

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA  
DEPARTAMENTO DE INFORMÁTICA  
VALPARAÍSO - CHILE



# “DESARROLLO DE UN FRAMEWORK PARA LA SIMULACIÓN Y VISUALIZACIÓN DE ALGORITMOS DISTRIBUIDOS”

ALEX HERNÁN MUSSO ROJAS

MEMORIA PARA OPTAR AL TÍTULO DE  
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Raúl Monge  
Profesor Correferente: Erika Rosas

Agosto - 2018

## **DEDICATORIA**

Dedicado a mis padres, hermano y abuela Celia, quien siempre se sintió orgullosa por mis logros y le hubiese encantado estar en esta instancia.

## **AGRADECIMIENTOS**

Solo me queda agradecer a todos los que hicieron esto posible. A mis padres y hermano por acompañarme y apoyarme en cada paso, por inspirarme a ser mejor y superarme. A mis abuelos y familia por su preocupación y apoyo incondicional. A mis amigos por todos los buenos momentos, por las noches de estudio y por hacer el proceso más ameno. A mis profesores por creer en mis capacidades, por todo el esfuerzo que pusieron para formarme como profesional, y por la cercanía de parte de todo el Departamento de Informática. Por último, agradecer a mi profesor guía Raúl Monge, por estar siempre dispuesto a ayudar y guiarme durante todo el proceso para realizar este trabajo.

## RESUMEN

**Resumen**— Con el fin de apoyar el proceso de enseñanza en la asignatura de Sistemas Distribuidos en la UTFSM, en el presente trabajo se desarrolló un *framework* para simular y visualizar algoritmos distribuidos a través del lenguaje de programación *Java* y la librería *JavaFX*. Luego, haciendo uso de este, se procedió a implementar el Algoritmo del Matón, Algoritmo de Chang/Roberts, Algoritmo de Eco y Algoritmo de Suzuki-Kasami, algoritmos distribuidos considerados lo suficientemente representativos con respecto a los diversos tipos que existen en el área. Así, se obtuvieron resultados satisfactorios, con una estructura simple e intuitiva para programar el código, junto con una interfaz en la cual se pueden modificar parámetros que afectan directamente en la ejecución, crear, modificar, importar o exportar redes de nodos, controlar la ejecución del algoritmo, y adicionalmente obtener información de cada nodo y mensaje dentro de la red, de acuerdo a lo que el usuario quiera mostrar.

**Palabras Clave**— algoritmos distribuidos; simulación; visualización; framework; educación

## ABSTRACT

**Abstract**— In order to support the teaching process inside the subject of Distributed Systems in the UTFSM, in the present work a framework was developed to simulate and visualize distributed algorithms through the Java programming language and the JavaFX library. Then, using it, it was proceeded to implement the Bully Algorithm, Chang/Roberts Algorithm, Echo Algorithm and Suzuki-Kasami Algorithm, distributed algorithms considered representative enough with respect to the various existing types in the area. Thus, satisfactories results were obtained, with a simple and intuitive structure to program the code, together with an interface which allows to modify parameters that affect directly into the execution, create, modify, import or export networks of nodes, control the execution of the algorithm, and additionally obtain information of each node and message within the network, according to what the user wants to show.

**Keywords**— distributed algorithms; simulation; visualization; framework; education

## GLOSARIO

UTFSM: Universidad Técnica Federico Santa María.

JDK: *Java Development Kit*.

IDE: *Integrated Development Environment*.

# ÍNDICE DE CONTENIDOS

|                                                                                           |    |
|-------------------------------------------------------------------------------------------|----|
| <b>RESUMEN</b> . . . . .                                                                  | IV |
| <b>ABSTRACT</b> . . . . .                                                                 | IV |
| <b>GLOSARIO</b> . . . . .                                                                 | V  |
| <b>ÍNDICE DE FIGURAS</b> . . . . .                                                        | IX |
| <b>ÍNDICE DE TABLAS</b> . . . . .                                                         | X  |
| <b>INTRODUCCIÓN</b> . . . . .                                                             | 1  |
| <b>CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA</b> . . . . .                                      | 2  |
| 1.1 Contexto de la Memoria . . . . .                                                      | 2  |
| 1.2 Macro Problema y Situación Actual . . . . .                                           | 2  |
| 1.3 Definición del Problema . . . . .                                                     | 4  |
| 1.4 Participantes en la problemática . . . . .                                            | 6  |
| 1.5 Problemática en el Mediano Plazo . . . . .                                            | 6  |
| 1.6 Objetivos de la Memoria . . . . .                                                     | 8  |
| 1.6.1 Objetivo General . . . . .                                                          | 8  |
| 1.6.2 Objetivos Específicos . . . . .                                                     | 8  |
| 1.7 Alcance y Restricciones de la Memoria . . . . .                                       | 9  |
| 1.8 Competencias y Estado Actual . . . . .                                                | 9  |
| 1.8.1 DAVE: Distributed Algorithm Visualization Engine . . . . .                          | 9  |
| 1.8.2 A New Tool for the Simulation And Visualization of Distributed Algorithms . . . . . | 11 |
| 1.8.3 VADE: Visualization of Algorithms in Distributed Environments . . . . .             | 11 |
| 1.8.4 Toolkit DAJ: Distributed Algorithms in JAVA . . . . .                               | 13 |
| 1.8.5 Características similares entre las competencias . . . . .                          | 15 |
| <b>CAPÍTULO 2: MARCO CONCEPTUAL</b> . . . . .                                             | 16 |
| 2.1 Simulación de Algoritmos Distribuidos . . . . .                                       | 16 |
| 2.1.1 Estructura . . . . .                                                                | 16 |
| 2.1.2 Funcionamiento . . . . .                                                            | 17 |
| 2.1.3 Comunicación . . . . .                                                              | 17 |
| 2.1.4 Aplicaciones . . . . .                                                              | 18 |
| 2.2 Visualización de Algoritmos Distribuidos . . . . .                                    | 19 |
| 2.2.1 Representación . . . . .                                                            | 19 |
| 2.2.2 Topologías . . . . .                                                                | 20 |
| 2.2.3 Anillo . . . . .                                                                    | 20 |
| 2.2.4 Árbol . . . . .                                                                     | 20 |
| 2.2.5 Grafo General . . . . .                                                             | 21 |

|                                                            |                                                    |           |
|------------------------------------------------------------|----------------------------------------------------|-----------|
| 2.3                                                        | Algoritmos Distribuidos y su aprendizaje . . . . . | 21        |
| 2.3.1                                                      | Visualización . . . . .                            | 21        |
| 2.3.2                                                      | Dinamicidad . . . . .                              | 22        |
| 2.3.3                                                      | Variabilidad . . . . .                             | 23        |
| <b>CAPÍTULO 3: PROPUESTA DE SOLUCIÓN . . . . .</b>         |                                                    | <b>24</b> |
| 3.1                                                        | Diseño conceptual . . . . .                        | 24        |
| 3.1.1                                                      | Objeto Node . . . . .                              | 24        |
| 3.1.2                                                      | Objeto Canal . . . . .                             | 25        |
| 3.1.3                                                      | Objeto Network . . . . .                           | 26        |
| 3.1.4                                                      | Interfaz Gráfica . . . . .                         | 27        |
| 3.1.5                                                      | Modelo de Comunicación . . . . .                   | 29        |
| 3.1.6                                                      | Técnica de Interrupción . . . . .                  | 29        |
| 3.2                                                        | Ambiente de programación . . . . .                 | 31        |
| <b>CAPÍTULO 4: IMPLEMENTACIÓN DE LA SOLUCIÓN . . . . .</b> |                                                    | <b>34</b> |
| 4.1                                                        | Clases del Framework . . . . .                     | 34        |
| 4.1.1                                                      | App . . . . .                                      | 34        |
| 4.1.2                                                      | Parameter . . . . .                                | 35        |
| 4.1.3                                                      | Network . . . . .                                  | 36        |
| 4.1.4                                                      | Node . . . . .                                     | 37        |
| 4.1.5                                                      | VisualNode . . . . .                               | 37        |
| 4.1.6                                                      | Program . . . . .                                  | 38        |
| 4.1.7                                                      | Channel . . . . .                                  | 38        |
| 4.1.8                                                      | VisualChannel . . . . .                            | 39        |
| 4.1.9                                                      | MessageQueue . . . . .                             | 39        |
| 4.1.10                                                     | Message . . . . .                                  | 39        |
| 4.1.11                                                     | Screen . . . . .                                   | 40        |
| 4.1.12                                                     | AlgorithmView . . . . .                            | 40        |
| 4.1.13                                                     | SettingsWindow . . . . .                           | 42        |
| 4.1.14                                                     | Assertion . . . . .                                | 44        |
| 4.1.15                                                     | Constants . . . . .                                | 44        |
| 4.1.16                                                     | VectorClock . . . . .                              | 44        |
| 4.1.17                                                     | LogHandler . . . . .                               | 45        |
| 4.2                                                        | Guía de Uso . . . . .                              | 46        |
| 4.2.1                                                      | Herramientas de Programación . . . . .             | 46        |
| 4.2.2                                                      | Clase App . . . . .                                | 46        |
| 4.2.3                                                      | Clase Program . . . . .                            | 48        |
| 4.2.4                                                      | Extensión de Clase Network . . . . .               | 54        |
| 4.2.5                                                      | Clase Message . . . . .                            | 54        |
| 4.2.6                                                      | Clase Node . . . . .                               | 54        |
| 4.2.7                                                      | Clase Channel . . . . .                            | 54        |
| 4.2.8                                                      | Clase Constants . . . . .                          | 55        |
| 4.2.9                                                      | Compilación y Ejecución . . . . .                  | 56        |

|                                                  |    |
|--------------------------------------------------|----|
| <b>CAPÍTULO 5: VALIDACIÓN DE LA SOLUCIÓN</b>     | 58 |
| 5.1 Algoritmo del Matón                          | 58 |
| 5.1.1 Implementación                             | 58 |
| 5.1.2 Resultados                                 | 61 |
| 5.2 Algoritmo de Chang/Roberts                   | 62 |
| 5.2.1 Implementación                             | 62 |
| 5.2.2 Resultados                                 | 65 |
| 5.3 Algoritmo de Eco                             | 66 |
| 5.3.1 Implementación                             | 67 |
| 5.3.2 Resultados                                 | 69 |
| 5.4 Algoritmo de Suzuki-Kasami                   | 71 |
| 5.4.1 Implementación                             | 71 |
| 5.4.2 Resultados                                 | 77 |
| 5.5 Evaluación Global                            | 78 |
| <b>CAPÍTULO 6: CONCLUSIONES</b>                  | 80 |
| 6.1 Objetivos y Resultados                       | 80 |
| 6.2 Reflexión y Aprendizaje Logrado              | 81 |
| 6.3 Contribuciones y Aplicaciones de la Solución | 82 |
| 6.4 Trabajo Futuro                               | 83 |
| <b>ANEXOS</b>                                    | 85 |
| <b>REFERENCIAS BIBLIOGRÁFICAS</b>                | 86 |
| <b>ÍNDICE DE PALABRAS CLAVE</b>                  | 87 |

## ÍNDICE DE FIGURAS

|    |                                                                                                                         |    |
|----|-------------------------------------------------------------------------------------------------------------------------|----|
| 1  | Árbol de Causas y Efectos. . . . .                                                                                      | 5  |
| 2  | Ejemplo de la representación del paso de un algoritmo distribuido en las diapositivas de la asignatura. . . . .         | 7  |
| 3  | Ejemplo de funcionamiento del simulador DAVE. . . . .                                                                   | 10 |
| 4  | Ejemplo de funcionamiento de <i>A New Tool for the Simulation And Visualization of Distributed Algorithms</i> . . . . . | 12 |
| 5  | Ejemplo de funcionamiento del simulador VADE. . . . .                                                                   | 13 |
| 6  | Ejemplo de funcionamiento del Toolkit DAJ. . . . .                                                                      | 14 |
| 7  | Ejemplo de la representación de un algoritmo distribuido. . . . .                                                       | 20 |
| 8  | Ejemplo de topología de anillo en un algoritmo distribuido. . . . .                                                     | 21 |
| 9  | Ejemplo de topología de árbol en un algoritmo distribuido. . . . .                                                      | 22 |
| 10 | Ejemplo de topología de grafo general en un algoritmo distribuido. . . . .                                              | 23 |
| 11 | Diagrama de interacción entre el usuario y el <i>framework</i> . . . . .                                                | 24 |
| 12 | Modelo del objeto <i>Node</i> . . . . .                                                                                 | 25 |
| 13 | Diagrama de flujo del <i>thread</i> en <i>Node</i> . . . . .                                                            | 26 |
| 14 | Modelo del objeto <i>Channel</i> . . . . .                                                                              | 27 |
| 15 | Diagrama de flujo del <i>thread</i> en <i>Channel</i> . . . . .                                                         | 28 |
| 16 | Modelo del objeto <i>Network</i> . . . . .                                                                              | 29 |
| 17 | Diagrama de flujo del <i>thread</i> en <i>Network</i> . . . . .                                                         | 30 |
| 18 | Esquema de la interfaz gráfica del <i>framework</i> . . . . .                                                           | 31 |
| 19 | Modelo de comunicación del <i>framework</i> . . . . .                                                                   | 32 |
| 20 | Diagrama de relación entre clases del <i>framework</i> . . . . .                                                        | 35 |
| 21 | Implementación del Algoritmo del Matón. . . . .                                                                         | 63 |

|    |                                                        |    |
|----|--------------------------------------------------------|----|
| 22 | Implementación del Algoritmo de Chang/Roberts. . . . . | 66 |
| 23 | Implementación del Algoritmo de Eco. . . . .           | 70 |
| 24 | Implementación del Algoritmo de Suzuki-Kasami. . . . . | 79 |

## ÍNDICE DE TABLAS

|   |                                                                            |    |
|---|----------------------------------------------------------------------------|----|
| 1 | Matriz de Influencia e Interés de los <i>Stakeholders</i> . . . . .        | 6  |
| 2 | Cuadro comparativo de los lenguajes de programación más conocidos. . . . . | 32 |

## INTRODUCCIÓN

Hoy en día, dentro de la asignatura de Sistemas Distribuidos realizada en la UTFSM, se presenta un problema en el área de docencia relacionado a la enseñanza de los **algoritmos distribuidos**. Debido a la falta de herramientas, la forma en que se está explicando su funcionamiento es a través de fotos que indiquen cada paso de estos. En consecuencia, considerando que son algoritmos completamente **dinámicos**, cambiando constantemente a lo largo del tiempo, el método de enseñanza utilizado no se hace suficiente para facilitar y profundizar su comprensión.

Como efecto de lo anterior, dentro del presente trabajo se propondrá una solución consistente en un *framework* el cual permita simular y visualizar algoritmos distribuidos, y el que se abordará de la siguiente manera:

- En primer lugar, dentro del capítulo **Definición del Problema**, se analizará en detalle el problema descrito, describiendo la metodología del curso, involucrados en el problema, competencias actuales que existen en el área, además de los objetivos de esta memoria.
- Luego, en el capítulo **Marco Conceptual**, se analizarán todos los factores que componen a un algoritmo distribuido, junto con toda la información necesaria a la hora de simularlos y visualizarlos, considerando los tipos, topologías, entre otros.
- Una vez hecho lo anterior, se comenzará a diseñar la solución en **Propuesta de Solución**, indicando las herramientas a utilizar, las clases principales del *framework* y cómo será la lógica de este para lograr simular un algoritmo distribuido, y por lo tanto, paralelo.
- Después de ya haber hecho el diseño, se irá directamente a implementar la solución en **Implementación de la Solución**. Para esto, haciendo uso del lenguaje de programación *Java* y la librería para interfaces gráficas *JavaFX*, se seguirá una metodología de desarrollo *bottom-up*, comenzando con la parte de simulación del *framework*, para luego terminar con la visualización a través de la interfaz gráfica. Así, dentro del capítulo se detallarán cada una de las clases que lo componen, además de analizar los métodos esenciales que logran su funcionamiento, para finalmente crear una guía de uso dirigida al usuario que vaya a utilizarlo.
- Una vez hecha la implementación, se utilizará el *framework* en el capítulo **Validación de la Solución** para implementar cuatro algoritmos distribuidos como casos de prueba, verificando que la solución resuelve el problema y logra los objetivos propuestos.
- Finalmente, se harán las conclusiones respectivas de todo el trabajo, basándose en los resultados y contrastándolo con los objetivos planteados inicialmente. Se considerarán los aportes, contribuciones, además de propuestas para un trabajo futuro de la solución.

# **CAPÍTULO 1**

## **DEFINICIÓN DEL PROBLEMA**

### **1.1. Contexto de la Memoria**

En primer lugar, todo lo que se aborde en este trabajo será desarrollado dentro del contexto educativo del Departamento de Informática de la UTFSM, dirigido específicamente a la asignatura INF-343 Sistemas Distribuidos que hoy es parte de la Licenciatura en Ciencias de la Ingeniería de la carrera de Ingeniería Civil Informática. Así, este trabajo pretende ser una contribución para mejorar la calidad docente de la asignatura, intentando integrar herramientas que permitan realizar cambios metodológicos que fortalezcan los procesos de aprendizaje de los estudiantes. En consecuencia, el profesor y los estudiantes de la asignatura son los usuarios involucrados dentro de la problemática identificada.

### **1.2. Macro Problema y Situación Actual**

Actualmente, la situación de la asignatura de Sistemas Distribuidos dentro de la Universidad presenta un escenario complejo. Esto debido a que, por una parte, la asignatura abarca una gran cantidad de contenidos teóricos que requieren de la totalidad de las clases presenciales para transmitirse y enseñarse, mientras que por otra parte, el tiempo restante para realizar actividades prácticas se hace cada vez menor. Es aquí donde identificamos el macro problema a tratar, referido al programa de la asignatura.

El programa actual de Sistemas Distribuidos comprende los siguientes contenidos:

#### **1. Introducción General a los Sistemas Distribuidos:**

- Motivación
- Definición y características de un sistema distribuido
- Ventajas y desventajas
- Sistemas centralizados vs. distribuidos
- Técnicas de distribución
- Modelos y arquitecturas
- Tendencias

#### **2. Programación Distribuida y Comunicación**

- Paradigmas de programación

- Programación distribuida
- Comunicación orientada a mensajes
- *Sockets* en TCP/IP
- *Multicasting*
- Protocolos de comunicación de *Middleware*
- Invocación remota
- *Middleware* orientado mensajería (MoM).

### **3. Fundamentos Teóricos de Computación Distribuida**

- Modelos de Sistemas Distribuidos
- Tiempo, causalidad y ordenamiento de eventos
- Relojes lógicos
- Sincronización de relojes
- Observaciones válidas
- Historias causales y relojes de vector
- Cortes, estados globales y consistencia

### **4. Algoritmos Distribuidos**

- Algoritmos distribuidos básicos
- Elección distribuida
- Instantánea distribuida
- Detección de término
- Detección de *deadlock*
- Exclusión mutua distribuida

### **5. Base de Datos y Transacciones Distribuidas**

- Distribución de datos
- Transacciones distribuidas
- Control de concurrencia de transacciones y serialización
- Tolerancia a fallas y alta disponibilidad
- Recuperación de errores en la ejecución de transacciones y almacenamiento de datos
- Protocolos de compromiso para transacciones distribuidas

### **6. Replicación de Datos**

- Enfoques de replicación de datos
- Semánticas de consistencia
- Protocolos de replicación centrados en los datos
- Consistencia eventual
- Protocolos de replicación centrados en los usuarios

## 7. Seguridad en Sistemas Distribuidos

- Seguridad de información
- Criptografía
- Algoritmos de cifrado
- Firma digital
- Infraestructura de Clave Pública
- Seguridad en comunicaciones y aplicaciones

Estos contenidos se profundizan a través de clases expositivas sobre la materia, lectura de material bibliográfico complementario, desarrollo de guías y tareas de programación que permiten profundizar desde la práctica muchos de los conceptos aprendidos teóricamente. Así, tal como se puede apreciar, se comprende una gran cantidad de materia conceptual, con gran teoría detrás, considerando que se analizan los distintos tipos de sistemas distribuidos que existen, junto con la forma de comunicación, arquitectura, entre otros. A esto se le suma el hecho de que es una cantidad considerable de materia la cual está planificada para tratarse durante todo el semestre, solo dando espacio para hacer dos tareas prácticas, generalmente cubriéndose aspectos de programación distribuida que incluye *Multicasting*, *Sockets* y comunicación basada en mensajes, como es el caso de *MoM*<sup>1</sup>. En estas tareas se implementa con bastante esfuerzo al menos un algoritmo distribuido cubierto en la materia.

Considerando lo anterior, el efecto que se produce en la asignatura es que no se puede aplicar la mayoría de la materia a situaciones reales y prácticas, quedando gran parte sin cubrir, teniendo como consecuencia una baja profundización de los contenidos transmitidos y escaso dominio sobre los temas conversados, debido a la falta de un escenario práctico en donde los estudiantes puedan desenvolverse, creando dudas y resolviéndolas de la misma manera; y en todas las situaciones donde sean ellos mismos quienes interactúan, sin necesitar al profesor como intermediario.

### 1.3. Definición del Problema

Es en base a todo lo mencionado, que es posible comenzar a definir el problema identificado y que se pretende solucionar con esta memoria.

---

<sup>1</sup>Siglas para referenciar a *Message-Oriented Middleware* o *Middleware orientado a Mensajería*.

Para innovar en el aprendizaje, el curso de Sistemas Distribuidos se encuentra actualmente en un proceso de actualización, buscando nuevos aportes posibles para mejorar las metodologías y profundizar más en los contenidos por parte de los estudiantes. Así, uno de los principales requerimientos en este contexto es el poder representar de una forma más interactiva, comprensible e intuitiva cada uno de los algoritmos distribuidos estudiados. Actualmente, estos son representados a través de diapositivas estáticas, las cuales explican paso por paso desarrollo del algoritmo, sin poder ver más allá del ejemplo o de lo que se muestra, tales como el estado actual de los nodos, el mensaje que se transmite, y otra información que pudiera ser relevante para una mejor comprensión. En consecuencia, se ha propuesto integrar tecnologías en la clase, para simular y visualizar de una manera interactiva, intuitiva y con capacidades de *debugging*<sup>2</sup> de estos algoritmos. Sin embargo, el escenario es que hay una falta de herramientas para realizar este propósito, encontrándose el problema central del presente trabajo y el cual se detalla en el siguiente diagrama (Ver figura 1).



Figura 1: Árbol de Causas y Efectos.

Fuente: Elaboración Propia.

<sup>2</sup>Es el proceso de analizar internamente cómo funciona un algoritmo, pudiendo hacer pruebas para encontrar posibles errores o entender su funcionamiento.

## 1.4. Participantes en la problemática

Tal como se identificó anteriormente, el problema mencionado involucra al profesor y estudiantes de la asignatura de Sistemas Distribuidos de la UTFSM. Estos comprenden un universo de, en promedio, 100 personas en total cada semestre en el que se dicta el curso, lo cual generalmente es un semestre al año, con pequeñas variaciones entre uno y otro. Sin embargo, hay más actores o *stakeholders*<sup>3</sup> involucrados, como lo es el mismo Departamento de Informática y la Universidad. Así, es posible hacer su representación en la siguiente matriz (Ver tabla 1).

Tabla 1: Matriz de Influencia e Interés de los *Stakeholders*.

Fuente: Elaboración Propia.

| Stakeholder                 | Influencia                                                                                                 | Interés                                                                                                                       |
|-----------------------------|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Profesor de la asignatura   | <i>Alto</i> . Será finalmente quien decida su metodología de enseñanza y cómo hará la clase.               | <i>Alto</i> . Existe gran interés por el aprendizaje de sus estudiantes y de aportar con lo que se enseña.                    |
| Estudiante de la asignatura | <i>Alto</i> . Serán los consumidores directos del contenido y el enfoque del profesor.                     | <i>Alto</i> . Existe gran interés por los contenidos y por el adquirir nuevos conocimientos.                                  |
| Departamento de Informática | <i>Medio</i> . Restringen la carga académica del ramo a una cierta cantidad de horas.                      | <i>Medio</i> . Se tiene interés por la formación y perfil de egreso del estudiante.                                           |
| UTFSM                       | <i>Bajo</i> . No influye directamente en el curso, pero sí entrega la infraestructura y las oportunidades. | <i>Medio</i> . Se tiene interés por el perfil de egreso de sus estudiantes y el reconocimiento de ellos en el ámbito laboral. |
| Otras universidades         | <i>Bajo</i> . No influyen en el ramo.                                                                      | <i>Medio</i> . Hay interés por el aporte que se haga al área de la docencia y al área de Sistemas Distribuidos.               |

## 1.5. Problemática en el Mediano Plazo

Actualmente, considerando que nos encontramos en una sociedad continuamente con mayor tecnología, incorporándose esta al mismo tiempo en las salas de clases, cada vez se hace

<sup>3</sup>Son todos los actores que se vean afectados o estén interesados de alguna manera en el problema.

más necesario innovar en las metodologías educativas. Así, es importante hacer uso de los recursos tecnológicos que hoy en día están presentes y darles un buen uso educativo con el fin de mejorar y renovar las formas de enseñar, logrando de esta forma un mejor aprendizaje.

Teniendo en cuenta lo anterior, si las dificultades mencionadas en el problema persisten en el corto o mediano plazo, es claro que la asignatura comenzaría a quedar obsoleta, perdiendo los estudiantes la motivación por cursarla al percibirlo como una con netamente contenido teórico, baja práctica y sin visualizar las posibles aplicaciones y aportes del contenido en el mundo real. En otras palabras, el curso se mantendría de la misma manera y seguiría conservándose la misma situación actual, donde no se puede escalar, aplicar ni interactuar con los contenidos tal como se desearía, y donde las diapositivas no son suficientes para explicar completamente el funcionamiento de los algoritmos distribuidos, tal como se aprecia en la figura 2. En este caso, por ejemplo, basta con que el estudiante quiera analizar el algoritmo con otro conjunto de nodos o un nodo inicial diferente para ver la falta de una herramienta con la que se pueda interactuar y variar las condiciones del ejercicio para hacer pruebas con otros escenarios.

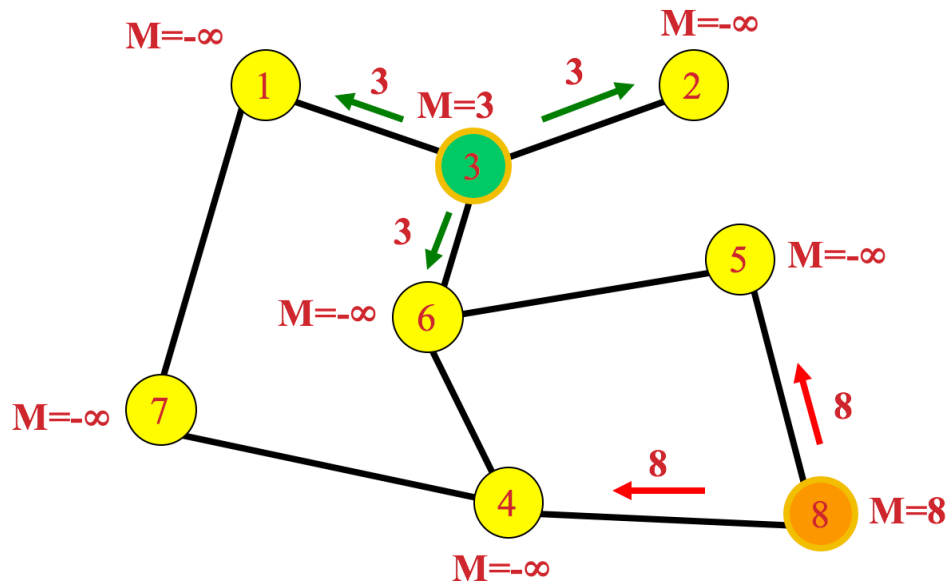


Figura 2: Ejemplo de la representación del paso de un algoritmo distribuido en las diapositivas de la asignatura.

Fuente: [Monge(2017)].

Enfocándose en el tema anterior es cuando se ve la utilidad de la presente memoria y el por qué conviene hacerla. Así, con este trabajo se pretende apaciguar y cubrir la necesidad identificada a través de una herramienta interactiva para los estudiantes, donde sean ellos mismos quienes puedan aplicar, desarrollar y descubrir autónomamente lo aprendido. En consecuencia, se busca aumentar el interés de los estudiantes en seguir aprendiendo e inducirse en el área de los Sistemas Distribuidos, así como dar una base para que innoven y creen nuevas aplicaciones a partir de esto, probando nuevos escenarios en la vida real, ha-

ciendo variaciones y mejoras en los algoritmos, entre muchas otras opciones para dar paso a la creatividad de los alumnos y cumplir con el objetivo del curso.

Por último, es importante analizar el aporte general que se haría en el curso y las innovaciones que podrían hacerse. Considerando que existiera una herramienta para simular y visualizar algoritmos distribuidos, podría modificarse el sistema de evaluación. Así, en vez de hacer solo dos tareas durante todo el semestre, las cuales no se justifican por el bajo contenido que cubren con respecto a la cantidad de materia del curso, podrían hacerse actividades prácticas pequeñas para ciertas clases, en donde, por ejemplo, se solicite simular un algoritmo distribuido, modificarlo, o sacar ciertas conclusiones a partir de éste. De esta forma, la modalidad de la clase cambiaría, ya que sería el mismo estudiante quien aprende haciendo, siendo una metodología que lo fuerza a estar activo y concentrado, profundizando y entendiendo correctamente la teoría que se encuentra detrás del contenido en sí. Finalmente, este caso podría aumentar el alcance del ramo de Sistemas Distribuidos, pudiéndose incluso hacer proyectos que se traten de la creación o modificación de algoritmos distribuidos, siendo el profesor y estudiante capaces de dar un cierre y aplicación real a todo lo enseñado.

## 1.6. Objetivos de la Memoria

Dada la problemática ya mencionada, los objetivos de la presente memoria consistirán en lo siguiente:

### 1.6.1. Objetivo General

- Desarrollar demostraciones que permitan visualizar computacionalmente diferentes tipos de algoritmos distribuidos, de manera de facilitar la comprensión de su diseño y funcionamiento en el aprendizaje.

### 1.6.2. Objetivos Específicos

- Analizar *toolkits* existentes para la simulación y visualización de algoritmos distribuidos.
- Contrastar tecnologías actuales para programar Sistemas Distribuidos y construir interfaces de usuario.
- Diseñar un *framework* que permita una programación simple e intuitiva de algoritmos distribuidos.
- Construir algoritmos distribuidos para probar y ejemplificar el uso del *framework* implementado.

## 1.7. Alcance y Restricciones de la Memoria

El alcance de este trabajo comprende la implementación de un *framework* para simular y visualizar algoritmos distribuidos, así como la construcción de algunos de estos utilizando el *toolkit* para ejemplificar su uso y capacidades, al mismo tiempo que se verifica su correcto funcionamiento.

Con respecto a las restricciones, al ser un *framework* no hay impedimentos con respecto al *hardware*, debido a que los casos de prueba serían con algoritmos distribuidos lo suficientemente simples para ser simulados en un entorno normal y sin requerir mucho procesamiento. Así, solo habrían límites en cuanto a la cantidad de algoritmos a ejemplificar y a las capacidades de *debugging* de la interfaz, donde la profundización de esta estará marcada por el tiempo disponible y las tecnologías existentes disponibles y posibles de utilizar.

## 1.8. Competencias y Estado Actual

Hoy en día se ha trabajado mucho en el área de algoritmos distribuidos al momento de poder simularlos y visualizarlos. A pesar de existir varias herramientas para este propósito, lo que ocurre es que gran parte de éstas se encuentran obsoletas, con tecnologías que ya no se usan y dejan de adaptarse a todos los recursos y avances que actualmente se han hecho; por ejemplo, en lo que a interfaz y experiencia de usuario de trata. Además, la mayoría no cubren todas las necesidades ideales que se requieren para resolver el problema, y dejan de ser intuitivas para los estudiantes, mientras que otros nunca fueron implementados y fueron solamente desarrollados como un concepto, o pensados para áreas más generales, tales como programación paralela<sup>4</sup>.

Dentro de las distintas alternativas que se encuentran, es posible destacar las siguientes:

### 1.8.1. DAVE: Distributed Algorithm Visualization Engine

Tal como se menciona en [McGinnis(1998)], *DAVE* fue construida como una herramienta complementaria a un curso de Sistemas Distribuidos, con el fin de permitir a los propios estudiantes construir y automatizar procesos distribuidos con entradas y salidas de una forma gráfica, creando nodos y uniéndolos de la misma manera. Por otra parte, es una herramienta que fue pensada con el fin de servir como interfaz para simuladores desarrollados en el lenguaje *IOA*<sup>5</sup>.

A pesar de encontrarse el código del *software* en [McGinnis(1998)], es más representativo

---

<sup>4</sup>Forma de programación en la que muchas instrucciones se ejecutan simultáneamente.

<sup>5</sup>Lenguaje formal utilizado para describir procesos computacionales los cuales generalmente están modelados como sistemas o algoritmos distribuidos.

analizar sus funcionalidades a través de la interfaz que provee (Ver figura 3). Con esto, se aprecia la red de nodos, sus estados, y las funciones para enviar o recibir mensajes de cada uno. Además, se logra ver controles para reproducir, detener, o analizar paso por paso el algoritmo, lo que le da un gran valor como simulador de algoritmos distribuidos. Sin embargo, es un programa que utiliza tecnologías obsoletas. Se encuentra programado en una versión antigua de *Java*<sup>6</sup> con la librería de gráficos AWT<sup>7</sup>, la cual ya hoy en día han sido mayormente reemplazadas por otras versiones mejoradas.

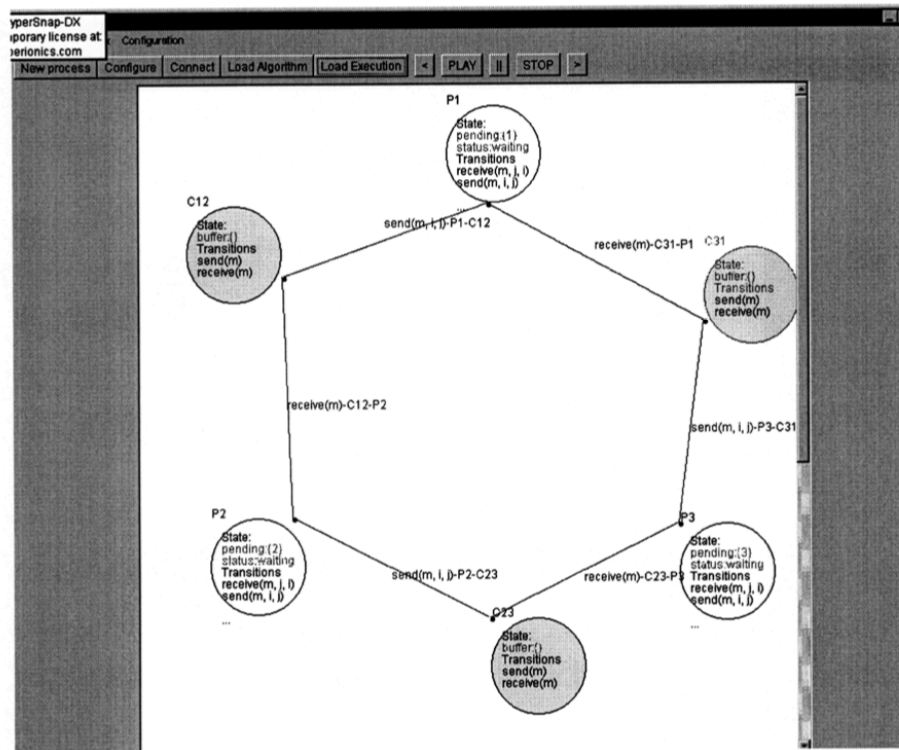


Figura 3: Ejemplo de funcionamiento del simulador DAVE.

Fuente: [McGinnis(1998)].

Por último, una de las desventajas y mayores problemas de DAVE es el factor de tiempo de ejecución. Ya una vez desarrollado, se hicieron pruebas y se obtuvieron tiempos notoriamente lentos. A pesar de haberse utilizado un computador no muy potente, se obtuvieron tiempos que desmotivaban el uso del programa y afectaban claramente la experiencia de usuario, siendo un problema sin resolver y con múltiples posibles causas, tal es como el mismo lenguaje utilizado o por tener un código mal optimizado.

<sup>6</sup>Lenguaje de programación de propósito general, orientado a objetos.

<sup>7</sup>La *Abstract Window Toolkit*, una librería de Java lanzada en 1995 para hacer interfaces de usuario y que hoy en día se encuentra en gran parte reemplazada por la librería *Swing*.

### 1.8.2. A New Tool for the Simulation And Visualization of Distributed Algorithms

En [Gruner(2000)] se presentó una nueva propuesta para simular y visualizar algoritmos distribuidos. El enfoque y el problema que los creadores quisieron cubrir con esta herramienta es, por una parte, la necesidad de los investigadores para visualizar algoritmos distribuidos durante la fase de diseño y *debugging*, así como al momento de presentarlos, considerando que pueden existir problemas de grafos los cuales no son determinísticamente<sup>8</sup> solucionables por computación distribuida. Por otro lado, está la necesidad ya comentada en otros puntos, donde los profesores puedan explicar más en detalle los algoritmos distribuidos, y los estudiantes sean capaces de manipularlos y experimentar con ellos, pudiendo ver los resultados o el comportamiento de los procesos durante la ejecución en caso de ser necesario.

Con respecto a esta aproximación, es importante aclarar primeramente que solamente es un concepto propuesto para implementar un simulador, y solo se hicieron experimentos con un prototipo<sup>9</sup> de este aplicado a algoritmos distribuidos específicos; es decir, nunca fue implementado completamente. Considerando esto, es posible analizar las funcionalidades de la propuesta a través de su interfaz (Ver figura 4). Así, se comienzan a encontrar funcionalidades similares entre simuladores, donde hay controles para ver el algoritmo, pudiendo reproducir o pausar, además de identificar cada nodo y las operaciones que ocurren en cada enlace.

Este prototipo fue desarrollado en *Java*, y su arquitectura está compuesta por un visualizador o interfaz, la programación del algoritmo distribuido, y el simulador, el cual se encarga de hacer la conexión entre el algoritmo y el visualizador. Finalmente, se encontraron algunas inconsistencias en las pruebas que se hicieron, demostrando que aún el prototipo estaba recién empezando a ser implementado y quedando una gran cantidad de trabajo por delante para poder generalizarlo a algoritmos con grafos más grandes y que requieran mayor cantidad de recursos. En consecuencia, no es posible que sea utilizado e integrado a una clase.

### 1.8.3. VADE: Visualization of Algorithms in Distributed Environments

Así como se explica en [Moses(1998)], VADE nace de la necesidad de un programa que asista al momento de diseñar algoritmos, en el proceso de *debugging*, y al enseñar distintos algoritmos distribuidos a los estudiantes. Todo esto debido a la dificultad que hay de entender estos, considerando la comunicación entre procesos y la sincronización<sup>10</sup> entre ellos.

Dada la dificultad anterior es que se propuso este simulador, diseñado para que programadores puedan escribir animaciones de una forma rápida y eficaz, haciendo uso de las librerías de VADE. Analizando la interfaz de este (Ver figura 5), además de encontrarse nuevamente

---

<sup>8</sup>Modelo en donde, dada una misma entrada, siempre se producirá la misma salida, invariablemente.

<sup>9</sup>Representación limitada y no completa de un *software*, que permite probarlo en situaciones reales o explorar su uso.

<sup>10</sup>Forma de coordinar procesos que se ejecutan de forma paralela.

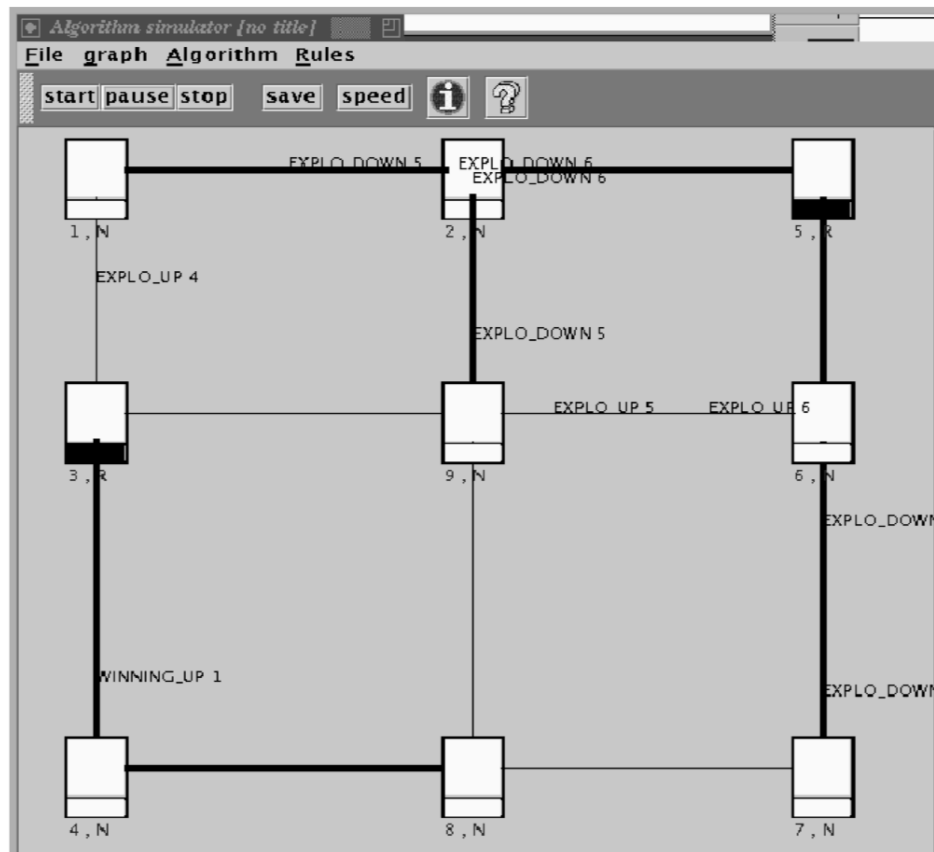


Figura 4: Ejemplo de funcionamiento de *A New Tool for the Simulation And Visualization of Distributed Algorithms*.

Fuente: [Gruner(2000)].

los controles para reproducir y pausar, se ven nuevas funciones como la capacidad de seleccionar un nodo líder para la ejecución del algoritmo. Por otra parte, se ve la información que el programa entrega, donde es posible apreciar los nodos, sus estados, los mensajes circulando alrededor del grafo, entre otros. Con esto, a pesar de presentar una interfaz confusa, se logra representar exitosamente toda la información necesaria para entender el funcionamiento del algoritmo.

A diferencia de otras aproximaciones, VADE fue creado con una arquitectura diferente: toda la ejecución del algoritmo es realizada en un servidor, mientras que toda la visualización de este es vista desde el navegador del cliente, a través de una *Applet Java*<sup>11</sup>. De esta manera, se ofrecen beneficios como la protección del código del algoritmo distribuido, además de asegurar un costo de comunicación bajo, al solo enviar operaciones de alto nivel<sup>12</sup> a través de la red. Junto con esto, se presenta un *framework* con un gran grado de accesibilidad y

<sup>11</sup>Aplicación pequeña, escrita en *Java*, capaz de ser incrustada en una página web, siendo descargada y utilizada al momento de visitarla.

<sup>12</sup>Las operaciones de alto nivel son procedimientos que son fácilmente entendibles por la capacidad cognitiva humana, englobando procesos con gran complejidad pero representándose de una manera simple.

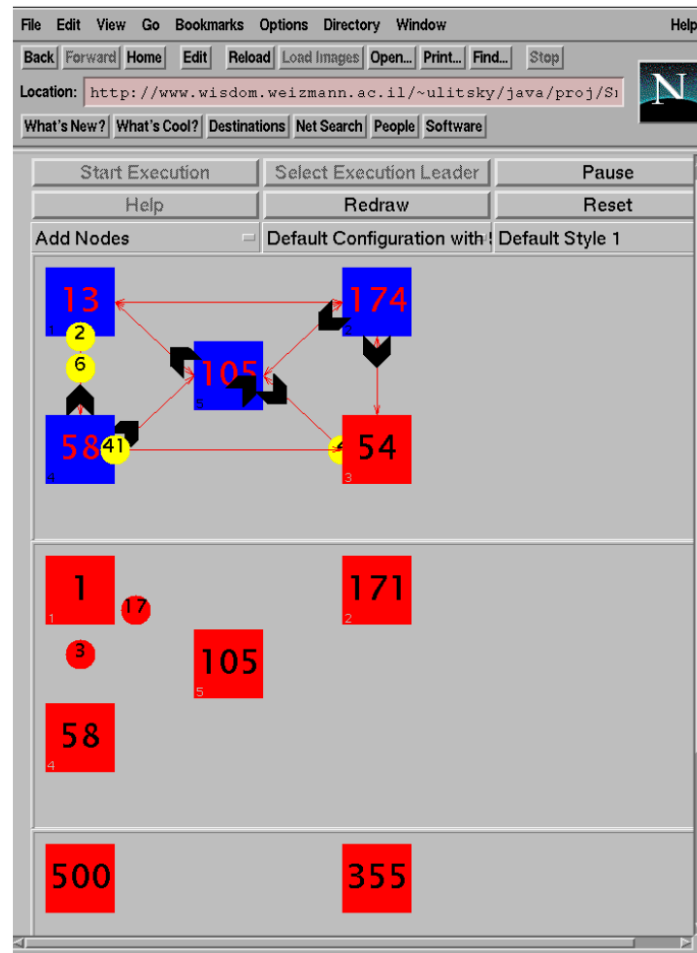


Figura 5: Ejemplo de funcionamiento del simulador VADE.  
Fuente: [Moses(1998)].

flexibilidad, mientras que, en consecuencia de todas sus características, permite incrustar animaciones a través de documentos en línea. Sin embargo, a pesar de tener ciertas particularidades innovadoras, lo cierto es que hoy en día las *Applets Java* se encuentran obsoletas, y han sido reemplazadas por nuevas tecnologías que se encuentran mayormente optimizadas, sin considerar el hecho de que no se provea el *software* y sus librerías para hacer uso de este.

#### 1.8.4. Toolkit DAJ: Distributed Algorithms in JAVA

Como último caso importante, se encuentra en [Schreiner(2002)] una nueva propuesta al área de Sistemas Distribuidos. El *toolkit*<sup>13</sup> DAJ nació de la dificultad que tienen los profesores para enseñar algoritmos distribuidos y su programación. Más aún, por la falta de herra-

<sup>13</sup>Conjunto de herramientas que buscan ayudar y agilizar el proceso de desarrollo de una tarea específica.

mientas de fácil acceso y accesibilidad independiente de la plataforma para implementar algoritmos enseñados en clases e investigar y analizar su comportamiento de una forma dinámica e interactiva. Adicional a lo último, a pesar de existir herramientas, estas son más complejas de implementar e impiden la observación simple del algoritmo en sí.

Es en efecto de lo anterior que se propuso el *toolkit DAJ*, el cual provee un *framework* para programar de forma simple e intuitiva algoritmos distribuidos, y visualizar su comportamiento de forma dinámica. Analizando un ejemplo de su funcionamiento (Ver figura 6), se ve la gran cantidad de información que este entrega. Junto con tener controles para avanzar paso por paso en el algoritmo, se ve una representación más intuitiva de los nodos. Sus estados están representados por colores, de la misma forma que se hace con la transmisión de mensajes, teniendo la particularidad de poder obtener más información del nodo al posicionarse el usuario encima de este, y reposicionar los elementos de la interfaz a través de la interacción directa con ellos.

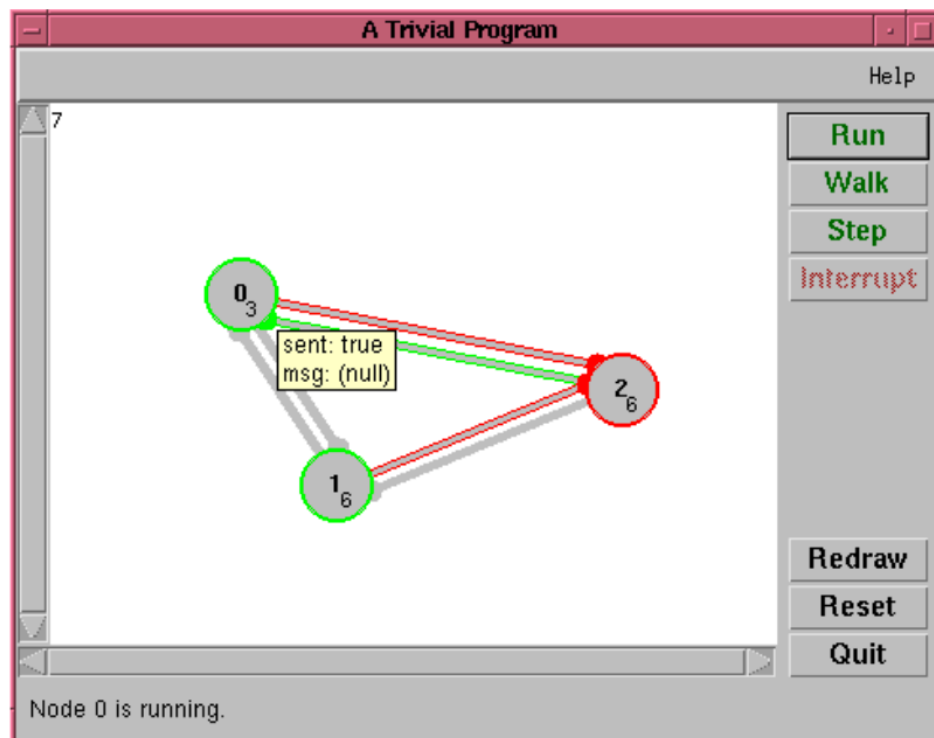


Figura 6: Ejemplo de funcionamiento del *Toolkit DAJ*.  
Fuente: [Schreiner(2002)].

Con respecto a la implementación de *DAJ*, este fue desarrollado utilizando el lenguaje *Java* y la librería de gráficos *AWT*. En cuanto a su uso, presenta modalidades de ejecución en el escritorio, o en una página web a través de una *Applet Java*, similar a como lo hace *VADE*. Así, parece ser una buena herramienta para simular y visualizar algoritmos distribuidos, debido a todas las facilidades que provee, habiendo sido implementados aproximadamente 15 algoritmos como casos de prueba del *toolkit*, lográndose resultados exitosos. A pesar de esta

buena propuesta, sigue ocurriendo el mismo problema con las demás alternativas, como lo es el hecho de utilizar tecnologías obsoletas que no se adaptan a los recursos que hoy en día se tienen.

#### **1.8.5. Características similares entre las competencias**

A modo de cierre entre todas las alternativas analizadas, es posible identificar y tener en cuenta ciertos rasgos similares en cada una de ellas. La principal es que todos se encuentran programados en *Java*, generalmente con la librería de gráficos *AWT*, para asegurar la ejecución del programa independientemente de la plataforma. Además, se destaca lo esenciales que fueron considerados en cada propuesta los controles para reproducir, detener y avanzar paso por paso durante la visualización del algoritmo distribuido, cubriendo la necesidad de analizarlos en profundidad, de forma interactiva y dinámica. Por último, se hace importante notar lo bien que debe estar construida la interfaz para poder representar toda la información posible del algoritmo, danado paso a buenos y malos visualizadores.

## CAPÍTULO 2

### MARCO CONCEPTUAL

Al momento de introducirse en el área de simulación y visualización de algoritmos distribuidos, se hace importante reconocer todos los factores que influyen en la aplicación de esta, analizando los diferentes aspectos para poder cubrir todos los casos posibles a través del *framework* a crear.

#### 2.1. Simulación de Algoritmos Distribuidos

Basándose en [Attiya(2004)], para simular un algoritmo distribuido se deben considerar todos los factores que esto comprende:

- Estructura
- Funcionamiento
- Comunicación
- Aplicaciones

##### 2.1.1. Estructura

Un algoritmo al ser distribuido implica un trabajo en conjunto dentro de una red a través del envío de mensajes. Así, la composición de este estará formada por:

- **Nodos:** Serán los puntos, sitios, o procesos de la red que ejecutan la serie de instrucciones que componen al algoritmo y se comunican entre sí. Más a bajo nivel, corresponden a procesos monohebra concurrentes, es decir, procesos concurrentes, donde cada uno se ejecuta de forma secuencial. Generalmente se diferencian a través de identificadores únicos, con la característica de que siempre habrá al menos un nodo, denominado **iniciador**, quien se encargará de comenzar el algoritmo y desencadenar eventos en los demás nodos, en caso de corresponder, a través de mensajes. Por último, cada nodo tendrá un **estado**, el cual indicará en qué fase del algoritmo se encuentra, y cambiará de acuerdo a eventos como la recepción o envío de mensajes.
- **Arcos:** Serán los canales encargados de conectar a los nodos entre sí, y a través de los cuales viajarán los mensajes enviados. Estos se subdividen en dos tipos:

- **Unidireccionales:** Son canales que pueden transmitir mensajes en un solo sentido, es decir, desde un nodo A a un nodo B, pero no al revés.
- **Bidireccionales:** Son canales que pueden transmitir mensajes en ambos sentidos, por lo que los dos nodos conectados a través este canal pueden comunicarse entre sí. Se consideran también como dos canales unidireccionales con misma dirección y distinto sentido.

### 2.1.2. Funcionamiento

Un algoritmo distribuido, además de ser una red de nodos que ejecutan un mismo conjunto de instrucciones, posee una cualidad fundamental: la **conurrencia**, como se hace ver en [Tel(2000)]. Esto quiere decir que cada punto de la red ejecutará el algoritmo al mismo tiempo y de forma asincrónica, dependiendo su comportamiento solamente de los mensajes entrantes. Con esto, es posible observar que en cada ejecución del mismo programa el orden de los eventos puede variar debido a la aleatoriedad en que estos ocurren, donde, por ejemplo, un mensaje puede tardar más en entregarse que otro, o un nodo en comenzar a trabajar; todo esto considerando que la lógica del algoritmo no fuerce a un grado de sincronismo global en la red por algún motivo. Además, considerando la lógica de los distintos algoritmos, suelen encontrarse el uso de **parámetros** iniciales los cuales cambian el comportamiento de estos, tales como el nodo que inicia el algoritmo, valores iniciales de variables, entre muchos otros, dándole a la ejecución la cualidad de ser **no determinista**.

Por otro lado, se identifican dos tipos de eventos de los cuales depende el flujo de ejecución del programa en cada nodo, tal como se menciona en [Monge(2017)]:

- **Algoritmo principal:** Serán las instrucciones que cada nodo ejecutará al comenzar el programa. Generalmente en este se inicializan variables y, en caso de los nodos iniciadores, se envían los primeros mensajes.
- **Recepción de mensajes:** Este conjunto de instrucciones será el algoritmo que se ejecutará cada vez que un nodo recibe un mensaje. Es completamente independiente del algoritmo principal, y constituye generalmente los momentos de cambio en el nodo, ya sea de variables o estado.

### 2.1.3. Comunicación

Tal como se ha mencionado anteriormente, los nodos en un algoritmo distribuido se comunican a través del envío y recepción de mensajes. En cuanto a esto, pueden encontrarse tres formas de comunicación:

- **Unicast:** Corresponde al envío de un mensaje desde un nodo a solo uno de sus vecinos.

- **Multicast:** Corresponde al envío de un mensaje desde un nodo a varios de sus vecinos, pero no a todos. Se ven casos como el envío de un mensaje a todos menos al vecino desde donde se recibió un mensaje anterior, o a los vecinos que cumplen con una condición específica, entre otros.
- **Broadcast:** Corresponde al envío de un mensaje desde un nodo a todos los vecinos de este en la topología de red. Se usa generalmente para difundir un mensaje a lo largo de la red o anunciar el término del algoritmo.

#### 2.1.4. Aplicaciones

A pesar de haber varias aplicaciones en los que se usan algoritmos distribuidos, como en [Fokkink(2013)], [Lynch(1996)] y [Englert(2006)], es posible clasificar cada una de ellas en los siguientes tipos:

- **Recorrido:** Su objetivo es recorrer todos los nodos dentro de la red. Se utiliza generalmente para generar un árbol de recorridos a lo largo de la red.
- **Elección:** Su objetivo es elegir un único nodo representante o coordinador de la red. Se utiliza, por ejemplo, en coordinaciones al recuperarse de fallas, elección de un nodo principal en la red, entre otros.
- **Instantánea distribuida:** Su objetivo es obtener un estado global consistente del algoritmo, esto es, obtener una imagen de este en un momento dado reflejando todos los eventos y cambios que han ocurrido, siempre respetando el orden de ellos.
- **Detección de término:** Su objetivo es, tal como lo dice el nombre, detectar el término del algoritmo, entendiendo que esto ocurre cuando todos los procesos en cada nodo han terminado y ya no hay mensajes en tránsito a través de los canales.
- **Sincronización distribuida:** Su objetivo es asegurarse que una sección de instrucciones se ejecute sincrónicamente, es decir, por solo un nodo a la vez a lo largo de la red. Esta sección a ejecutar se denomina *sección crítica* y se controla generalmente a través de un elemento denominado *token*, el cual es único y se envía a través de mensajes, asegurando con esto que un solo nodo lo tenga a la vez.

Cabe destacar que no se considerarán para esta memoria los algoritmos pertenecientes al área de **instantánea distribuida**, debido a que su representación difiere de todas las demás, forma parte de un caso específico y no están hechos para ser simulados en tiempo real como los otros casos.

## 2.2. Visualización de Algoritmos Distribuidos

Una vez ya analizados los factores a tener en cuenta con respecto a la simulación de un algoritmo distribuido, se hace necesario también pensar de qué manera visualizarlo para finalmente entender cómo se está ejecutando y qué cambios están ocurriendo durante su ejecución:

### 2.2.1. Representación

La principal representación de un algoritmo distribuido es a través de un **grafo** (Ver figura 7). A la vez, esta se descompone en varios componentes que indican diferentes factores del mismo algoritmo:

- **Nodo:** Cada nodo dentro del algoritmo se representa como un **círculo** en el grafo. Adicionalmente, dentro de este se suele agregar un **texto corto** con el fin de rotular el nodo y diferenciarlo de los demás, generalmente, como se analizó anteriormente, a través de un identificador numérico único o un número dinámico que cambia de acuerdo a la lógica del algoritmo y sirve para mostrar información relevante de forma rápida.

Por otro lado, también influye lo que es el **color** del nodo, el cuál será el encargado de indicar el estado en que se encuentra. En consecuencia, cada vez que un nodo cambie de color, significará que este cambió de estado.

Por último, está el caso de los nodos iniciadores. Para estas situaciones, habrá un rol especial donde se utilizará una **circunferencia** de diferente color al borde del nodo, la cual indicará que este es uno de los que inician el algoritmo.

- **Arco:** Cada arco o canal dentro del algoritmo se representa como una **línea recta** conectando dos nodos. El sentido de este estará determinado por la punta de una **flecha** o, en caso de ser bidireccional, esta se omitirá. Como caso alternativo, este último caso también puede representarse como dos canales con las flechas apuntando a sentidos diferentes.

Como un factor adicional, el **color** del arco puede usarse para representar **estados** del canal, tales como cuando tiene mensajes en tránsito, cuando se encuentra desocupado, o para un uso personalizado de acuerdo a lo que se quiera representar dentro del algoritmo.

- **Etiquetas:** Por último, es importante hacer notar casos en donde no es posible representar toda la información del algoritmo en los factores mencionados anteriormente. Para estas situaciones, lo que generalmente se hace es utilizar etiquetas o agregar texto adicional junto a los nodos y/o arcos, indicando datos tales como el valor actual de variables, identificar qué tipo de mensaje se está enviando y hacia dónde, entre otros.

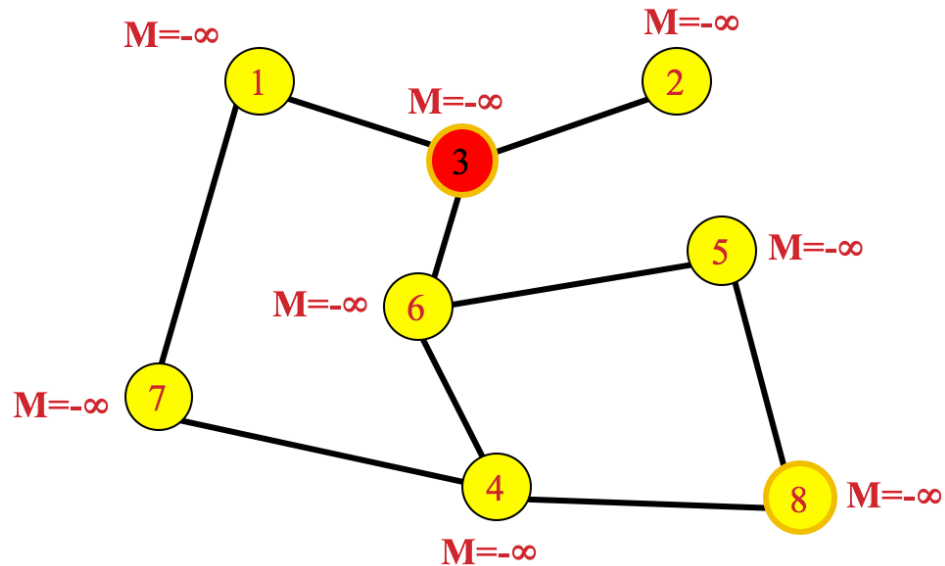


Figura 7: Ejemplo de la representación de un algoritmo distribuido.  
Fuente: [Monge(2017)].

### 2.2.2. Topologías

Considerando todas las aplicaciones de algoritmos distribuidos que existen, es posible identificar tres topologías de redes en común entre ellos:

#### 2.2.3. Anillo

En una topología de anillo (Ver figura 8) se presenta un orden de nodos que forman un **ciclo**. A la vez, este último tiene la característica de ser un ciclo **dirigido**, es decir, los nodos se encuentran conectados en un solo sentido. Así, el fin de esto suele ser enviar mensajes a lo largo de toda la red basándose en el **principio de extinción de mensajes**, es decir, que a medida que transcurra el algoritmo solo el mensaje que cumpla con una condición prevalecerá sobre los demás e indicará el término del algoritmo.

#### 2.2.4. Árbol

En una topología de árbol (Ver figura 9) se presenta un orden de nodos donde, haciendo la analogía con un árbol, existen **hojas** y una **raíz**. La raíz será un único nodo el cual no tiene nodos que lo preceden, es decir, puede considerarse como el origen de la red. Por otro lado, las hojas serán los nodos al extremo de la red y que marcan su fin, dicho de otro modo, no tienen nodos que los sigan.

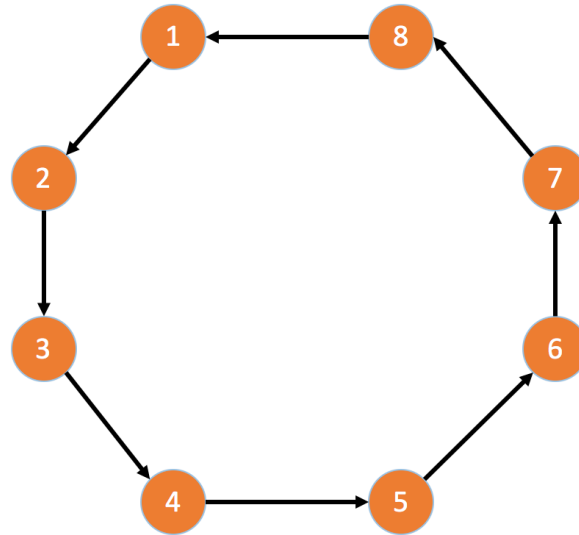


Figura 8: Ejemplo de topología de anillo en un algoritmo distribuido.  
Fuente: Elaboración Propia.

#### 2.2.5. Grafo General

En una topología de grafo general (Ver figura 10) se presentan todas las topologías que no coinciden con las anteriores mencionadas. Así, pueden encontrarse casos como un grafo donde todos sus nodos están completamente conectados entre sí, dentro de muchos más.

### 2.3. Algoritmos Distribuidos y su aprendizaje

Luego de analizar cada uno de los componentes que influyen a la hora de confeccionar un algoritmo distribuido, se hace imprescindible estudiar su aplicación en la educación, tema central de este trabajo. Y es así donde se comienza a ver la complejidad que conlleva el aprendizaje en esta área, considerando la importancia de:

- Visualización
- Dinamicidad
- Variabilidad

#### 2.3.1. Visualización

En primer lugar, un algoritmo distribuido debe poder ser **visualizado**. Sin su visualización, el aprendizaje no se puede lograr debido a que no habría otra forma de comprender qué

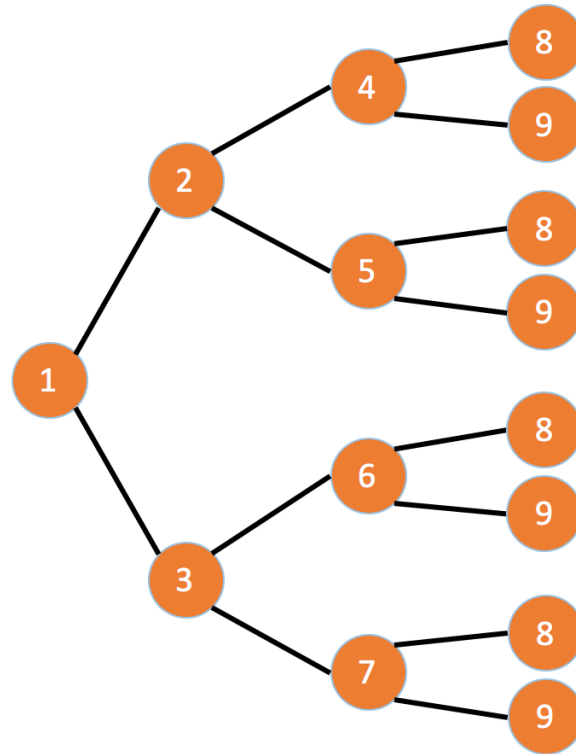


Figura 9: Ejemplo de topología de árbol en un algoritmo distribuido.

Fuente: Elaboración Propia.

está ocurriendo durante su ejecución, a pesar de tener una lógica ya conocida. Además, esta debe ser lo suficientemente representativa para poder mostrar toda la información que el algoritmo conlleva, tomando en cuenta todas las variables, nodos, mensajes, arcos, estados, entre otros, que pueden llegar a utilizarse.

### 2.3.2. Dinamicidad

Otro factor prioritario a la hora de comprender un algoritmo distribuido es su **dinamicidad**. Esto quiere decir que no basta con conocer la representación de este en un determinado momento de su ejecución, ya que está en un constante cambio a medida que transcurre el **tiempo**. En consecuencia, esta deberá ser también capaz de dar a conocer el estado del algoritmo en todo segundo, entregando junto con esto la posibilidad de detenerse en ciertos tiempos para analizar los pasos que se están haciendo y cómo se está comportando el programa en general, además de conocer su estado y variables.

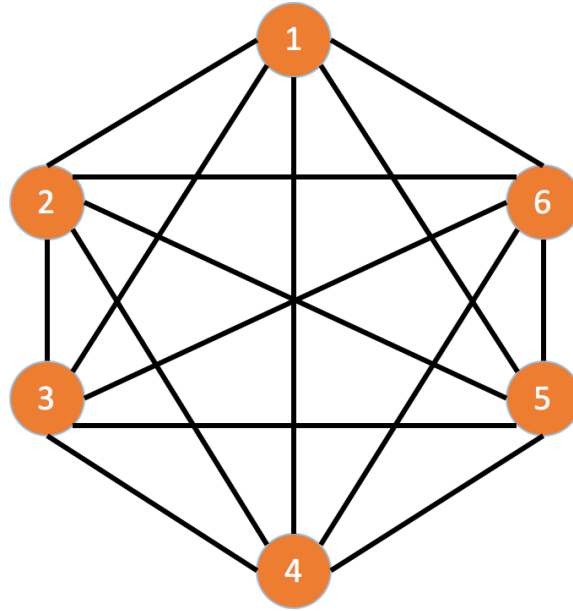


Figura 10: Ejemplo de topología de grafo general en un algoritmo distribuido.  
Fuente: Elaboración Propia.

### 2.3.3. Variabilidad

Un último factor que marca la diferencia durante el aprendizaje de un algoritmo distribuido es su **variabilidad** que es lo que le da la cualidad de ser **no determinista**. Si bien la lógica de un algoritmo ya está dada y es siempre la misma, pueden haber casos en donde su comportamiento esté determinado de acuerdo a **parámetros** establecidos inicialmente. Esto significa que no basta con poder representar y visualizar su estado en todo momento, sino que también se deben poder reflejar los cambios en este a partir de las condiciones que se establezcan en su inicio. Esto aplica también al grafo o **red de nodos** que se vaya a utilizar, el cual también puede variar en cada ejecución, siempre manteniendo las mismas instrucciones a ejecutar. Por último, incluso si se mantuviera todo lo anterior constante, esta característica estará siempre presente a través de los cambios en la **velocidad de ejecución**, lo cual puede verse afectado por la aleatoriedad del tiempo que cada mensaje demorará en viajar o el tiempo que tomará cada nodo en comenzar a ejecutar el algoritmo.

## CAPÍTULO 3

### PROPUESTA DE SOLUCIÓN

#### 3.1. Diseño conceptual

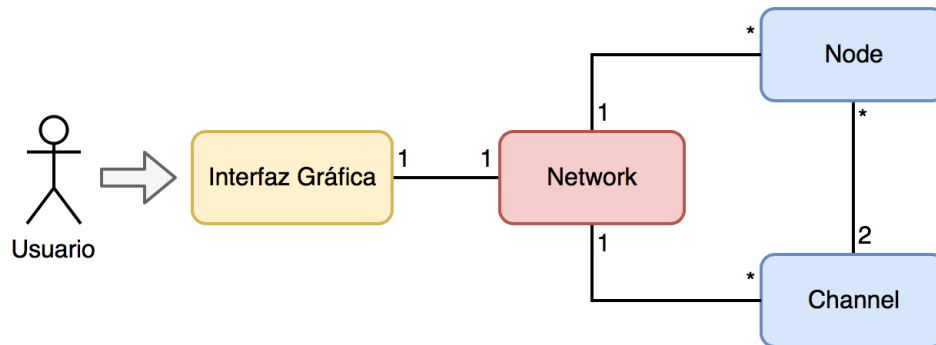


Figura 11: Diagrama de interacción entre el usuario y el *framework*.

Fuente: Elaboración Propia.

Considerando el grado de paralelismo de los algoritmos distribuidos y su relevancia en estos, se propone una solución (Ver figura 11) basada directamente en *threads*, de manera de acercarse lo más posible a una ejecución **paralela**. Así, se presentan los objetos que compondrán la lógica del *framework*:

- Objeto *Node*
- Objeto *Canal*
- Objeto *Network*
- Interfaz Gráfica
- Modelo de Comunicación
- Técnica de Interrupción

##### 3.1.1. Objeto Node

El objeto *Node* será el cual representará a cada nodo dentro de la red. Así, este contendrá una serie de componentes que le otorgarán todas las características necesarias para funcionar correctamente (Ver figura 12):

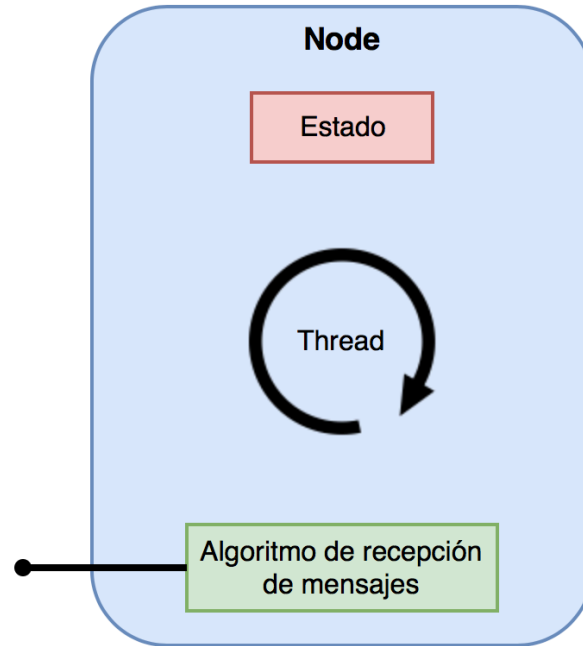


Figura 12: Modelo del objeto *Node*.

Fuente: Elaboración Propia.

- **Estado:** Será todo lo que forme parte del estado interno del nodo, esto es, su estado lógico dentro del algoritmo, su nombre, posición, si es iniciador o no, entre otros.
- **Thread:** Será el conjunto de instrucciones perteneciente al **algoritmo principal** del nodo, las cuales se ejecutarán como un proceso independiente. (Ver figura 13).
- **Algoritmo de recepción de mensajes:** Será el conjunto de instrucciones que se ejecutarán cada vez que se reciba un mensaje en el nodo. Generalmente, esto provocará cambios en el nodo modificando su **estado**.

### 3.1.2. Objeto Canal

De manera similar a la anterior, el objeto *Channel* será el cual representará a cada canal dentro de la red. En consecuencia, contendrá una serie de componentes que lo conformarán (Ver figura 14):

- **Estado:** Será todo lo que forme parte del estado interno del canal, esto es, si está con mensajes en tránsito, su posición, entre otros.
- **Thread:** Será el proceso independiente encargado netamente de **entregar mensajes**, agregando un tiempo de espera aleatorio antes de pasárselos al nodo de destino (Ver

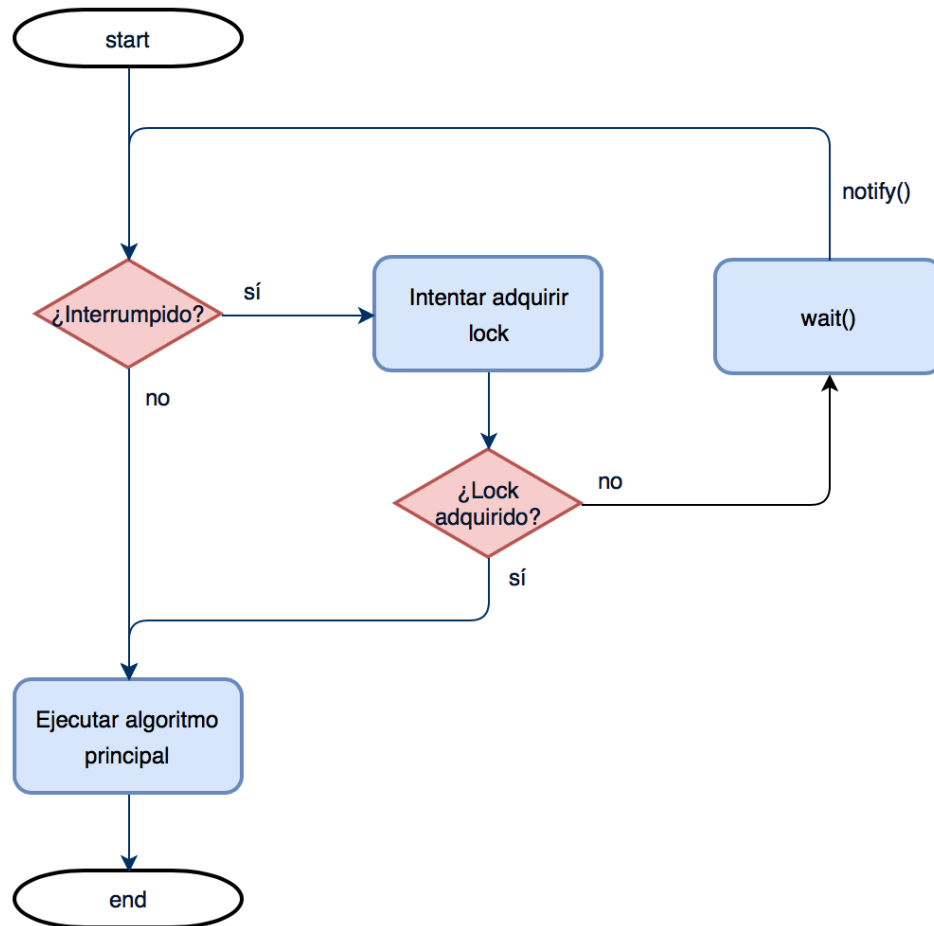


Figura 13: Diagrama de flujo del *thread* en *Node*.  
Fuente: Elaboración Propia.

figura 15), para luego llamar al algoritmo de recepción de mensajes en este. Cabe destacar que se considerarán todos como canales unidireccionales, por lo que arcos bidireccionales serán representados como dos de estos, pero con distinto sentido.

- **Cola de mensajes:** Será la cola donde se irán acumulando los mensajes a entregar, y desde la cual el *thread* del canal los irá enviando.

### 3.1.3. Objeto Network

El objeto *Network* será el que representará a la red de nodos y, por lo tanto, contendrá a los dos objetos anteriores (Ver figura 16). Su función será coordinar a estos, y contendrá lo siguiente:

- **Estado:** Será el estado de la ejecución del algoritmo, esto es, si está interrumpido,

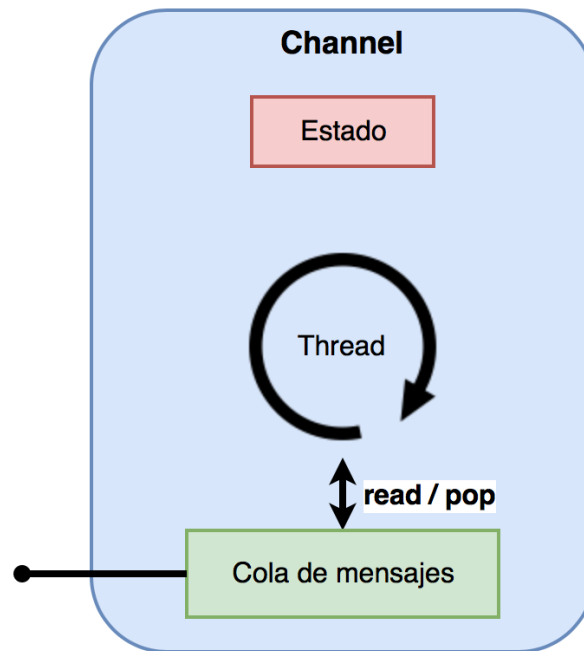


Figura 14: Modelo del objeto *Channel*.

Fuente: Elaboración Propia.

finalizado o ejecutándose.

- **Nodos:** Serán los nodos pertenecientes a esta red.
- **Canales:** Serán los canales que conectan a los nodos pertenecientes a esta red.
- **Thread:** Será el proceso principal que se encargará de iniciar el algoritmo, considerando nodos y canales (Ver figura 17), y por otro lado, de crear un proceso independiente para detectar el término del algoritmo.

#### 3.1.4. Interfaz Gráfica

La interfaz gráfica del programa, parte fundamental para la visualización del algoritmo, tendrá las siguientes funcionalidades (Ver figura 18):

- **Vista del algoritmo:** Será la sección donde se visualizará directamente el funcionamiento del algoritmo, analizando los nodos, arcos, sus estados, los cambios en el tiempo, entre toda la otra información que se pueda obtener de este.
- **Parámetros:** Será la sección donde el usuario pueda dar valor y modificar parámetros iniciales que influyan de alguna manera en la ejecución del algoritmo de forma rápida.

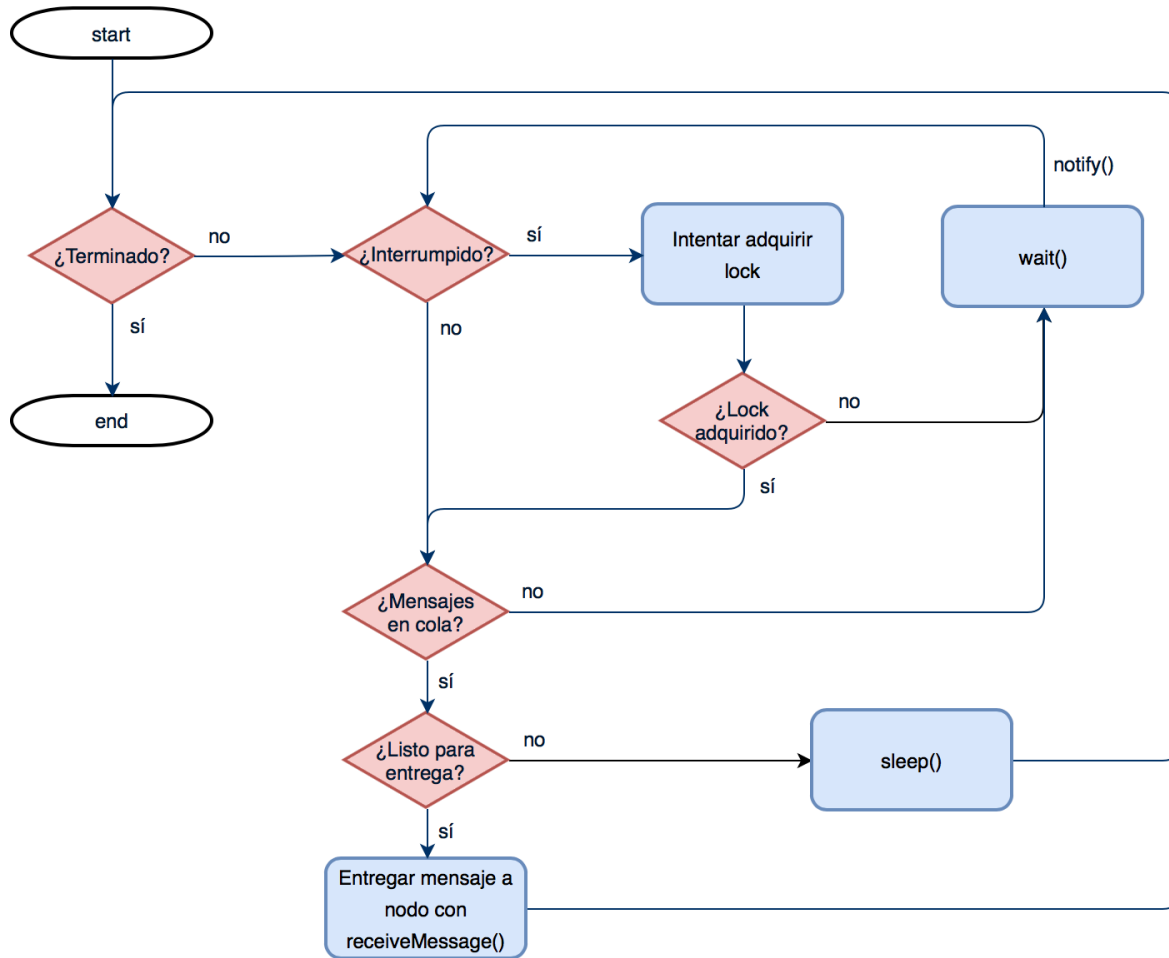


Figura 15: Diagrama de flujo del *thread* en *Channel*.

Fuente: Elaboración Propia.

- **Barra de estado:** Será la sección que indique información de acuerdo al estado de ejecución del algoritmo, u otra que sea importante mencionar.
- **Menú:** Será la sección con todos los botones de acción que el usuario tendrá a disposición. Se encontrarán los siguientes:
  - **Run:** Corresponde al botón para iniciar o resumir la ejecución del algoritmo.
  - **Step:** Corresponde al botón para avanzar en la ejecución del algoritmo solo por una cantidad fija de instrucciones para luego detenerse nuevamente.
  - **Interrupt:** Corresponde al botón para interrumpir la ejecución del algoritmo.
  - **Reset:** Corresponde al botón para reiniciar la ejecución del algoritmo, volviendo todo a su estado inicial.
  - **Settings:** Corresponde al botón para ingresar a la sección de configuración del programa. El fin de esta es poder personalizar las redes a utilizar. Esto compren-

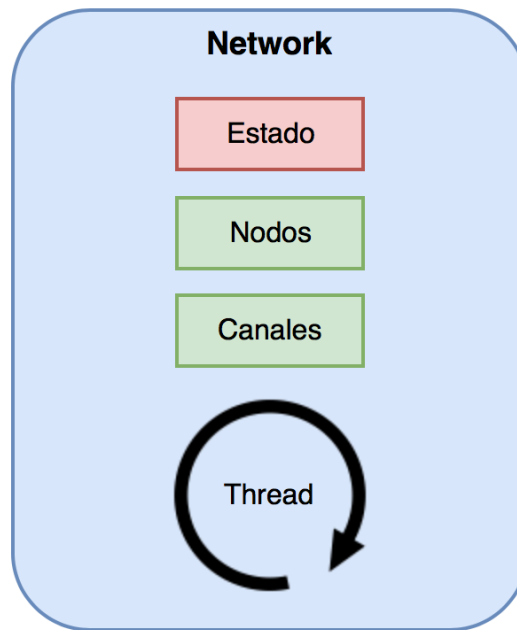


Figura 16: Modelo del objeto *Network*.  
Fuente: Elaboración Propia.

de poder crear, importar y exportar, además de modificarlas agregando o eliminando nodos o canales.

- **Exit:** Corresponde al botón para salir del programa.

### 3.1.5. Modelo de Comunicación

Una vez analizados los objetos principales del *framework* a implementar, es importante ver ahora cómo se comunicarán. Así, viendo la figura 19, es posible ver las relaciones entre ellos, destacando que el usuario solo interactuará con la interfaz gráfica al momento de la ejecución, mientras que esta se encargará de comunicarse con el algoritmo y modificarlo cuando corresponda, considerando su lógica, estado, entre otros, para a la vez representarlo en tiempo real.

### 3.1.6. Técnica de Interrupción

Al considerar un programa que se ejecuta paralelamente con múltiples *threads*, una de las grandes interrogantes que surgen es qué técnica utilizar para lograr interrumpir cada uno de ellos de forma eficiente, con la condición de que luego se puedan resumir. En otras palabras, al no haber un modelo sincrónico, no es posible detener la ejecución en un solo proceso ya que hay múltiples de estos, y cada uno ejecutando una instrucción diferente de forma

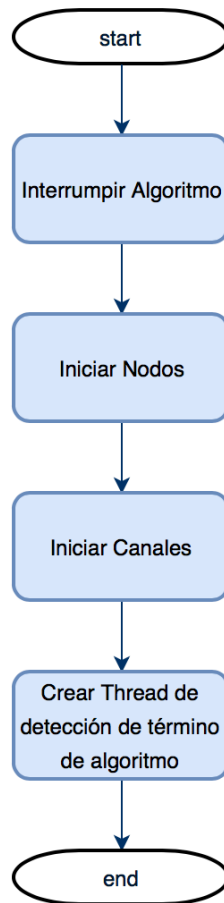


Figura 17: Diagrama de flujo del *thread* en *Network*.

Fuente: Elaboración Propia.

independiente. Es por esto que se decidió establecer en el diseño del *framework* **puntos de interrupción**. Así, cada vez que un *thread* pase por estos puntos, este deberá revisar si debe interrumpirse la ejecución o no.

Si bien un algoritmo distribuido puede tener una gran cantidad de componentes diferentes en su lógica, aún es posible identificar ciertas funciones en común que serán imprescindibles en ella, y que por lo tanto son las más indicadas para agregar puntos de interrupción:

- **Enviar mensaje:** Cada vez que un nodo deba enviar un mensaje se llamará a una función específica, convergiendo su ejecución a este punto de interrupción.
- **Recibir mensaje:** Cada vez que un nodo reciba un mensaje se llamará a una función específica que lo consuma de acuerdo a la lógica del algoritmo, siendo otra instancia para agregar un punto de interrupción.
- **Cambiar de estado:** Cada vez que un nodo cambia de estado significa un cambio y un

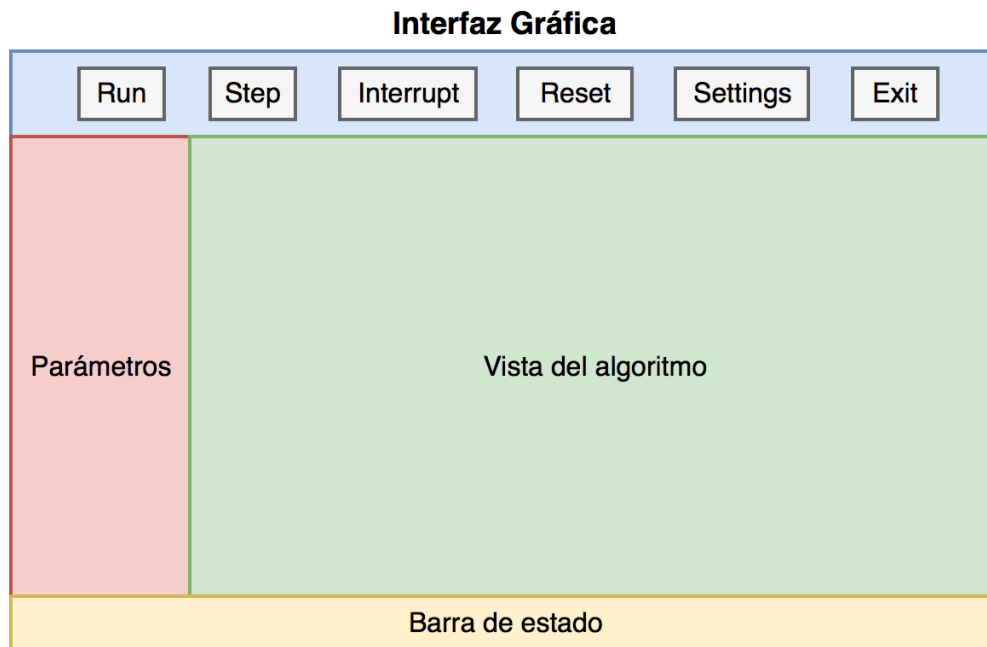


Figura 18: Esquema de la interfaz gráfica del *framework*.

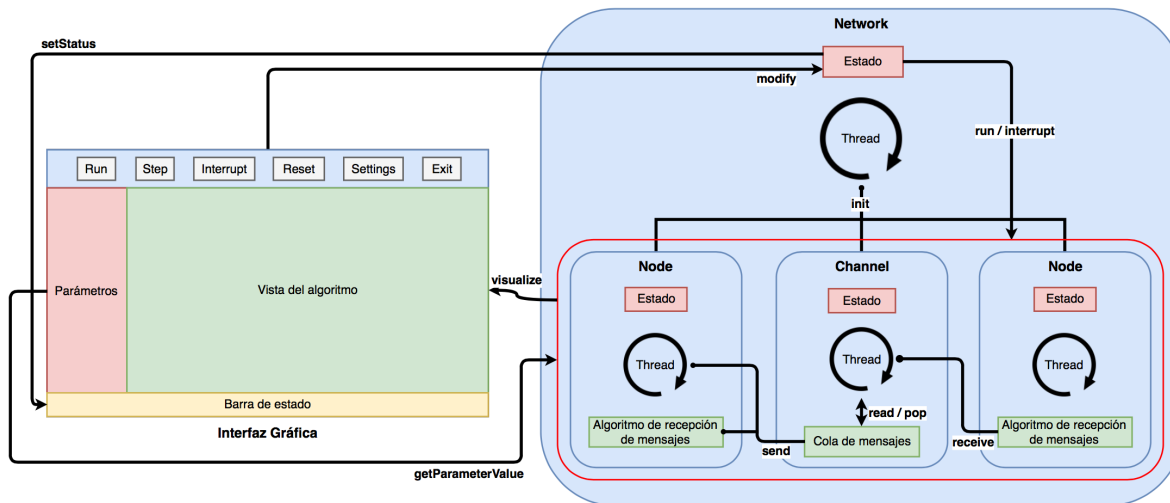
Fuente: Elaboración Propia.

paso importante en la ejecución del algoritmo, por lo que se hace necesario interrumpirse en caso de que el usuario lo requiera.

- **Actualizar nombre de nodo:** Cada vez que un nodo cambia su nombre durante la ejecución del algoritmo también significa un cambio en su ejecución, agregándose en consecuencia otro punto de interrupción en este lugar.
- **Inicio de cada sección de la ejecución:** Por último, tal como se puede apreciar en los diagramas de flujo explicados anteriormente, en el inicio de ejecución para los nodos y arcos también se agregarán puntos de interrupción, siendo también parte importante del algoritmo.

### 3.2. Ambiente de programación

Al momento de elegir el ambiente de programación para este trabajo, deben considerarse algunos factores fundamentales para su desarrollo (Ver tabla 2). En primer lugar, debe ser un lenguaje de alto nivel lo suficientemente **intuitivo** para simplificar el proceso de programación por parte del usuario que vaya a hacer uso del *framework*, de manera que se enfoque más en el código del algoritmo distribuido a implementar que en el cómo hacerlo. Junto con lo anterior, este debe también ser **eficiente**, es decir, tiene que ser capaz de optimizar la suficiente cantidad de recursos para que la simulación y visualización del código sea fluida, sin agregar distractores que puedan dificultar la comprensión de lo que está ocurriendo. Por

Figura 19: Modelo de comunicación del *framework*.

Fuente: Elaboración Propia.

último, una cualidad no menor a tener en cuenta es la **compatibilidad** del programa, asegurando que este se pueda ejecutar en cualquier computador sin importar el sistema operativo que este tenga y de forma simple.

Tabla 2: Cuadro comparativo de los lenguajes de programación más conocidos.

Fuente: Elaboración Propia.

|                   | <i>Python</i>   | <i>Java</i>              | <i>C/C++</i>                                                |
|-------------------|-----------------|--------------------------|-------------------------------------------------------------|
| <b>Intuitivo</b>  | Alto Nivel      | Alto Nivel               | Bajo Nivel                                                  |
| <b>Eficiente</b>  | Interpretado    | Compilado e Interpretado | Compilado                                                   |
| <b>Compatible</b> | Multiplataforma | Multiplataforma          | Depende del Sistema Operativo y Arquitectura del Procesador |

En consecuencia de todo lo mencionado anteriormente, y considerando que se tomaron en cuenta solamente los tres lenguajes de programación que más se conocen hoy en día con el fin de simplificar la tarea al estudiante al momento de programar, se llegó a dos alternativas: *Python* y *Java*. Si bien el primero se encuentra cada vez en mayor uso por parte de los usuarios y se usa más en el ámbito científico, lo cierto es que dentro del área generalmente se ha trabajado siempre con el lenguaje de programación *Java*. Esto se justifica, ya que es un lenguaje que lleva ya una gran cantidad de tiempo trabajando con *multithreading*<sup>14</sup> y es relativamente más intuitivo a la hora de programar, junto con ser más robusto al pertenecer, por una parte, a los lenguajes compilados<sup>15</sup>. En conclusión, se determinó por usar este lenguaje para el desarrollo del *framework* en su versión 8.

<sup>14</sup>Habilidad de ejecutar múltiples procesos de forma concurrente o paralela.

<sup>15</sup>Son lenguajes de programación que son traducidos a un lenguaje de bajo nivel a través de un compilador, junto con todas las optimizaciones que esto implica.

Con respecto a la interfaz gráfica, parte esencial de la visualización del programa, si bien están como alternativas las librerías oficiales *Abstract Window Toolkit*, *Swing* y *JavaFX*, se decidió utilizar el último conjunto de herramientas, perteneciente al kit de desarrollo de *Java 8*, debido a que corresponde a la librería más reciente que se está utilizando hoy en día y que cumple con todos los requerimientos necesarios para esta memoria, destacando su cualidad de ser compatible con los sistemas operativos *Windows*, *MacOS* y *Linux*.

Por último, en cuanto al ambiente de desarrollo, se utilizó el IDE<sup>16</sup> *IntelliJ IDEA*, siendo un editor muy completo y fácil de usar a la vez.

---

<sup>16</sup>Es un *Entorno de Desarrollo Integrado* el cual es capaz de proporcionar todas las herramientas necesarias para trabajar con el lenguaje de programación correspondiente.

## CAPÍTULO 4

### IMPLEMENTACIÓN DE LA SOLUCIÓN

En primer lugar, se hace necesario destacar que esta implementación se basó en la solución propuesta en [Schreiner(2002)], de la cual se rescataron algunas técnicas de visualización y su estructura. Sin embargo, la lógica en el *framework* propuesto en esta memoria es totalmente elaboración propia y difiere en su totalidad con respecto al trabajo mencionado.

Junto con lo anterior, se utilizó una metodología de desarrollo *bottom-up*. Así, se comenzó desarrollando la clase principal *App* responsable de controlar la aplicación de *JavaFX*, para luego comenzar a implementar la clase representante de la red *Network*. En consecuencia, al ser esta la encargada de controlar los nodos y canales, se procedió a implementar las clases *Node* y *Channel*, correspondientes a estos respectivamente. Con ello, se implementó la clase *Program*, una extensión de la clase *Node* responsable de ejecutar el algoritmo en cada nodo, para tener finalmente una base con la que se pueda simular un algoritmo distribuido. En consiguiente, se procedió a la implementación de la interfaz gráfica con las clases *Screen* y *AlgorithmView*, logrando una visualización del funcionamiento del algoritmo. Después de hecho todo lo mencionado, se comenzó con un proceso **iterativo**, con el fin de agregar características adicionales o arreglar errores, llevando a la creación de todas las clases explicadas en la sección siguiente.

Por último, se hace notar que para los propósitos de este *framework* solo se considerarán algoritmos distribuidos **sin fallas**, por lo que no habrá en ningún momento pérdida de mensajes.

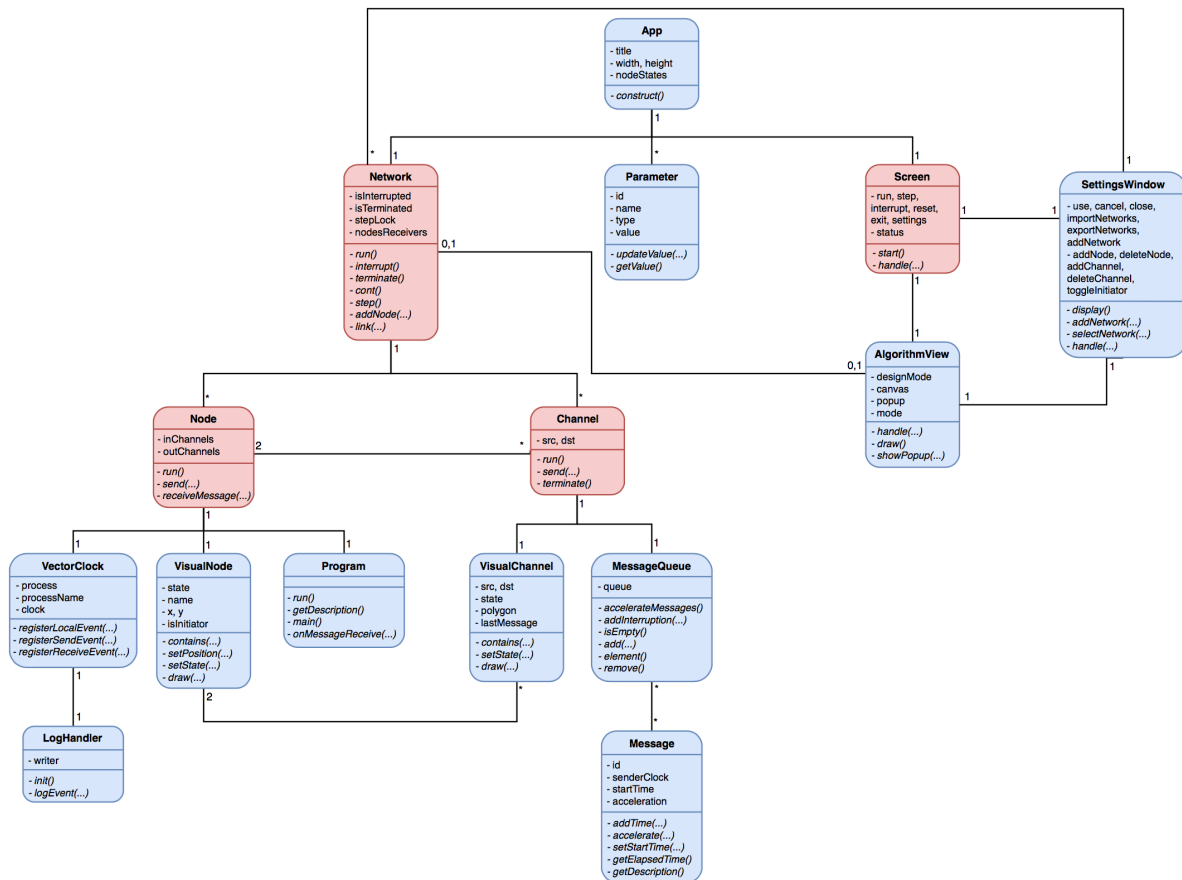
#### 4.1. Clases del Framework

Como punto de partida, es bueno comenzar mencionando las clases implementadas dentro del *framework*. Viendo la figura 20, se cuentan 15 en total, donde las que se encuentran en color rojo son las centrales al ser propuestas en el diseño conceptual del capítulo anterior.

Notando que en el diagrama solo se destacaron los campos y métodos más importantes, a continuación se explica el fin de cada una de las clases:

##### 4.1.1. App

Corresponde a la clase que da inicio a la aplicación de *JavaFX*, esto es, la encargada de instanciar y dar inicio a la red *Network* y a la interfaz gráfica *Screen*. Hereda de la clase *Application* de esta misma librería, y está hecha para ser heredada por el usuario que vaya a utilizar el

Figura 20: Diagrama de relación entre clases del *framework*.

Fuente: Elaboración Propia.

*framework*, indicando el título, ancho y alto del programa a través del método constructor. Además, ofrece el método *construct()*, el cual deberá implementar el usuario para construir la red que va a ejecutar el algoritmo. Así, se entregan métodos dentro de esta misma clase para agregar nodos (*addNode(...)*) y agregar canales (*link(...)*). Por otro lado, se entrega el método *setState(...)* para definir todos los estados lógicos de cada nodo indicando su color, o *getChannelColor(...)* en caso de querer hacer lo mismo pero con el color de los canales. Por último, es aquí donde el usuario agregará a través de *addParameter(...)* todos los parámetros a utilizar en el algoritmo distribuido, y que podrá modificar durante la ejecución del programa.

#### 4.1.2. Parameter

Será la clase que representará a los parámetros del algoritmo. Tendrá un identificador único para acceder a él, un nombre para representarlo en pantalla, el tipo, ya sea un número o un *string* para validar su valor al modificarlo, y el valor actual de este. Se utilizará el método *up-*

*dateValue(...)* cada vez que se modifique su valor al momento de la ejecución. Cabe destacar que solo se permitirá su modificación al inicio del algoritmo, y no una vez ya iniciado.

#### 4.1.3. Network

Será la clase que se describió en el diseño conceptual del capítulo anterior para representar a una red de nodos. Hereda de la clase *Thread*, y será el coordinador de toda la ejecución del algoritmo. Es por esto que tendrá variables que indiquen si la ejecución está interrumpida o no (*isInterrupted*), o si la red se destruyó (*isTerminated*). Esto último ocurre en el caso que se reinicie la ejecución del algoritmo, provocando la creación de una nueva red, y por lo tanto terminando y destruyendo la actual.

El estado de esta instancia, representado en las variables anteriores, es parte vital de la lógica del algoritmo ya que afectará directamente a las ejecuciones de cada nodo y canal, siendo posible verlo en los diagramas del diseño conceptual. Así, para este control, la clase contiene los métodos:

- *run()*: Utilizado para iniciar la ejecución del algoritmo. Se da comienzo a todos los *threads*, mientras que al mismo tiempo interrumpe la ejecución con el fin de solicitar una interacción del usuario para continuar.
- *cont()*: Utilizado para resumir la ejecución. Para esto se indica que no está interrumpida en *isInterrupted*, y se envían señales de *notify()* a cada uno de los *threads*, esto es, a los nodos, canales y a los *threads* encargados de ejecutar el método de recepción de mensajes en cada nodo (explicados más adelante).
- *pause()*: Utilizado para interrumpir la ejecución. Solo basta con indicar esto a través de *isInterrupted* para que cada nodo y canal se detenga al momento de llegar a un punto de interrupción. Adicional a esto, se envían excepciones *InterruptedException* a cada uno de ellos para los casos en que se encuentren suspendidos por el método *sleep*, forzándolos así a resumir la ejecución, e inmediatamente interrumpirla.
- *step()*: Utilizado para avanzar una cantidad de instrucciones fija en la ejecución. Para lograr esto, se hizo uso de una variable que actúa como *lock* denominada *stepLock*. Así, esta será un contador, y cada *thread* al momento de llegar a un punto de interrupción lo revisará. En caso de que este sea mayor a 0, significa que el *thread* puede continuar la ejecución y le restará 1. En caso contrario, se interrumpirá.

En conclusión, cada vez que se llame a este método lo que se hará simplemente será aumentar el contador en una cierta cantidad para permitir a algunos *threads* continuar su ejecución, a pesar de estar interrumpida la red. Adicional a esto, se envían señales de *notify()* a cada uno para despertarlos en caso de estar interrumpidos y así intenten adquirir el *lock*, y excepciones de *InterruptedException* para volver a la ejecución a los procesos suspendidos por el método *sleep*.

- *terminate()*: Utilizado para detener la ejecución del algoritmo y destruir todos los *threads*. Así, se indica esto a través de *isTerminated*, para luego enviar excepciones de *InterruptedException* con el fin despertarlos en caso de estar suspendidos y vuelvan a revisar el estado de la red, o directamente interrumpir su ejecución y forzarlos a retornar.

Por último, es importante destacar que esta clase también se encarga de detectar el término de la ejecución del algoritmo. Para hacer esto, se crea un *thread* el cual esperará, en primer lugar, el término de la ejecución de los algoritmos principales de cada nodo. Luego, se asegurará de que todos los canales no tengan mensajes en tránsito, a través del contador *runningChannelsCount*, y, además, todos los nodos hayan procesado los mensajes recibidos a través del método respectivo, a través del contador *runningNodesReceiversCount*. En consecuencia, se indicará a los canales que ya no habrán más mensajes a enviar, forzándolos a terminar su ejecución. A continuación, se asegura de que todos los *threads* creados para procesar los mensajes que llegan en cada nodo hayan terminado. Por último, siempre y cuando la red de nodos no se encuentre destruida, se le indica a la interfaz gráfica que la ejecución del algoritmo terminó, produciendo los cambios respectivos en ella.

#### 4.1.4. Node

Será la clase que se describió en el diseño conceptual del capítulo anterior para representar a un nodo. Hereda de la clase *Thread*, y su función esencial será ejecutar el algoritmo principal de este a través del método *run()*, considerando el punto de interrupción respectivo. Contiene todos los canales entrantes y salientes, y su funcionalidad está complementada con las clases *VisualNode* y *Program*, explicadas más adelante.

Por otro lado, entrega los métodos de *setState(...)* para cambiar su estado lógico, y *receiveMessage(...)*, que es el cual utilizará cada canal al momento de entregarle un mensaje a este nodo. Para eso, este último método se encarga de aumentar el contador *runningNodesReceiversCount* de la clase *Network* y crear un nuevo *thread*, el que ejecutará el algoritmo de recepción de mensajes de manera independiente, para que el canal siga procesando los demás mensajes por entregar.

#### 4.1.5. VisualNode

Si bien la clase anterior ya representa a un nodo, esta es la que compone solamente la ejecución del algoritmo. Ahora, para representarlo visualmente, se creó una nueva clase, utilizada netamente para dibujarlo en la interfaz gráfica. Así, la clase *VisualNode* contiene la posición del nodo en la pantalla, su nombre, estado, y una variable que indica si es iniciador o no. Luego, cada vez que se actualice alguno de los valores anteriores, se llama al método *draw()* para dibujarlo en la pantalla nuevamente.

Se destaca el método *contains(...)*, que será utilizado para eventos del *mouse* en pantalla, para detectar si está dentro del dibujo del nodo o no.

#### 4.1.6. Program

Para representar al nodo ya se tiene la clase *Node* encargada de ejecutar el algoritmo, y la clase *VisualNode* para visualizarlo en pantalla. Sin embargo, hace falta una clase la cual pueda modificar el usuario y donde se le permita definir el algoritmo a ejecutar, sin modificar otros elementos propios del *framework*. Para eso se creó la clase *Program*, destacando los métodos abstractos *main()* y *onMessageReceive(...)* los cuales corresponden al algoritmo principal y al algoritmo de recepción de mensajes, respectivamente. Están declarados para ser directamente implementados por el usuario de acuerdo al programa a ejecutar, y los cuales son utilizados en la clase *Node*.

Esta clase corresponde a la conexión entre la representación del nodo en el *framework* y el código implementado por el usuario, es por esto que aquí se entregan todos los métodos que este puede utilizar, tales como enviar mensajes, cambiar su estado o nombre, o saber si el nodo es iniciador. Por último, se entrega el método *getDescription(...)*, el cuál permitirá mostrar información adicional del nodo durante el tiempo de ejecución, quedando a libre disposición de ser sobrescrito, en caso de querer mostrar información correspondiente a la lógica del algoritmo.

#### 4.1.7. Channel

Será la clase que se describió en el diseño conceptual del capítulo anterior para representar a un canal. Hereda de la clase *Thread*, conecta un nodo de origen y de destino, y su función esencial será transportar los mensajes del primero al segundo. Para esto, tendrá una cola de mensajes la cual revisará en un *loop* infinito dentro del método *run()* y, una vez transcurrido un tiempo especificado, le entregará el mensaje al nodo de destino llamando al método de *receiveMessage(...)* respectivo, para luego seguir procesando los mensajes restantes. Es importante destacar al mismo tiempo otros casos que pueden ocurrir aquí. Por un lado, si la cola de mensajes está vacía, el *thread* se suspenderá hasta que vuelva a recibir una señal de *notify()*. Además, cada vez que ocurra una interrupción en la ejecución, el proceso se encargará de no considerar el tiempo que esta dure al momento de revisar si un mensaje estuvo el suficiente tiempo en la cola para ser entregado. Para eso, se le resta el tiempo de interrupción al tiempo acumulado que lleva cada mensaje esperando para ser enviado. De manera contraria, cuando se llama al método *step()* en la clase *Network* y siempre y cuando el *thread* del canal logre adquirir el *lock*, en vez de restar el tiempo a los mensajes se sumará un tiempo de aceleración para simular que se avanzó durante la ejecución. Junto con lo anterior, con el fin de optimizar recursos, si el próximo mensaje en la cola no se encuentra listo aún para ser entregado, el canal se suspenderá por un tiempo hasta que sí lo esté.

Al mismo tiempo, esta clase ofrece el método *send(...)*, para enviar los mensajes desde el nodo de origen. Dentro de este, en primer lugar, se establece el tiempo en que entra el mensaje a la cola, para que luego el *thread* pueda saber cuánto tiempo ha pasado desde que el mensaje ingresó. Por último, se agrega a la cola dentro del canal y se envía una señal de *notify* para resumir su ejecución en caso de estar suspendida.

Como último punto, es importante hacer notar el método *terminate()*. Debido a que el canal estará en un *loop* infinito, nunca terminará su ejecución, incluso cuando no tenga mensajes en tránsito. Para esto, se hizo uso de la variable *finish*, para indicarle al *thread* que el algoritmo ya terminó y puede dejar de esperar más mensajes. En caso de que el *thread* se encuentre suspendido, se envía una excepción *InterruptedException* para resumirlo y termine su ejecución.

#### 4.1.8. VisualChannel

Al igual que la representación de un nodo, se separó la funcionalidad de un canal entre la clase *Channel*, la cual se encarga de procesar los mensajes y entregarlos, y la clase *VisualChannel* que está hecha netamente para representarlo en pantalla. Así, esta contiene la posición del canal en la pantalla, construido como un polígono en forma de flecha, y el estado del canal. Para esto último, se considera *EMPTY*, cuando no hay mensajes en tránsito, *BLOCKED*, cuando hay mensajes en tránsito pero el primer mensaje a entregar aún no está listo, y *READY*, cuando el primer mensaje de la cola está listo para ser entregado. Con respecto a los colores para representar estos estados, se ofrece al usuario modificarlos sobrescribiendo el método *getChannelColor(...)*, donde se pasará el último mensaje entregado como argumento en caso de querer utilizarlo.

Se destaca el método *contains(...)*, que será utilizado para eventos del *mouse* en pantalla, para detectar si está dentro del polígono del canal o no.

#### 4.1.9. MessageQueue

Corresponde a la cola de mensajes. Tendrá métodos sincrónicos para agregar, obtener y eliminar mensajes de la cola, asegurando su consistencia. Se agregan los métodos *accelerateMessages(...)* para agregar un tiempo de aceleración a cada uno de los mensajes en la cola, y *addInterruption(...)* para agregar un tiempo de interrupción en ellos.

#### 4.1.10. Message

Será la clase que representará un mensaje. Tendrá un número identificador único *id*, el instante en que ingresó a la cola de mensajes *startTime*, y un valor que representará la acelera-

ción *acceleration* con respecto al tiempo transcurrido. Para obtener esto último se utiliza el método *getElapsedTime()*, el cual entrega el tiempo transcurrido entre que el mensaje entró a la cola y el tiempo actual. Adicional a esto, se le suma la variable de aceleración, la cual es inicialmente 0, en caso de que se quiera avanzar en el tiempo y entregar este mensaje más rápido. El fin de la aceleración es útil al momento de que se quiera simular un avance en la ejecución del algoritmo, a través de *step()*.

Cabe destacar que para agregar un tiempo de interrupción al mensaje desde la clase *MessageQueue* solo basta con aumentar la variable *startTime*. Además, se hace notar el método *getDescription(...)* que cumple la misma función del método mencionado en la clase *Program*. Esto es, poder mostrar información adicional del mensaje que se encuentra en tránsito a través del canal. Queda a disposición del usuario en caso de querer sobrescribirlo.

#### 4.1.11. Screen

Será la clase que se describió en el diseño conceptual del capítulo anterior para representar a la interfaz gráfica del *framework*. Esta contendrá todos los botones para interactuar de clase *Button*, los campos para modificar los parámetros como *TextField*, la barra de estado para mostrar el estado del algoritmo como un elemento *Label*, y la vista de la ejecución del algoritmo distribuido como la clase *AlgorithmView*, explicada más adelante.

En conclusión, toda la funcionalidad esencial de esta clase estará centrada en el método *handle(...)*, donde se manejan todas las interacciones con la interfaz. Esto es, salir del programa, ejecutar el algoritmo, pausar el algoritmo, avanzar una cantidad de instrucciones fija en el algoritmo o reiniciarlo. Principalmente se encarga de llamar al método respectivo de la clase *Network*, y habilitar o deshabilitar botones de acuerdo a lo que le esté permitido hacer al usuario según el estado de ejecución. Esta clase también se encarga de actualizar los parámetros ingresados por la interfaz cada vez que se comienza a correr el algoritmo.

#### 4.1.12. AlgorithmView

Esta es la clase que provoca directamente la visualización del algoritmo, incluyendo todos los nodos y canales, mostrando su funcionamiento de forma dinámica. Para esto, se hace uso de un elemento *canvas*, en el cual se dibuja la red. Esto se logra con el método *draw()*, el cual simplemente llama a los métodos *draw()* de cada nodo y canal. Por otro lado, todas las interacciones con esta vista se manejarán con el método *handle(...)*.

Se hace necesario destacar que pueden haber dos modos para esta vista, definida por la variable *designMode*. Si es verdadera, significa que la clase está en modo diseño, lo que indica que su fin es solamente construir la red de nodos, agregando o eliminando nodos y canales, y reposicionarlos en caso de querer el usuario. Por el contrario, si no está en modo

de diseño, significa que la vista está para visualizar la ejecución del algoritmo distribuido, sin posibilidades de modificar la red.

Para las interacciones con el usuario se capturan los eventos del *mouse*. Cuando el modo no es de diseño, esto incluye solamente eventos de movimiento, en donde se verifica la posición del puntero; si está sobre un nodo o un canal, se muestra la información adicional de este llamando al método *getDescription()* respectivo. En caso de que la clase esté en modo de diseño, se consideran otros eventos:

- **MOUSE\_PRESSED:** Cuando se comienza a presionar el botón izquierdo del *mouse*. En este caso, solo se verifica si se seleccionó un nodo o canal, y se guarda su identificador para utilizarlo en otros eventos posteriores.
- **MOUSE\_DRAGGED:** Cuando se suelte el botón izquierdo del *mouse* luego de haberlo mantenido presionado. Si se seleccionó un nodo previamente, significará que se intentó reposicionar este a través de la pantalla, por lo que se actualiza la posición del nodo a donde está el *mouse* y se vuelve a dibujar.
- **MOUSE\_RELEASED:** Cuando se suelte el botón izquierdo del *mouse*. Solo se captura este evento para poder diferenciar si corresponde al evento de *MOUSE\_DRAGGED* o no, es decir, identificar si anteriormente hubo un *click* o se mantuvo presionado el botón durante un tiempo. Para esto, se usa la variable *dragging*, la que en caso de ser verdadera, y por lo tanto, parte del evento de arrastrar, simplemente se reinicia el indicador de nodo seleccionado, para que no haga efecto.
- **MOUSE\_CLICKED:** Cuando se presione y suelte el botón izquierdo del *mouse*. Este evento puede tener múltiples efectos, dependiendo del estado en que se encuentre la vista:
  - **DELETE\_NODE:** En caso de que la vista se encuentre en este estado, significa que el usuario está intentando borrar un nodo de la red. Por lo tanto, se captura el último nodo seleccionado, a través de *selectedNode*, se elimina de la red, se redibuja todo y se vuelve al estado normal.
  - **ADD\_CHANNEL:** En caso de que la vista se encuentre en este estado, significa que el usuario está intentando agregar un canal a la red. Para esto, significa que el usuario deberá seleccionar primero un nodo de origen y un nodo de destino. Utilizando las variables *prevSelectedNode* y *selectedNode*, se verifica que los dos nodos hayan sido seleccionados, se agrega el canal que los une, se dibuja en la pantalla, y se vuelve al estado normal.
  - **DELETE\_CHANNEL:** En caso de que la vista se encuentre en este estado, significa que el usuario está intentando borrar un canal de la red. Por lo tanto, se captura el último canal seleccionado, a través de *selectedChannel*, se elimina de la red, se redibuja todo y se vuelve al estado normal.

- **TOGGLE\_INITIATOR:** En caso de que la vista se encuentre en este estado, significa que el usuario está intentando modificar el estado de iniciador de un nodo. Por lo tanto, se captura el último nodo seleccionado, se cambia su estado de *isInitiator* a lo contrario, es decir, si no es iniciador pasa a ser iniciador o viceversa, se redibuja el nodo y se vuelve al estado normal.
- **NONE:** En caso de que la vista se encuentre en este estado, significa que el usuario está intentando cambiar el estado inicial de un nodo de la red. Por lo tanto, se captura el último nodo seleccionado, se cambia su estado al siguiente valor posible, se redibuja el nodo y se vuelve al estado normal.

#### 4.1.13. SettingsWindow

Esta clase representará a la ventana de configuración del *framework*. Esta permitirá, en tiempo de ejecución, modificar la red que vaya a correr el algoritmo definido. Presionando el botón de *Settings* en la ventana de *Screen*, se utilizará el método *display()* para entrar a este modo de diseño. Así, se mostrará una lista de las redes actuales cargadas, mostrando la seleccionada a través de una vista con la clase *AlgorithmView* en modo de diseño, y ofreciéndose la alternativa de agregar otra, eliminar, importar o exportar. Todas estas interacciones se manejarán desde el método *handle(...)*, entregando las siguientes opciones:

- **Close:** Será para cerrar la ventana de configuración.
- **Add Network:** Será para agregar una nueva red a la lista. Esto simplemente crea una nueva clase *Network* vacía, y lo agrega al arreglo de redes guardadas en la clase de *SettingsWindow*.
- **Delete Network:** Será para eliminar una red de la lista. Esta opción se presentará como un botón - al lado del nombre de la red a eliminar, provocando, al ser presionado, que la red seleccionada se elimine del arreglo de redes guardadas en la clase de *SettingsWindow*.
- **Use:** Será para utilizar la red seleccionada para correr el algoritmo, y volver a la pantalla de ejecución. Solo se permitirá cuando la red seleccionada tenga al menos un nodo.
- **Add node:** Será para agregar un nodo dentro de la red seleccionada. Para esto, se crea una nueva instancia de un nodo con un nombre genérico "N", una posición aleatoria, estado inicial con valor 0, y modo no iniciador, para luego agregarlo a la red actual y dibujarlo en pantalla. Es importante hacer notar que para que esto sea posible, el usuario deberá definir un constructor para la clase *Program* el cual no reciba parámetros. De esta manera, se le podrá entregar una nueva instancia de esta clase al nodo y crearlo exitosamente.

- *Delete node*: Será para eliminar un nodo dentro de la red seleccionada. El único fin de esto es indicarle a la instancia de *AlgorithmView* que cambie su estado a *DELETE\_NODE*.
- *Add channel*: Será para agregar un canal dentro de la red seleccionada. El único fin de esto es indicarle a la instancia de *AlgorithmView* que cambie su estado a *ADD\_CHANNEL*.
- *Delete channel*: Será para eliminar un canal dentro de la red seleccionada. El único fin de esto es indicarle a la instancia de *AlgorithmView* que cambie su estado a *DELETE\_CHANNEL*.
- *Toggle initiator*: Será para cambiar el valor de *isInitiator* de un nodo dentro de la red seleccionada. El único fin de esto es indicarle a la instancia de *AlgorithmView* que cambie su estado a *TOGGLE\_INITIATOR*.
- *Cancel*: Será para indicarle a la instancia de *AlgorithmView* que vuelva su estado a normal y no haga cambios en la red.
- *Export networks*: Para poder reutilizar las redes de nodos de la ejecución actual con otros algoritmos, se ofrece la opción de exportarlas en un archivo externo. Para esto, se crea un archivo con un arreglo de objetos *JSON*, con extensión *.daf*. Cada uno de estos objetos se escribirá de acuerdo a la siguiente estructura de ejemplo:

```
{
  "nodes": [
    {
      "name": "N1",
      "x": 110.0,
      "y": 85.0,
      "state": 0,
      "isInitiator": true
    },
    {
      "name": "N2",
      "x": 382.5,
      "y": 117.5,
      "state": 0,
      "isInitiator": false
    }
  ],
  "channels": [
    {
      "src": 0,
      "dst": 1
    },
    {
```

```
        "src":1 ,
        "dst":0
    }
    ]
}
```

Así, habrá una lista de nodos y canales. Para cada nodo se indicarán su nombre, posición, estado inicial, y si es iniciador o no. Para cada canal, solo basta con indicar cuál nodo es el de origen, y cuál es el de destino. Cabe destacar que con el fin de simplificar la identificación de los nodos en esto último, se indica la posición del nodo en el arreglo.

- *Import networks*: Será utilizado para importar un archivo *.daf* con redes previamente exportadas por la opción de *Export networks*. Se encargará de leer el archivo *JSON*, crear las redes indicadas, e instanciar los nodos y canales de cada una a través de los métodos *addNode(...)* y *link(...)*.

#### 4.1.14. Assertion

Es una clase utilitaria que sirve para asegurarse de que cierta condición se cumpla antes de ejecutar algún método, a través de *test(...)*. En caso contrario, se saldrá del programa.

#### 4.1.15. Constants

Es la clase que contendrá todas las constantes que influyan en el funcionamiento del *framework*. Esto incluye tiempos de espera antes de entregar un mensaje, tamaño de los nodos y canales, tamaño de texto, entre otros.

#### 4.1.16. VectorClock

Como característica adicional del *framework*, se quiso facilitar el trabajo a futuro de este. Considerando que los algoritmos distribuidos como los de **instantánea distribuida** tienen una representación de diagrama espacio-temporal lo cual se escapa del objetivo de esta memoria, se decidió implementar **relojes vectoriales**. Esto debido a que ellos permiten indicar todos los eventos ocurridos en cada ejecución del algoritmo distribuido, con la propiedad de poder organizarlos de forma cronológica y mostrar sus relaciones de causalidad. Así, cada nodo será un proceso que tendrá su propia instancia de *VectorClock*, el cual contendrá un arreglo representando el reloj vectorial, un identificador del nodo y su nombre, y que registrará cada evento en un archivo *log* con la ayuda de la clase *LogHandler*, explicada más

adelante, entregando la posibilidad de que en otro trabajo pueda ser procesado y se obtenga la representación requerida.

Con respecto a los eventos a registrar, se encuentran los siguientes:

- **Evento de envío:** Ocurrirá cada vez que un nodo envíe un mensaje. Cada mensaje tendrá un identificador único, e incluirá la instancia de reloj vectorial del proceso en este punto, para que pueda ser utilizado por el nodo de destino al recibir el mensaje.
- **Evento de recepción:** Ocurrirá cada vez que un nodo reciba un mensaje. Se utilizará el reloj vectorial incluido en el mensaje y el identificador de este.
- **Evento local:** Serán todos los eventos locales que no corresponden a los anteriores. Esto es, cambiar el estado del nodo, cambiar su nombre, o un evento local personalizado el cual quiera registrar el usuario.

Es importante hacer notar que se le entrega al usuario en la clase *Program* un método para registrar un evento local personalizado a través de *registerLocalEvent(...)*, y métodos para obtener la descripción del evento que se esté registrando. El fin de esto último es que el usuario pueda modificar y decidir qué información se registrará en cada tipo de evento, sobrescribiendo *getSendEventInfo(...)* para eventos de envío, *getReceiveEventInfo(...)* para eventos de recepción, *getStateChangeEventInfo(...)* para eventos locales en donde se cambie el estado del nodo y *getNameChangeEventInfo(...)* para eventos locales en donde se cambie el nombre del nodo.

#### 4.1.17. LogHandler

Esta clase se encargará simplemente de registrar los relojes vectoriales en un archivo de *log*. Cada evento escrito indicará en orden, separado por un guión, lo siguiente:

- Número de proceso o nodo que genera el evento
- Reloj vectorial
- Nombre del nodo
- Tipo de evento
- Identificador del mensaje (en caso de ser un evento de envío o recepción)
- Descripción del evento

Cabe destacar que se agregarán indicadores para cuando se inicie o cierre el programa, a través del texto *Starting program* y *Closing program*, respectivamente.

## 4.2. Guía de Uso

### 4.2.1. Herramientas de Programación

Para comenzar a utilizar el *framework* deben instalarse las herramientas necesarias para su correcto funcionamiento. Esto simplemente consiste en instalar el *kit* de desarrollo de *Java* 8, también conocido *Java Development Kit 8*<sup>17</sup>. Se recomienda además instalar un *IDE* como *Eclipse*<sup>18</sup> o *IntelliJ IDEA*<sup>19</sup> para facilitar el proceso de desarrollo, compilación y ejecución del programa que utilizará el *framework*. Una vez hecho esto, basta con agregar el módulo del *framework* **daf** al directorio de trabajo y comenzar a implementar las clases mencionadas a continuación, siempre preocupándose de importar los componentes necesarios de este según corresponda.

Cabe destacar que para esta memoria se trabajó con *Java SE Development Kit 1.8.0\_45* y el *IDE IntelliJ IDEA 2016.3.1*, todo en el sistema operativo *MacOS High Sierra 10.13.6*.

### 4.2.2. Clase App

Como parte del programa que dará inicio a toda la ejecución, se deberá implementar una nueva clase que herede de la clase *App* del *framework*. Una vez hecho esto, se deberá implementar primero el método **main(...)** requerido por *Java* para lanzar la aplicación de *JavaFX*:

```
public static void main( String args [] ) {  
    launch ( );  
}
```

Luego, se deberá crear un constructor para la nueva clase, donde simplemente se llame al constructor de la clase *App*, con el método **super(...)**, entregando los siguientes parámetros, en orden:

- (*String*) **title**: Será el título de la ventana donde se ejecutará el programa.
- (*double*) **width**: Será el ancho de la ventana donde se ejecutará el programa.
- (*double*) **height**: Será el alto de la ventana donde se ejecutará el programa.
- (*Class*) **programClass**: Será la clase *.class* que hereda de la clase *Program* del *framework*, y que se implementará en la sección siguiente.

---

<sup>17</sup><http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

<sup>18</sup><https://www.eclipse.org>

<sup>19</sup><https://www.jetbrains.com/idea/>

- (Class) **networkExtensionClass** [OPCIONAL]: Será la clase *.class* que formará parte de la extensión la clase *Network* del *framework*, explicada más adelante.

Por otra parte, se deberá implementar el método abstracto **void construct()** de la clase *App* en la nueva clase, destinado para construir la red de nodos inicial con la que se ejecutará el programa, y donde se indicarán además otros componentes de la aplicación. Así, en este se podrán utilizar los siguientes métodos:

- **Node addNode(...)**: Será el método utilizado para agregar un nodo en la red. Retornará la instancia de este último como resultado, y requiere los siguientes parámetros, en orden:
  - (Program) **program**: Será una instancia de la implementación de la clase *Program*, explicada en la sección siguiente, la cual se utilizará para este nodo.
  - (String) **name**: Será el nombre del nodo que se mostrará por pantalla.
  - (double) **x**: Será la coordenada x de la posición del nodo en la pantalla.
  - (double) **y**: Será la coordenada y de la posición del nodo en la pantalla.
  - (int) **state**: Será el estado inicial del nodo.
  - (boolean) **isInitiator**: Será una variable booleana donde se indique si el nodo es iniciador o no. Influye principalmente para agregar este rol especial en la visualización, pero también puede utilizarse para la lógica del algoritmo.
- **void link(...)**: Será el método utilizado para unir dos nodo en la red a través de un canal. Aquí se utilizarán las instancias de los nodos retornadas por el método *addNode(...)*, y requiere los siguientes parámetros, en orden:
  - (Node) **src**: Será la instancia del nodo de origen en el canal.
  - (Node) **dst**: Será la instancia del nodo de destino en el canal.
  - (boolean) **bidirectional**: Será una variable booleana donde se indique si el canal es bidireccional o no. Equivale a ocupar el método *link(...)* dos veces, intercambiando de posición el nodo de origen y destino.
- **void addParameter(...)**: Será el método utilizado para agregar un parámetro en la interfaz gráfica al momento de comenzar la ejecución, y el cual podrá ser utilizado después en la lógica del algoritmo. Requiere los siguientes parámetros, en orden:
  - (int) **id**: Será el número identificador del parámetro, utilizado posteriormente cuando se quiera obtener su valor.
  - (String) **name**: Será el nombre del parámetro, el cual lo identificará en pantalla.
  - (Class) **type**: Será la clase que indique el tipo de parámetro. Las opciones posibles son *String.class*, *Integer.class*, *Double.class* y *Boolean.class*.

- (T) **initialValue**: Será el valor inicial del parámetro. El tipo de este valor estará definido por la clase *type* entregada en el parámetro anterior.
- **void setNodeStates(...)**: Será el método utilizado para definir todos los estados lógicos de un nodo en el algoritmo. Solo requiere un parámetro:
  - (Color[]) **nodeStates**: Será un arreglo de objetos de la clase *javafx.scene.paint.Color*. En este se indicarán los colores de cada uno de los estados posibles del nodo, donde estos últimos estarán definidos por la posición del color en el arreglo. Por ejemplo, si se entrega el arreglo *new Color[]{ Color.RED, Color.BLUE }*, significa que el estado 0 tendrá color rojo y el estado 1 tendrá color azul.

Por último, y como caso opcional, también se da la posibilidad de modificar el color de los estados de los canales. Para eso, se deberá sobrescribir el método **Color getChannelColor(...)**, preocupándose de que se retorne una instancia de la clase *javafx.scene.paint.Color*. Este método se llamará cada vez que un canal cambie de estado y se tenga que dibujar en pantalla, entregando la opción de ocupar los siguientes parámetros, en orden:

- (Channel.State) **state**: Será el estado al que cambió el canal. Tal como se explicó anteriormente, los valores pueden ser *EMPTY*, *BLOCKED* y *READY*, todos pertenecientes a *Channel.State*.
- (Message) **lastMessage**: Será el último mensaje que el canal ha entregado. En caso de que aún no haya entregado alguno, su valor será nulo.

#### 4.2.3. Clase Program

Como parte del programa donde se definirá el algoritmo distribuido a ejecutar, se deberá implementar una nueva clase que herede de la clase *Program* del *framework*, asegurándose de no definir ningún constructor.

Luego de lo anterior, deberán implementarse los siguientes métodos abstractos de la clase *Program* en la nueva clase:

- **void init()**: Corresponderá al código ejecutado al crear la instancia, equivalente al constructor. Cabe destacar que en caso de querer modificar el nombre del nodo correspondiente a la instancia de esta implementación antes de comenzar la ejecución del programa, se ofrece aquí el método **void setInitialNodeName(...)**, el que solo recibirá el parámetro (String) **name**, siendo el nuevo nombre del nodo.
- **void start()**: Corresponderá al código ejecutado al comenzar a ejecutar el *thread* del nodo, y antes de que el usuario presione el botón para comenzar a correr el algoritmo. Permite modificar variables internas, y la ventaja de este punto es que ya se tienen

todos los nodos de la red definidos, por lo que es posible obtener la cantidad de vecinos en la red o alguna información similar. Notar que este método ya es parte del *thread*, por lo que se deberá tener esto en consideración al acceder a variables compartidas.

- ***void main() throws InterruptedException***: Será la sección donde se definirá el algoritmo principal de cada nodo. Notar que el método puede arrojar una excepción *InterruptedException*, utilizada por el *framework* para detener la ejecución del *thread*.
- ***void onMessageReceive(...) throws InterruptedException***: Será la sección donde se definirá el algoritmo de recepción de mensajes de cada nodo. Notar que el método puede arrojar una excepción *InterruptedException*, utilizada por el *framework* para detener la ejecución del *thread*, y que recibirá los siguientes parámetros, en orden:
  - (Node) **src**: Será el nodo quien envió el mensaje.
  - (Message) **message**: Será el mensaje recibido en el nodo. En caso de haberse hecho una nueva implementación de la clase *Message*, explicado más adelante, será necesario hacer el respectivo *cast*<sup>20</sup> del mensaje dentro del método.

Dentro de todos los métodos mencionados anteriormente, se ofrecen los siguientes métodos de utilidad:

- ***boolean isInitiator()***: Será el método utilizado para saber si el nodo que está ejecutándose es iniciador o no, retornando un valor booleano. Este estará basado en la variable *isInitiator* del nodo, la cual se indica al momento de su instanciación. Si bien está pensado para fines de visualización, también puede ocuparse para la lógica del algoritmo, donde se ejecute una instrucción diferente en caso de que el nodo sea iniciador, por ejemplo.
- ***Node getNode()***: Será el método utilizado para obtener el nodo al que pertenece la instancia de la implementación de la clase *Program*. Retornará un objeto de la clase *Node*.
- ***Object getNetworkExtension()***: Será el método utilizado para obtener la instancia de la extensión de la clase *Network*, en caso de haberse implementado y la cual se explicará más adelante. Retornará un objeto de la clase *Object*, por lo que será necesario hacer el *cast* correspondiente.

Además, para todos los métodos anteriores, a excepción de *init(...)*, se ofrecen adicionalmente los siguientes métodos:

- ***Channel[] getInChannels()***: Será el método utilizado para obtener todos los canales entrantes al nodo o, en otras palabras, todos los canales que tengan como destino al nodo actual. Retornará un arreglo de objetos de la clase *Channel*.

---

<sup>20</sup>Procedimiento utilizado para transformar una variable de un tipo a otro.

- ***Channel[] getOutChannels()***: Será el método utilizado para obtener todos los canales salientes del nodo o, en otras palabras, todos los canales que tengan como origen al nodo actual. Retornará un arreglo de objetos de la clase *Channel*.

Luego, para fines de ejecución y por lo tanto solo para los métodos *main(...)* y *onMessageReceive(...)*, se ofrecen los siguientes métodos de utilidad para apoyar el proceso de desarrollo:

- ***Object getParameter(...)***: Será el método utilizado para obtener el valor de un parámetro definido anteriormente, en el método *construct()* de la implementación de la clase *App*. Retornará el parámetro como una instancia de la clase *Object*, por lo que será necesario hacer un *cast* a la clase que corresponda. Solo requiere un parámetro:
  - ***(int) id***: Será el número identificador del parámetro que se quiera obtener, definido al momento de su creación.
- ***void sleep(...) throws InterruptedException***: Será el método utilizado para suspender el *thread* por una cierta cantidad de tiempo. Si bien existe el método *Thread.sleep(...)*, este no considera el modelo lógico del *framework* para manejar las interrupciones, por lo que no se debe usar. Solo requiere un parámetro:
  - ***(long) sleepTime***: Será el tiempo, en milisegundos, en que se suspenderá el *thread*.
- ***void send(...) throws InterruptedException***: Será el método utilizado para enviar un mensaje, indicando la posición del canal saliente. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - ***(int) i***: Será la posición del canal saliente, donde se enviará el mensaje, en el arreglo retornado por el método *getOutChannels()*.
  - ***(Message) message***: Será el mensaje a enviar.
- ***void sendTo(...) throws InterruptedException***: Será el método utilizado para enviar un mensaje, indicando el nodo de destino. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - ***(Node) n***: Será el nodo de destino hacia quien va dirigido el mensaje. Cabe destacar que debe ser vecino del nodo desde donde se envía; en caso contrario, no se efectuará la operación.
  - ***(Message) message***: Será el mensaje a enviar.

- ***void sendThrough(...) throws InterruptedException***: Será el método utilizado para enviar un mensaje, indicando el canal saliente. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - (Channel) **c**: Será el canal saliente por donde se enviará el mensaje. Debe ser parte del arreglo retornado por el método *getOutChannels()*; en caso contrario, no se efectuará la operación.
  - (Message) **message**: Será el mensaje a enviar.
- ***void sendAt(...) throws InterruptedException***: Será el método utilizado para enviar un mensaje a través del primer canal saliente que cumpla con una condición de envío, es decir, recorriendo el arreglo retornado por *getOutChannels()*, al encontrar el primer canal que cumpla con la condición, se enviará el mensaje a través de este y se terminará la ejecución del método. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - (Message) **message**: Será el mensaje a enviar.
  - (SendCondition) **sendCondition**: Será la condición de envío. Para esto, se deberá entregar la instancia de una clase que implemente la interfaz *SendCondition* del *framework*. Específicamente, se deberá implementar el método ***boolean shouldSend(...)***, el cual retornará un valor booleano indicando si debe enviarse el mensaje o no a un canal específico, y que recibirá los siguientes parámetros:
    - (Channel) **c**: Será el canal saliente del nodo a través del cual se está evaluando si enviar el mensaje o no.
    - (Message) **message**: Será el mensaje a enviar. Corresponde a la misma instancia que se le entrega al método *sendAt(...)*.
- ***void multicast(...) throws InterruptedException***: Será el método utilizado para enviar un mensaje a través de todos los canales salientes que cumplan con una condición de envío. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - (Message) **message**: Será el mensaje a enviar.
  - (SendCondition) **sendCondition**: Será la condición de envío. Para esto, se deberá entregar la instancia de una clase que implemente la interfaz *SendCondition* del *framework*. Específicamente, se deberá implementar el método ***boolean shouldSend(...)***, el cual retornará un valor booleano indicando si debe enviarse el mensaje o no a un canal específico, y que recibirá los siguientes parámetros:
    - (Channel) **c**: Será el canal saliente del nodo a través del cual se está evaluando si enviar el mensaje o no.

- (Message) **message**: Será el mensaje a enviar. Corresponde a la misma instancia que se le entrega al método *multicast(...)*.
- **void broadcast(...) throws InterruptedException**: Será el método utilizado para enviar un mensaje a través de todos los canales salientes del nodo. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Solo requiere un parámetro:
  - (Message) **message**: Será el mensaje a enviar.
- **void broadcastExcept(...) throws InterruptedException**: Será el método utilizado para enviar un mensaje a través de todos los canales salientes del nodo, a excepción de uno indicado. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Requiere los siguientes parámetros, en orden:
  - (Node) **exc**: Será el canal a excluir del envío.
  - (Message) **message**: Será el mensaje a enviar.
- **void setState(...) throws InterruptedException**: Será el método utilizado para cambiar el estado lógico del nodo a otro indicado. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Solo requiere un parámetro:
  - (int) **state**: Será el estado lógico al cual cambiará el estado del nodo durante la ejecución del algoritmo distribuido.
- **void setNodeName(...) throws InterruptedException**: Será el método utilizado para cambiar el nombre del nodo a otro indicado, en tiempo de ejecución. El usuario deberá asegurarse de que, en caso de que arroje una excepción *InterruptedException*, esta llegue al método *main()* o *onMessageReceive(...)*, según corresponda. Solo requiere un parámetro:
  - (String) **name**: Será el nombre al cual se cambiará el nombre del nodo durante la ejecución del algoritmo distribuido.
- **void registerLocalEvent(...)**: Será el método utilizado para registrar un evento local del algoritmo. Esto se verá reflejado en el archivo de *log* donde se registran todos los eventos de este durante su ejecución. Solo requiere un parámetro:
  - (String) **info**: Será la descripción del evento local registrado.

Por otro lado, y como caso opcional, también se da la posibilidad de modificar otras características del algoritmo, sobrescribiendo los siguientes métodos dentro de la implementación de la clase *Program*:

- ***String getDescription()***: Será el que se ocupe cada vez que se vaya a mostrar información adicional del nodo en pantalla, al mover el *mouse* por encima de este durante la ejecución del algoritmo. En caso de querer modificarse, el usuario deberá sobrescribirlo preocupándose de que se retorne un objeto *String*.
- ***String getSendEventInfo(...)***: Será el que se ocupe cada vez que se vaya a registrar un evento de envío en el archivo de *log*, para obtener la descripción de este. En caso de querer modificarse, el usuario deberá sobrescribirlo preocupándose de que se retorne un objeto *String*, teniendo la opción de ocupar los siguientes parámetros, en orden:
  - (Node) ***dst***: Será el nodo hacia donde se envía el mensaje.
  - (Message) ***message***: Será el mensaje que se está enviando.
- ***String getReceiveEventInfo(...)***: Será el que se ocupe cada vez que se vaya a registrar un evento de recepción en el archivo de *log*, para obtener la descripción de este. En caso de querer modificarse, el usuario deberá sobrescribirlo preocupándose de que se retorne un objeto *String*, teniendo la opción de ocupar los siguientes parámetros, en orden:
  - (Node) ***src***: Será el nodo que envió el mensaje.
  - (Message) ***message***: Será el mensaje que se recibió.
- ***String getChangeStateEventInfo(...)***: Será el que se ocupe cada vez que se vaya a registrar un evento local correspondiente a un cambio de estado de un nodo en el archivo de *log*, para obtener la descripción de este. En caso de querer modificarse, el usuario deberá sobrescribirlo preocupándose de que se retorne un objeto *String*, teniendo la opción de ocupar el siguiente parámetro:
  - (int) ***newState***: Será el nuevo estado lógico al que cambió el nodo.
- ***String getChangeNameEventInfo(...)***: Será el que se ocupe cada vez que se vaya a registrar un evento local correspondiente a un cambio de nombre de un nodo en el archivo de *log*, para obtener la descripción de este. En caso de querer modificarse, el usuario deberá sobrescribirlo preocupándose de que se retorne un objeto *String*, teniendo la opción de ocupar el siguiente parámetro:
  - (String) ***newName***: Será el nuevo nombre al que cambió el nodo.

Por último, cabe destacar que dentro de la implementación de la clase *Program*, el usuario será libre de agregar métodos y variables que lo ayuden en la escritura del código para simular y visualizar el algoritmo distribuido, siendo importante hacer notar que *main()* y *onMessageReceive(...)* se ejecutarán en *threads* diferentes, por lo que se deberá tener esto en consideración al momento de leer o escribir variables, u ocupar métodos, para mantener la consistencia de los datos.

#### 4.2.4. Extensión de Clase Network

Si bien la clase *Network* del *framework* no puede ser implementada o modificada de ninguna manera, se ofrece implementar una clase que sea una extensión de esta. El fin de esto es que el usuario pueda definir variables o métodos a nivel de la red, accesibles por cualquier nodo, y donde cada red tenga una instancia diferente. No hay restricciones a la hora de la implementación de esto más que considerar que se está trabajando con *threads*, por lo que deberá asegurarse la consistencia de los datos.

#### 4.2.5. Clase Message

En caso de querer crear una nueva implementación de los mensajes a enviar durante la ejecución del algoritmo, se deberá crear una nueva clase que herede de la clase *Message* del *framework*. Una vez hecho esto, el usuario quedará libre de agregar los métodos y variables que desee, y que puedan apoyarlo en la ejecución del programa.

Cabe destacar que, al igual que la clase *Program*, se ofrece también la posibilidad de modificar la información adicional que se muestra del mensaje en tránsito, al mover el *mouse* por encima del canal por el que viaja durante la ejecución del algoritmo. Para esto, se deberá sobrescribir el método ***String getDescription()***, teniendo el usuario que preocuparse de retornar un objeto *String*.

#### 4.2.6. Clase Node

Si bien esta clase no está hecha para ser heredada, sí ofrece métodos que pueden serle útil al usuario al momento de obtener la instancia de un nodo a través de alguno de los métodos mencionados anteriormente:

- ***int getIndex()***: Se podrá utilizar este método para obtener el identificador único del nodo, esto es, su posición en el arreglo de nodos que contiene la clase *Network*.
- ***Program getProgram()***: Se podrá utilizar este método para obtener la instancia de la implementación de la clase *Program* asociada al nodo. Será necesario hacer el *cast* respectivo para poder ingresar a los métodos y variables que se hayan definido dentro de esta.

#### 4.2.7. Clase Channel

Al igual que la clase *Node*, esta clase no está hecha para ser heredada. Sin embargo, también ofrece métodos que pueden serle útil al usuario al momento de obtener la instancia de un

canal a través de alguno de los métodos mencionados anteriormente:

- **Node getSrc():** Se podrá utilizar este método para obtener el nodo de origen del canal.
- **Node getDst():** Se podrá utilizar este método para obtener el nodo de destino del canal.

#### 4.2.8. Clase Constants

Tal como se mencionó anteriormente, esta clase contendrá parámetros que influirán en el funcionamiento del *framework* y los cuales quedan a disposición del usuario para ser modificados directamente:

- (*double*) **FONT\_SIZE:** Corresponde al tamaño de la fuente utilizada para mostrar el nombre del nodo en pantalla.
- (*int*) **PREF\_SEP\_SIZE:** Corresponde al tamaño de separación preferido utilizado entre los distintos elementos de la interfaz gráfica. Se utiliza para márgenes, rellenos, separaciones, entre otros cálculos, por lo que si hay algún problema con las separaciones en la interfaz, o algún elemento no se ve completamente, se recomienda modificar este parámetro hasta que se vea una mejora.
- (*Point2D*) **POPUP\_MOUSE\_OFFSET:** Corresponde al *offset* o distancia entre el puntero del *mouse* y la posición del *popup* mostrado durante la ejecución del algoritmo para entregar información adicional de los nodos o mensajes.
- (*double*) **NODE\_RADIUS:** Corresponde al radio de cada nodo del algoritmo distribuido representado en pantalla.
- (*double*) **CHANNEL\_WIDTH:** Corresponde al ancho de canal del algoritmo distribuido representado en pantalla.
- (*double*) **ARROW\_HEAD\_ANGLE:** Corresponde al ángulo de la punta de la flecha utilizada en cada canal del algoritmo distribuido representado en pantalla para indicar su sentido.
- (*double*) **ARROW\_HEAD\_LENGTH:** Corresponde al tamaño de la punta de la flecha utilizada en cada canal del algoritmo distribuido representado en pantalla para indicar su sentido.
- (*long*) **MESSAGE\_SENDING\_TIME:** Corresponde al tiempo, en milisegundos, en que un mensaje estará listo para ser entregado dentro de un canal.
- (*long*) **NODE\_PROGRAM\_MIN\_DELAY:** Corresponde al tiempo aleatorio mínimo, en milisegundos, que cada nodo tomará antes de comenzar a ejecutar el algoritmo principal.

- (*long*) **NODE\_PROGRAM\_MAX\_DELAY**: Corresponde al tiempo aleatorio máximo, en milisegundos, que cada nodo tomará antes de comenzar a ejecutar el algoritmo principal.
- (*int*) **STEP\_SIZE**: Corresponde a la cantidad de instrucciones que se ejecutan cada vez que se llama al método *step()*.
- (*int*) **STEP\_TIME\_SIZE**: Corresponde al tiempo que se simula que se avanzó cada vez que se llama al método *step()*. Es utilizado para adelantar tiempos de entrega de mensajes, o tiempo de suspensión de *threads*, por ejemplo.
- (*String*) **LOG\_FILE**: Corresponde al nombre del archivo de *log* que se utilizará para registrar los eventos del algoritmo. En caso de querer cambiar su ubicación, se deberá indicar también la ruta completa.

#### 4.2.9. Compilación y Ejecución

En caso de que se haya ocupado un *IDE*, tal como se recomendó, bastará con correr el programa desde allí para que este automáticamente haga la compilación y ejecución de la aplicación. En caso contrario, se podrá hacer uso de la terminal del sistema operativo. Para esto, en primer lugar, se deberá tener un directorio raíz el cual contenga una carpeta con todas las clases del *framework*, y otra carpeta con todas las clases que hacen uso del primero y corresponden a la implementación del algoritmo distribuido a simular y visualizar.

Una vez se tenga la estructura anterior, se deberá ingresar a la terminal, ir al directorio raíz mencionado y compilar todos los archivos *.java*, utilizando el comando *javac*, localizado en la carpeta *bin* del *JDK* y asegurándose que su ubicación se encuentre en la variable de entorno **PATH** del sistema operativo. Es importante notar que el *framework* requiere el módulo *javafx-mx.jar*, incluido dentro del *Java Development Kit*, por lo que solo bastará indicar en el comando su ubicación. Así, para compilar los dos módulos, se deberán ejecutar dos comandos con la siguiente forma:

```
javac -classpath "<DIRECTORIO JDK>/lib/javafx-mx.jar:" daf/*.java
javac <Módulo>/*.java
```

donde **<Módulo>** corresponde al nombre del módulo donde se implementó el algoritmo distribuido. Notar el carácter de separación al final de la ruta que se indica con *classpath*, el cual corresponderá a : en sistemas *UNIX*, y a ; en sistemas *Windows*.

Por último, luego de haber compilado los módulos y, en consecuencia, haberse creado archivos *.class*, bastará con utilizar *java*, localizado en la carpeta *bin* del *JDK* y asegurándose que su ubicación se encuentre en la variable de entorno **PATH** del sistema operativo, ejecutando el siguiente comando:

```
java -classpath "<DIRECTORIO JDK>/lib/javafx-mx.jar:" <Módulo>.Main
```

donde **<Módulo>** corresponde al nombre del módulo donde se implementó el algoritmo distribuido. Notar que un punto antepone a la clase *Main*, indicándole a *Java* que corresponde a un módulo y no a una sola clase.

## CAPÍTULO 5

### VALIDACIÓN DE LA SOLUCIÓN

Para validar el *framework* propuesto, se decidió elegir cuatro algoritmos distribuidos a implementar, los cuales representen los distintos tipos que pueden haber en cuanto a simulación y representación:

- Algoritmo del Matón
- Algoritmo de Chang/Roberts
- Algoritmo de Eco
- Algoritmo de Suzuki-Kasami

Cabe destacar que se omitirán tanto las implementaciones de la clase *App* como las modificaciones del método *getDescription(...)*, para solamente enfocarse en el código que está directamente relacionado con el algoritmo distribuido, y no en la red u otros aspectos de la aplicación.

#### 5.1. Algoritmo del Matón

También conocido como *Bully Algorithm*, corresponde a un algoritmo distribuido de **elección** en una topología de **grafo general**. Se basa en una red de nodos completamente interconectada, cada uno con número identificador único, y que tiene como fin elegir como coordinador al que tenga el número más alto, basándose en el principio de extinción de mensajes.

Es parte del programa del ramo de Sistemas Distribuidos y es lo suficientemente representativo con respecto a los algoritmos de elección en una topología de grafo general, debido a que todos buscan el mismo fin inundando toda la red de mensajes con los identificadores de cada nodo, para que finalmente prevalezca el mayor.

##### 5.1.1. Implementación

En primer lugar, se decidió trabajar con una red de 8 nodos, todos conectados entre sí. Además, se definió un parámetro llamado *No answer time* para indicar el tiempo que debe pasar para que un nodo no reciba una respuesta y, en consecuencia, pase a ser coordinador. Por otro lado, se definieron tres estados lógicos para cada nodo:

- **0:** Si el nodo es candidato, es decir, está dentro de las opciones para pasar a ser el coordinador. Será el estado inicial y está representado por el color celeste.
- **1:** Si el nodo ya no es candidato, es decir, está descartado de la opciones para pasar a ser el coordinador. Está representado por el color rojo.
- **2:** Si el nodo es coordinador. Debido a lo anterior, solo un nodo tendrá este estado al momento de terminar el algoritmo. Está representado por el color verde.

Por otra parte, se consideró necesario una implementación de la clase *Message* donde se pueda indicar qué tipo es (de elección, respuesta o para indicar el nodo coordinador) y adjuntar el identificador del nodo que lo envía, quedando de la siguiente forma:

```
public class Msg extends Message {
    private int senderId;
    private int type;

    public Msg(int id, int type) {
        senderId = id;
        this.type = type;
    }

    public int getSenderId() {
        return senderId;
    }

    public int getType() {
        return type;
    }
}
```

Así, para escribir el algoritmo, se hizo la siguiente implementación de la clase *Program*:

```
public class Prog extends Program {
    private static Random random = new Random();
    private int id;
    private int coordinator;
    private int state;
    private boolean candidate;
    private AtomicInteger msgSent = new AtomicInteger(0);

    @Override
    protected void init() {
        id = random.nextInt(99);
        coordinator = -1;
        state = 0;
        candidate = true;
        setInitialNodeName(String.valueOf(id));
    }

    @Override
    protected void start() {
    }

    @Override
    protected void main() throws InterruptedException {
        multicast(new Msg(id, 0), (c, message) -> {
            Prog program = (Prog) c.getDst().getProgram();
            int dstId = program.getId();
            if (dstId > id) {

```

```

        msgSent.incrementAndGet();
    }

    return dstId > id;
});
sleep((Integer) getParameter(0)); // Wait for answers
if (candidate && msgSent.get() == 0) { // Candidate chosen
    setState(2);
    coordinator = id;
    broadcast(new Msg(id, 2));
}
}

@Override
protected void onMessageReceive(Node src, Message message) throws InterruptedException {
    Msg msg = (Msg) message;
    int type = msg.getType();
    if (type == 0) { // ELECTION
        if (msg.getSenderId() < id) {
            sendAt(new Msg(id, 1), (c, _message) -> {
                Prog program = (Prog) c.getDst().getProgram();
                int dstId = program.getId();

                return dstId == msg.getSenderId();
            }); // OK
        }
    } else if (type == 1) { // OK
        if (msg.getSenderId() > id) {
            candidate = false;
            msgSent.decrementAndGet();
            setState(1);
        }
    } else { // COORDINATOR
        coordinator = msg.getSenderId();
    }
}

public int getId() {
    return id;
}
}

```

Para comenzar, se le dio a cada nodo un número identificador aleatorio. Por otra parte, se crearon los siguientes campos de utilidad:

- **(int) coordinator:** Será el número identificador del coordinador elegido en la red.
- **(int) state:** Será el estado actual del nodo. Su utilidad será simplemente poder mostrar información adicional en pantalla.
- **(boolean) candidate:** Será una variable booleana que indique si el nodo es candidato o no para ser elegido coordinador.
- **(AtomicInteger) msgSent:** Será un contador que indique la cantidad de mensajes enviados y, por lo tanto, de los que se debería esperar una respuesta. Se hizo uso de la clase *AtomicInteger* para asegurar la consistencia del contador, debido al trabajo con *threads*.

Luego, para construir el algoritmo principal, se comenzó enviando un mensaje en cada nodo a todos los que tengan un identificador mayor, adjuntando el propio, haciendo uso del método *multicast(...)*, para luego esperar una cantidad de tiempo igual a la entregada a través del parámetro *No answer time*, y finalmente revisar si el nodo fue elegido coordinador, verificando si aún es candidato y si se recibió respuesta de todos los mensajes enviados. En caso de ser coordinador, se cambia el estado y se notifica esto a todos los nodos de la red a través del método *broadcast(...)*, y adjuntando su identificador en el mensaje.

Paralelamente, para construir el algoritmo de recepción de mensajes, se verifica primero el tipo de mensaje que se recibió. Si es **0**, significa que es un mensaje de elección desde un nodo esperando respuesta, por lo que se revisa identificador en el mensaje y se envía un mensaje de respuesta en caso de que el identificador del remitente es menor al del nodo que recibió. Si es **1**, significa que será un mensaje de respuesta, por lo que basta con disminuir el contador *msgSent* e indicar a través de *candidate* que el nodo ya no es candidato para ser elegido. Por último, si es **2**, significa que es un mensaje notificando el identificador del nodo coordinador elegido, por lo que solo se guarda en la variable *coordinator*.

Se hace importante destacar que se ocupó una forma simple de asignar un identificador a cada nodo a través de un número aleatorio. Sin embargo, esto no asegura que no se pueda repetir alguno, por lo que es posible implementar una extensión de la clase *Network*, donde a nivel de la red se vayan guardando los números ya ocupados, y se eviten estas colisiones.

### 5.1.2. Resultados

Luego de haber implementado el algoritmo distribuido, se procedió a ejecutar el programa, obteniendo como resultado lo que se ve en la figura 21, detallado en cinco pasos:

1. Se aprecia la red con todos los nodos conectados entre sí, cada uno con su número identificador único, además del parámetro *No answer time*, habilitado para modificarse antes de comenzar a ejecutar el algoritmo.
2. Una vez comenzado el algoritmo, se ve cómo empiezan a transitar mensajes entre los nodos, y donde se muestra la descripción de uno de ellos, siendo el identificador de la clase para casos en que no se sobrescribe el método *getDescription(...)*. Esta es el momento donde se comienzan a enviar mensajes de elección y respuesta.
3. Luego de transcurrido un tiempo, se aprecia en este paso nodos en rojo, lo que significa que pasaron al estado 1, dejando de ser candidatos para ser elegidos como coordinador. Además, se muestra la información adicional de uno de ellos donde, a través de *getDescription(...)*, se implementó para mostrar el identificador y estado del nodo.
4. Una vez cumplido el tiempo indicado por el parámetro *No answer time*, el nodo con número identificador 94 concluye que fue elegido coordinador ya que no recibió res-

puesta. Esto se verifica al ver todos los demás nodos en estado 1, siendo este el único restante. Así, les envía un mensaje a todos notificando esta elección.

5. Finalmente, concluye el algoritmo, con el nodo 94 elegido como coordinador, y todos los demás nodos notificados de esto.

## 5.2. Algoritmo de Chang/Roberts

También conocido como *The Chang/Roberts algorithm*, corresponde a un algoritmo distribuido de **elección** en una topología de **anillo**. Se considera una variante del Algoritmo del Matón, con el mismo fin pero en una red con topología diferente.

Es también parte del programa del ramo de Sistemas Distribuidos y es un buen representante de los algoritmos de elección, ya que complementa al Algoritmo del Matón añadiendo una nueva topología, donde no necesariamente se debe conocer toda la red entre sí.

### 5.2.1. Implementación

En primer lugar, se decidió trabajar con una red de 8 nodos, conectados por canales unidireccionales formando una topología de anillo. Se definieron dos nodos iniciadores, por lo tanto dos candidatos a ser coordinador, y no se encontró necesario utilizar parámetros. Por otro lado, solo bastó definir dos estados lógicos para cada nodo:

- **0**: Solo indica que el nodo no ha sido escogido coordinador. Es el estado inicial y está representado por el color celeste.
- **1**: Indica que el nodo fue elegido coordinador. Está representado por el color verde.

Por otra parte, se consideró necesario una implementación de la clase *Message* donde simplemente se pueda adjuntar el número identificador del nodo que lo envía, quedando de la siguiente manera:

```
public class Msg extends Message {  
    private int id;  
  
    public Msg(int id) {  
        this.id = id;  
    }  
  
    public int getId() {  
        return id;  
    }  
}
```

Así, para escribir el algoritmo, se hizo la siguiente implementación de la clase *Program*:

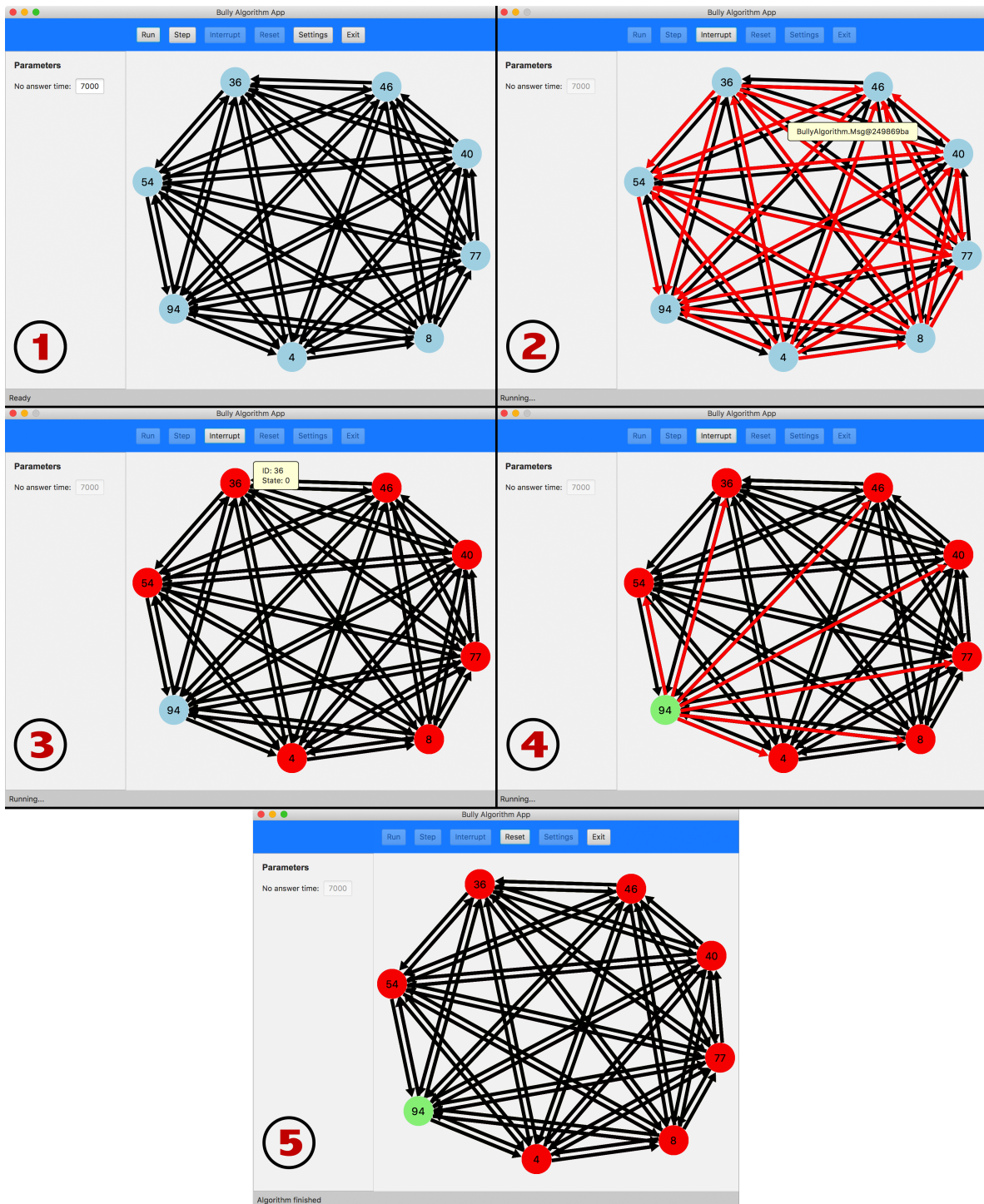


Figura 21: Implementación del Algoritmo del Matón.  
Fuente: Elaboración Propia.

```

public class Prog extends Program {
    private static Random random = new Random();
    private int id;
    private AtomicInteger coordinator = null;

    @Override
    protected void init() {
        id = random.nextInt(99);
        coordinator = new AtomicInteger(-1);
        setInitialNodeName(String.valueOf(coordinator.get()));
    }

    @Override
    protected void start() {
    }

    @Override
    protected void main() throws InterruptedException {
        if (isInitiator()) {
            synchronized (coordinator) {
                coordinator.set(id);
                setNodeName(String.valueOf(coordinator.get()));
            }
            send(0, new Msg(id));
        }

        @Override
        protected void onMessageReceive(Node src, Message message) throws InterruptedException {
            Msg msg = (Msg) message;
            if (coordinator.get() < msg.getId()) {
                synchronized (coordinator) {
                    coordinator.set(msg.getId());
                    setNodeName(String.valueOf(coordinator.get()));
                }
                send(0, new Msg(msg.getId()));
            } else if (coordinator.get() == msg.getId()) {
                // I'm the leader
                setState(1);
            }
        }
    }
}

```

Para comenzar, se le dio a cada nodo un número identificador aleatorio. Además, se creó el campo (*AtomicInteger*) **coordinator**, el cual registrará el número identificador del nodo que por el momento se considera como coordinador de la red. Este tendrá valor inicial -1, indicando que no hay un posible coordinador debido a que no han transitado mensajes, y su valor se mostrará como el nombre del nodo en pantalla, para tener una vista rápida de lo que está ocurriendo en el algoritmo.

Luego, para construir el algoritmo principal solo bastó verificar si el nodo es iniciador o no. En caso de que así sea, se actualizará la variable *coordinator* con su número identificador, se hará el cambio correspondiente en el nombre, y se comenzará enviando un mensaje a través del único canal saliente que tendrá, incluyendo su identificador en este. En caso contrario, no se hará ninguna operación.

Paralelamente, para construir el algoritmo de recepción de mensajes, se verifica el número identificador adjunto en el mensaje. En caso de que este sea mayor al actual número re-

gistrado en *coordinator*, se actualiza este último con el identificador recibido y se refleja su cambio en pantalla a través de *setNodeName(...)*; todo esto dentro de una sección *synchronized* para asegurar la consistencia de la variable, para luego transmitir el número recibido a través de un nuevo mensaje al siguiente nodo. Por otro lado, en caso de que el número identificador recibido sea igual al almacenado en *coordinator*, significa que el nodo que recibe fue elegido coordinador dentro de la red, por lo que cambia su estado a 1. Por último, si ninguno de los dos casos anteriores ocurre, simplemente se extinguirá el mensaje, lo que significa que hay otro nodo más indicado para ser coordinador.

Por último, al igual que en el Algoritmo del Matón, se hace importante notar que aquí tampoco se asegura la unicidad de los números identificadores de cada nodo, por lo que es posible implementar una extensión de la clase *Network*, donde a nivel de la red se vayan guardando los números ya ocupados, y se eviten estas colisiones.

### 5.2.2. Resultados

Luego de haber implementado el algoritmo distribuido, se procedió a ejecutar el programa, obteniendo como resultado lo que se ve en la figura 22, detallado en cuatro pasos:

1. Se aprecia la red con los nodos conectados en una topología de anillo, cada uno con nombre igual a -1, indicando que no se ha elegido un posible coordinador en ninguno de ellos.
2. Una vez comenzado el algoritmo, se ve cómo empiezan a transitar mensajes entre los nodos, todo generado desde los dos iniciadores que se identifican con una circunferencia de distinto color en el borde. Así, se ve que algunos ya tienen algún posible coordinador registrado, además de la descripción de un mensaje en tránsito, donde se ve el número identificador que este último transmite, lo cual se implementó a través de *getDescription(...)* de la clase *Message*.
3. Luego de transcurrido un tiempo, se aprecia que los nodos comienzan a converger en considerar al nodo con número identificador 32 como el coordinador de la red. Además, se ve la información adicional de uno de los nodos iniciadores, donde se ve que su identificador es el número 14, el cual se fue extinguiendo a lo largo de la ejecución.
4. Finalmente, concluye el algoritmo, indicando al nodo iniciador con número identificador 32 como coordinador y, por lo tanto, con estado 1, y todos los demás nodos notificados de esto.

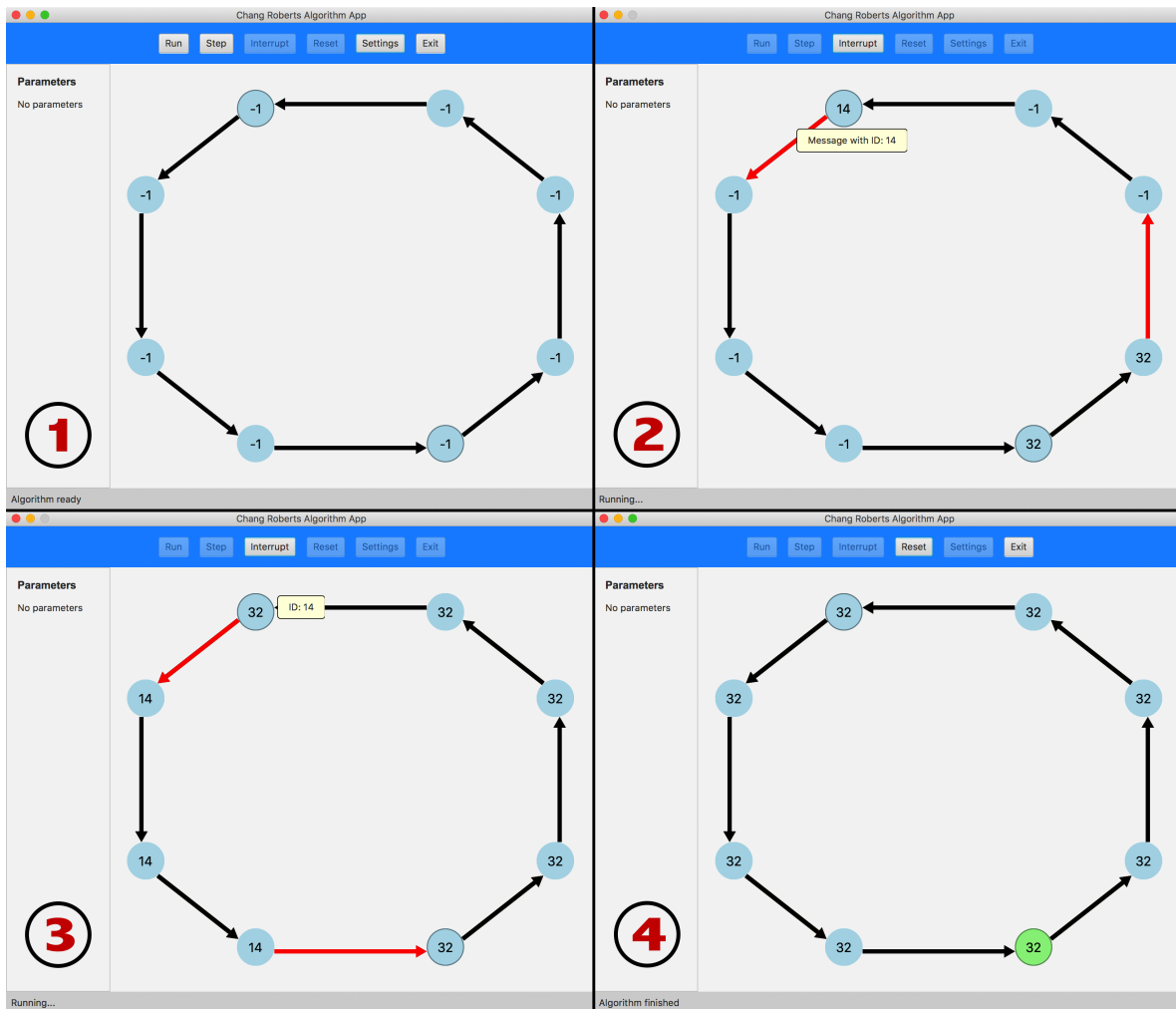


Figura 22: Implementación del Algoritmo de Chang/Roberts.  
Fuente: Elaboración Propia.

### 5.3. Algoritmo de Eco

También conocido como *The Echo algorithm*, corresponde a un algoritmo distribuido de **recorrido** en una topología de **árbol**. Su fin es recorrer toda la red de nodos a través de mensajes exploradores, e identificar la ruta de regreso a través de mensajes de eco, los cuales se envían desde cada nodo una vez haya recibido mensajes de todos sus vecinos.

Forma al igual que los anteriores parte del programa del ramo de Sistemas Distribuidos y es una muestra clara de cómo funcionan los algoritmos de recorrido, identificando las rutas donde transitó cada tipo mensaje, al mismo tiempo que recorre toda la red.

### 5.3.1. Implementación

En primer lugar, se decidió trabajar con una red de 5 nodos conectados en forma de árbol, siendo el nodo raíz el iniciador, y no se vio la necesidad de utilizar parámetros. Por otro lado, se definieron tres estados lógicos para cada nodo:

- **0:** Si el nodo no ha sido comprometido, es decir, no ha recibido mensajes. Será el estado inicial y está representado por el color celeste.
- **1:** Si el nodo fue comprometido pero aún se espera el mensaje de eco correspondiente a uno o más de los mensajes exploradores enviados. Está representado por el color verde.
- **2:** Si el nodo fue comprometido y recibió un mensaje de eco por cada mensaje explorador enviado. Está representado por el color lavanda o violeta claro.

Por otra parte, se consideró necesario una implementación de la clase *Message* donde se pueda indicar si es de tipo explorador o eco, agregando un campo para esto y quedando de la siguiente manera:

```
public class Msg extends Message {
    private int mode; // 0: EXPLORER, 1: ECHO

    public Msg(int mode) {
        this.mode = mode;
    }

    public boolean isExplorer() {
        return mode == 0;
    }

    public boolean isEcho() {
        return mode == 1;
    }
}
```

Así, para escribir el algoritmo, se hizo la siguiente implementación de la clase *Program*:

```
public class Prog extends Program {
    private static int cont = 1;
    private int id;
    private Node origin;
    private boolean compromised = false;
    private int msgReceived = 0;
    private int state = 0;

    @Override
    protected void init() {
        this.id = cont++;
    }

    @Override
    protected void start() {
    }
}
```

```

@Override
protected void main() throws InterruptedException {
    if (isInitiator()) {
        compromised = true;
        broadcast(new Msg(0));    // Explorer
        state = 1;
        setState(1);
    }
}

@Override
protected void onMessageReceive(Node src, Message message) throws InterruptedException {
    Msg msg = (Msg) message;
    int neighborsNum = getInChannels().length;

    if (msg.isExplorer()) {
        if (!compromised) {
            compromised = true;
            msgReceived = 0;
            origin = src;
            broadcastExcept(src, new Msg(0));    // Explorer
            state = 1;
            setState(1);
        }
    }

    msgReceived++;
    if (msgReceived == neighborsNum) {
        compromised = false;

        if (!isInitiator()) {
            sendTo(origin, new Msg(1)); // Echo
        }

        state = 2;
        setState(2);
    }
}
}

```

Para comenzar, se le dio a cada nodo un número identificador generado a través de un contador estático (*int*) **cont**, solo con fines de mostrar mayor información en pantalla. Por otra parte, se crearon los siguientes campos de utilidad:

- (*Node*) **origin**: Será el nodo de origen, es decir, el nodo del que se recibió el primer mensaje explorador y, posteriormente, al que se le deberá responder con un mensaje de eco.
- (*boolean*) **compromised**: Será una variable booleana que indique si el nodo ha sido comprometido o no.
- (*int*) **msgReceived**: Será una contador que indique la cantidad de mensajes recibidos, con el fin de saber cuántos más deberían esperarse.
- (*int*) **state**: Será una variable que indique el estado lógico actual del nodo. Su utilidad será simplemente poder mostrar información adicional en pantalla.

Luego, para construir el algoritmo principal solo bastó verificar si el nodo es iniciador o no. En caso de que así sea, se indicará que el nodo está comprometido a través de *compromised*, se enviará un mensaje explorador a todos los vecinos de este en la red a través de *broadcast(...)*, y se actualizará su estado a 1. En caso contrario, no se hará ninguna operación.

Paralelamente, para construir el algoritmo de recepción de mensajes, primero se obtiene la cantidad de vecinos del nodo en la red a través del tamaño del arreglo entregado por *getInChannels(...)*, siendo lo mismo que el tamaño del arreglo entregado por *getOutChannels(...)*, debido a que son todos canales bidireccionales. Luego, en caso de que el mensaje sea del tipo explorador y el nodo aún no haya sido comprometido, se actualizarán las variables correspondientes de la misma forma que lo hizo el iniciador, para luego guardar el nodo que envió el mensaje en *origin*, y enviar un mensaje explorador a todos los vecinos menos a este último, a través de *broadcastExcept(...)*. Después, se aumentará el contador de *msgReceived*, y se verificará si ya se recibió respuesta de todos los nodos vecinos. En caso de que así sea, el nodo dejará de estar comprometido y, siempre y cuando no sea el nodo iniciador, se enviará un mensaje de eco al nodo de origen a través de *sendTo(...)*, para finalmente actualizar su estado a 2.

Cabe destacar que para este caso se sobrescribió el método de *getChannelColor(...)* en la implementación de la clase *App*, con el fin de identificar los canales donde no ha pasado ningún mensaje, donde ha pasado un mensaje explorador y donde ha pasado un mensaje de eco, quedando de la siguiente manera:

```
@Override
protected Color getChannelColor(Channel.State state, Message lastMessage) {
    if (lastMessage != null) {
        Msg msg = (Msg) lastMessage;
        if (msg.isExplorer()) {
            return Color.DARKRED;
        } else {
            return Color.GREEN;
        }
    } else {
        return Color.BLACK;
    }
}
```

### 5.3.2. Resultados

Luego de haber implementado el algoritmo distribuido, se procedió a ejecutar el programa, obteniendo como resultado lo que se ve en la figura 23, detallado en cuatro pasos:

1. Se aprecia la red con los nodos conectados en una topología de árbol, con el nodo iniciador correspondiendo a la raíz en esta, y cada uno con un nombre para identificarlos, pero que no tienen relación alguna con el algoritmo.
2. Una vez comenzado el algoritmo, se ve cómo empiezan a enviarse mensajes exploradores desde el nodo iniciador con estado 1, a través de la descripción de uno de los

mensajes en tránsito y el color rojo de los canales.

3. Luego de transcurrido un tiempo, se aprecia que ya transitaban mensajes exploradores por toda la red, y cómo desde los nodos hoja del árbol se empiezan a enviar mensajes de eco, reflejándose en sus estados y el color verde de los canales. Se ve también la información adicional de uno de los nodos, donde se indica su estado y número de *thread*.
4. Finalmente, concluye el algoritmo con todos los nodos en estado 2, indicando que todos enviaron los mensajes de eco respectivos y quedando esto evidenciado a través del color de los canales, donde se ve la ruta que tomaron.

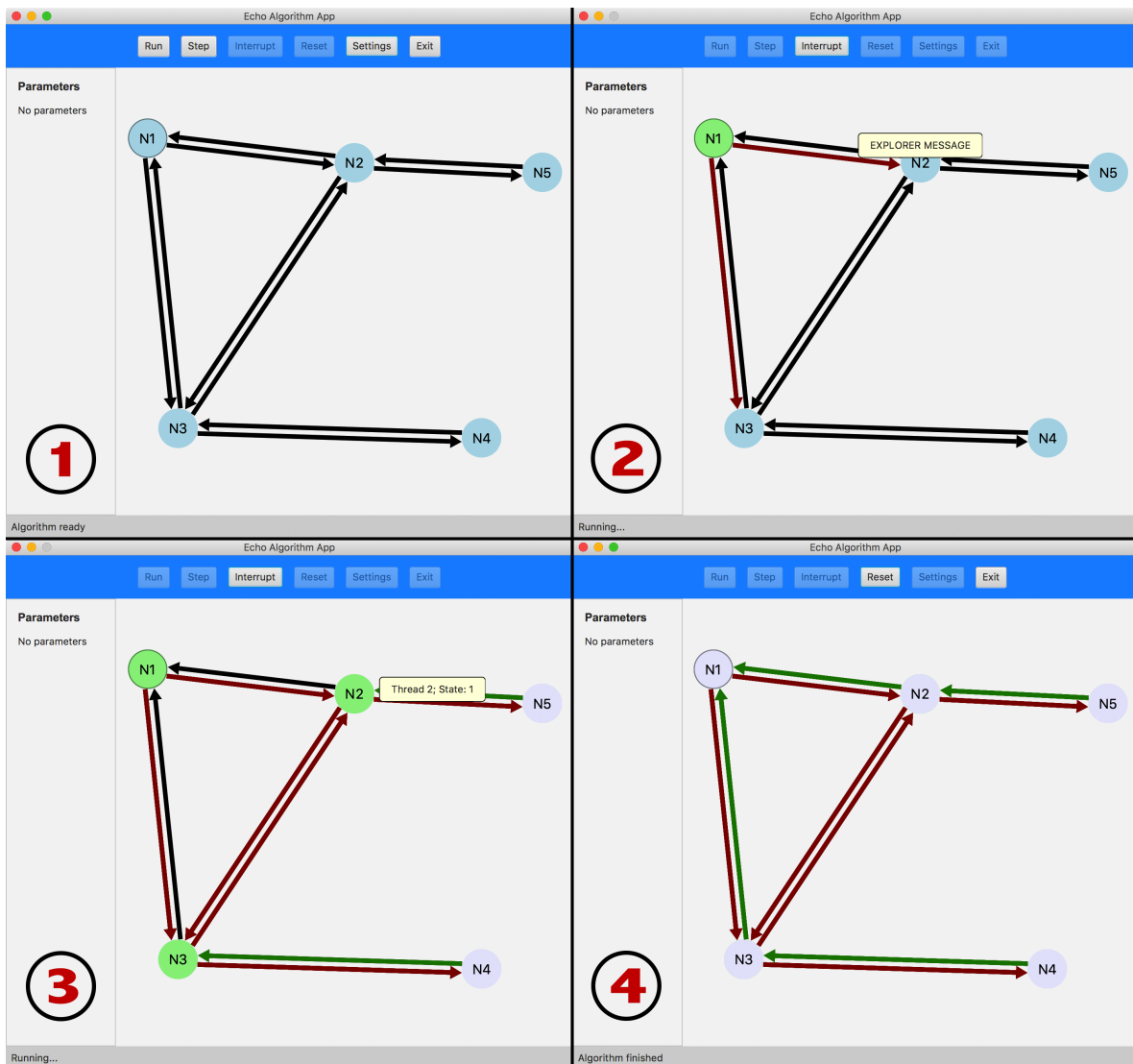


Figura 23: Implementación del Algoritmo de Eco.  
Fuente: Elaboración Propia.

## 5.4. Algoritmo de Suzuki-Kasami

También conocido como *The Suzuki-Kasami algorithm*, corresponde a un algoritmo distribuido de **sincronización distribuida** en una topología de **grafo general**. Así, su fin es asegurar que cada proceso o nodo de la red ejecute una denominada *sección crítica* de forma sincrónica, es decir, solo uno a la vez. Para lograr esto, se utiliza un objeto único denominado *token*, transmitido a través de mensajes, el cual permite al nodo que lo obtiene ejecutar la sección crítica, mientras que los demás lo solicitan y quedan en espera de este. Es importante destacar que no tiene un término definido, ya que está considerado en una ejecución constante de procesos.

Este último también es parte del programa del ramo de Sistemas Distribuidos y es característico de los algoritmos de sincronización distribuida, ya que tal como lo dice, se asegura de sincronizar la ejecución de una sección de instrucciones logrando exclusión mutua entre los procesos, donde todos deben conocerse para lograr una coordinación a nivel de red.

### 5.4.1. Implementación

En primer lugar, se decidió trabajar con una red de 4 nodos, todos conectados entre sí, y donde haya un nodo iniciador. Además, se definió un parámetro llamado *Critical Section Time*, con identificador 0, para indicar el tiempo que cada proceso ocupa para estar dentro de la sección crítica, al momento de obtener el *token*. Junto con esto, también se definió un parámetro llamado *Waiting Time*, con identificador 1, para indicar cuánto tiempo pasará desde que el proceso salió de la sección crítica para que vuelva a solicitar el *token*.

Por otro lado, se definieron cuatro estados lógicos para cada nodo:

- **0:** A veces identificado como *IDLE*, indicará que el proceso está ocioso, debido a que ya ocupó el *token* pero no hay más procesos que lo soliciten. Será el estado inicial y está representado por el color blanco.
- **1:** A veces identificado como *BUSY*, indicará que el proceso está con el *token* y dentro de la sección crítica. Está representado por el color rojo.
- **2:** A veces identificado como *TOKEN\_IDLE*, indicará que el proceso solicitó el *token* y está en espera de este. Está representado por el color amarillo.
- **3:** A veces identificado como *WAITING*, indicará que el proceso entregó el *token* a otro y está esperando un tiempo para volver a solicitarlo. Está representado por el color cian.

Por otra parte, siendo el *token* parte esencial del algoritmo, se encontró necesario crear una clase que lo represente:

```

public class Token {
    private Queue<Integer> queue;
    private int[] ln;

    public Token(int size) {
        queue = new LinkedList<>();
        ln = new int[size];
        for (int i=0; i < ln.length; i++) {
            ln[i] = 0;
        }
    }

    public int getLN(int i) {
        return ln[i];
    }

    public void setLN(int i, int value) {
        ln[i] = value;
    }

    public boolean queueContains(int id) {
        return queue.contains(id);
    }

    public boolean isQueueEmpty() {
        return queue.isEmpty();
    }

    public void addId(int id) {
        queue.add(id);
    }

    public int popId() {
        return queue.remove();
    }
}

```

En esta, se contendrá una cola *queue* donde se almacenen los identificadores de los nodos solicitantes, y un arreglo *ln* donde se contenga el número de secuencia de la última solicitud realizada por cada nodo, y la cual inicialmente es 0. Junto con esto, se establecen métodos para obtener acceder y modificar los valores de la cola y el arreglo *ln*.

Con respecto a los mensajes, se consideró necesario una implementación donde se pueda indicar el identificador del nodo que envía el mensaje, el tipo de este último (de solicitud o de envío del *token*), y que permita adjuntar el número de secuencia o el *token*, según corresponda. Todo esto se ve reflejado en el código siguiente:

```

public class Msg extends Message {
    private int senderId;
    private int type;           // 0: Request; 1: Token
    private int sn;
    private Token token;

    public Msg(int id, int type, Token token) {
        senderId = id;
        this.type = type;
        sn = -1;
        this.token = token;
    }

    public Msg(int id, int type, int sn) {

```

```

        senderId = id;
        this.type = type;
        this.sn = sn;
        token = null;
    }

    public int getSenderId() {
        return senderId;
    }

    public int getSN() {
        return sn;
    }

    public Token getToken() {
        return token;
    }

    public boolean isRequest() {
        return type == 0;
    }
}

```

Así, utilizando lo anterior, se llegó a la siguiente implementación de la clase *Program*:

```

public class Prog extends Program {
    private static int cont = 0;
    private int id;
    private AtomicInteger[] rn;
    private Token token;
    private int state;

    @Override
    protected void init() {
        token = null;
        state = 0; // IDLE
        id = cont++;
        setInitialNodeName(String.valueOf(id));
    }

    @Override
    protected void start() {
        int neighborsNum = getOutChannels().length;

        cont = 0; // Reset counter for next execution

        rn = new AtomicInteger[neighborsNum + 1];
        for (int i=0; i < rn.length; i++) {
            rn[i] = new AtomicInteger(0);
        }
    }

    private synchronized boolean hasToken() {
        return token != null;
    }

    public int getId() {
        return id;
    }

    private void setInternalState(int state) throws InterruptedException {
        this.state = state;
        setState(this.state);
    }
}

```

```

private void inCriticalSection() throws InterruptedException {
    setInternalState(1); // BUSY
    sleep((Integer) getParameter(0));
}

private void waiting() throws InterruptedException {
    setInternalState(3); // WAITING
    sleep((Integer) getParameter(1));
}

private void requestToken() throws InterruptedException {
    setInternalState(2); // TOKEN_IDLE
    int sn = rn[id].incrementAndGet();
    broadcast(new Msg(id, 0, sn)); // Request
}

private void sendToken(int dstId) throws InterruptedException {
    sendAt(new Msg(dstId, 1, token), (c, message) -> {
        Prog program = (Prog) c.getDst().getProgram();
        return program.getId() == dstId;
    });
    token = null;
}

private void withToken() throws InterruptedException {
    // Enter in CS
    inCriticalSection();

    // Update token queue values
    token.setLN(id, rn[id].get());

    // Add missing requesters to token queue
    synchronized (rn) {
        Channel[] outChannels = getOutChannels();
        for (Channel c : outChannels) {
            Prog program = (Prog) c.getDst().getProgram();
            int dstId = program.getId();

            if (!token.queueContains(dstId)) {
                synchronized (rn[dstId]) {
                    if (rn[dstId].get() == token.getLN(dstId) + 1) {
                        token.addId(dstId);
                    }
                }
            }
        }
        if (token.isQueueEmpty()) {
            setInternalState(0); // IDLE
        }
    }

    // Send token
    if (!token.isQueueEmpty()) {
        sendToken(token.popId());
        // Wait for a while
        waiting();
        requestToken();
    }
}

@Override
protected void main() throws InterruptedException {
    if (isInitiator()) { // Create Token
        token = new Token(getOutChannels().length + 1);
        withToken();
    } else {

```

```

        requestToken();
    }
}

@Override
protected void onMessageReceive(Node src, Message message) throws InterruptedException {
    Msg msg = (Msg) message;

    if (msg.isRequest()) { // Request
        int senderId = msg.getSenderId();
        synchronized (rn[senderId]) {
            rn[senderId].set(Math.max(rn[senderId].get(), msg.getSN()));

            if (hasToken() && state == 0 && rn[senderId].get() == token.getLN(senderId) + 1) {
                // Has token and is IDLE
                synchronized (token) {
                    sendToken(senderId);
                }
                waiting();
                requestToken();
            }
        }
    } else { // Token received
        token = msg.getToken();
        withToken();
    }
}
}

```

Para comenzar, se le dio a cada nodo un número identificador generado a través de un contador estático (*int*) **cont**. Por otra parte, se crearon los siguientes campos de utilidad:

- (*AtomicInteger[]*) **rn**: Será el arreglo que registrará el número de secuencia mayor recibido de cada proceso, donde la posición coincidirá con el identificador de este último.
- (*Token*) **token**: Será la variable que contendrá al *token* cuando se reciba.
- (*int*) **state**: Será una variable que indique el estado lógico actual del nodo.

Así, se iniciaron las variables respectivas en *init(...)* a excepción de *rn*. La razón de esto es que al momento en que se crea la instancia de la implementación de la clase *Program*, la red aún no está completa, es decir, aún no se encuentran todos los nodos instanciados. En consecuencia, no es posible saber con seguridad la cantidad de procesos que tendrá la red (a no ser que se trabaje con una red conocida). Es por esto que se hizo uso del método *start()*, donde ya se tiene la seguridad que la red está completa y, por lo tanto, es posible obtener la cantidad de nodos a través de la número de canales salientes, instanciando así el arreglo *rn* con el tamaño indicado. Como detalle, se aprovecha de volver a dejar en 0 el contador, para mantener la consistencia entre los identificadores y las posiciones en el arreglo en futuras ejecuciones del mismo algoritmo, debido a su característica de ser estático.

Luego, para organizar de mejor manera el código, se decidió implementar métodos que ejecuten ciertas tareas de acuerdo al estado del nodo, y que van directamente relacionadas con el funcionamiento del algoritmo:

- ***boolean hasToken()***: Indica si el nodo tiene el *token* o no. Se utiliza *synchronized* para evitar inconsistencias entre el algoritmo principal y el algoritmo de recepción de mensajes.
- ***void setInternalState(...) throws InterruptedException***: Se utiliza para cambiar el estado del nodo, actualizando la variable *state* e indicando el cambio internamente a través de *setState(...)*. Recibe como parámetro (*int*) ***state***, correspondiente al nuevo estado.
- ***void inCriticalSection() throws InterruptedException***: Serán las instrucciones a ejecutar al momento que el nodo entra en la sección crítica. En este caso, basta simplemente con cambiar el estado a *BUSY*, y suspender el *thread* por la cantidad de tiempo indicada en el parámetro *Critical Section Time*. Se agrega en la definición *throws InterruptedException* para hacer llegar la excepción, en caso de que ocurra, al algoritmo principal o algoritmo de recepción de mensajes, según corresponda.
- ***void waiting() throws InterruptedException***: Serán las instrucciones a ejecutar luego de que el nodo entregue el *token* a otro proceso. Basta con cambiar el estado a *WAITING*, y suspender el *thread* por la cantidad de tiempo indicada en el parámetro *Waiting Time*. Se agrega en la definición *throws InterruptedException* para hacer llegar la excepción, en caso de que ocurra, al algoritmo principal o algoritmo de recepción de mensajes, según corresponda.
- ***void requestToken() throws InterruptedException***: Serán las instrucciones a ejecutar cada vez que el nodo quiera solicitar el *token*. Para esto, se cambia el estado a *TOKEN\_IDLE*, se incrementa y obtiene el número de secuencia correspondiente al proceso actual, y se envía la solicitud a todos los vecinos en la red a través del método *broadcast(...)*. Se agrega en la definición *throws InterruptedException* para hacer llegar la excepción, en caso de que ocurra, al algoritmo principal o algoritmo de recepción de mensajes, según corresponda.
- ***void sendToken(...) throws InterruptedException***: Serán las instrucciones a ejecutar por el nodo cada vez que vaya a enviar el *token* a otro. Recibe el parámetro (*int*) ***dstId***, correspondiente al identificador del proceso de destino, y, haciendo uso de *sendAt(...)*, le envía al nodo correspondiente un mensaje con el *token*, para luego cambiar la variable *token* a *null*, indicando que ya no lo posee. Se agrega en la definición *throws InterruptedException* para hacer llegar la excepción, en caso de que ocurra, al algoritmo principal o algoritmo de recepción de mensajes, según corresponda.
- ***void withToken() throws InterruptedException***: Serán las instrucciones a ejecutar cada vez que el nodo tenga el *token* y quiera entrar a la sección crítica, generalmente luego de haberlo recibido. Para esto, se entra primero a la sección crítica llamando a *inCriticalSection()*. Luego, obteniendo acceso único a la variable *rn*, se actualiza la cola en el *token*, agregando el identificador de los procesos que no estén en esta y que cumplan con que el número de secuencia registrado en el arreglo del nodo actual sea

igual al último número de secuencia registrado en el *token* más 1. Finalmente, se verifica si la cola del *token* está vacía. Si esto es verdadero, significa que ya no hay procesos a quien enviar el *token*, por lo que el nodo pasa a estado *IDLE*. En caso contrario, se enviará el *token* al próximo nodo dentro de la cola de este, para luego esperar un tiempo, llamando a *waiting()*, y volver a solicitarlo, llamando a *requestToken()*. Se agrega en la definición *throws InterruptedException* para hacer llegar la excepción, en caso de que ocurra, al algoritmo principal o algoritmo de recepción de mensajes, según corresponda.

Ya con las herramientas anteriores, se procedió a construir el algoritmo principal, donde solo bastó verificar si el nodo es iniciador o no. En caso de que así sea, significa que es quien comenzará el algoritmo, por lo que creará el *token* y lo ocupará a través de *withToken()*. En caso contrario, lo solicita.

Paralelamente, para construir el algoritmo de recepción de mensajes, primero se revisa de qué tipo es. Si es una solicitud, se actualiza el arreglo *rn* con el máximo número de secuencia entre el registrado y el recibido a través del mensaje. Luego, en caso de que el actual nodo se encuentre en estado *IDLE*, tenga el *token* y corresponda enviar este al proceso solicitante, se utilizará el método *sendToken(...)* para pasárselo. Finalmente, en consecuencia, se esperará un tiempo para volver a solicitarlo. Por otro lado, en caso de que el mensaje no sea una solicitud, y por lo tanto contenga el *token*, se guardará este y entrará a la sección *withToken()*.

Cabe destacar aquí la importancia del número de identificador, ya que además de identificar a cada nodo, la posición de los arreglos *rn* y *ln* deben coincidir con este. Es por esto que, si bien la implementación se pensó para ejecutar varias veces el mismo algoritmo en la misma red, no está hecha para ser ejecutada en redes creadas durante la ejecución. Lo anterior debido a que el método *start()* no correrá al modificar la red en modo de diseño, dentro de *Settings*, obteniendo números identificadores que no partirán en 0 y, por lo tanto, no coincidirán con la posición en los arreglos. Para darle solución esto, es posible implementar una extensión de la clase *Network*, la cual tenga un contador interno y se encargue de entregar el siguiente identificador disponible. Con esto, se asegura que cada red tenga su propio contador, evitando así inconsistencias.

#### 5.4.2. Resultados

Luego de haber implementado el algoritmo distribuido, se procedió a ejecutar el programa, obteniendo como resultado lo que se ve en la figura 24, detallado en cuatro pasos:

1. Se aprecia la red con todos los nodos conectados entre sí, y nodo iniciador con número identificador igual a 1. Además, se ven los dos parámetros definidos *Critical Section Time* y *Waiting Time*, disponibles para ser modificados antes de comenzar la ejecución del algoritmo.

2. Una vez comenzado el algoritmo, se ve que el proceso 1, al ser el iniciador, comienza entrando a la sección crítica, mientras que los demás nodos solicitan el *token*, tal como se ve en la descripción de uno de los mensajes en tránsito, identificando a la vez el número de secuencia enviado desde el proceso 0.
3. Luego de que el nodo 1 haya salido de la sección crítica, se aprecia cómo pasa al estado *WAITING*, al mismo tiempo que envía el *token* al proceso 0, mientras que los demás nodos siguen esperando.
4. Finalmente, el nodo 0 recibe el *token* y entra a la sección crítica. Se ve además cómo quedó el arreglo del nodo 1, implementado a través del método *getDescription()* en la implementación de la clase *Program*, siendo este el único que aún no ha solicitado el *token*, debido a que fue el iniciador y aun no sale del estado *WAITING*.

Se hace notar que el algoritmo no tiene un término definido, por lo que los pasos anteriores explicados son suficientes para detallar cómo será su comportamiento a lo largo del tiempo.

## 5.5. Evaluación Global

Luego de haber hecho las cuatro implementaciones anteriores, es posible decir que se lograron resultados exitosos. Si bien los mayores esfuerzos en la programación se centraron en el Algoritmo de Suzuki-Kasami, aún así fue una forma simple de implementación. Esto es, la mayor parte del código se centró en los métodos destinados al algoritmo principal y algoritmo de recepción de mensajes, que es lo esencial de un algoritmo distribuido, disminuyendo los esfuerzos para indicar otros aspectos del mismo. Así, se destaca el uso de parámetros tales como tiempos de espera, posibles de modificar directamente en la interfaz, facilitando en gran parte la cualidad de variabilidad en el programa y sin tener que modificar directamente el código para luego volver a ejecutarlo.

Por último, se ve que se ofrecen los suficientes lugares para mostrar toda la información necesaria del algoritmo, considerando nombre de los nodos, descripción de los nodos y descripción de los mensajes, pudiendo personalizarlos de acuerdo a lo que sea importante para el usuario, de forma libre.

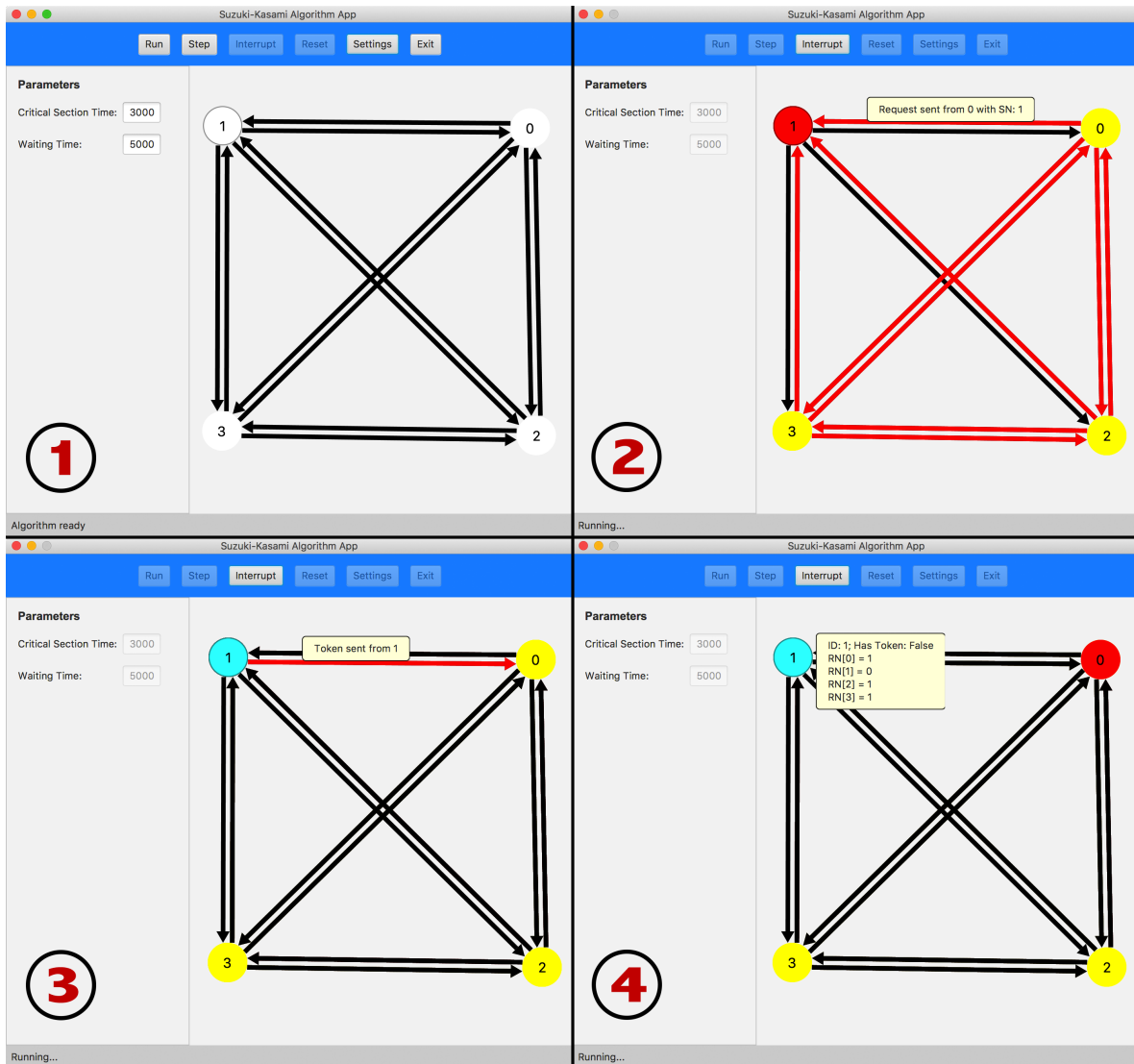


Figura 24: Implementación del Algoritmo de Suzuki-Kasami.  
Fuente: Elaboración Propia.

## CAPÍTULO 6

### CONCLUSIONES

#### 6.1. Objetivos y Resultados

A modo de conclusión, y basándose en todo el desarrollo presentado anteriormente, es posible decir que se cumplió con los objetivos propuestos dentro de esta memoria. Una vez analizados los *toolkits* existentes para simular y visualizar algoritmos distribuidos dentro del capítulo de **Definición del Problema**, y basándose en todos sus aspectos detallados en el capítulo de **Marco Conceptual**, se procedió a diseñar un *framework* que permita una programación simple e intuitiva de lo último contrastando las tecnologías actuales para programar y construir interfaces de usuario en el capítulo de **Propuesta de Solución**, con el fin de después implementarlo en el capítulo de **Implementación de la Solución** y validarlo con cuatro algoritmos distribuidos de prueba en el capítulo de **Validación de la Solución**. Así, se logró una solución la cual permite simular y visualizar distintos tipos de algoritmos distribuidos, con ciertas características adicionales que lo hacen diferente a otras propuestas, tales como personalizar las redes de nodos o modificar parámetros durante la ejecución, pero sobre todo, ocupar el mismo procesador de la máquina para ejecutar los *threads* de forma paralela, permitiendo simular al máximo esta última cualidad.

Con respecto a los resultados obtenidos en los cuatro casos de prueba utilizados, es posible concluir que fueron satisfactorios. Por el lado de la simulación, se llegó a una forma intuitiva de programar, donde el usuario principalmente se preocupa de la implementación directa del algoritmo distribuido, y solo en caso de quererlo así, poder modificar otros detalles de la interfaz o el funcionamiento. Así, el fin de haber considerado los cuatro algoritmos que se eligieron, fue obtener justamente una retroalimentación del *framework*, cubriendo los distintos casos que podrían encontrarse. Y, en efecto, fueron de mucha utilidad, ya que permitieron identificar algunas necesidades que podría haber tenido el usuario, requiriendo, por ejemplo, una mayor abstracción de los métodos otorgados o una forma de registrar datos a nivel de red, entre otros, llevando, en consecuencia, a la implementación de esto en el *framework*. Por otro lado, en el ámbito de la visualización, se construyó una interfaz donde se identifica claramente cada área y las interacciones que el usuario tiene disponibles. El poder modificar los parámetros en este momento facilita en gran parte la simulación, al no tener que crear diferentes instancias de ejecución para variar los casos de prueba, de la misma forma que lo hace el poder modificar la red de nodos o importar alguna creada anteriormente, acelerando el proceso de desarrollo y apoyando al usuario para que solo se preocupe en lo que es el funcionamiento del algoritmo distribuido.

Es por todo lo anterior que se logró maximizar la abstracción y calidad de las herramientas entregadas al usuario a través del *framework* de la forma más indicada posible, logrando resultados que le permiten hacer una implementación **simple e intuitiva** de cada algoritmo distribuido, minimizando en gran parte el **esfuerzo** que esto comprende, y, de forma adi-

cional, agregarle una cualidad de **variabilidad**, pudiendo hacer pequeñas modificaciones del programa en forma **eficiente**.

## 6.2. Reflexión y Aprendizaje Logrado

Como reflexión de la memoria realizada, se destaca lo complejo que puede llegar a ser el trabajar con *threads*. Esto debido a todo lo que hay que considerar para lograr un correcto funcionamiento sin producir *deadlocks*<sup>21</sup> y, al mismo tiempo, mantener la consistencia de los datos. Fue un proceso difícil el cual requirió varias pruebas, debido a que, al ser cada ejecución no determinista, podrían haberse encontrado, por un lado, ejecuciones sin errores, mientras que en otras haber detectado un mal funcionamiento. Además, se tuvo que revisar en detalle todo el código escrito, considerando los métodos o variables compartidas accedidas por diferentes *threads*, entre otros.

En cuanto al área de los sistemas distribuidos, es un ámbito trabajoso en donde miles de casos pueden ocurrir, y, por lo tanto, muchas pruebas son necesarias. Para esto, se hace imprescindible contar con herramientas que permitan trabajar con ellos de una manera segura y lo más representativa posible. Sin embargo, es importante destacar que, aún haciendo múltiples pruebas, es difícil tener la seguridad de que un sistema distribuido estará libre de errores. Por un lado, el algoritmo utilizado puede tener un error en su lógica que, por ejemplo, solo ocurrirá cuando los *threads* se ejecuten en un orden y manera específica, siendo muy poco probable detectarlo en tiempo de ejecución. Es por esto que la mayor importancia estará en el análisis del código de forma detallada, considerando su lógica e implementación. Por otro lado, un sistema distribuido, al trabajar con diferentes nodos o sitios, puede tener fallas a lo largo de su ejecución, esto es, que un punto en la red deje de funcionar correctamente. Así, será también necesario establecer algoritmos que permitan recuperarse de estas situaciones, sin afectar el funcionamiento del sistema completo y permitiendo su estabilidad. Por lo tanto, en resumen, este tipo de sistemas deberá estar en un proceso de mejora constante, de análisis y supervisión continua, pero por sobre todo, con las herramientas necesarias para optimizar este proceso y simplificarlo en la mayor forma posible.

Es de acuerdo a todo lo mencionado anteriormente que se logró un gran aprendizaje a lo largo del proceso de desarrollo de este trabajo, desde manejo de la interfaz gráfica con *JavaFX*, considerando la construcción y el manejo de las interacciones del usuario con esta, hasta el trabajo con *threads*, debiendo asegurar la consistencia de los datos a través de operaciones atómicas junto con el uso de las declaraciones *synchronized* y *volatile*. En esto último se destaca también la estrategia ideada para pausar y resumir los *threads*. Al no tener control alguno sobre la programación de su ejecución, no había forma de detenerlos, por lo que la creación de puntos de interrupción en situaciones clave, utilizadas por todo algoritmo distribuido, tales como enviar mensajes, recibir mensajes, entre otras, fue una buena manera

---

<sup>21</sup>Situación en donde cada uno de los *threads* queda bloqueado esperando a que otro se libere, y en consecuencia, sin poder resumirse.

de abordar el problema y que se alinea con las metodologías utilizadas en épocas anteriores dentro de los sistemas operativos para simular trabajo con *multithreading*.

### 6.3. Contribuciones y Aplicaciones de la Solución

La contribución de este trabajo es hacia todas las áreas que trabajen con sistemas distribuidos. Se aporta con una forma simple e intuitiva de poder analizar el funcionamiento de algoritmos distribuidos, donde una vez implementado en código *Java*, es posible manipularlo a través de parámetros y diferentes redes de nodos en tiempo real. Así, se facilita el proceso de desarrollo y análisis, pudiendo el usuario enfocarse netamente en el algoritmo, y minimizar el tiempo que se ocupa en manejar otros elementos necesarios pero que salen de lo que es la ejecución del programa.

Es debido a lo mencionado que las aplicaciones de este trabajo estarán enfocadas primeramente en el área de la educación, permitiendo explicar el funcionamiento de algoritmos distribuidos de una forma lo suficientemente representativa para facilitar la comprensión por parte del estudiante. Además, por ejemplo, se podrán otorgar tareas o trabajos donde se proponga analizar en profundidad algún algoritmo distribuido, manipulándolo y viendo cómo varía su funcionamiento a través de diversas situaciones. A esto se le suma poder proponer **mejoras** a este y demostrarlas haciendo uso del mismo *framework*, o incluso crear un algoritmo completamente **desde cero**, con el fin de resolver un problema propuesto, entre muchas opciones más.

A pesar de lo anterior, la aplicación de la solución no está restringida por el área en que se aplique. Fuera de la educación, también puede ocuparse en otros ámbitos donde, por ejemplo, se ocupe algún algoritmo distribuido para el funcionamiento de un sistema. Debido a lo delicado que puede ser hacer un cambio en este, podría utilizarse el *framework* para simular cómo funcionaría el sistema en caso de modificarse, o analizarlo para buscar una falla que esté ocurriendo, siendo un sinfín de posibilidades.

En detalle, con respecto a los participantes involucrados en la problemática de este trabajo, es posible identificar los siguientes aportes en cada uno:

- **Profesor de la asignatura:** El *framework* que se ofrece aporta directamente al profesor de la asignatura, ya que es quien podrá hacer uso de este para dar a conocer el funcionamiento de los algoritmos distribuidos dentro del contenido número 4 en el programa de su curso, pudiendo, tal como se mencionó, entregar actividades que busquen que los estudiantes manipulen, analicen y propongan mejoras a los algoritmos distribuidos enseñados, siendo una forma más interactiva en comparación a la forma de enseñanza actual, que se basa en mostrar fotos de cada paso del algoritmo, limitando la interactividad con este, y además, siendo más complicado mostrar su funcionamiento en distintos escenarios.

- **Estudiante de la asignatura:** Será quien haga uso del *framework* para el aprendizaje, por lo que el aporte también es directo. Permitirá una mayor comprensión de los algoritmos distribuidos, interacción con estos, pero por sobre todo, apoyará el proceso de estudio, y servirá incluso como una herramienta motivacional para darle al estudiante la posibilidad de proponer un algoritmo diferente, o analizar uno en mayor profundidad. Podrá ejecutarse el mismo algoritmo en diferentes redes de nodos, con diferentes parámetros, por lo que el poder variar la ejecución para ver todos los casos posibles que pueden ocurrir es clave dentro del aporte que se hace hacia este actor.
- **Departamento de Informática:** Considerando que el Departamento de Informática está conformado por profesores y estudiantes, todo lo mencionado anteriormente se podrá expandir. Es decir, el aporte no solo será para el profesor y estudiantes de la asignatura de Sistemas Distribuidos, sino para todo profesor y estudiante participante de algún ramo del departamento que pueda hacer uso del *framework* y aplicarlo en su asignatura, por ejemplo, en el área de redes. Además, por otra parte, también se aporta al perfil de egreso del estudiante, y por lo tanto, a su formación, reforzando el aprendizaje y, por lo tanto, logrando un mayor dominio del tema.
- **UTFSM:** El aporte en esta institución estará principalmente enfocado en el perfil de egreso del estudiante y en su reconocimiento dentro del área laboral. Así, el *framework* apoya en la formación, lo que puede llevar a aplicaciones dentro del área de sistemas distribuidos hechos por la iniciativa del mismo estudiante, y así, lograr reconocimiento luego de egresar por sus trabajos realizados. En otras palabras, la solución de este trabajo entrega una oportunidad para que la institución sea reconocida dentro del ámbito, ya que, al mismo tiempo, es una herramienta que puede utilizarse en las asignaturas de todas las sedes de la universidad, y no solo en una como se podría pensar.
- **Otras Universidades:** Por último, en cuanto a los demás usuarios externos, el aporte del *framework* es directo hacia el área de docencia y el área de sistemas distribuidos. Si bien este se enfocó en una asignatura específica, queda abierta la posibilidad de que sea utilizado para cualquier fin en donde pueda ser de utilidad, tales como instituciones de investigación, desarrollo de *software*, entre muchos otros más.

## 6.4. Trabajo Futuro

Si bien se logró el objetivo pudiendo representar diversos tipos de algoritmos distribuidos, lo cierto es que se tuvo que excluir los algoritmos de tipo **instantánea distribuida**, debido a que su representación era diferente. Para esto, tal como se explicó en capítulos anteriores, se implementó un archivo de *log*, donde se registrarán todos los eventos ocurridos durante la ejecución del algoritmo implementado. Así, queda para trabajo futuro procesar este archivo y poder crear el **diagrama espacio-temporal** correspondiente. En esto, la complejidad estaría principalmente en la visualización, debido a que dentro del archivo ya está toda la

información necesaria. Considerando las propiedades de los relojes vectoriales, es posible organizar los eventos de cada proceso y establecer las relaciones de causalidad correspondientes, por lo que basta con leer el archivo, guardarlo en un formato fácil de procesar y llevar esto al diagrama.

Como propuesta a lo anterior, se podría implementar esta opción dentro del mismo *framework*, es decir, podría agregarse un botón o una vista nueva dentro del mismo, en la cual se permita visualizar el diagrama espacio-temporal del algoritmo recién ejecutado, ya que, si bien no todos pueden ser representados por un grafo, lo cierto es que todos tienen un diagrama espacio-temporal que representa los eventos ocurridos en ellos. Por otro lado, en caso de no querer seguir esta propuesta, bastaría con crear un programa totalmente aparte que procese los datos y muestre la representación respectiva en pantalla, donde la ventaja sería tener la libertad de utilizar las herramientas y lenguaje de programación que la persona que lo vaya a implementar prefiera, sin restricciones de ningún tipo más que poder leer archivos.

Cabe destacar que **se dejará a disposición el código del *framework* listo para cualquiera que lo requiera**, con el fin de utilizarlo, analizarlo, extenderlo o modificarlo en caso de querer adaptarlo a alguna otra necesidad en algún trabajo a futuro. Así, por ejemplo, se podrá aplicar la herramienta implementada en la clase de Sistemas Distribuidos u otra con contenido similar, y, a través de la retroalimentación que provean los estudiantes al hacer uso de esta, hacer mejoras, incluyendo, por ejemplo, mayor aleatoriedad en el funcionamiento o agregando métodos a nivel de usuario que permitan abstraer alguna operación requerida y que no formó parte de los casos de prueba. Lo mismo aplica también en caso de que se quieran medir tiempos en los que ocurren algunos eventos, pudiendo agregar una marca de tiempo en los registros del archivo de *log*, por ejemplo, a través de una variable que simule un cronómetro a nivel de red en la clase *Network*.

Además, complementando lo mencionado, el profesor puede requerir más adelante simular y visualizar algoritmos distribuidos que contengan fallas. Así, para este fin también se podrá obtener el código del *framework* y modificarlo para simular estos casos. Como recomendación, una forma simple de hacerlo sería modificar, dentro de la clase *Channel*, la sección donde se llama al método *receiveMessage(...)* del nodo de destino. En esta, se podrá establecer una forma no determinista, por ejemplo a través de una variable aleatoria, en donde de acuerdo al valor que esta obtenga decidir si se debe entregar el mensaje o no, para luego eliminarlo de la cola de mensajes. Así, para los casos en que no se entregue el mensaje se podrá simular una **falla** en el algoritmo y lograr el propósito deseado. Sin embargo, es importante hacer notar que no todos los algoritmos distribuidos están contruidos considerando ocurrencia de fallas, por lo que la propuesta anterior se podría controlar a través de un parámetro donde se indique si estas están permitidas en el algoritmo a implementar o no.

## ANEXOS

### ANEXO A: Implementaciones utilizando el Framework

Junto a esta memoria, además de incluir el *framework* implementado, se adjuntarán cuatro directorios:

- **BullyAlgorithm:** Será el código correspondiente a la implementación del Algoritmo del Matón utilizando el *framework*, con todas las mejoras correspondientes.
- **ChangRobertsAlgorithm:** Será el código correspondiente a la implementación del Algoritmo de Chang/Roberts utilizando el *framework*, con todas las mejoras correspondientes.
- **EchoAlgorithm:** Será el código correspondiente a la implementación del Algoritmo de Eco utilizando el *framework*, con todas las mejoras correspondientes.
- **SuzukiKasamiAlgorithm:** Será el código correspondiente a la implementación del Algoritmo de Suzuki-Kasami utilizando el *framework*, con todas las mejoras correspondientes.

## REFERENCIAS BIBLIOGRÁFICAS

- [Attiya(2004)] J. Attiya, Hagit y Welch, *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004, vol. 19.
- [Englert(2006)] D. y. M. G. y. S. A. Englert, Burkhard y Kowalski, "Distributed algorithms," 2006. [Online]. Available: <http://compalg.inf.elte.hu/~tony/Informatikai-Konyvtar/03-Algorithms%20of%20Informatics%201,%202,%203/Distributedf29May.pdf>
- [Fokkink(2013)] W. Fokkink, *Distributed algorithms: an intuitive approach*. MIT Press, 2013.
- [Gruner(2000)] M. y. B. M. Gruner, Stefan y Mosbah, "A new tool for the simulation and visualization of distributed algorithms," 2000.
- [Lynch(1996)] N. A. Lynch, *Distributed algorithms*. Elsevier, 1996.
- [McGinnis(1998)] S. W. McGinnis, "Dave : the distributed algorithm visualization engine," May 1998. [Online]. Available: <http://hdl.handle.net/1721.1/47588>
- [Monge(2017)] R. Monge, "Capítulo IV: Algoritmos distribuidos [Presentación PowerPoint]," 2017.
- [Moses(1998)] Z. y. T. A. y. U. L. Moses, Yoram y Polunsky, "Algorithm visualization for distributed environments," pp. 71–78, 154, Oct 1998.
- [Schreiner(2002)] W. Schreiner, "A java toolkit for teaching distributed algorithms," *SIGCSE Bull.*, vol. 34, no. 3, pp. 111–115, Jun. 2002. [Online]. Available: <http://doi.acm.org/10.1145/637610.544449>
- [Tel(2000)] G. Tel, *Introduction to distributed algorithms*. Cambridge university press, 2000.

## ÍNDICE DE PALABRAS CLAVE

AlgorithmView, 40, 42, 43  
App, 34, 46, 47, 50, 58, 69  
Assertion, 44  
Channel, 25, 38, 39, 49, 50, 84  
Constants, 44  
LogHandler, 44, 45  
Message, 39, 49, 54, 59, 62, 65, 67  
MessageQueue, 39, 40  
Network, 26, 34, 36–38, 40, 42, 47, 49,  
54, 61, 65, 77, 84

Node, 24, 37, 38, 49, 54  
Parameter, 35  
Program, 37, 38, 40, 42, 45–49, 52–54,  
62, 67, 73, 75, 78  
Screen, 34, 40, 42  
SettingsWindow, 42  
VectorClock, 44  
VisualChannel, 39  
VisualNode, 37, 38