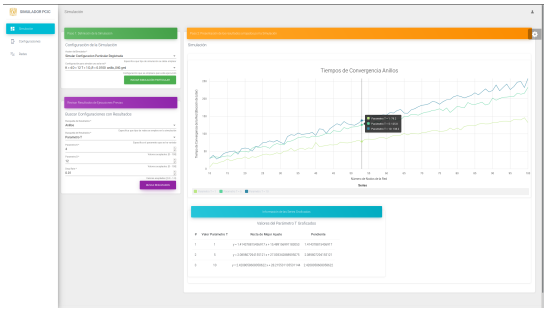
Figura 5-5: Estructura del proyecto del cliente del Simulador empleando Angular y *request* para iniciar una Simulación. Parámetros K = 4, D = 12, T = 1 y Drop Rate = 0.01 para todas las redes de anillos registradas



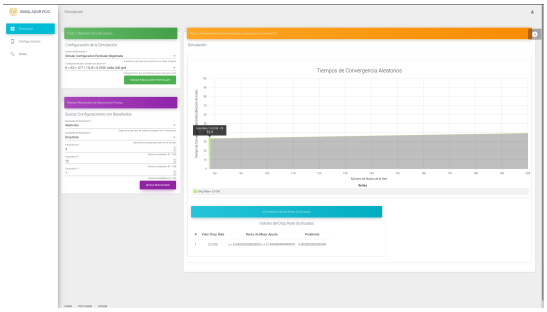
(a) Ejecución finalizada de Simulación para una configuración particular de un anillo.



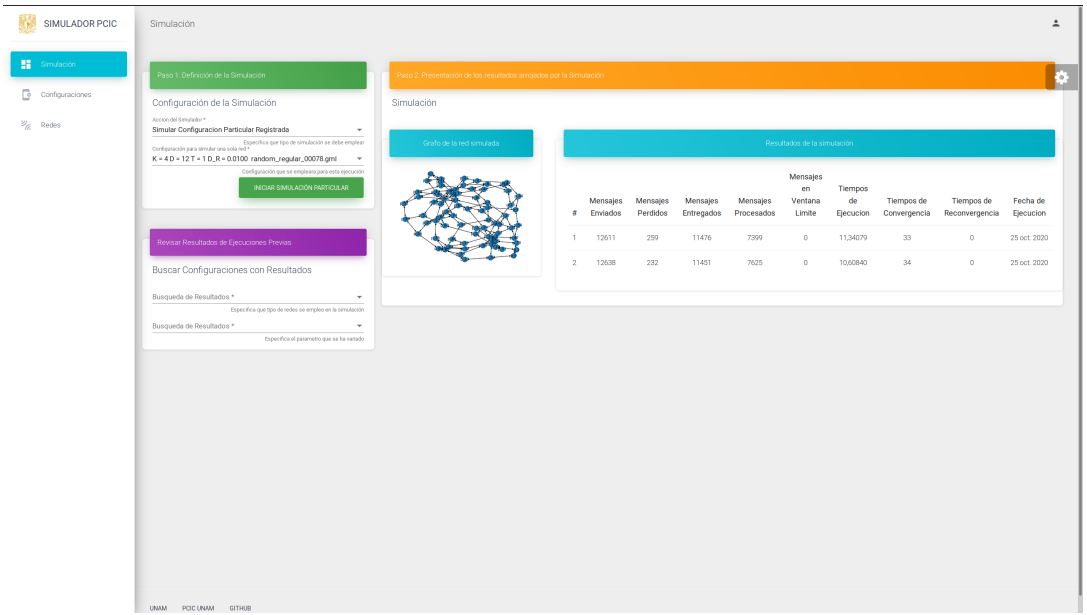
(b) Vista de presentación de resultados para una configuración particular.



(c) Consulta de resultados para anillos variando Parámetro T.

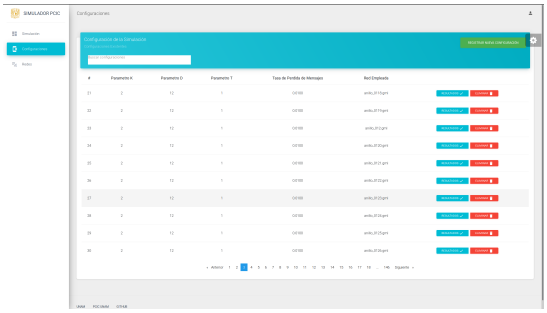


(d) Consulta de resultados para redes aleatorias regulares variando Parámetro T.

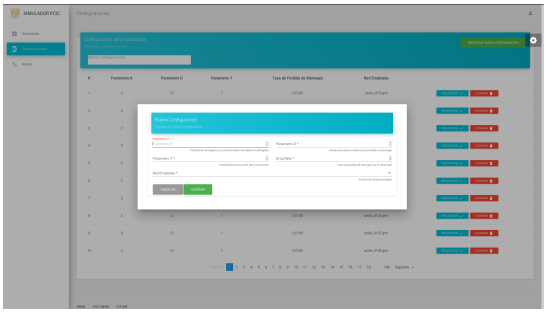


(e) Vista de presentación de resultados para una configuración particular de una red aleatoria.

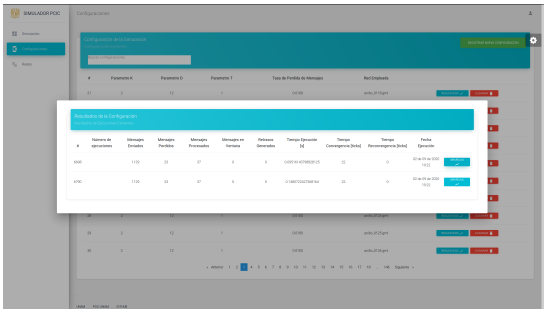
Figura 5-6: Vistas del Cliente Web. Apartado de Simulación y Búsqueda



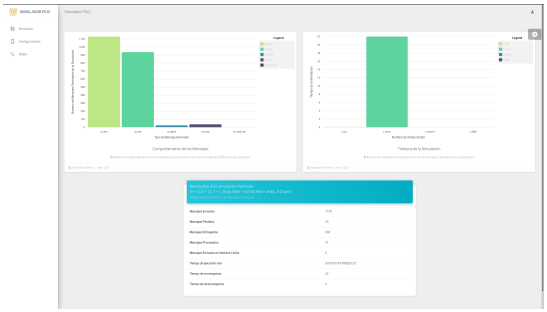
(a) Vista para el apartado de Configuraciones.



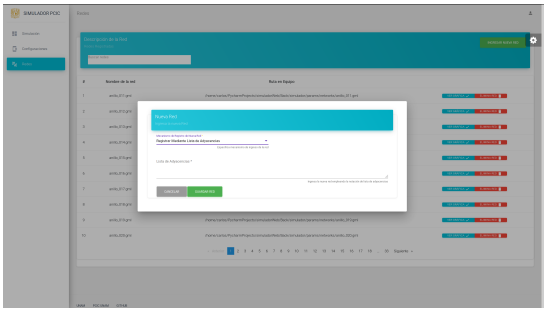
(b) Formulario de alta de una nueva Configuración.



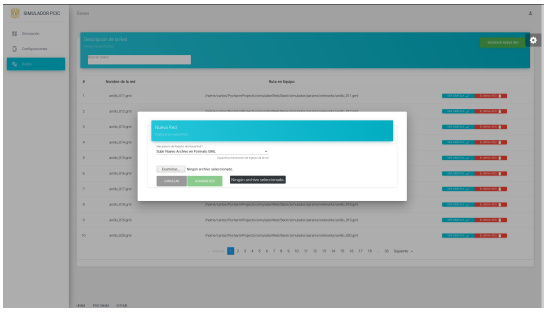
(c) Revisión de resultados asociados a una Configuración.



(d) Generación de Gráficos de resultados asociados a una Configuración.

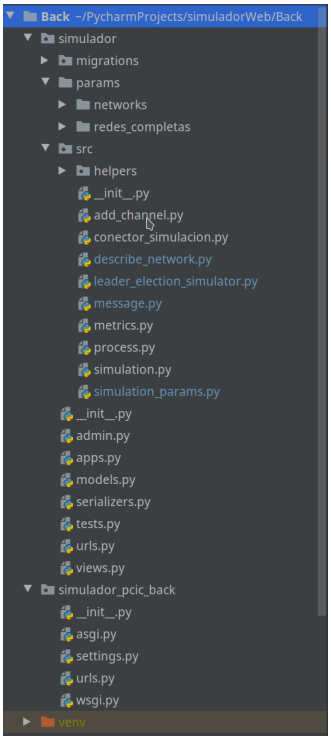


(e) Alta de una red descrita como una lista de adyacencias.

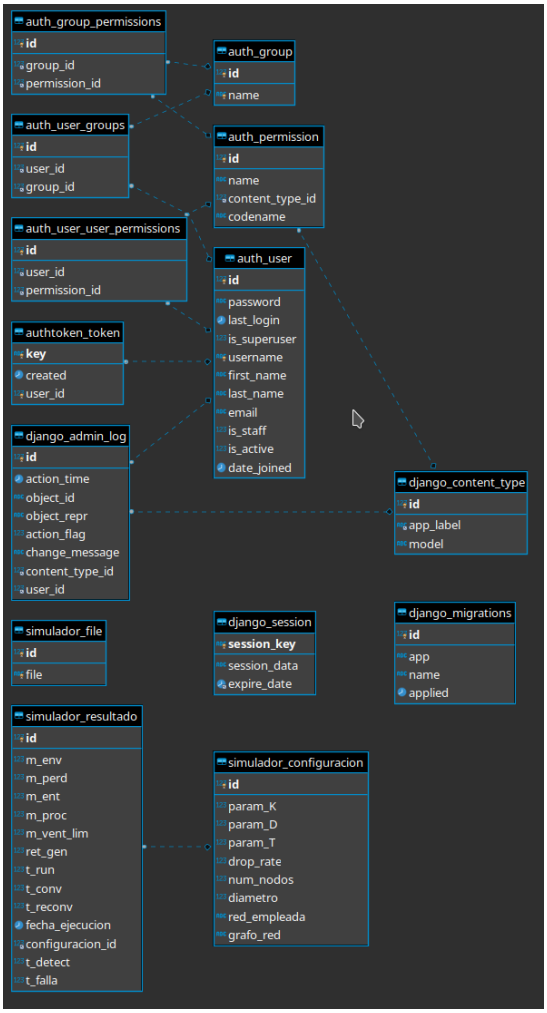


(f) Alta de una red mediante archivo con descripción en notación GML.

Figura 5-7: Vistas del Cliente Web. Apartado de Configuración, Resultados y Redes



(a) Estructura del Proyecto para el Servidor en el *back-end*.



(b) Modelo Entidad-Relación de la BD conectada al *back-end*.

Figura 5-8: Estructura del Servidor del Simulador Empleando Django y Modelo ER de la DB

Capítulo 6

Resultados

Concluida la revisión de lo correspondiente al desarrollo del simulador creado, este capítulo se enfoca en la revisión de los resultados arrojados por el mismo, la metodología para la extracción, el procesamiento y el análisis de los datos.

6.1. Infraestructura que Soporta el Simulador

6.1.1. Descripción de la Infraestructura que Soporta el Simulador

Las simulaciones para su realización en el número requerido tanto de ejecuciones simultáneas como de procesos involucrados se realizaron con la ayuda del Laboratorio Universitario de Cómputo de Alto Rendimiento (LUCAR) del Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas (IIMAS) en un servidor con 48 procesadores y 256 GB de memoria RAM ejecutando el sistema operativo Ubuntu server 19.04.

Para la implantación del simulador desarrollado en el servidor proporcionado se requirió la instalación de los paquetes listados a continuación:

- npm: el sistema de gestión de paquetes por defecto para Node.js.
- mysql: nuestro DBMS (Sistema de Gestión de Bases de Datos Relacional, por sus siglas en inglés) desarrollado bajo licencia dual: Licencia pública general/Licencia comercial por

Oracle Corporation y considerado como la base de datos de código abierto más popular del mundo.

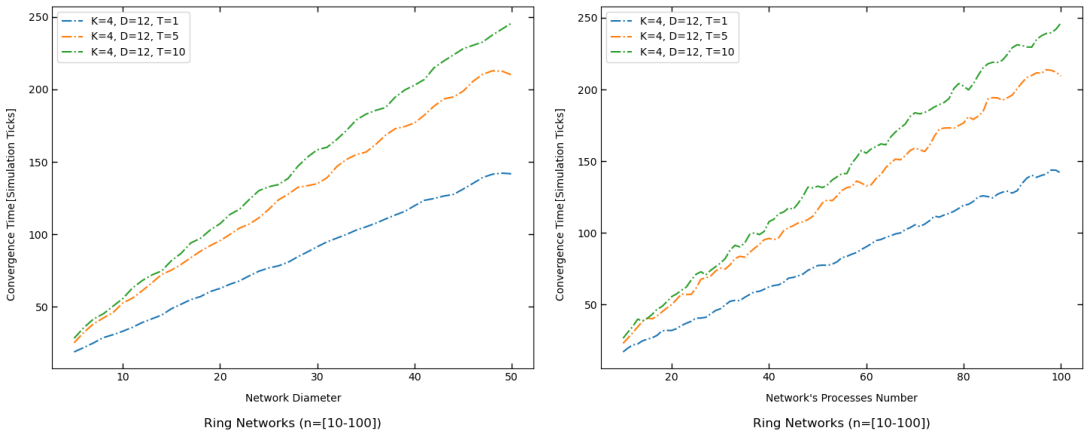
- nodejs: un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.
- python3: lenguaje de programación interpretado.
- python3-pip: un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python.
- python3-venv: una herramienta para crear espacios de trabajo (*workspaces*) para una aplicación de Python.
- httpie: un cliente para HTTP en linea de comandos para consumir la API del simulador.

Con estas utilidades instaladas en el servidor es posible realizar la configuración y despliegue del simulador en su conjunto.

6.2. Descripción de los Experimentos Realizados

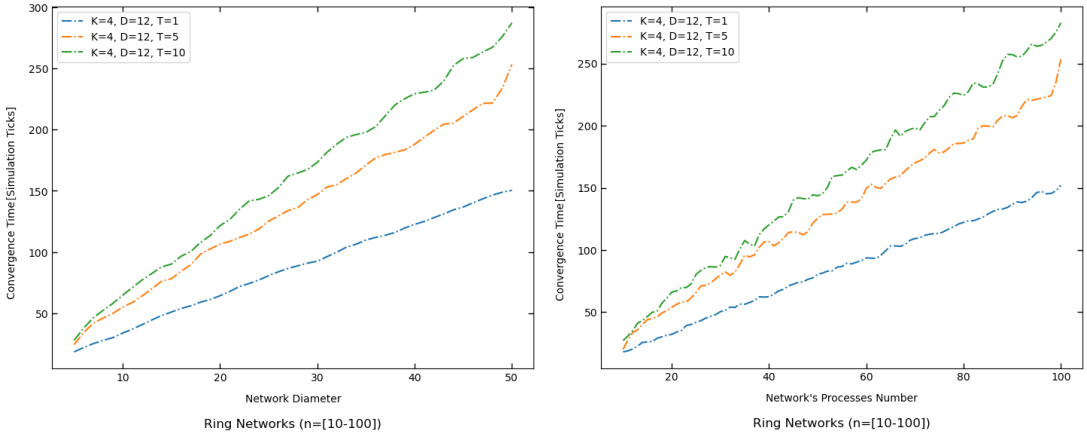
Recordando brevemente que las familias de redes involucradas en los experimentos son anillos de n número de nodos y redes aleatorias regulares[1] (*Random Regular Graphs*). El primer generador de redes, de NetworkX, requiere únicamente un parámetro n donde al seleccionar un entero como parámetro de entrada el generador devuelve un grafo de ciclo con n nodos, para estos experimentos esta n fue incrementada, en intervalos de 1, desde 10 nodos hasta los 400. Para el segundo generador se requieren al menos dos parámetros, d y n, donde d denota el grado de cada nodo y n nuevamente corresponde al número de nodos donde se debe preservar la relación donde el producto nxd debe resultar en un número par, en este grupo de experimentos la d elegida fue 3 iniciando en una n de 100 e incrementándola en 100 hasta alcanzar los 10 000

[1]En teoría de grafos una red regular es aquella en la que cada vértice tiene el mismo número de vecinos. En otras palabras todos los vértices tienen el mismo grado. Adicionalmente, al ser una red aleatoria regular esta familia contiene el espacio de probabilidades de los grafos regulares en n vértices, donde $3 \leq r \leq n$ cuando nr es par. En otras palabras, el componente aleatorio establece el mecanismo con el cual vértices nuevos que deban incorporarse al grafo deben ser agregados manteniendo el grado de cada uno de ellos y los existentes. [34]



(a) Tasa de pérdidas del 1 %

(b) Tasa de pérdidas del 1 %



(c) Tasa de pérdidas del 10 %

(d) Tasa de pérdidas del 10 %

Figura 6-1: Tiempo de Convergencia en Simulaciones de Anillos de Hasta 100 Nodos sin Fallas de Procesos.

nodos, punto desde el cual los incrementos fueron en intervalos de 10 000 hasta alcanzar los 50 000 nodos, este número siendo el límite permitido por los recursos computacionales a nuestra disposición para ejecuciones simultaneas en la infraestructura descrita en la sección anterior, no así, del algoritmo.

Una vez que estos grafos son devueltos por sus generadores correspondientes son utilizados para producir archivos GML (Graph Modeling Language) a partir de los que se construyen los objetos efectivos empleados en la simulación, donde los nodos se vuelven procesos y las aristas se vuelven canales ADD.

6.2.1. Descripción de los Experimentos sin Fallas de los Procesos

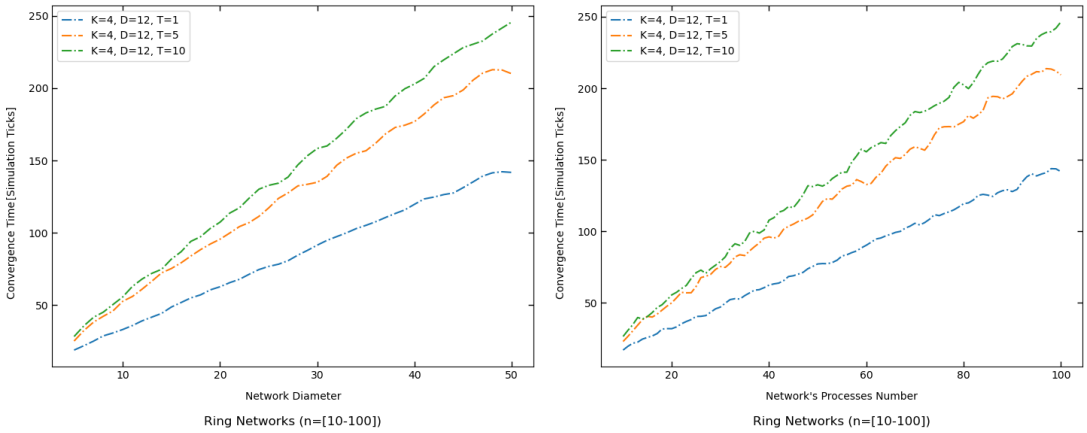
Experimentos en Anillos de n Procesos

Para cada una de las gráficas que describen el comportamiento del tiempo de convergencia para anillos aquí mostradas se realizaron 10 ejecuciones para cada una de las configuraciones. Esto es, al final de las 10 ejecuciones que comparten parámetros idénticos para K, D, T, tasa de pérdidas y red de anillos involucrada, se tomaron los resultados obtenidos para el tiempo de convergencia y se calculó su promedio. Se variaron los parámetros para tres distintos valores del parámetro T (que corresponde a la periodicidad del envío de los mensajes en los procesos) para los valores 1, 5 y 10. En las mismas condiciones se varió la tasa de pérdidas para los valores 1 %, 10 % y 20 % para los casos de redes de hasta 100 nodos, resultando en los 6 gráficos de la Figura 6-1.

Los experimentos de la Figura 6-2 mantiene las mismas condiciones que los experimentos descritos en la Figura 6-1, cambiando únicamente la tasa de mensajes que los canales ADD pueden llegar a perder para los valores de 1 %, 20 % y 99 %, la segunda diferencia es el número de nodos máximo de las redes simuladas alcanzando hasta 400 nodos.

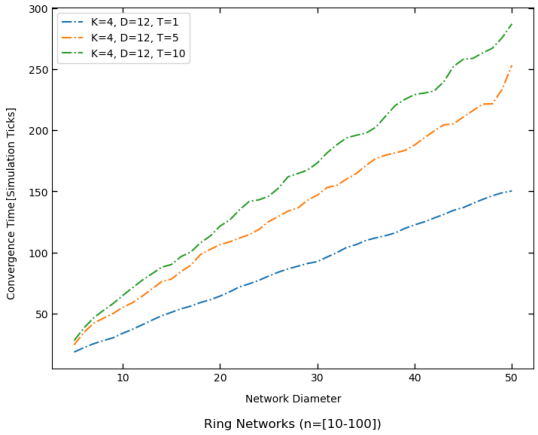
Experimentos en Redes Aleatorias Regulares

Para el caso de las gráficas que describen el comportamiento de redes aleatorias regulares se realizaron 5 ejecuciones que compartían parámetros idénticos para K, D, T, tasa de pérdidas y red involucrada. Nuevamente se varió el parámetro T para valores 1, 5 y 10. Resultando en la Figura 6-4 y Figura 6-5. Estas figuras presentan los resultados de la simulación para el tiempo de convergencia contra el diámetro y el número de nodos respectivamente. En estas gráficas adicionalmente se muestran los modelos de ajuste que fueron obtenidos con ayuda de SciPy, particularmente del paquete Numpy y su método polyfit para el caso de los resultados de diámetro de la red contra tiempo de convergencia, mientras que para el modelo de ajuste para el caso del número de nodos contra tiempo de convergencia se empleó el método de mínimos cuadrados en el caso logarítmico.

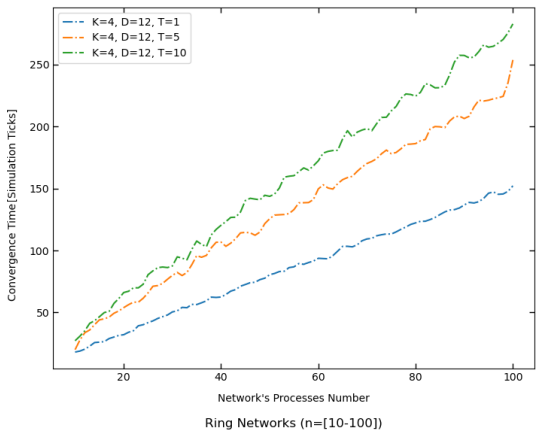


(a) Tasa de pérdidas del 1 %

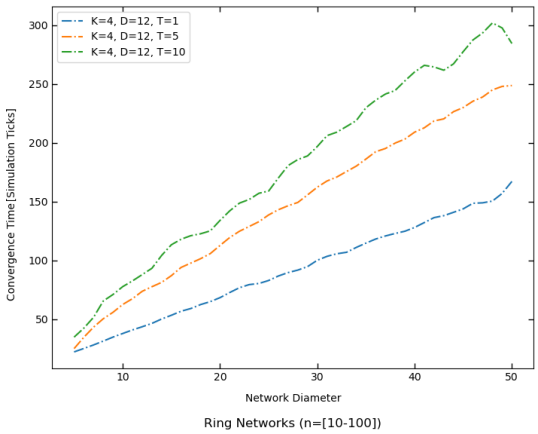
(b) Tasa de pérdidas del 1 %



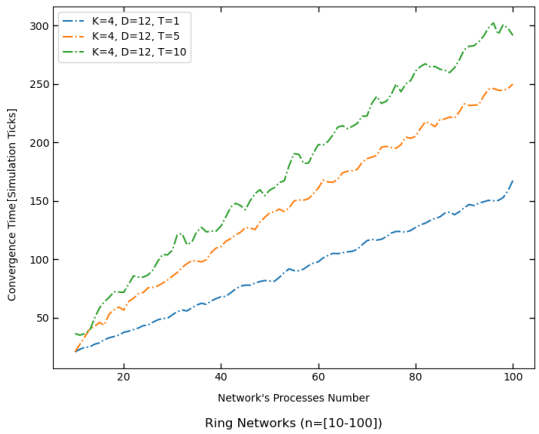
(c) Tasa de pérdidas del 10 %



(d) Tasa de pérdidas del 10 %

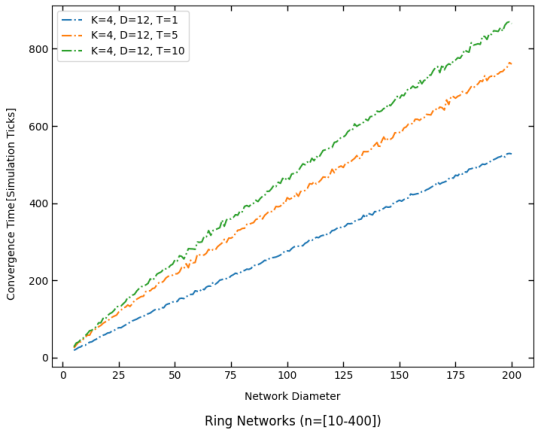


(e) Tasa de pérdidas del 20 %

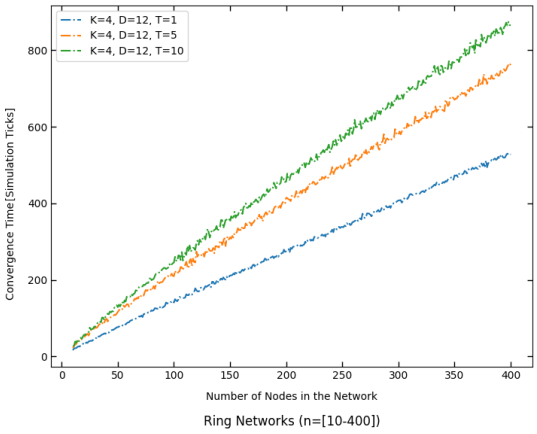


(f) Tasa de pérdidas del 20 %

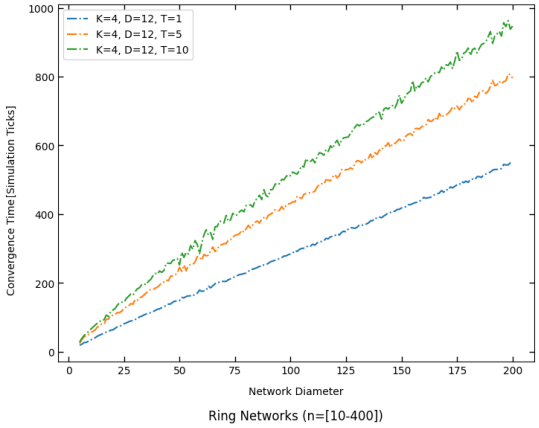
Figura 6-2: Tiempo de Convergencia en Simulaciones de Anillos de Hasta 100 Nodos sin Fallas de Procesos.



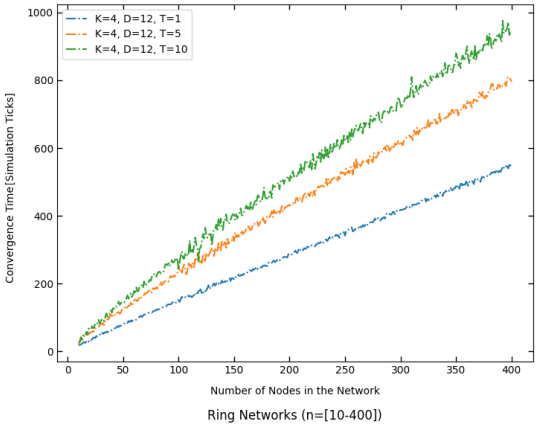
(a) Tasa de pérdidas del 1 %



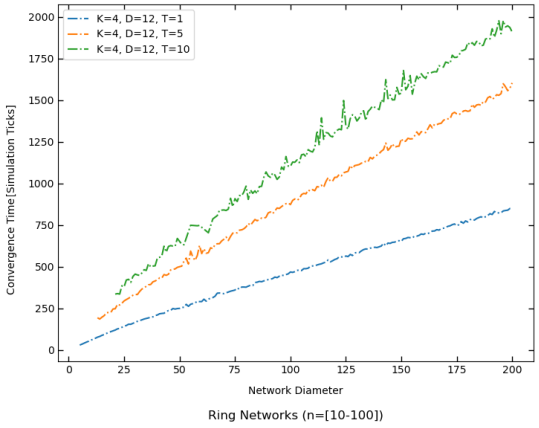
(b) Tasa de pérdidas del 1 %



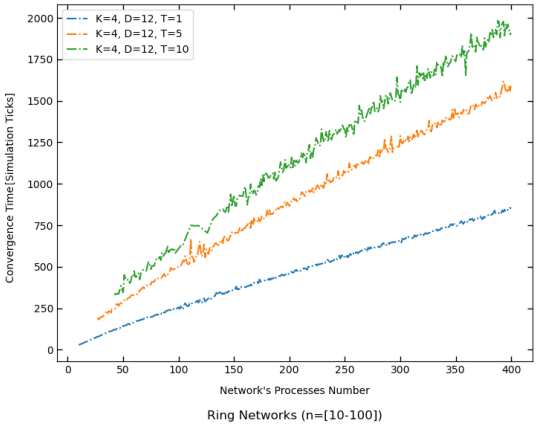
(c) Tasa de pérdidas del 10 %



(d) Tasa de pérdidas del 10 %



(e) Tasa de pérdidas del 99 %



(f) Tasa de pérdidas del 99 %

Figura 6-3: Tiempo de Convergencia en Simulaciones de Anillos de Hasta 400 Nodos sin Fallas de Procesos.

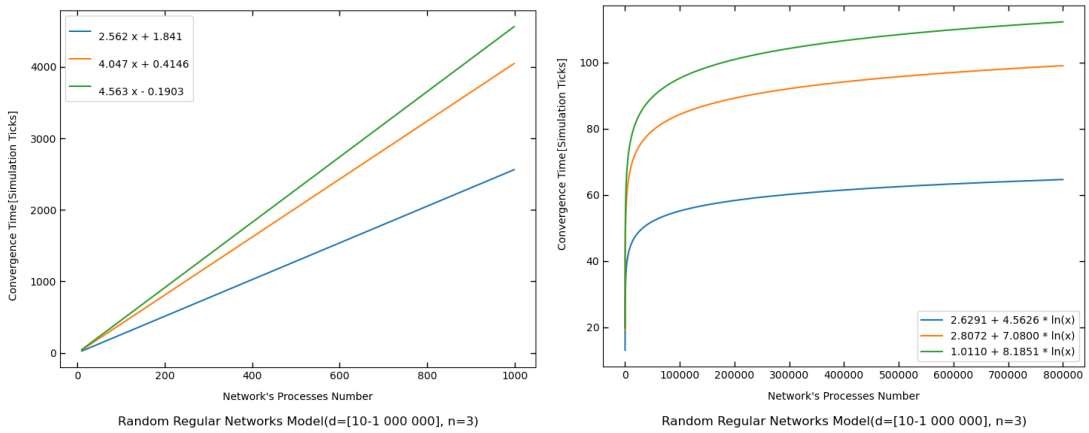
Los modelos obtenidos para simulaciones de redes aleatorias en el caso de diámetro de la red contra el tiempo de convergencia son los siguientes:

- Para T = 1: $2.56x + 1.895$
- Para T = 5: $4.044x + 0.4912$
- Para T = 10: $4.557x - 0.0805$

Nuevamente en el caso de redes aleatorias, pero ahora al enfrentar el número de nodos contra el tiempo de convergencia se obtuvieron los siguientes modelos:

- Para T = 1: $2.629071134692593 + 4.562624683320318 * ln(x)$
- Para T = 5: $2.807055303664013 + 7.080396876308954 * ln(x)$
- Para T = 10: $1.048466931488686 + 8.180601639514453 * ln(x)$

Empleando estos modelos obtenidos de los experimentos se generan los gráficos de la Figura 6-3 para estimar el comportamiento en redes regulares aleatorias para tiempos de convergencia.



(a) Modelo para diámetro de red en redes aleatorias

(b) Modelo para número de nodos en redes aleatorias

Figura 6-4: Estimación de tiempo de convergencia empleando los modelos obtenidos

6.2.2. Descripción de los Experimentos con Fallas de Procesos

Para las simulaciones de anillos en el caso que existen fallas de los procesos, particularmente la falla del proceso que inicialmente se propaga como candidato a líder, se realizaron las simu-

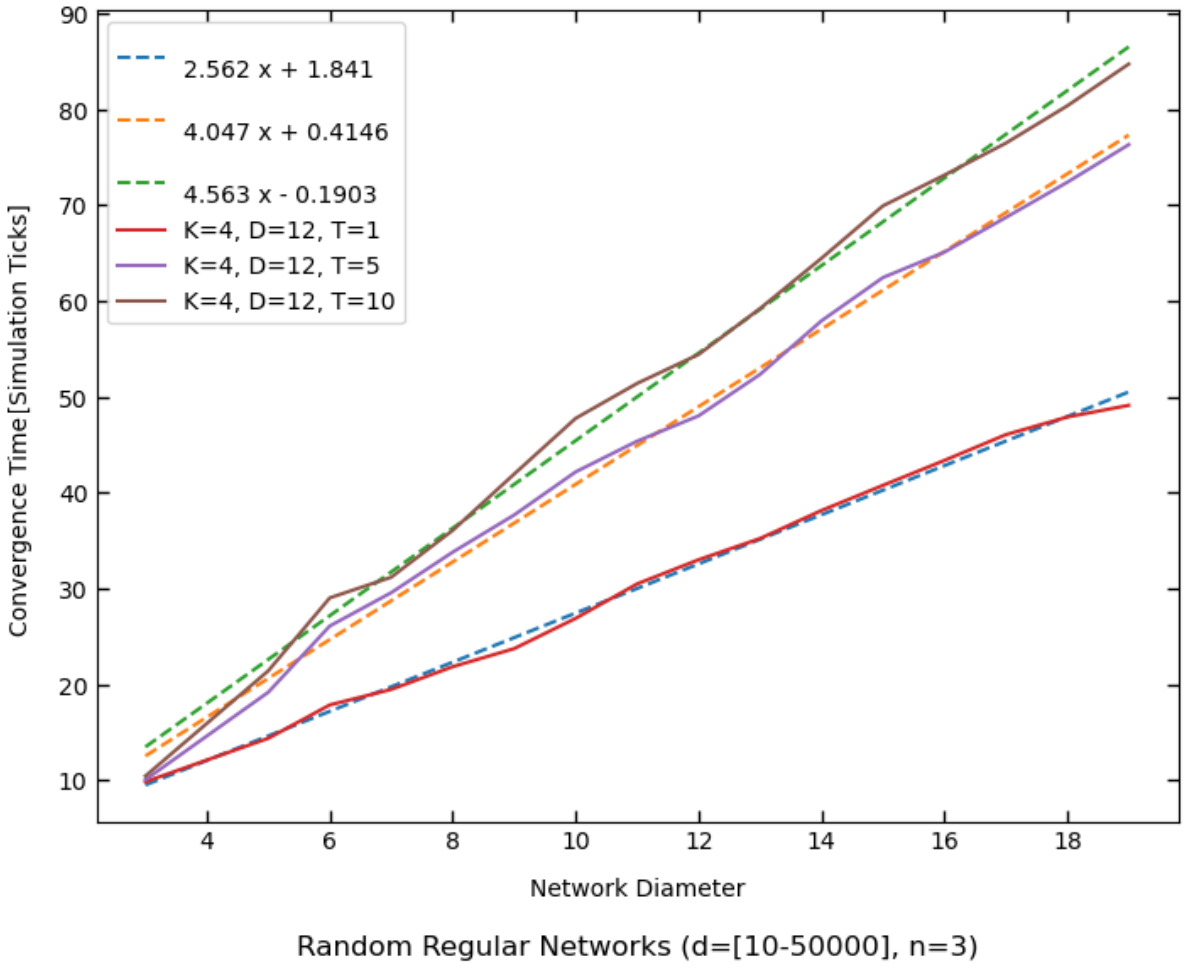


Figura 6-5: Comportamiento de los tiempos de convergencia para redes regulares aleatorias de hasta 50 000 nodos. Tasa de pérdidas del 1 % para distintos valores del Parámetro T

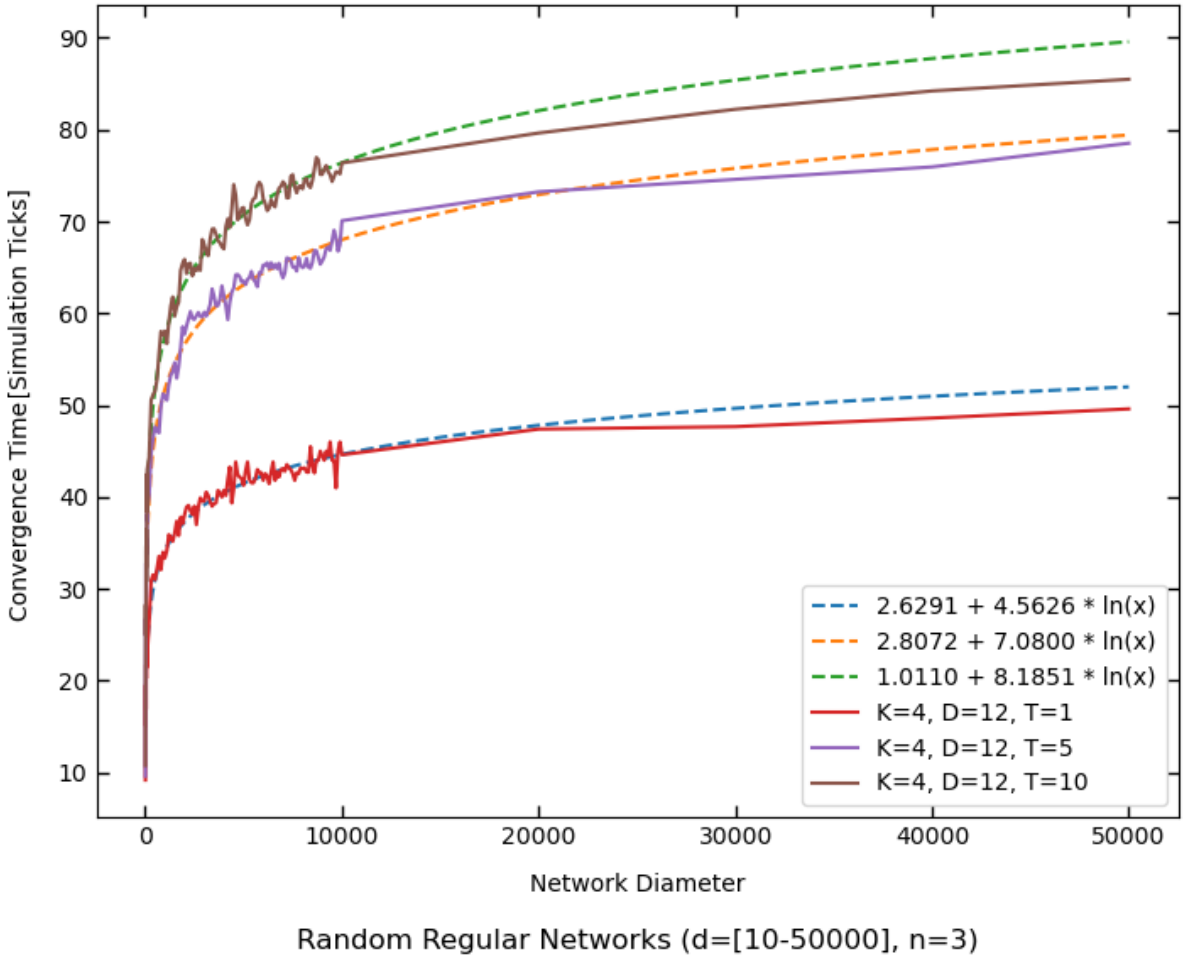


Figura 6-6: Comportamiento de los tiempos de convergencia para redes regulares aleatorias de hasta 50 000 nodos. Tasa de pérdidas del 1 % para distintos valores del Parámetro T

laciones conservando los mismos parámetros que en el caso de los experimentos sin fallas en los procesos, recordando estos parámetros y valores respectivamente tenemos $K = 4, D = 12, T = 1$ y una tasa de pérdidas de los mensajes del 1 %. Los resultados de estas simulaciones se presentan en la Figura 6-6 para la comparativa del diámetro de la red contra el tiempo de convergencia y en la Figura 6-7 para el comportamiento del número de nodos contra el tiempo de convergencia.

En estas simulaciones el escenario inicial es el mismo que para las simulaciones sin fallas de los procesos. El algoritmo inicia en t_0 y continua su ejecución normal hasta el instante promedio en el que ejecuciones previas del simulador habían alcanzado la elección de un líder para la red que actualmente se esta simulando, en las figuras este tiempo se presenta con la curva naranja. Es en este instante en el que el candidato a líder falla que se inicia un temporizador en los procesos que con ayuda de un observador externo permite medir el tiempo promedio que los procesos emplean en desechar el líder que ha fallado y proponerse nuevamente, a ellos mismos, como candidatos a líder iniciando así la nueva búsqueda de un líder. Este tiempo, al que llamamos de detección de la falla, se muestra con la curva en color morado. Por último, el tiempo que el algoritmo emplea en encontrar el proceso correcto con el nuevo identificador menor a partir de ese tiempo de detección se ilustra con la curva en color azul. Nótese el incremento en el tiempo de reconvergencia que la desconexión en la red producto de la falla del proceso que originalmente era propuesto como líder ocasiona.

Con esto concluimos la revisión de los resultados para los experimentos realizados en este trabajo, el lector puede reconocer que detrás de la medida en la que hemos centrado nuestra atención en este trabajo, el tiempo de convergencia y de reconvergencia, dentro de una red y más aun un sistema como el modelado y simulado aquí se pueden extraer mucha información que puede resultar de interés dependiendo de nuestra intención y objeto de estudio (*throughput*, latencia, uso de *bandwidth*, justicia, etc.).

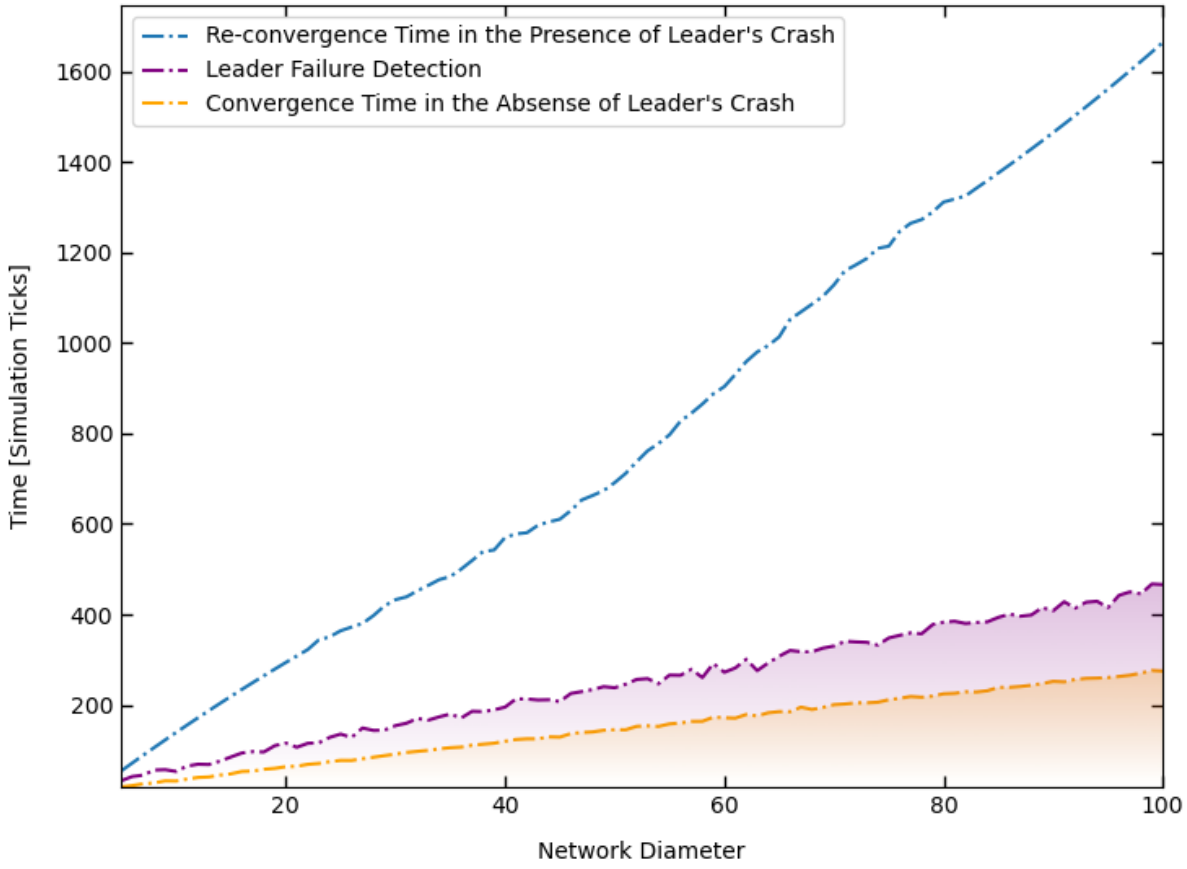


Figura 6-7: Comportamiento de los tiempos de convergencia para anillos de hasta 200 procesos. Tasa de pérdidas del 1 % cuando el líder falla en el instante previo a alcanzar la convergencia por primera vez

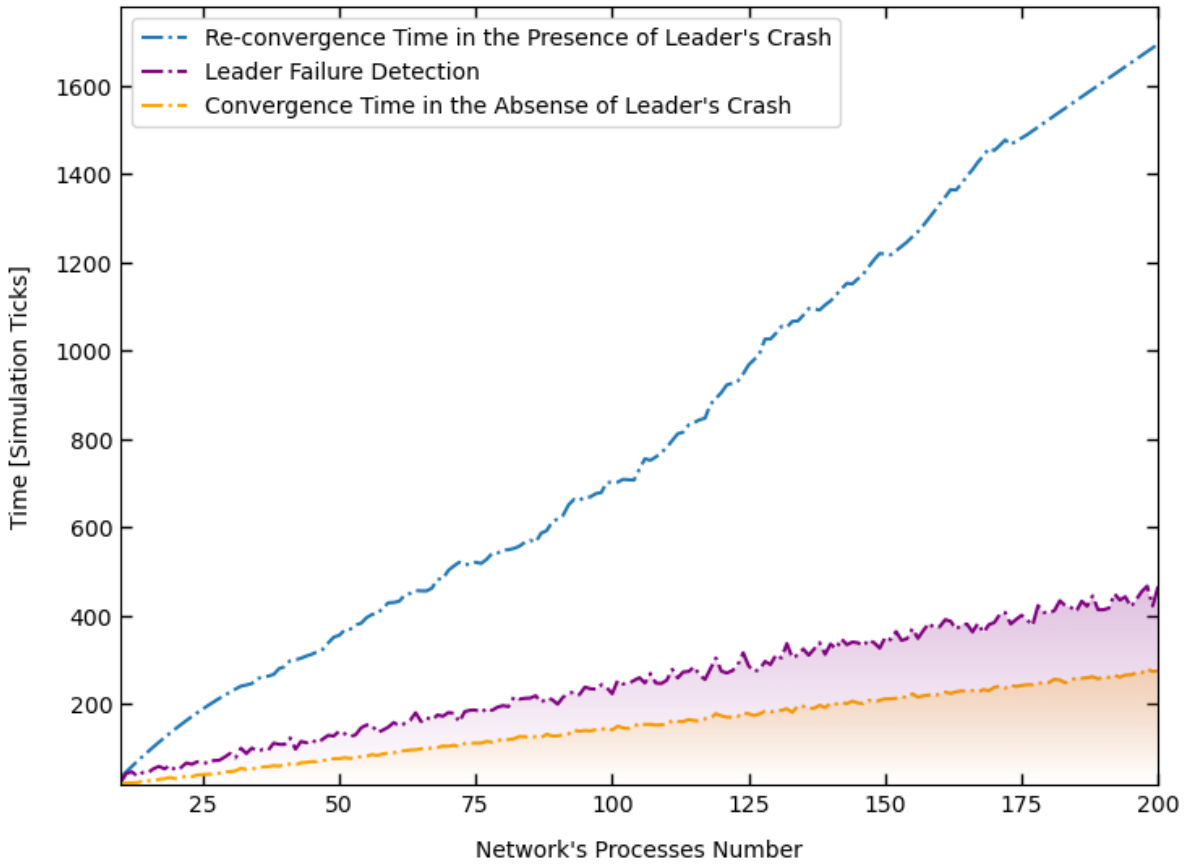


Figura 6-8: Comportamiento de los tiempos de convergencia para anillos de hasta 200 procesos. Tasa de pérdidas del 1 % cuando el líder falla en el instante previo a alcanzar la convergencia por primera vez

Capítulo 7

Conclusiones

El modelo de comunicación ADD, parcialmente síncrono parcialmente *débil*, empleado en el el sistema simulado en este trabajo, en conjunto con procesos dentro de una red susceptibles a fallar presentan un entorno retador para el correcto desempeño de un sistema distribuido. El algoritmo descrito en [29] presenta un avance en la dirección correcta en la disminución del tamaño de los mensajes que son suficientes para lograr un consenso en forma de elección de un líder en una red conectada. Los resultados presentados respaldan un desempeño que permitiría la implementación en sistemas del mundo real, considerando memoria, procesamiento y tiempos de convergencia, incluso ante las situaciones adversas en la comunicación presentadas. Muchas pruebas de desempeño quedan abiertas, entre ellas el impacto del número de mensajes en el canal de comunicación, el procesamiento requerido por un nodo de alguna red que tenga que ejecutar el algoritmo en el mundo físico, aspectos de memoria que se incrementan a medida que un mayor número de nodos se unen a la red. Todos estas líneas permanecen abiertas para ser abordadas en trabajos posteriores.

Se ha validado el funcionamiento correcto del algoritmo de elección de líder que previamente había sido descrito exclusivamente de forma teórica. Se ha caracterizado el comportamiento del algoritmo con detalle suficiente para encontrar modelos que describan el desempeño que se puede esperar en al menos dos familias de redes de interés sin importar el número de nodos o el diámetro que esta red posea.

Adicionalmente y quizás de mayor valor para aquellos interesados en la simulación de sistemas similares, la estructura del simulador desarrollado otorga la flexibilidad de adaptarse a diversos algoritmos que empleen el paso de mensajes para realizar su comunicación. Y cuando menos presenta una posible solución a problemáticas similares mostrando como la integración de tecnologías, paquetes y bibliotecas de software pueden emplearse en conjunto para escenarios de cómputo similares.

Apéndice A

Clases Auxiliares Seleccionadas del Simulador en el *back-end*

Código 3: Clase Principal de la Simulación - simulation.py

```python
"""
    Universidad Nacional Autõnoma de Mẽxico
    Posgrado en Ciencias e Ingenieria en Computaciõn
    Simulador de Algoritmo de Eleccion de Líder (Sergio Rajsbaum, Karla Vargas)

    Autor: Carlos Lõpez
"""

import networkx as nx
import simpy
import logging
import time

import matplotlib.pyplot as plt
from io import BytesIO
import base64

from simulador.models import Configuracion, Resultado
from django.db.models import Avg, Max, Min, Sum
from simulador.src.describe_network import DescribeNetwork
from simulador.src.simulation_params import SimulationParams
from simulador.src.leader_election_simulator import LeaderElectionSimulator
from simulador.src.metrics import Metrics

log = logging.getLogger(__name__)


class ActualSimulator:
```

```
29
30      def __init__(self, config, bandera=False):
31          self.config = config
32          self.bandera_single = bandera
33
34      def inicia_simulador(self):
35          if self.config['verbose']:
36              logging.basicConfig(format='%(asctime)s %(message)s - %(filename)s', level=
                    logging.INFO)
37          else:
38              logging.basicConfig(format='%(asctime)s %(message)s - %(filename)s', level=
                    logging.WARN)
39
40          # Si se pasa como argumento una red descrita en un archivo .gml, se procede a
                parsearla directamente
41          if self.config['red_empleada'] is not None:
42              log.info('Inicia Parseo de la Red Distribuida')
43              # Creaciõn de la red mediante la definiciõn del archivo proporcionado
44              graph = nx.read_gml('{}'.format(self.config['red_empleada']))
45
46          log.info('Creaciõn de ambiente de SimPy')
47          # Creaciõn del ambiente de SimPy
48          env = simpy.Environment()
49
50          # Modulo que genera el reporte de la simulaciõn y las metricas
51          metricas = Metrics(self.config)
52
53          tiempo_promedio = Resultado.objects.filter(configuracion=self.config['id']).exclude
                (t_conv='0').aggregate(Avg('t_conv'))
54
55          if self.config['falla'] == 'True':
56              try:
57                  self.config['tiempo_promedio_convergencia'] = int(tiempo_promedio['
                        t_conv__avg'])
58              except TypeError:
59                  return "Error"
60          else:
61              self.config['tiempo_promedio_convergencia'] = 0
62
63          nx_graph = DescribeNetwork(env, graph, self.config, metricas)
64          red = nx_graph.construye_red(self.config)
65
66          if 'random' in self.config['red_empleada']:
67              if self.config['param_T'] == 1:
68                  duracion = 55
69              else:
70                  duracion = 55 * self.config['param_T']
71          elif 'anillo' in self.config['red_empleada']:
72              if self.config['falla'] == 'True':
73                  duracion = red.number_of_nodes() * 3 * self.config['param_T'] * 4
74              else:
75                  duracion = red.number_of_nodes() * 3 * self.config['param_T']
76          elif 'barabasi' in self.config['red_empleada']:
```

```
77                 if self.config['param_T'] == 1:
78                     duracion = 25
79                 else:
80                     duracion = 25 * self.config['param_T']
81 
82             if self.bandera_single:
83                 nx.draw_circular(red, with_labels=True)
84                 figfile = BytesIO()
85                 plt.savefig(figfile, transparent=True, format='png')
86                 figfile.seek(0)
87                 figdata_png = base64.b64encode(figfile.getvalue())
88                 Configuracion.objects.filter(id=self.config['id']).update(grafo_red=figdata_png
                        )
89 
90             plt.clf()
91 
92             # Creaciõn de los Parametros de la Simulaciõn
93             params = SimulationParams(red, self.config)
94             print("Diametro de la red: {}".format(nx.algorithms.diameter(red)))
95             Configuracion.objects.filter(id=self.config['id']).update(
96                                                                 num_nodos=red.
                                                                     number_of_nodes(),
97                                                                 diametro=nx.algorithms.
                                                                     diameter(red))
98 
99             # Creaciõn del objeto LeaderElectionSimulator, recibe ambiente de Simpy y los pará
                    metros
100             simulador = LeaderElectionSimulator(env, params, metricas)
101 
102             tiempo_inicio = time.time()
103             # Inicia la simulacion y ejecuta la duracion especificada
104             simulador.start()
105 
106             print("Se simulará hasta el instante {}".format(duracion))
107             env.run(until=duracion)
108 
109             # Tiempo de finalizacion
110             tiempo_fin = time.time()
111 
112             metricas.get_metricas()
113             metricas.set_t_run(tiempo_inicio, tiempo_fin)
114 
115             metricas.exporta_resultados(red.number_of_nodes())
```

Código 4: Creación de la Red - descibe_network.py

```
1  """
2      Crea la red con los nodos como Procesos que ejecutan el Algoritmo de Eleccion de Lider
3      y las aristas como Canales ADD. Lo anterior respeta la red descrita en un archivo GML.
4      Autor: Carlos Lõpez
5  """
6 
7  import networkx as nx
```

```
8  import logging
9
10 from simulador.src.add_channel import ADDChannel
11 from simulador.src.process import Process
12
13 log = logging.getLogger(__name__)
14
15
16 class DescribeNetwork:
17
18     def __init__(self, env, graph, config, metricas):
19         self.env = env
20         self.config = config
21         self.graph = graph
22         self.metricas = metricas
23
24     def construye_red(self, args):
25         red = nx.DiGraph()
26         nodos = []
27         nodos_cnt = 0
28
29         for _ in self.graph.nodes():
30             proceso = Process(self.env, nodos_cnt, self.graph.number_of_nodes(), args, self
                   .metricas, self.config)
31             nodos.append(proceso)
32             red.add_node(nodos[nodos_cnt])
33             nodos_cnt += 1
34
35         for e in self.graph.edges():
36             source = nodos[int(e[0])]
37             target = nodos[int(e[1])]
38             canal_1 = ADDChannel(self.env, self.config, source, target, self.metricas, args
                   )
39             canal_2 = ADDChannel(self.env, self.config, target, source, self.metricas, args
                   )
40             red.add_edge(target, source, object=canal_1)
41             red.add_edge(source, target, object=canal_2)
42         return red
```

Código 5: Creación de Procesos Simpy- leader_election_simulator.py

```
1  """
2      Clase LeaderElection
3      Esta clase describe el comportamiento descrito por el algoritmo (Eventual Leader
4      Election ADD Model) para el inicio del algoritmo de la simulacion.
5  """
6
7  import logging
8  import simpy
9
10 from simulador.src.metrics import Metrics
11 from simulador.src.simulation_params import SimulationParams
12
```

```
log = logging.getLogger(__name__)


class LeaderElectionSimulator:
    def __init__(self, env: simpy.Environment, params: SimulationParams, metricas: Metrics):
        self.env = env
        self.params = params
        self.metricas = metricas
        self.contador_convergencia = 0

    def start(self):
        """
        Inicia la simulaciõn.
        """
        log.info("Iniciando la simulacion")
        # Generacion de Mensajes alive en todos los nodos y envio a sus vecinos por los canales correspondientes
        for nodo in self.params.network.nodes():
            neighbors = list(self.params.network.neighbors(nodo))
            for neighbor in neighbors:
                canal = self.params.network.get_edge_data(neighbor, nodo)['object']
                self.env.process(nodo.send_alive(canal, self.params))
                self.env.process(neighbor.on_alive(canal))
```

Código 6: Clase Mensaje - message.py

```
"""
    Abstraccion de los mensajes que se comunican los nodos con la finalidad de elegir un lider
    id_nodo: Identificador del nodo que genera el mensaje

    Autor: Carlos Lõpez
"""


class Message:

    def __init__(self, lider, hopbound):
        self.lider = lider
        self.hopbound = hopbound

    def __str__(self):
        return "(alive: (Lider {}, Hopbound {}))".format(self.lider, self.hopbound)
```

Código 7: Clase de Métricas- metrics.py

```
"""
    Modulo encargado de realizar el reporte de los datos que genera la simulaciõn

    Autor: Carlos Lõpez
"""

```

```
7  import logging
8
9  from simulador.models import Resultado
10
11 log = logging.getLogger(__name__)
12
13
14 class Metrics:
15     def __init__(self, args):
16         self.metricas = {}
17         self.reset_metricas()
18         self.args = args
19
20     def reset_metricas(self):
21         """Inicializa todas las mẽtricas"""
22         # Mensajes generados
23         self.metricas['m_gen'] = 0
24         # Mensajes enviado
25         self.metricas['m_env'] = 0
26         # Mensaje Entregado
27         self.metricas['m_ent'] = 0.0
28         # Mensajes procesados
29         self.metricas['m_proc'] = 0
30         # Transmisiones recibidas en limite de ventana de entrega
31         self.metricas['m_vent_lim'] = 0
32         # Paquetes perdidos por el canal
33         self.metricas['m_perd'] = 0
34         # Retraso promedio en los canales
35         self.metricas['ret_prom'] = 0.0
36         # Tiempo de convergencia de desde el estado inicial (medido en ticks de simulaciõn)
37         self.metricas['t_conv'] = 0
38         # Tiempo de convergencia de la red despuẽs de que el lider deja de responder
39         self.metricas['t_reconv'] = 0.0
40         # Tiempo de ejecuciõn medido por el sistema operativo
41         self.metricas['t_run'] = 0.0
42         # Numero de retrasos generados por el canal
43         self.metricas['ret_gen'] = 0
44         # Tiempo que los procesos tardan en darse cuenta que el lider elegido ha fallado
45         self.metricas['t_detect'] = 0.0
46         # Tiempo en el que el lider falla
47         self.metricas['t_falla'] = 0.0
48         # Retraso promedio generado por el canal
49         self.metricas['ret_prom'] = 0.0
50
51         self.metricas['exe_num'] = 0
52
53         self.metricas['lideres'] = 0
54
55         self.metricas['nuevos_lideres'] = 0
56
57     # Llamada cuando un mensaje ha sido generado y enviado al canal
58     def mensaje_generado(self):
59         self.metricas['m_gen'] += 1
```

```
60
61      # Llamada cuando un mensaje ha sido generado y enviado al canal
62      def mensaje_enviado(self):
63          self.metricas['m_env'] += 1
64
65      # Llamada cuando un mensaje ha sido generado y enviado al canal
66      def mensaje_entregado(self):
67          self.metricas['m_ent'] += 1
68
69      def mensaje_enviado_limite(self):
70          self.metricas['m_vent_lim'] += 1
71
72      def mensaje_perdido(self):
73          self.metricas['m_perd'] += 1
74
75      # Llamada cuando un mensaje ha sido recibido y procesado en el receptor
76      def mensaje_procesado(self):
77          self.metricas['m_proc'] += 1
78
79      def set_t_run(self, tiempo_inicio, tiempo_fin):
80          self.metricas['t_run'] = tiempo_fin - tiempo_inicio
81
82      def tiempo_deteccion(self, tiempo_actual):
83          self.metricas['t_detect'] = tiempo_actual
84
85      def calc_ret_prom(self):
86          if self.metricas['ret_gen'] > 0:
87              self.metricas['ret_prom'] \
88                  = self.metricas['retraso_total_generado'] / self.metricas['ret_gen']
89          else:
90              self.metricas['ret_prom'] = 0
91
92      def set_t_conv(self, ultimo_cambio_lider):
93          self.metricas['t_conv'] = ultimo_cambio_lider
94
95      def set_t_reconv(self, tiempo_reconv):
96          self.metricas['t_reconv'] = tiempo_reconv
97
98      def set_t_falla(self, tiempo_falla):
99          self.metricas['t_falla'] = tiempo_falla
100
101     def get_num_lideres(self):
102         return self.metricas['lideres']
103
104     def get_nuevo_num_lideres(self):
105         return self.metricas['nuevos_lideres']
106
107     def set_t_deteccion(self, tiempo_falla_lider, tiempo_regreso_lider, t_conv):
108         if t_conv != 0:
109             self.metricas['t_detect'] = tiempo_regreso_lider - t_conv
110         else:
111             self.metricas['t_detect'] = tiempo_regreso_lider - tiempo_falla_lider
112
```

```
113     def nodos_lider_encontrado(self):
114         self.metricas['lideres'] += 1
115 
116     def nodos_nuevo_lider_encontrado(self):
117         self.metricas['nuevos_lideres'] += 1
118 
119     def get_metricas(self):
120         self.calc_ret_prom()
121         self.calc_m_gen()
122         return self.metricas
123 
124     def get_t_conv(self):
125         return self.metricas['t_conv']
126 
127     def calc_m_gen(self):
128         self.metricas['m_gen'] = self.metricas['m_env'] + self.metricas['m_perd']
129 
130     def acumula_t_deteccion(self, t_conv_promedio, t_deteccion):
131         self.metricas['t_detect'] += (t_deteccion - t_conv_promedio)
132 
133     def exporta_resultados(self, num_nodos):
134         resultado = Resultado(
135             m_env=self.metricas['m_env'],
136             m_perd=self.metricas['m_perd'],
137             m_proc=self.metricas['m_proc'],
138             m_ent=self.metricas['m_ent'],
139             m_vent_lim=self.metricas['m_vent_lim'],
140             ret_gen=self.metricas['ret_gen'],
141             t_run=self.metricas['t_run'],
142             t_conv=self.metricas['t_conv'],
143             t_reconv=self.metricas['t_reconv'],
144             t_falla=self.metricas['t_falla'],
145             t_detect= self.metricas['t_detect'] / num_nodos,
146             configuracion_id=self.args['id']
147         )
148         resultado.save()
```

Código 8: Conector de la Simulación *front-end back-end* conector_simulacion.py

```
1  import json
2  import os
3  from concurrent.futures.process import ProcessPoolExecutor
4 
5  import django
6  from django.core import serializers
7  from django.db import transaction, connection
8 
9  from simulador.models import Configuracion, Resultado
10 from simulador.src.simulation import ActualSimulator
11 
12 
```

```
13  def subprocess_setup():
14      django.setup()
15
16
17  def paraleliza_ejecuciones(configuracion):
18      simulacion_real = ActualSimulator(configuracion)
19      simulacion_real.inicia_simulador()
20
21
22  class ConectorSimulacion:
23
24      def __init__(self, configuracion_base):
25          self.archivos = []
26          self.configuraciones = []
27          self.configuracion_base = configuracion_base
28
29      def inicia_single(self):
30
31          configuracion = {'param_K': int(self.configuracion_base['param_K']),
32                           'param_D': int(self.configuracion_base['param_D']),
33                           'param_T': int(self.configuracion_base['param_T']),
34                           'drop_rate': float(self.configuracion_base['drop_rate']),
35                           'red_empleada': self.configuracion_base['red_empleada'],
36                           'verbose': False,
37                           'id': int(self.configuracion_base['id']),
38                           'falla': self.configuracion_base['falla'],
39                           }
40
41          simulacion_real = ActualSimulator(configuracion, True)
42          simulacion_real.inicia_simulador()
43
44          resultados = Resultado.objects.filter(configuracion_id=self.configuracion_base['id'
                ])
45          qs_string = serializers.serialize('json', resultados)
46
47          qs_json = json.loads(qs_string)
48
49          for resultado in qs_json:
50              del resultado['pk']
51              del resultado['model']
52
53          return qs_json
54
55      def inicia_bulk(self):
56
57          with ProcessPoolExecutor(max_workers=40, initializer=subprocess_setup) as executor:
58              django.db.connections.close_all()
59              executor.map(paraleliza_ejecuciones, self.configuraciones)
60
61          res = [sub['id'] for sub in self.configuraciones]
62          resultados = Resultado.objects.select_related('configuracion').filter(
                configuracion_id__in=res)
63
```

```
64          lista_resultados = []
65
66          for resultado in resultados:
67              lista_resultados.append({'m_env': resultado.m_env,
68                                       'm_perd': resultado.m_perd,
69                                       'm_ent': resultado.m_ent,
70                                       'm_proc': resultado.m_proc,
71                                       'm_vent_lim': resultado.m_vent_lim,
72                                       'ret_gen': resultado.ret_gen,
73                                       't_run': resultado.t_run,
74                                       't_conv': resultado.t_conv,
75                                       't_reconv': resultado.t_reconv,
76                                       'configuracion_id': resultado.configuracion_id,
77                                       'numero_nodos': resultado.configuracion.num_nodos,
78                                       'red_empleada': resultado.configuracion.red_empleada,
79                                       })
80          return lista_resultados
81
82      def save_configuraciones_bulk(self):
83          for archivo in self.archivos:
84              posible_configuracion_existente = Configuracion.objects.filter(
85                  param_K=int(self.configuracion_base['param_K']),
86                  param_D=int(self.configuracion_base['param_D']),
87                  param_T=int(self.configuracion_base['param_T']),
88                  drop_rate=float(self.configuracion_base[
89                                  'drop_rate']),
90                  red_empleada=archivo)
91
92              existe = posible_configuracion_existente.count()
93
94              if existe == 0:
95                  configuracion = Configuracion(param_K=self.configuracion_base['param_K'],
96                                                param_D=self.configuracion_base['param_D'],
97                                                param_T=self.configuracion_base['param_T'],
98                                                drop_rate=self.configuracion_base['drop_rate'
                                                      ],
99                                                red_empleada=archivo,
100                                               )
101                 configuracion.save()
102                 self.configuraciones.append({
103                     'param_K': int(self.configuracion_base['param_K']),
104                     'param_D': int(self.configuracion_base['param_D']),
105                     'param_T': int(self.configuracion_base['param_T']),
106                     'drop_rate': float(self.configuracion_base['drop_rate']),
107                     'red_empleada': archivo,
108                     'verbose': False,
109                     'id': configuracion.pk,
110                     'falla': self.configuracion_base['falla'],
111                 }
112                 )
113             else:
114                 self.configuraciones.append({
115                     'param_K': int(self.configuracion_base['param_K']),
```

```
                    'param_D': int(self.configuracion_base['param_D']),
                    'param_T': int(self.configuracion_base['param_T']),
                    'drop_rate': float(self.configuracion_base['drop_rate']),
                    'red_empleada': archivo,
                    'verbose': False,
                    'id': posible_configuracion_existente[0].pk,
                    'falla': self.configuracion_base['falla']
                })

    def get_nombres_archivos(self):
        path = os.path.abspath(os.path.dirname(__file__))

        if os.name == 'nt':
            path += "\\..\\params\\networks\\"
        else:
            path += "/../params/networks/"

        for r, d, f in os.walk(path):
            for file in f:
                if '.gml' and 'anillo' in file:
                    self.archivos.append(os.path.join(r, file))
        self.archivos.sort()

    def get_nombres_archivos_aleatorias(self):
        path = os.path.abspath(os.path.dirname(__file__))

        if os.name == 'nt':
            path += "\\..\\params\\networks\\"
        else:
            path += "/../params/networks/"

        for r, d, f in os.walk(path):
            for file in f:
                if '.gml' and 'random' in file:
                    self.archivos.append(os.path.join(r, file))
        self.archivos.sort()

    def get_nombres_archivos_barabasi(self):
        path = os.path.abspath(os.path.dirname(__file__))

        if os.name == 'nt':
            path += "\\..\\params\\networks\\"
        else:
            path += "/../params/networks/"

        for r, d, f in os.walk(path):
            for file in f:
                if '.gml' and 'barabasi' in file:
                    self.archivos.append(os.path.join(r, file))
        self.archivos.sort()
```

Apéndice B

Código de Componentes Seleccionados del *front-end*

Código 1: Servicio de consumo de Simulación del *front-end* al *back-end*

simulacion.service.ts

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { ConfiguracionService } from 'app/_services/configuracion.service';
import {MatDialog} from '@angular/material/dialog'
import { Configuracion } from 'app/_models/configuracion';
import { first } from 'rxjs/operators';
import { SimulacionService } from 'app/_services/simulacion.service';
import Swal from 'sweetalert2';
import { Resultado } from 'app/_models/resultado';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/forkJoin';
import { compileNgModule } from '@angular/compiler';

@Component({
  selector: 'app-simulacion',
  templateUrl: './simulacion.component.html',
  styleUrls: ['./simulacion.component.css']
})
export class SimulacionComponent implements OnInit {

  grafo: any
  loading = false
  simula_uno = false
  simula_muchos = false
  muestra_botones = false
```

```
27
28    configuraciones: Configuracion[]
29    resultados_promediados = []
30
31    bulkFormGroup: FormGroup;
32    singleFormGroup: FormGroup;
33
34    opciones = [
35      { indice: 1, accion: "Simular Configuracion Particular Registrada" },
36      { indice: 2, accion: "Definir Configuración y Simular Anillos N = [10-1000]" },
37      { indice: 3, accion: "Definir Configuración y Simular Redes Aleatorias  N = [10-1000]"
            },
38      { indice: 4, accion: "Definir Configuración y Simular Redes de Barabasi  N = [10-1000]"
            }
39    ]
40
41    tipos_redes = [
42      { indice: 1, nombre: "Anillos"},
43      { indice: 2, nombre: "Aleatorios"},
44      { indice: 3, nombre: "Barabasi"}
45    ]
46
47    variables = [
48      { indice: 1, nombre: "Drop Rate" },
49      { indice: 2, nombre: "Parametro T" }
50    ]
51
52    resultado_simulacion_single: Object;
53    resultado_simulacion_bulk: [];
54
55    resultados: Resultado[]
56
57    sinResultados = true;
58    simulacion_single_terminada = false;
59    simulacion_bulk_terminada = false;
60    loading_configuraciones = false;
61    busqueda_terminada = false;
62
63    fecha_actual: number
64    busquedaDropFormGroup: FormGroup;
65    busquedaParamTFormGroup: FormGroup;
66    muestra_botones_busqueda_param_t: boolean;
67    muestra_botones_busqueda_drop: boolean;
68    busqueda_drop: boolean;
69    busqueda_param_T: boolean;
70
71    titulo_grafica: string
72    tipo_series: string
73    valores_drop_unicos: number[];
74    valores_t_unicos: number[];
75
76    tipo_red: string;
77
```

```
78    resultados_t_conv_clasificiacion: any[];
79    regresiones: any[];
80    pendientes: any[];
81    ordenadas: any[];
82    opcion: number;
83
84    // Miembros para graficaciõn
85    legend: boolean = true;
86    legendPosition: string = "below";
87    showLabels: boolean = true;
88    animations: boolean = true;
89    xAxis: boolean = true;
90    yAxis: boolean = true;
91    showYAxisLabel: boolean = true;
92    showXAxisLabel: boolean = true;
93    xAxisLabel: string;
94    yAxisLabel: string;
95    yScaleMax: number;
96    nombre_series: string = "Series";
97
98    colorScheme = {
99      domain: ['#BCE784', '#5DD39E', '#348AA7', '#525174', '#513B56', '#1CFEBA','#95F2D9', '#
          B8CDF8', '#9D8DF1', '#41463D']
100   };
101
102   datos: any[];
103   etiquetas: number[];
104   series: any[];
105
106
107   constructor(
108     private formBuilder: FormBuilder,
109     private configuracionService: ConfiguracionService,
110     private simulacionService: SimulacionService,
111     public dialog: MatDialog) { }
112
113   ngOnInit(){
114     this.bulkFormGroup = this.formBuilder.group({
115       param_K: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
116       param_D: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
117       param_T: ['', [Validators.required, Validators.min(1), Validators.max(100)],],
118       drop_rate: ['', [Validators.required, Validators.min(0), Validators.max(0.9999)]],
119     });
120
121     this.busquedaDropFormGroup = this.formBuilder.group({
122       param_K: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
123       param_D: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
124       param_T: ['', [Validators.required, Validators.min(1), Validators.max(100)],],
125     });
126
127     this.busquedaParamTFormGroup = this.formBuilder.group({
128       param_K: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
129       param_D: ['', [Validators.required, Validators.min(1), Validators.max(100)]],
```

```
130         drop_rate: ['', [Validators.required, Validators.min(0), Validators.max(0.9999)],],
131       });
132
133       this.singleFormGroup = this.formBuilder.group({
134         configuracion: ['', Validators.required],
135       })
136     }
137
138
139     cambia_accion(opcion: number){
140
141       this.opcion = opcion
142       // Una configuración
143       if (opcion == 1){
144         this.loading_configuraciones = true
145          this.configuracionService.getConfiguraciones().pipe(first())
146           .subscribe(
147             response => {
148               this.configuraciones = response
149               this.busqueda_terminada = false
150               this.simula_uno = true
151               this.simula_muchos = false
152               this.loading_configuraciones = false
153             },
154             error => {
155               console.error(error);
156               this.loading_configuraciones = false
157             }
158           )
159           return
160       }
161       // Muchas configuraciones
162       if (opcion >= 2){
163         this.simula_uno = false
164         this.muestra_botones = false
165         this.simula_muchos = true
166         return
167       }
168     }
169
170     cambia_tipo_red(tipo_red: string){
171       // Anillos
172       if (tipo_red == 'Anillos'){
173         this.tipo_red = tipo_red
174         return
175       }
176       // Aleatorios
177       if (tipo_red == 'Aleatorios'){
178         this.tipo_red = tipo_red
179         return
180       }
181     }
182
```

```
183
184   cambia_variable(variable: number){
185     // Varia Drop Rate
186     if (variable == 1){
187       this.busqueda_drop = true
188       this.busqueda_param_T = false
189       return
190     }
191     // Varia Parametro T
192     if (variable == 2){
193       this.busqueda_drop = false
194       this.busqueda_param_T = true
195       return
196     }
197   }
198
199   verifica(){
200     if(this.bulkFormGroup.invalid){
201       this.muestra_botones = false
202     }else{
203       this.muestra_botones = true
204     }
205   }
206
207   verificaBusquedaDrop(){
208     if(this.busquedaDropFormGroup.invalid){
209       this.muestra_botones_busqueda_drop = false
210     }else{
211       this.muestra_botones_busqueda_drop = true
212     }
213   }
214
215   verificaBusquedaT(){
216     if(this.busquedaParamTFormGroup.invalid){
217       this.muestra_botones_busqueda_param_t = false
218     }else{
219       this.muestra_botones_busqueda_param_t = true
220     }
221   }
222
223   busca_resultados_drop(){
224     this.loading = true;
225     this.sinResultados = false
226     this.simulacion_bulk_terminada = false
227     this.simulacion_single_terminada = false
228     let parametros = []
229     const controls = this.busquedaDropFormGroup.controls;
230     for (const name in controls) {
231       parametros.push(name, controls[name].value);
232     }
233     this.simulacionService.busca_resultados_drop(parametros[1], parametros[3], parametros
          [5], this.tipo_red)
234     .pipe(first())
```

```
235       .subscribe(
236         res => {
237           this.loading = false
238           this.grafica_resultados_drop(res)
239         },
240         error => {
241           Swal.fire(
242             'Error',
243             'Algo ha salido mal ' +  + '[' + error.status + ']' + error.error,
244             'error'
245           )
246           this.loading = false
247         }
248       )
249     }
250
251     busca_resultados_param_t(){
252       this.loading = true;
253       this.sinResultados = false
254       this.simulacion_bulk_terminada = false
255       this.simulacion_single_terminada = false
256       let parametros = []
257       const controls = this.busquedaParamTFormGroup.controls;
258       for (const name in controls) {
259         parametros.push(name, controls[name].value);
260       }
261       this.simulacionService.busca_resultados_param_T(parametros[1], parametros[3],
              parametros[5], this.tipo_red)
262       .pipe(first())
263       .subscribe(
264         res => {
265           this.loading = false
266           this.grafica_resultados_param_T(res)
267         },
268         error => {
269           Swal.fire(
270             'Error',
271             'Algo ha salido mal ' +  + '[' + error.status + ']' + error.error,
272             'error'
273           )
274           this.loading = false
275         }
276       )
277     }
278
279     grafica_resultados_drop(configuraciones: Configuracion[]) {
280       if(configuraciones.length == 0){
281         this.sinResultados = true;
282         this.busqueda_terminada = true;
283       }else{
284         this.sinResultados = false;
285         this.busqueda_terminada = true;
286         this.simulacion_bulk_terminada = false
```

```
287         this.simulacion_single_terminada = false
288
289         let clasificaciones = []
290         let configuraciones_clasificacion = []
291         this.valores_drop_unicos = configuraciones.map(item => item.drop_rate).filter((value,
                index, self) => self.indexOf(value) === index)
292         this.etiquetas = configuraciones.map(item => item.num_nodos).filter((value, index,
                self) => self.indexOf(value) === index)
293
294         for(let i = 0; i < this.valores_drop_unicos.length; i++){
295           for(let j = 0; j < configuraciones.length; j++){
296             if(configuraciones[j].drop_rate == this.valores_drop_unicos[i])
297               configuraciones_clasificacion.push(configuraciones[j])
298           }
299           clasificaciones.push(configuraciones_clasificacion)
300           configuraciones_clasificacion = []
301         }
302
303         let t_conv_clasificacion = []
304         this.resultados_t_conv_clasificiacion = []
305
306         for(let i = 0; i < clasificaciones.length; i++){
307           for(let j = 0; j < clasificaciones[i].length; j++){
308             let promedio_t_conv_configuracion = 0
309             for(let k = 0; k < clasificaciones[i][j].resultados.length; k++){
310               promedio_t_conv_configuracion += clasificaciones[i][j].resultados[k].t_conv
311             }
312             promedio_t_conv_configuracion /= clasificaciones[i][j].resultados.length;
313             t_conv_clasificacion.push(promedio_t_conv_configuracion)
314           }
315           this.resultados_t_conv_clasificiacion.push(t_conv_clasificacion)
316           t_conv_clasificacion = []
317         }
318
319         // Inicia creacion de los arreglos de datos para las gráficas
320         this.titulo_grafica = "Tiempos de Convergencia Variando Drop Rate"
321         this.tipo_series = "Drop Rate"
322
323         let observables: Observable<any>[] = [];
324
325         for(let i = 0; i < this.resultados_t_conv_clasificiacion.length; i++){
326           observables.push(this.simulacionService.get_regresion(this.
                resultados_t_conv_clasificiacion[i]))
327         }
328
329         Observable.forkJoin(observables).subscribe(
330           res => this.muestra_modelos(res)
331         )
332
333
334         // Inicia poblado de datos para graficaciõn
335         this.xAxisLabel = "Nũmero de Nodos de la Red"
336         this.yAxisLabel = "Tiempo de Convergencia de la Red (Elecciõn de Líder)"
```

```
337
338         if(this.tipo_red == 'Aleatorios')
339           this.yScaleMax = 100
340         else
341           this.yScaleMax = 0
342         this.series = []
343         this.datos = []
344         // Numero de clasificaciones (series)
345         for (let i = 0; i < this.valores_drop_unicos.length; i ++){
346           // Resultados en cada serie
347           for(let j = 0; j < this.resultados_t_conv_clasificiacion[i].length; j++){
348             this.series.push({"name": this.etiquetas[j], "value": this.
                  resultados_t_conv_clasificiacion[i][j]})
349           }
350           this.datos.push({"name": "Drop Rate  = " + this.valores_drop_unicos[i], "series":
                this.series})
351           this.series = []
352         }
353
354         // Fin Datos de Graficaciõn
355
356
357       }
358
359     }
360
361     grafica_resultados_param_T(configuraciones: Configuracion[]) {
362       if(configuraciones.length == 0){
363         this.sinResultados = true;
364         this.busqueda_terminada = true;
365       }else{
366         this.sinResultados = false;
367         this.busqueda_terminada = true;
368         this.simulacion_bulk_terminada = false
369         this.simulacion_single_terminada = false
370
371
372         let clasificaciones = []
373         let configuraciones_clasificacion = []
374         this.valores_t_unicos = configuraciones.map(item => item.param_T).filter((value,
              index, self) => self.indexOf(value) === index)
375         this.etiquetas = configuraciones.map(item => item.num_nodos).filter((value, index,
              self) => self.indexOf(value) === index)
376
377
378         for(let i = 0; i < this.valores_t_unicos.length; i++){
379           for(let j = 0; j < configuraciones.length; j++){
380             if(configuraciones[j].param_T == this.valores_t_unicos[i])
381               configuraciones_clasificacion.push(configuraciones[j])
382           }
383           clasificaciones.push(configuraciones_clasificacion)
384           configuraciones_clasificacion = []
385         }
```

```
386
387         let t_conv_clasificacion = []
388         this.resultados_t_conv_clasificiacion = []
389
390         for(let i = 0; i < clasificaciones.length; i++){
391           for(let j = 0; j < clasificaciones[i].length; j++){
392             let promedio_t_conv_configuracion = 0
393             for(let k = 0; k < clasificaciones[i][j].resultados.length; k++){
394               promedio_t_conv_configuracion += clasificaciones[i][j].resultados[k].t_conv
395             }
396             promedio_t_conv_configuracion /= clasificaciones[i][j].resultados.length;
397             t_conv_clasificacion.push(promedio_t_conv_configuracion)
398           }
399           this.resultados_t_conv_clasificiacion.push(t_conv_clasificacion)
400           t_conv_clasificacion = []
401         }
402
403         // Inicia creacion de los arreglos de datos para las gráficas
404         this.titulo_grafica = "Tiempos de Convergencia Variando T"
405         this.tipo_series = "Parámetro T"
406
407         let observables: Observable<any>[] = [];
408
409         for(let i = 0; i < this.resultados_t_conv_clasificiacion.length; i++){
410           observables.push(this.simulacionService.get_regresion(this.
                  resultados_t_conv_clasificiacion[i]))
411         }
412
413         Observable.forkJoin(observables).subscribe(
414           res => this.muestra_modelos(res)
415         )
416
417         // Inicia poblado de datos para graficaciõn
418         this.xAxisLabel = "Nũmero de Nodos de la Red"
419         this.yAxisLabel = "Tiempo de Convergencia de la Red (Elecciõn de Líder)"
420
421         if(this.tipo_red == 'Aleatorios')
422           this.yScaleMax = 100
423         else
424           this.yScaleMax = 0
425         this.series = []
426         this.datos = []
427         // Numero de clasificaciones (series)
428         for (let i = 0; i < this.valores_t_unicos.length; i ++){
429           // Resultados en cada serie
430           for(let j = 0; j < this.resultados_t_conv_clasificiacion[i].length; j++){
431             this.series.push({"name": this.etiquetas[j], "value": this.
                    resultados_t_conv_clasificiacion[i][j]})
432           }
433           this.datos.push({"name": "Parametro T = " + this.valores_t_unicos[i], "series":
                  this.series})
434           this.series = []
435         }
```

```
436
437         // Fin Datos de Graficaciõn
438       }
439
440     }
441     muestra_modelos(res: any[]): void {
442
443       this.pendientes = []
444       this.ordenadas = []
445
446       for (let i = 0; i < res.length; i++){
447         this.pendientes.push(res[i][0]['pendiente'])
448         this.ordenadas.push(res[i][0]['ordenada'])
449       }
450
451
452     }
453
454     iniciar_single(){
455       this.loading = true
456       this.simulacion_bulk_terminada = false
457       this.simulacion_single_terminada = false
458       this.busqueda_terminada = false
459       this.sinResultados = true
460       const formData = new FormData();
461       Object.keys(this.singleFormGroup.value).forEach(key => {
462         Object.keys(this.singleFormGroup.value[key]).forEach(key_c =>{
463           formData.append(key_c, this.singleFormGroup.value[key][key_c])
464         });
465       });
466
467       var t0 = performance.now()
468       this.simulacionService.ejecutaConfiguracion(formData).pipe(first())
469         .subscribe(
470           res => {
471             var t1= performance.now()
472             Swal.fire(
473               'Exito!',
474               'Simulaciõn Terminada en ' + ((t1 - t0) / 1000.0).toFixed(5) + " [s]",
475               'success'
476             ),
477             this.resultado_simulacion_single = res
478             this.loading = false
479             this.muestra_resultados_single()
480           },
481           error => {
482             console.log(error)
483             Swal.fire(
484               'Error interno del servidor',
485               'Algo ha salido mal ' + '[' + error.status + ']' + error.error,
486               'error'
487             ),
488             this.loading = false
```

```
489         }
490       )
491   }
492
493
494   muestra_resultados_single() {
495
496     this.resultados = []
497
498     for (let resultado in this.resultado_simulacion_single)
499       this.resultados.push(this.resultado_simulacion_single[resultado]['fields'])
500
501       this.configuracionService.getConfiguracion(this.resultados[0]['configuracion']).pipe(
              first())
502       .subscribe(
503         res => {
504           this.grafo = ''+ res.grafo_red.slice(2,-1)
505           this.simulacion_single_terminada = true
506           this.sinResultados = true
507         }
508       )
509   }
510
511   iniciar_bulk_barabasi(){
512     this.simulacion_single_terminada = false
513     this.simulacion_bulk_terminada = false
514     this.sinResultados = true
515     this.busqueda_terminada = false
516     const formData = new FormData();
517     const controls = this.bulkFormGroup.controls;
518     for (const name in controls) {
519       formData.append(name, controls[name].value);
520     }
521     Swal.fire({
522       title: 'Seguro que quieres iniciar la simulaciõn?',
523       text: "La simulaciõn de todas estas redes puede llevar varios minutos.",
524       icon: 'warning',
525       showCancelButton: true,
526       confirmButtonColor: '#5cb85c',
527       cancelButtonColor: '#d9534f ',
528       confirmButtonText: 'Si, iniciar.'
529     }).then((result) => {
530       if (result.value) {
531         this.loading = true
532         var t0 = performance.now()
533         this.simulacionService.ejecutaConfiguracionBulkBarabasi(formData).pipe(first())
534           .subscribe(
535             res => {
536               var t1 = performance.now()
537               Swal.fire(
538                 'Exito!',
539                 'Simulaciones Terminadas en ' + ((t1 - t0) / 1000.0).toFixed(5) + " [s]" ,
540                 'success'
```

```
541                 ),
542                 this.resultado_simulacion_bulk = res
543                 this.loading = false
544                 this.muestra_resultados_bulk('barabasi')
545               },
546               error => {
547                 console.log(error)
548                 Swal.fire(
549                   'Error interno del servidor',
550                   'Algo ha salido mal ' + '[' + error.status + '] ' + error.error,
551                   'error'
552                 ),
553                 this.loading = false
554               }
555             )
556         }
557       })
558
559     }
560
561     iniciar_bulk_aleatorias(){
562       this.simulacion_single_terminada = false
563       this.simulacion_bulk_terminada = false
564       this.sinResultados = true
565       this.busqueda_terminada = false
566       const formData = new FormData();
567       const controls = this.bulkFormGroup.controls;
568       for (const name in controls) {
569         formData.append(name, controls[name].value);
570       }
571       Swal.fire({
572         title: 'Seguro que quieres iniciar la simulaciõn?',
573         text: "La simulaciõn de todas estas redes puede llevar varios minutos.",
574         icon: 'warning',
575         showCancelButton: true,
576         confirmButtonColor: '#5cb85c',
577         cancelButtonColor: '#d9534f ',
578         confirmButtonText: 'Si, iniciar.'
579       }).then((result) => {
580         if (result.value) {
581           this.loading = true
582           var t0 = performance.now()
583           this.simulacionService.ejecutaConfiguracionBulkAleatorias(formData).pipe(first())
584             .subscribe(
585               res => {
586                 var t1 = performance.now()
587                 Swal.fire(
588                   'Exito!',
589                   'Simulaciones Terminadas en ' + ((t1 - t0) / 1000.0).toFixed(5) + " [s]" ,
590                   'success'
591                 ),
592                 this.resultado_simulacion_bulk = res
593                 this.loading = false
```

```
              this.muestra_resultados_bulk('aleatorias')
            },
            error => {
              console.log(error)
              Swal.fire(
                'Error interno del servidor',
                'Algo ha salido mal ' + '[' + error.status + '] ' + error.error,
                'error'
              ),
              this.loading = false
            }
          )
      }
    })

  }

  iniciar_bulk(){
    this.simulacion_single_terminada = false
    this.simulacion_bulk_terminada = false
    this.sinResultados = true
    this.busqueda_terminada = false
    const formData = new FormData();
    const controls = this.bulkFormGroup.controls;
    for (const name in controls) {
      formData.append(name, controls[name].value);
    }
    Swal.fire({
      title: 'Seguro que quieres iniciar la simulaciõn?',
      text: "La simulaciõn de todas estas redes puede llevar varios minutos.",
      icon: 'warning',
      showCancelButton: true,
      confirmButtonColor: '#5cb85c',
      cancelButtonColor: '#d9534f ',
      confirmButtonText: 'Si, iniciar.'
    }).then((result) => {
      if (result.value) {
        this.loading = true
        var t0 = performance.now()
        this.simulacionService.ejecutaConfiguracionBulk(formData).pipe(first())
          .subscribe(
            res => {
              var t1 = performance.now()
              Swal.fire(
                'Exito!',
                'Simulaciones Terminadas en ' + ((t1 - t0) / 1000.0).toFixed(5) + " [s]" ,
                'success'
              ),
              this.resultado_simulacion_bulk = res
              this.loading = false
              this.muestra_resultados_bulk('anillos')
            },
            error => {
```

```
647                 console.log(error)
648                 Swal.fire(
649                   'Error interno del servidor',
650                   'Algo ha salido mal ' + '[' + error.status + '] ' + error.error,
651                   'error'
652                 ),
653                 this.loading = false
654               }
655             )
656         }
657       })
658
659   }
660
661 muestra_resultados_bulk(tipo_redes: string){
662
663   var resultados_por_num_nodos = []
664   var mensajes_enviados = []
665   var mensajes_perdidos = []
666   var mensajes_entregados = []
667   var mensajes_procesados = []
668   var mensajes_en_ventana_limite = []
669   var retrasos_generados = []
670   var tiempos_ejecucion = []
671   var tiempos_convergencia = []
672   var  tiempos_reconvergencia = []
673
674   this.etiquetas = []
675   this.etiquetas = this.resultado_simulacion_bulk.map(item => item['numero_nodos']).filter
          ((value, index, self) => self.indexOf(value) === index)
676
677   let max = Math.max(...this.etiquetas)
678   console.log(max)
679   for (let i = 2; i <= max; i++){
680     var result = this.resultado_simulacion_bulk.
681     filter(
682       obj => {
683               return obj['numero_nodos'] == i
684       })
685       if(result.length > 0)
686         resultados_por_num_nodos.push(result)
687     }
688
689     console.log(resultados_por_num_nodos)
690
691     for(let i = 0; i < resultados_por_num_nodos.length; i ++){
692       let m_env = 0.0
693       let m_perd = 0.0
694       let m_ent = 0.0
695       let m_proc = 0.0
696       let m_vent_lim = 0.0
697       let ret_gen = 0.0
698       let t_run = 0.0
```

```
699         let t_conv = 0.0
700         let t_reconv = 0.0
701         let num_resultados = 0
702         for(let j in resultados_por_num_nodos[i]){
703           m_env += resultados_por_num_nodos[i][j]['m_env']
704           m_perd += resultados_por_num_nodos[i][j]['m_perd']
705           m_ent += resultados_por_num_nodos[i][j]['m_ent']
706           m_proc += resultados_por_num_nodos[i][j]['m_proc']
707           m_vent_lim += resultados_por_num_nodos[i][j]['m_vent_lim']
708           ret_gen += resultados_por_num_nodos[i][j]['ret_gen']
709           t_run += resultados_por_num_nodos[i][j]['t_run']
710           t_conv += resultados_por_num_nodos[i][j]['t_conv']
711           t_reconv += resultados_por_num_nodos[i][j]['t_reconv']
712
713           num_resultados += 1
714         }
715         m_env /= num_resultados
716         m_perd /= num_resultados
717         m_ent /= num_resultados
718         m_proc /= num_resultados
719         m_vent_lim /= num_resultados
720         ret_gen /= num_resultados
721         t_run /= num_resultados
722         t_conv /= num_resultados
723         t_reconv /= num_resultados
724
725         this.resultados_promediados.push({'m_env': m_env, 'm_perd': m_perd, 'm_ent': m_ent, '
                m_proc': m_proc,
726                                           'm_vent_lim': m_vent_lim, 'ret_gen': ret_gen, '
                                                t_run': t_run, 't_conv': t_conv, 't_reconv':
                                                t_reconv})
727         mensajes_enviados.push(m_env)
728         mensajes_perdidos.push(m_perd)
729         mensajes_entregados.push(m_ent)
730         mensajes_procesados.push(m_proc)
731         mensajes_en_ventana_limite.push(m_vent_lim)
732         retrasos_generados.push(ret_gen)
733         tiempos_ejecucion.push(t_run)
734         tiempos_convergencia.push(t_conv)
735         tiempos_reconvergencia.push(t_reconv)
736
737       }
738
739       this.fecha_actual = Date.now()
740
741       // Tiempos de Convergencia
742       // Inicia poblado de datos para graficaciõn
743       this.xAxisLabel = "Nũmero de Nodos de la Red"
744       this.yAxisLabel = "Tiempo de Convergencia de la Red (Elecciõn de Líder)"
745
746       if(this.opcion == 3)
747         this.yScaleMax = 100
748       else
```

```
749         this.yScaleMax = 0
750       this.series = []
751       this.datos = []
752 
753       for (let i = 0; i < tiempos_convergencia.length; i ++){
754         this.series.push({"name": this.etiquetas[i], "value": tiempos_convergencia[i]})
755       }
756       this.datos.push({"name": "Tiempo de Convergencia", "series": this.series})
757       // Fin Datos de Graficaciõn Tiempos de Convergencia
758 
759       this.simulacion_bulk_terminada = true
760 
761     }
762 
763 
764     cambia_configuracion(configuracion: Configuracion){
765       this.muestra_botones = true
766     }
767 
768     findInvalidControls(form: FormGroup) {
769       const invalid = [];
770       const controls = form.controls;
771       for (const name in controls) {
772           if (controls[name].invalid) {
773               invalid.push(name);
774           }
775       }
776       return invalid;
777     }
778 
779     onSelect(data): void {
780       console.log('Item Seleccionado', JSON.parse(JSON.stringify(data)));
781     }
782 
783     onActivate(data): void {
784       console.log('Activar', JSON.parse(JSON.stringify(data)));
785     }
786 
787     onDeactivate(data): void {
788       console.log('Desactivar', JSON.parse(JSON.stringify(data)));
789     }
790 
791 }
```

Código 2: Servicio de consumo de Simulación del *front-end* al *back-end*

simulacion.service.ts

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient, HttpParams } from '@angular/common/http';
3  import { Configuracion } from '../_models/configuracion';
4  import { Resultado} from '../_models/resultado';
5  import { Observable } from 'rxjs';
```

```
6
7  @Injectable({
8    providedIn: "root"
9  })
10 export class SimulacionService {
11
12   private local = true
13   private baseUrl
14
15   constructor(private http: HttpClient) {
16     if (this.local){
17       this.baseUrl = 'http://localhost:8000/simulador/';
18
19     }else{
20       this.baseUrl = 'http://187.168.219.12:8000/simulador/';
21     }
22
23   }
24
25
26   busca_resultados_drop(param_K, param_D, param_T, tipo_red) {
27     let params = new HttpParams().set("param_K",param_K).set("param_D", param_D).set("
           param_T", param_T).set("tipo_red", tipo_red);
28     return this.http.get<Configuracion[]>(this.baseUrl+'busqueda', {params: params});
29   }
30
31   get_regresion(resultados): Observable<any>  {
32     let params = new HttpParams().set("resultados",resultados); //Create new HttpParams
33     return this.http.get(this.baseUrl+'regresion', {params: params});
34   }
35
36   busca_resultados_param_T(param_K, param_D, drop_rate, tipo_red) {
37     let params = new HttpParams().set("param_K",param_K).set("param_D", param_D).set("
           drop_rate", drop_rate).set("tipo_red", tipo_red);
38     return this.http.get<Configuracion[]>(this.baseUrl+'busqueda', {params: params});
39   }
40
41   ejecutaConfiguracion(configuracion) {
42     return this.http.post(this.baseUrl+'simulacion/single', configuracion);
43   }
44
45   ejecutaConfiguracionBulkAleatorias(configuracion_base) {
46     return this.http.post<[]>(this.baseUrl+'simulacion/bulk_aleatorias', configuracion_base
           );
47   }
48
49   ejecutaConfiguracionBulkBarabasi(configuracion_base: FormData) {
50     return this.http.post<[]>(this.baseUrl+'simulacion/bulk_barabasi', configuracion_base);
51   }
52
53   ejecutaConfiguracionBulk(configuracion_base) {
54     return this.http.post<[]>(this.baseUrl+'simulacion/bulk', configuracion_base);
55   }
```

```
56  }
```

Bibliografía

[1] Ciprian Dobre, Florin Pop, Alexandru Costan, Mugurel Ionut Andreica, and Valentin Cristea. Robust failure detection architecture for large scale distributed systems. *Proc. of the 17th International Conference on Control Systems and Computer Science*, pages 433–444, May 2009.

[2] Karla Vargas and Sergio Rajsbaum. An eventually perfect failure detector for networks of arbitrary topology connected with add channels using time-to-live values. *49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, Oregon, USA, June 27, 2019. Versión de revista en Channels Using Time-To-Live Values. Parallel Process. Lett. 30(2): 2050006:1-2050006:23 (2020)*, 2019.

[3] D. Tian, K. Wu, and X. Li. A novel adaptive failure detector for distributed systems. *2008 International Conference on Networking, Architecture, and Storage*, pages 215–221, June 2008.

[4] H. Cahng and C. Lo. A consensus-based leader election algorithm for wireless ad hoc networks. *2012 International Symposium on Computer, Consumer and Control*, pages 232–235, 2012.

[5] S. Sastry and S.M. Pike. Eventually perfect failure detectors using add channels. *Proc. 5th Int. Symposium on Parallel and Distributed Processing and Applications (ISPA)*, 4742(2):13–17, 2008.

[6] S. Rajsbaum and M. Raynal. A short introduction to failure detectors for asynchronous

distributed systems. *ACM Special Interest Group on Algorithms and Computation Theory News*, 36(1):53–70, March 2005.

[7] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[8] M. Patterson M. Fisher, N.Lynch. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 1985.

[9] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed computing principles, algorithms and systems.* Cambridge University Press, 2012.

[10] T.D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[11] M.Larrea, A. Fernández, and S. Arévalo. Implementing the weakest failure detector for solving consensus. *SRDS 2000*, 2005.

[12] M.Larrea, C. Martín, and E. Jiménez. Implementing the omega failure detector in the crash-recovery failure model. *Journal of Computer and System Sciences*, 75(4):178–189, 2008.

[13] Leslie Lamport Massachusetts Computer Associates. Time, clocks, and the ordering of events in a distributed system, Jul 1978.

[14] M. Raynal. *Distributed Algorithms for Message-Passing Systems*, pages 3–6. Springer, 2013.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Massachusetts Computer Associates, Inc.*, 1978.

[16] M. Raynal. *Distributed Algorithms for Message-Passing Systems*, pages 77–80. Springer, 2013.

[17] M.R.Stiglitz and C. Blanchard. *IEEE Standard Dictionary of Electrical and Electronics Terms Sec. 4.* IEEE, 1992.

[18] Renato Iida Le Wang and Alexander M. Wyglinski. Vehicular network simulation environment via discrete even system modeling. *Journal of Computer and System Sciences*, 75(4):178–189, 2019.

[19] Christos G. Cassandras and Lafortune Stéphane. *Introduction to discrete event systems.* Springer, 2011.

[20] Overview - Simpy 4.0.2 - Documentation. `https://simpy.readthedocs.io/en/latest/index.html`. Accesado: 2020-09-30.

[21] M. Raynal A. Mostéfaoui, Mourgaya. Asynchronus implementation of failure detectors. *Proceedings of the 33rd International Conference on Dependable Systems and Networks*, 2003.

[22] O. Sens M. Bertier, Marin. Implementation and performance evaluation of an adaptable failure detector. *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, pages 354–363, 2002.

[23] A. Fernández M. Larrea, S. Arévalo. Eficient algorithms to implement unreliable failure detectors in partially synchronus systems. *Proceedings of the 13th International Symposium on Distributes Computing*, pages 34–48, 1999.

[24] F. Tronel C. Fetzer, M. Raynal. An adaptive failure detection protocol. *Proceedings of the 7th Pacific Rim International Symposium on Dependable Computing*, pages 146–153, 2001.

[25] M. Süsskraut C. Fetzer, U. Schmid. On the possibility of consensus in asynchronous systems with finite average response times. *Proceedings of the 25th International Conference on Distributed Computing Systems*, pages 271–280, 2005.

[26] A. Lafuente M. Larrea. Communication-eficient implementation of failure detector classes $\diamond P$ and $\diamond Q$. *Proceedings of the 19th International Symposium on Distributed Computing Systems*, pages 495–496, 2005.

[27] J.L. Welch S. Kumar. Implementing $\diamond P$ with bounded messages on network of add channels. *Parallel Processing Letters*, 29:12–24, 2019.

[28] M. Hutle. An efficient failure detector for sparsely connected netwoks. *IASTED International Conference on Parallel and Distributed Computing*, 29:314–341, 1996.

[29] K. Vargas S. Rajsbaum, M.Raynal. Leader election in arbitrarily connected networks with process crashes and weak channel reliability. *Enviado a 35th IEEE International Parallel and Distributed Processing Symposium*, 2020.

[30] Documentation. Examples - Event Latency. `https://simpy.readthedocs.io/en/latest/examples/latency.html`. Accesado: 2020-09-30.

[31] NetworkX. Network Analysis in Python. `https://networkx.org/`. Accesado: 2020-10-25.

[32] NetworkX. Reference - Graph Generators. `https://networkx.org/documentation/stable/reference/generators.html`. Accesado: 2020-10-26.

[33] NetworkX. Reference - Algorithms. `https://networkx.org/documentation/stable/reference/algorithms/index.html`. Accesado: 2020-10-26.

[34] J. Kim and Van Hanh Vu. Generating random regular graphs. *Combinatorica*, 26:683–708, 12 2006.