Universidad Zaragoza

Trabajo Fin de Grado

Diseño e Implementación de Tolerancia a Fallos con Baja Latencia en Simulación Distribuida

Design and Implementation of Fault Tolerance with Low Latency in Distributed Simulation

Autor

Vela Tambo, Javier

Directores

Arronategui Arribalzaba, Unai

Bañares Bañares, José Ángel

ESCUELA DE INGENIERÍA Y ARQUITECTURA
Grado de Ingeniería Informática

2022

Diseño e Implementación de Tolerancia a Fallos con Baja Latencia en Simulación Distribuida

RESUMEN

El proyecto forma parte de la investigación en torno a un simulador distribuido de sistemas de eventos discretos. La naturaleza distribuida y escalable del simulador implica la presencia de fallos, por lo tanto, es fundamental contar con mecanismos de tolerancia a fallos en el sistema. En un entorno distribuido, existe un compromiso entre las prestaciones y la tolerancia a fallos, ya que el aumento en el número de mensajes y la sincronización, conlleva un incremento en la latencia y una reducción en el rendimiento. Obtener un rendimiento óptimo en una simulación tolerante a fallos es un desafío. Por lo tanto, el objetivo del proyecto es proponer un modelo de tolerancia a fallos original que preserve las prestaciones del sistema en ausencia de fallos.

La tolerancia a fallos implementada se basa en la replicación, la cual está adaptada para la simulación conservativa. Se aprovechan los mensajes y tiempos propios de la simulación para preservar la consistencia y se desacopla la ejecución de las réplicas, lo que reduce la cantidad de mensajes y sincronización necesarios. Mediante el desacoplamiento se logra una consistencia laxa que converge tras un fallo como resultado de un algoritmo diseñado para mantener el registro del estado de la simulación. Estos mecanismos también incluyen la detección de fallos entre nodos vecinos y la recuperación en caso de fallo. Adicionalmente, el diseño incluye un proceso externo que permite la incorporación dinámica de nuevos nodos para retomar la simulación después de un fallo.

Se ha logrado un rendimiento óptimo en la simulación mediante un enfoque innovador en el diseño de mecanismos de tolerancia a fallos. En ausencia de fallos, el coste adicional se limita al envío de eventos para persistir el estado en las réplicas, sin necesidad de sincronización adicional. Los resultados demuestran la eficacia de las estrategias implementadas para tolerar múltiples fallos durante la ejecución de la simulación. Además, se ha optimizado la implementación base mediante una gestión eficiente de conexiones y datos, reduciendo el tiempo de ejecución de manera significativa. Como resultado, el proyecto sienta una base sólida para futuras investigaciones y mejoras en la tolerancia a fallos de simulación distribuida.

Design and Implementation of Fault Tolerance with Low Latency in Distributed Simulation

SUMMARY

The project on which the paper focuses is part of the research line on a distributed simulator for discrete event systems. The distributed and scalable nature of the simulator implies the presence of faults, hence the need for fault tolerance mechanisms in the system. In a distributed environment, there is a trade-off between performance and fault tolerance, as an increase in the number of messages and synchronization leads to higher latency and reduced performance. Achieving optimal performance in a fault-tolerant simulation is a challenge. Therefore, the objective of the project is to propose an original fault tolerance model that preserves system performance in the absence of faults.

The implemented fault tolerance is based on replication, which is adapted for conservative simulation. Simulation-specific messages and timing are leveraged to maintain consistency, while the execution of replicas is decoupled, reducing the required messaging and synchronization. Through the decoupling a loose consistency is achieved, which converges after a failure as a result of an algorithm designed to keep track of the simulation state. These mechanisms also include fault detection among neighboring nodes and recovery in case of failure. Additionally, the design incorporates an external process which allows for dynamic incorporation of new nodes to resume the simulation after a failure.

By adopting an innovative approach in the design of fault tolerance mechanisms, optimal performance has been achieved in the simulation. In the absence of faults, the additional cost is limited to sending events to persist the state in the replicas, without the need of additional synchronization. The results demonstrate the effectiveness and correctness of the implemented strategies in tolerating multiple failures during simulation execution. Furthermore, the base implementation has been optimized through efficient management of connections and data, significantly reducing the execution time. As a result, the project establishes a solid foundation for future research and improvements in fault tolerance in distributed simulation.

AGRADECIMIENTOS

A mis padres y mis abuelos, quienes me han brindado un apoyo incondicional en cada paso que he dado y mediante su trabajo y sacrificio me han provisto de oportunidades y educado para convertirme en la persona que soy.

A mi hermano, mi pareja y mis amigos, quienes me han ayudado a desconectar en momentos de estrés, han estado a mi lado durante los momentos de trabajo y estudio, y siempre me han preguntado y mostrado interés.

A mis directores, mis compañeros y mis profesores, quienes con su conocimiento y dedicación, me han acompañado durante mi crecimiento académico y despertado en mí la pasión por este camino.

Índice general

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Capítulo 1

Introducción

1.1. Motivación

La creciente complejidad de sistemas en diversos campos, como la fabricación, logística, salud y otros, ha generado la necesidad de contar con herramientas eficaces que permitan analizar y predecir el comportamiento de estos sistemas. El modelado de estos sistemas conlleva a la creación de modelos de gran escala, donde la simulación se convierte en la única herramienta viable para su análisis. Sin embargo, debido a la complejidad de estos modelos, se requiere una considerable cantidad de recursos computacionales para llevar a cabo las simulaciones necesarias. En este sentido, el uso de un simulador distribuido, en lugar de uno centralizado, permite simular modelos más grandes en menos tiempo. Sin embargo, la simulación distribuida también introduce la posibilidad de fallos en los nodos distribuidos.

Para abordar esta problemática de fallos, es necesario analizar exhaustivamente todos los posibles fallos, las situaciones en las que pueden ocurrir y las soluciones correspondientes. En los sistemas de eventos discretos, la temporalización y el orden de los mensajes añaden complejidad al análisis. En entornos de producción a gran escala, la integración de técnicas de tolerancia a fallos es un requisito, ya que es probable que ocurra un fallo durante un tiempo prolongado de ejecución.

La simulación distribuida debe maximizar el rendimiento de los sistemas y la red en la que se ejecuta. Sin embargo, existe un compromiso entre la tolerancia a fallos del sistema y las prestaciones de la simulación, las cuales tienen gran implicación de latencia. Es decir, si los mensajes entre nodos tardan en llegar o existe un excesivo intercambio de mensajes, la simulación entera se ralentiza. Esto significa que la simulación avanza tan rápido como el nodo participante más lento. El uso de técnicas de tolerancia a fallos genéricas añade gran sobrecarga de número de mensajes y tiempos de espera debido a la sincronización, lo cual genera un aumento de la latencia que no se pretende asumir. Por ello, es necesario diseñar técnicas adaptadas y optimizadas para minimizar la degradación de la funcionalidad.

1.2. Objetivos y Alcance

El objetivo de este trabajo de investigación es diseñar mecanismos de tolerancia a fallos para un simulador distribuido de sistemas de eventos discretos. La idea es diseñar e implementar un modelo de replicación adaptado a la simulación con una estrategia conservativa. Nuestro objetivo principal es garantizar que estos mecanismos tengan un impacto mínimo en la latencia de la simulación, manteniendo así un buen rendimiento. En concreto, nuestro objetivo es asegurar que el impacto en las prestaciones del simulador sea mínimo durante la operativa normal de simulación, es decir, en situaciones sin fallos.

Nos enfocaremos en abordar los fallos de parada, que ocurren cuando un nodo del simulador deja de funcionar. Este tipo de fallos puede provocar la pérdida del estado interno del nodo y todos los eventos que aún no han sido procesados o enviados. Es crucial asegurarnos de que el fallo de un nodo de simulación no interrumpa por completo la simulación, sino que pueda continuar de manera adecuada.

Además del objetivo principal de implementar mecanismos de tolerancia a fallos, también nos planteamos un objetivo secundario. Queremos modificar y optimizar la estructura e implementación del simulador original, que está desarrollado en el lenguaje de programación Rust. Esta modificación nos permitirá integrar la tolerancia a fallos de manera más eficiente y lograr un mejor rendimiento general del simulador.

1.3. Contexto y Trabajo Relacionado

Este trabajo se enmarca dentro del proyecto *Simbots Swarm* llevado a cabo por el grupo *Computer Science for Complex System Modeling* (COSMOS) del Departamento de Informática e Ingeniería de Sistemas (DIIS) de la Universidad de Zaragoza. El grupo COSMOS se dedica al desarrollo de sistemas distribuidos complejos. En investigaciones previas realizadas por Unai Arronategui, José Ángel Bañares y José Manuel Colom [1, 2, 3, 4], se presenta una metodología dirigida por el modelo para la simulación distribuida de eventos discretos basada en redes de Petri. Además, se propone un lenguaje de modelado basado en componentes y un compilador que genera código eficiente para la simulación.

En los últimos años, múltiples alumnos han aportado al proyecto a través de sus Trabajos de Fin de Grado o Máster. Sergio Herrero [5] desarrolla el simulador centralizado y el compilador en Java, sentando las bases para los proyectos posteriores. Álvaro Santamaría [6] desarrolla el simulador distribuido en Rust que sirve como punto de partida para este trabajo. Al mismo tiempo, Fidel Reviriego [7] amplía el desarrollo del simulador distribuido con una gestión más específica de los *lookaheads*. Aunque este proyecto se basa en las implementaciones anteriores [6, 7], se han realizado cambios estructurales en el simulador como parte de esta investigación. A pesar de los cambios realizados en la implementación, es importante destacar que la metodología de simulación se ha mantenido sin modificaciones.

1.4. Estructura del Documento

A partir de este punto, la estructura del documento se organiza de la siguiente manera. Después de esta introducción, en el Capítulo 2 se presentan los conceptos generales de la simulación y se revisa el estado del arte en el campo de la tolerancia a fallos en simulación distribuida. Luego, en el Capítulo 3, se realiza un análisis de los requisitos de la tolerancia a fallos. A continuación, en el Capítulo 4, se detalla en profundidad la solución propuesta, que incluye los mecanismos de tolerancia a fallos y el diseño del *simbot*. Posteriormente, en el Capítulo 5, se describen los experimentos realizados y se presentan los resultados obtenidos. Finalmente, en el Capítulo 6, se presentan las conclusiones del trabajo. Además, en el Capítulo 7, se incluye la bibliografía utilizada.

Dado el alcance extenso del trabajo y la importancia de documentar los detalles para futuros alumnos que participen en el proyecto, se han desarrollado varios anexos complementarios al documento principal. Estos anexos se distribuyen de la siguiente manera. En el Anexo A, se proporciona una descripción detallada de la simulación distribuida y del simulador utilizado. En el Anexo B, se explican en detalle las características del nuevo diseño del *simbot*. En el Anexo C, se aborda la implementación en Rust del *simbot*. En el Anexo D, se detallan dos servicios auxiliares: el *Debug Server* y el *Replica Provisioner*. En el Anexo E, se presentan resultados adicionales de los experimentos realizados. En el Anexo F, se proponen posibles mejoras para trabajos futuros. Por último, en el Anexo G, se incluye la planificación del proyecto y el control de horas dedicadas.

Capítulo 2

Conceptos y Estado del Arte

2.1. Simulación Distribuida

Con el objetivo de establecer los precedentes al diseño de los mecanismos de tolerancia a fallos se presenta una descripción general del simulador, de sus componentes y su ejecución. El análisis más detallado del simulador se describe en el Anexo A.

El simulador distribuido estudiado en este proyecto reproduce el comportamiento de sistemas de eventos discretos (DES). En un DES los cambios de las variables del estado del sistema son producidos por eventos en instantes discretos de tiempo. Para modelar el comportamiento del sistema se utilizan redes de Petri, que son representaciones matemáticas que permiten modelar sistemas concurrentes y distribuidos de manera sencilla. La evolución del estado del sistema viene definida por el disparo de transiciones, que, en nuestro simulador, son representadas utilizando *Linear Enabling Functions* (LEF). La ejecución de la simulación se distribuye para permitir soportar modelos de gran tamaño y escalabilidad. La red de Petri que modela el sistema es dividida y distribuida entre los distintos nodos de la simulación.

En proyectos anteriores, se optó por considerar una estrategia de simulación conservativa por ser más eficiente frente a la optimista, la cual ejerce un gran uso de memoria. Se parte de la hipótesis de que una buena estimación del *lookahead*, que puede ser obtenida del modelo de red de Petri, permite acelerar las simulaciones con estrategias conservativas. Además, la estrategia conservativa es predecible y controlable, lo que permite un mejor análisis del comportamiento del sistema.

La estrategia de simulación conservativa garantiza la restricción de causalidad local: todos los eventos entrantes siempre serán procesados en el orden correcto establecido por su etiqueta temporal. Para ello, los nodos distribuidos de la simulación se bloquean hasta que asegurarse que no existen futuros eventos con un tiempo menor de simulación al actual. Para evitar interbloqueos de nodos en espera, se introducen los *lookahead*, mensajes de eventos nulos que contienen el valor de tiempo mínimo para cualquier evento futuro. Este valor de tiempo es el horizonte temporal hasta el que pueden avanzar todos los sucesores sin recibir un evento del *simbot* remitente con una etiqueta temporal más antigua.

2.1.1. Partición del Modelo

Los procesos lógicos que conforman la simulación distribuida se denominan *simbots*. Cada *simbot* simula una partición del modelo, una subred. Una subred contiene transiciones de entrada y de salida conectadas a otras subredes, creando una red de *simbots* vecinos donde se distinguen predecesores y sucesores. Los *simbot* vecinos interactúan entre sí mediante el paso de mensajes. La Figura 2.1 muestra la estructura de *simbots* vecinos (predecesores y sucesores) y el flujo de eventos a través de sus conexiones.

2.1.2. Ejecución del *Simbot*

El *simbot* interpreta el modelo de la red de Petri codificado con LEF, que representan cada transición junto con su estado de sensibilización y red de dependencias. Después enviar los eventos iniciales, los *simbots* ejecutan un ciclo de simulación tras otro hasta llegar al ciclo final. Un ciclo de simulación consta los siguientes pasos: la recepción de mensajes por cada vecino predecesor, el procesamiento de la lista de eventos, la ejecución del modelo mediante el disparo de las transiciones sensibilizadas, la generación de eventos debido al avance de la simulación y el envío de los eventos externos correspondientes. En el caso de que en un ciclo no existan eventos para los sucesores, el *simbot* envía un *lookahead* a todos los sucesores que no les corresponde un evento.

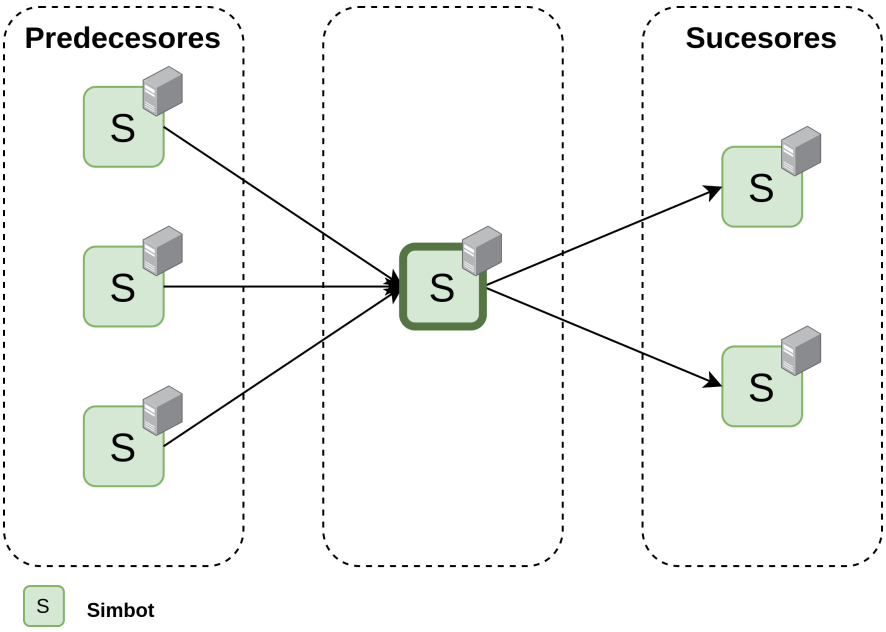

Figura 2.1: Estructura de *Simbots* Vecinos

Estructura y nomenclatura de los vecinos del *simbot* S_i. Sus predecesores se denotan como S_{i-1} y sus sucesores como S_{i+1}. Las flechas muestran el flujo de eventos de la simulación.

2.1.3. Comunicación entre Vecinos

Los mensajes de evento y *lookahead* son los únicos mensajes que cambian el estado de la simulación del *simbot*. El flujo de eventos y *lookahead* es de predecesores a sucesores. Existen otros mensajes de control de la simulación, por ejemplo, de sincronización de los nodos en la inicialización y finalización de la ejecución.

2.2. Estado del Arte

La tolerancia a fallos en simulación distribuida [8, 9] se puede clasificar en dos categorías principales: basada en replicación [10] o en *checkpointing* [11]. Ambas técnicas incurren en el uso de recursos computacionales y de red adicionales para mantener la operativa tolerante a fallos. La replicación mantiene copias de la aplicación y que, en el caso de un fallo, retoman el servicio. La replicación en la simulación distribuida puede utilizar mecanismos como el primario-copia [12] o técnicas de consenso [13, 14, 15] para conservar la consistencia del estado conjunto de las réplicas. La replicación conlleva la sincronización continua del estado de la simulación entre copias, lo que aumenta el número de mensajes. Además, modelos generales de replicación utilizan mensajes de confirmación para asegurar que todas las réplicas persisten el estado. Por otro lado, el *checkpointing* realiza copias periódicas del estado en disco (*snapshots*), que son restauradas tras un fallo mediante una operación de *rollback*. El proceso de guardado de una copia incrementa el tiempo de ejecución y el uso de memoria.

Los avances en tolerancia a fallos de simulación distribuida se han centrado alrededor de la simulación con estrategia optimista. Se desarrollan técnicas que hacen uso de los mecanismos de *rollback* inherentes a la simulación optimista [16, 17, 18, 19]. En ocasiones, la simulación conservativa permite la aplicación de las mismas técnicas [19]. Los métodos de tolerancia a fallos han sido desarrollados para arquitecturas o *frameworks* de simulación específicos como *Time Warp* [20, 18], *GAIA/ARTIS* [21, 22, 19] o *High Level Architecture* [23, 24]. La mayoría de las técnicas utilizan gran cantidad de recursos computacionales y de red, y aumentan la latencia de la simulación, por lo que se degradan las prestaciones de la simulación. Por otro lado, aquellas técnicas que se centran en conservar las prestaciones, son muy complejas y poco adaptables. El aumento de la latencia producido por los mecanismos de tolerancia a fallos afecta el tiempo de espera de los nodos distribuidos y reduce el rendimiento de la simulación.

Andras Varga *et al.* [25] proponen el factor de acoplamiento $\lambda = \frac{L \times E}{\tau * \times}$ para evaluar la rentabilidad de una simulación distribuida, teniendo en cuenta la latencia τ, el valor del *lookahead* L, la tasa de procesamiento de eventos P y la densidad de eventos del modelo E. Valores de $\lambda < 10$ se consideran bajos, cuando la simulación centralizada resulta más rentable que la distribuida. Por el contrario, con valores de $\lambda > 100$ se obtiene un buen rendimiento. La intuición de esta métrica es que para que salga rentable distribuir la simulación, los nodos deben estar simulando gran parte del tiempo de ejecución, en vez de esperando a los eventos.

El aumento de la latencia debido a los mecanismos de tolerancia a fallos disminuye el factor de acoplamiento, haciendo la simulación distribuida menos eficiente y rentable. Mientras que las líneas de investigación de los simuladores mencionados se centran en la interoperabilidad, nuestro simulador busca un sistema eficiente y escalable. Por ello, tras este análisis, se identifica la necesidad de un modelo de tolerancia a fallos igualmente eficiente, el cual preserve las prestaciones de la simulación distribuida sin introducir latencia en la simulación.

Rust [26] es un lenguaje de programación de sistemas que ha ganado popularidad debido a su enfoque en la seguridad de memoria y concurrencia. Comparado con lenguajes de programación de bajo nivel como C y C++, Rust tiene varias ventajas, como la capacidad de detectar errores en la gestión de memoria en tiempo de compilación mediante un sistema de *ownership* y *borrowing* [27]. Su estricta sintaxis otorga al programador un control más preciso sobre el ciclo de vida de los objetos en memoria, lo que puede evitar problemas comunes como las fugas de memoria y la liberación de memoria no válida, sin necesidad de un recolector de basura. Además, Rust ofrece un modelo de concurrencia seguro que evita condiciones de carrera [28]. Por estas razones, se ha optado por el uso de Rust para desarrollar el simulador distribuido.

Capítulo 3

Análisis de Tolerancia a Fallos

3.1. Tipo de Fallo Considerado

Para el estudio de la tolerancia a fallos se ha decidido tratar los fallos de parada de los nodos lógicos de la simulación, los *simbots*. Se define un fallo de caída cuando un *simbot* deja de funcionar correctamente, pierde su estado interno y no continúa su partición de la simulación. El análisis realizado es independientemente de que parte de la red está simulando cada *simbot*. Este proyecto se aborda con el objetivo de tolerar un fallo por subred del modelo simulada.

3.2. Suposiciones Iniciales

Se realizan las siguientes suposiciones para definir el alcance del diseño.

- El simulador resuelve los conflictos de la red de Petri de manera determinista, es decir, cada *simbot* simulando la misma partición de la misma red original genera los mismos eventos, con la misma estampilla temporal. El modelo simulado puede ser no determinista pero, el cálculo previo de la distribución de tiempos siguiendo una distribución cualquiera permite que la estrategia de simulación sea determinista.

- Existe un único *simbot* por máquina física, lo que significa que la caída de un nodo computacional solo corresponde con el fallo de un único *simbot*, a diferencia de como podría ocurrir en un entorno de virtualización.
- Se toleran aquellos fallos que ocurren una vez se ha inicializado el simulador correctamente y todos los *simbots* se encuentran ejecutando la simulación. Esta sección no incluye la inicialización de las estructuras de datos y las conexiones entre *simbots*. En el caso de ocurrir un fallo en la inicialización, la solución más sencilla es reiniciar la simulación.
- Existe un canal de comunicación de confianza entre todos los *simbots* que lo requieran dónde, si existe perdida de paquetes, estos se reenvían al destinatario sin involucrar a la aplicación y se conserva el orden de los mensajes. Además, el canal de comunicación no introduce particiones de red.
- Los *simbots* tienen suficientes recursos, principalmente, memoria para almacenar los mensajes y eventos que no se pueden enviar por el momento o que se deben almacenar para una posible recuperación futura.

3.3. Análisis de Requisitos

El objetivo del simulador distribuido es la simulación de modelos de gran tamaño durante largos periodos de tiempo, los cuales se pueden alargar durante días e incluso semanas. En simulaciones con un corto tiempo de ejecución, el reinicio de la simulación puede resultar una buena solución ante un fallo. Por el contrario, repetir simulaciones largas se convierte en una solución costosa.

La introducción de mecanismos capaces de tolerar y recuperar los fallos reduce el coste y el tiempo de ejecución frente a repetir la simulación. Pese a ello, un sistema tolerante conlleva una penalización en las prestaciones, comparado con la operativa de un sistema no tolerante a fallos. Para conservar las prestaciones, se debe reducir al máximo el número de mensajes y tiempos de espera de los mecanismos de tolerancia a fallos.

Nuestro diseño busca obtener un impacto mínimo, tolerando un degrado de la funcionalidad cuando un fallo ocurre. En simulaciones largas los fallos son poco frecuentes, por lo que la diferencia de tiempo de ejecución entre una simulación con fallos y una sin, debe resultar no significativa. Además, el diseño de tolerancia a fallos debe reducir al máximo el número de mensajes y tiempos de espera requeridos para conservar la consistencia.

Ante el fallo en un nodo, no se debe perder el estado de la simulación, sino que se tiene que persistir. Además, el estado no se debe alterar, ya que si el estado de un nodo tras un fallo difiere del anterior, los resultados de la simulación son incorrectos. El estado de un *simbot* está comprendido por el valor de los LEF, el contenido y el orden de la lista de eventos y los eventos ya recibidos aún por procesar. Además, debe existir cierto grado de sincronización para evitar el procesamiento incorrecto, desordenado o a destiempo de eventos, lo que puede invalidar el resultado de la simulación.

Para completar la tolerancia a fallos, además de la persistencia del estado, se deben incluir mecanismos de detección de fallos y recuperación ante la ocurrencia de estos. El sistema de detección de fallos debe identificar la caída de *simbots* y notificar efectivamente a los vecinos que requieran esta información. Las técnicas de recuperación diseñadas deben tener el mínimo impacto en las prestaciones y en la latencia de la simulación.

Por último, es necesario la implementación de un servicio que permita incluir dinámicamente *simbots* durante el transcurso de la simulación. Este proveedor debe aprovisionar e iniciar un *simbot* nuevo bajo demanda de cualquier nodo y habilitarlo para que se introduzca en la simulación.

3.4. Análisis del Diseño del *Simbot*

Este trabajo llega después de trabajos previos que abordan el desarrollo de un simulador centralizado y, tras este, la adaptación de un simulador distribuido. El diseño e implementación base del *simbot*, heredado de estos trabajos, es una primera iteración con carencias, las cuales impiden su extensión. El Anexo B desarrolla con detalle el análisis del diseño original del *simbot*.

Principalmente, el diseño original del *simbot* implementa una gestión de conexiones entre vecinos poco eficiente. Por cada mensaje a enviar, se crea una conexión TCP, se envía el mensaje y se cierra la conexión. Por otro lado, las estructuras de datos del simulador no son eficientes para realizar búsquedas, inserciones y eliminaciones. Además, la implementación mezcla indistintamente información lógica de la simulación con la información del entorno donde está siendo ejecutado, lo que entorpece que una o varias máquinas distintas ejecuten el mismo modelo y que esta máquina se cambie dinámicamente. Por último, la arquitectura del *simbot* y la organización del código impiden la ampliación de la funcionalidad debido a su poca modularización y separación en componentes. Todos estos puntos de mejora se deben abordar en un nuevo diseño del *simbot* para ejecutar el modo sencillo de simulación, completamente independiente de la tolerancia a fallos.

Capítulo 4

Diseño de Tolerancia a Fallos

La siguiente sección desarrolla los diferentes mecanismos de la tolerancia a fallos para la simulación: la replicación adaptada de los *simbots*, la detección y la recuperación de fallos.

4.1. Replicación

Como técnica principal para persistir el estado de un *simbot* se utiliza la replicación. En lugar de utilizar un modelo de replicación genérico, el cual puede añadir sobrecarga de mensajes y tiempos de espera, se ha diseñado un modelo original, ajustado al funcionamiento del simulador, para cumplir con los requisitos de latencia establecidos.

El diseño se basa en la replicación activa y se designa un *simbot* primario y réplica por partición del modelo. El estado de los *simbots* primario y réplica evoluciona igual debido a que reciben los mismos eventos de los predecesores y el simulador es determinista. Por ello, ambos *simbots* producen los mismos eventos, tanto internos como externos; pero únicamente el primario los envía a los sucesores, para evitar que se reciban duplicados. En el caso de un fallo, la réplica se convierte en primario y continua la simulación normalmente. De igual manera, ante el fallo de una réplica, el primario solicita una nueva réplica.

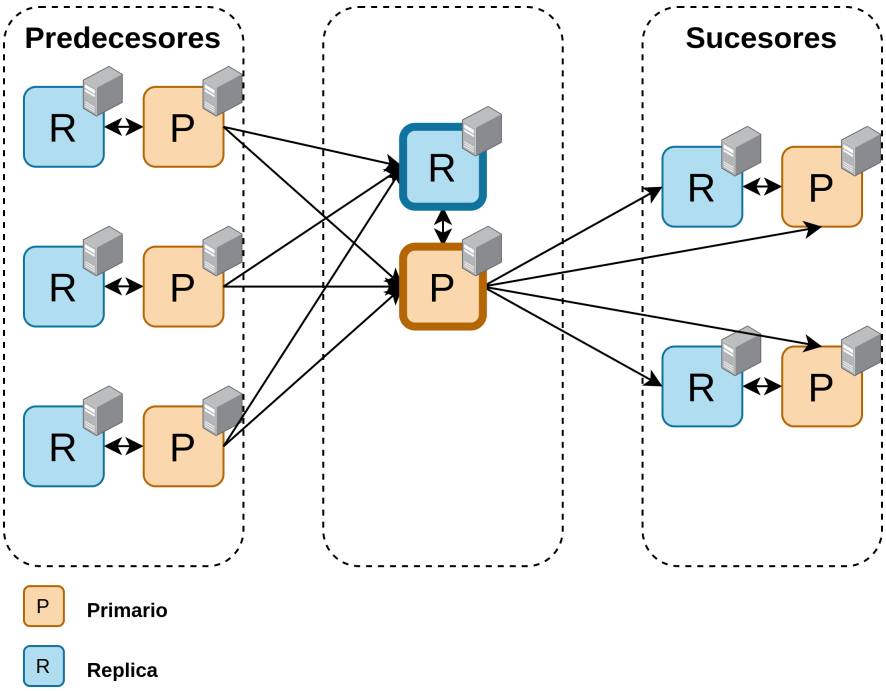

Figura 4.1: Estructura de *Simbots* Vecinos Replicados

Estructura y nomenclatura de los vecinos del *simbot*. El diagrama muestra los *simbots* primario y réplica de una subred y las flechas indican el flujo de eventos de la simulación y las conexiones necesarias. Se describe la misma estructura que la Figura 2.1, pero en simulación con tolerancia a fallos.

La Figura 4.1 presenta la nueva estructura de la simulación, en la que se duplica la cantidad de *simbots* y se muestra los canales de comunicación necesarios entre ellos. Se representa la misma estructura de vecinos *simbots* de la simulación que en la Figura 2.1 tras incluir la replicación.

4.1.1. Consistencia de Primario y Replica

La consistencia es un factor crítico en la replicación de datos. El primario y la réplica deben de ver los mismos eventos, en el mismo orden, para simular el modelo de la misma manera y, por ello producir los mismos eventos. La propiedad de consistencia garantiza que cuando la réplica o primario fallen el otro *simbot* puede continuar la simulación sin alterar el resultado final.

Pese a que una replicación síncrona más estricta permite asegurar que en todo momento se mantiene la consistencia del primario y réplica, estas técnicas conllevan un aumento de latencia, lo cual daña las prestaciones. Como alternativa para conservar la consistencia entre primario y réplica, se propone el uso de los mensajes y tiempos propios de la simulación. Esta, junto al desacoplamiento de la ejecución de las réplicas, son las ideas clave del diseño que posibilitan obtener una latencia baja.

Desacoplamiento de Primario y Réplica

La simulación ejecutada por un gran número de nodos, posiblemente en máquinas con distintas características, implica que no todos los *simbots* simulen a la misma velocidad. Llevar a cabo una consistencia más laxa, limitando la sincronización entre copias y desacoplando la ejecución del primario y la réplica, evita ralentizar uno de los *simbots*, y por consiguiente la simulación entera, si la ejecución del otro se demora. Es necesario integrar técnicas que, si un fallo ocurre cuando primario y réplica tienen diferente estado, converjan en una consistencia estricta.

El *simbot* primario es el encargado de enviar los eventos externos a sus vecinos sucesores, mientras que la réplica los genera, pero no los envía. En el caso de que la réplica falle, el primario puede continuar la simulación normalmente, independientemente de si la réplica iba más avanzada o no, ya que no ha enviado ningún evento. Por el contrario, si el primario falla, la réplica se puede encontrar en los tres casos expuestos a continuación.

Caso 1: Réplica va más retrasada que Primario. Cuando la réplica tome la responsabilidad de enviar los eventos a los sucesores, generará eventos que el primario ya ha enviado. Estos eventos no se deben enviar.

Caso 2: Réplica va más avanzada que Primario. La réplica habrá generado y no enviado los eventos que deben ser recibidos por los sucesores. Estos eventos deben ser almacenados y enviados por la réplica a sus respectivos sucesores tras el fallo del primario.

Caso 3: Réplica y Primario van a la par. La réplica puede continuar su ejecución normalmente, y retomar el envío de eventos a los sucesores.

Confirmaciones de Envío e Identificador de Eventos

En el Caso 1 y 2, la réplica debe conocer el estado del primario para decidir que acciones tomar tras el fallo. En estas situaciones la replicación se encuentra en una ventana de consistencia eventual, que en el momento del fallo debe converger. Para registrar el estado del primario se propone el envío de confirmaciones de los eventos enviados. A diferencia de otras técnicas de replicación generales, estas confirmaciones solo se realizan en un sentido: del primario a la réplica. El primario no tiene que esperar a una respuesta de la réplica tras la confirmación y la réplica no tiene que esperar a la confirmación para seguir simulando. Con esta técnica se reduce el número de mensajes y el tiempo de espera necesarios para conservar la consistencia.

A cada evento generado se le asigna un índice identificador único. Debido a la naturaleza determinista de la simulación, el mismo evento generado por el primario y por la réplica, tiene el mismo identificador asignado. Este identificador se incluye en los eventos externos y en las confirmaciones para poder identificar el avance del estado del primario frente al de la réplica.

Registro de Invalidación de Mensajes

Las confirmaciones garantizan a la réplica que un evento generado por el primario ha sido enviado a los sucesores de la subred. Esta seguridad solamente se puede obtener si la confirmación es enviada a la réplica tras el envío a los sucesores. Existe la posibilidad de que el nodo primario falle tras enviar un evento a uno o varios sucesores y antes de realizar la confirmación a la réplica. En este caso, la réplica, al tomar el control, reenviaría este evento, el cual estaría duplicado.

Para evitar esta duplicidad de mensajes, los sucesores deben de ser responsables de invalidar los posibles mensajes duplicados. La invalidación conlleva almacenar los mensajes recibidos, utilizando el identificador de los mensajes para detectar aquellos repetidos. Si cada confirmación es enviada tras un único envío a los sucesores, solo es necesario utilizar un registro para almacenar el último mensaje de cada predecesor.

Registro de Estado del Primario

Utilizando el identificador de las confirmaciones y comparándolo con el identificador de los mensajes generados, la réplica puede reconocer si va más atrasada o adelantada que el primario. Si se mantiene un registro del estado del primario, en caso del fallo de este, la réplica es capaz decidir que mensajes enviar o ignorar.

Utilizando el identificador del último mensaje confirmado por el primario antes de su fallo, la réplica conoce hasta qué mensaje no debe enviar a sus sucesores (Caso 1). No es necesario almacenar todas las confirmaciones del primario, ya que la réplica generará estos eventos en el futuro con el mismo identificador. Por el contrario, la réplica debe almacenar todos los mensajes que ha generado y que el primario no ha confirmado, ya que en el caso de que el primario falle, la réplica no puede generar los mensajes que se deben enviar desde su estado actual (Caso 2). La réplica asume que los mensajes que no han sido confirmados por el primario tampoco han sido enviados. Los posibles errores derivados de esta suposición se resuelven mediante los mecanismos invalidación discutidos anteriormente.

El Algoritmo 1 formaliza los procedimientos ejecutados por el *simbot* réplica para registrar el estado del primario. El procedimiento *registrarConfirmaciónRecibida* detalla las acciones al recibir un mensaje de confirmación del primario y el procedimiento *registrarEventoGenerado*, las acciones al generar un evento propio. La cola de mensajes Q^R representa la diferencia de estado entre la réplica y el primario. Q^R es la cola de mensajes generados por la réplica (y no enviados) que no han sido confirmados por el primario todavía. Por otro lado, M^R y M^P son los identificadores del último mensaje generado por la réplica y del último mensaje confirmado por el primario, respectivamente.

Algorithm 1 Procedimientos de Registro de Estado del Primario en Réplica

// Diferencia de estado entre Primario y Réplica. Inicialización: Lista vacía
$Q^R \leftarrow []$
// Último mensaje generado por Réplica. Inicialización: ID Mensaje nulo
$M^R \leftarrow 0$
// Último mensaje confirmado por Primario. Inicialización: ID Mensaje nulo
$M^P \leftarrow 0$

// Recepción de Confirmación de Evento del Primario
procedure REGISTRARCONFIRMACIÓNRECIBIDA(M : Confirmación)
    $M^P \leftarrow M_{ID}$
    if $M^R >= M^P$ **then**
        // Réplica va más avanzada o a la par con el Primario
        $Q^R.eliminarPrimero()$
    else
        // ($M^R < M^P$) Réplica va más retrasada
        $no - op$
    end if
end procedure

// Generación de Evento de la Réplica
procedure REGISTRAREVENTOGENERADO(M : Mensaje)
    $M^R \leftarrow M_{ID}$
    if $M^R > M^P$ **then**
        // Réplica va más avanzada
        $Q^R.insertarUltimo(M)$
    else
        // ($M^R <= M^P$) Réplica va más retrasada o a la par con el Primario
        $no - op$
    end if
end procedure

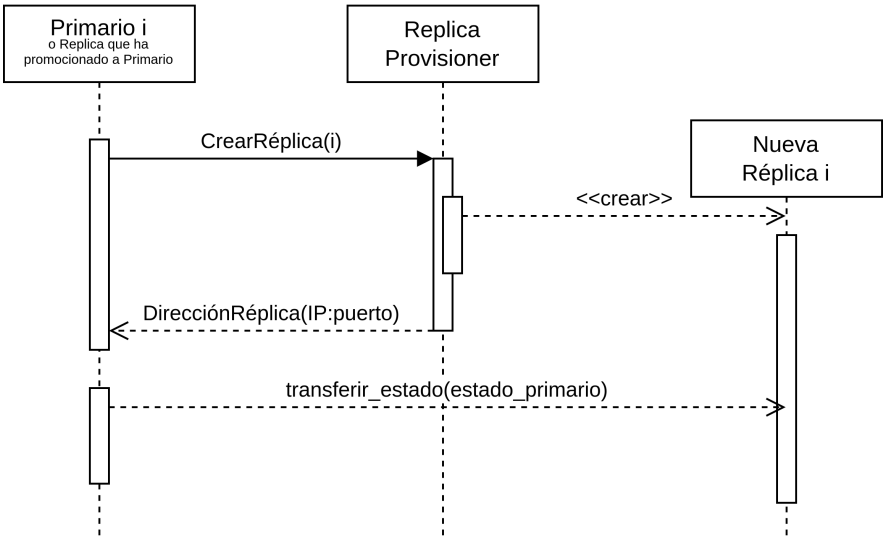

Figura 4.2: Diagrama de Secuencia de Petición de Réplica

Diagrama de secuencia de la interacción entre el *simbot* primario y el *Replica Provisioner* al solicitar una nueva réplica y la transmisión de su estado.

4.1.2. Incorporación Dinámica de Réplicas en la Simulación

Proveedor de Réplicas

El proveedor de réplicas es el servicio externo que prepara la infraestructura e inicia el *simbot* para ejecutar cierta subred del modelo, sea desde el principio o durante la simulación, tras un fallo. El método de aprovisionamiento del proveedor de réplicas es único al entorno de ejecución de la simulación, ya que el procedimiento para ejecutar un *simbot* difiere. Pese a ello, el proveedor de réplicas ofrece a los *simbots* la misma interfaz para realizar peticiones y recibir respuestas independientemente del entorno donde se ejecute.

El procedimiento es el siguiente. Primero, el *simbot* envía una petición de réplica al proveedor de réplicas. A continuación, el proveedor responde con la dirección de red donde la nueva réplica está escuchando para recibir el estado. La Figura 4.2 muestra la interacción esta interacción. El Anexo D explica en detalle el diseño e implementación del *Replica Provisioner*.

Copia del Estado de un *Simbot*

Una vez la réplica ya está escuchando a una conexión del primario, se le debe transferir todo el estado, lo que le permite continuar la simulación en el mismo punto que el primario. El estado completo del *simbot* está formado los datos listados a continuación.

- La información de la subred del modelo almacenada en los LEF.
- El reloj local.
- La lista de eventos del *simbot*.
- El contador del identificador de eventos.
- El LVHT de la simulación.
- La fase de la simulación en la que se encuentra el primario
- En el caso de encontrarse en la fase de envío, la lista de eventos externos por enviar.
- Los eventos y *lookaheads* de los predecesores por procesar. Estos mensajes se encuentran en los canales entre ambos hilos.
- Las direcciones de red asociadas a los *simbots* vecinos de la subred.

Sincronización de Primario y Predecesores

La incorporación de un nuevo *simbot* a la simulación conlleva sincronización entre todos los vecinos de la subred en la que se está incluyendo. Una vez la réplica está disponible para recibir su estado, se deben sincronizar los predecesores y el primario para no continuar simulando mientras se realiza la copia. En el caso de que los predecesores enviasen algún evento o el primario simulase, se corre el riesgo de que el estado no se haya persistido completamente en la réplica y la ejecución difiera al reanudar la simulación.

Inicialización de Conexiones de la Réplica

Una vez la réplica está preparada para simular, necesita establecer las conexiones necesarias con otros *simbots*, de acuerdo con la Figura 4.1, su primario y los vecinos predecesores. Mediante el mecanismo de inicialización de conexiones, la réplica contacta con sus vecinos, les comunica su índice de subred, e inicia una conexión permanente. A diferencia de los primarios, los cuales se coordinan entre ellos para iniciar las conexiones, las réplicas inician la conexión con todos los vecinos que lo requieren. Tras establecer todas las conexiones necesarias, la simulación puede continuar de manera normal.

A continuación, se detallan los mecanismos de detección y notificación de fallos en la Sección 4.2 y las secuencias de acciones de recuperación de la simulación ante un fallo en la Sección 4.3.

4.2. Detección y Notificación de Fallo

La detección de fallos dentro de un sistema tolerante es imprescindible para tomar las acciones necesarias ante un fallo. En el simulador distribuido, la parada de un *simbot* implica la parada de la simulación y la sincronización de los *simbots* vecinos hasta que el sistema vuelve a ser tolerante. Los fallos de *simbots* que no son vecinos no se necesitan detectar, ya que los mecanismos internos de bloqueo de la simulación conservativa en ausencia de eventos son suficientes para detener la ejecución cuando sea necesario.

4.2.1. Fallos de *Simbots* Vecinos

Un *simbot* debe detectar el fallo de su respectivo primario o réplica y, en el caso del primario, el de sus vecinos sucesores. Dado que todos estos *simbots* comparten conexiones, la notificación del cierre imprevisto del otro extremo se utiliza para detectar un fallo. Además, si el envío o la recepción de mensajes falla se detecta que el *simbot* del otro extremo de la conexión ha fallado.

Otras alternativas para la detección de paradas como el uso de latidos (*ping*) y sus respectivas confirmaciones conllevan un aumento indeseado del número de mensajes, lo que daña la latencia de la simulación. Por el contrario, la detección del cierre de las conexiones utilizando TCP no introduce tráfico adicional.

4.2.2. Fallos Fatales

Un fallo fatal es un fallo que no es tolerado, por lo que la simulación no se puede recuperar. Este fallo ocurre cuando el primario y la réplica simulando la misma subred no están disponibles y fallan antes de haber podido persistir su estado en una nueva réplica. El fallo fatal puede ser detectado por el *simbot* primario de cualquiera de las subredes vecinas predecesoras, ya que tiene conexiones con ambos primario y réplica. Ante la ocurrencia de un fallo fatal, todos los *simbots* de la simulación deben de ser notificados para abortar la simulación.

Para notificar el fallo fatal a todos los *simbots* de la simulación se comunica a un servicio centralizado, el cual lo retransmite al resto de la simulación. Para la notificación del fallo fatal se ha utilizado el *Debug Server*, el cual está diseñado para asistir con la depuración de la simulación, ya que este conoce a todos los *simbots* de la simulación.

4.3. Recuperación de Fallo

Una vez un fallo ha sido detectado y notificado a los *simbots* necesarios, la simulación se debe recuperar del fallo para reanudar su ejecución. La recuperación de un fallo dentro de una subred es llevada a cabo por el otro *simbot* de la subred, sea primario o réplica, pero requiere de la sincronización de sus vecinos.

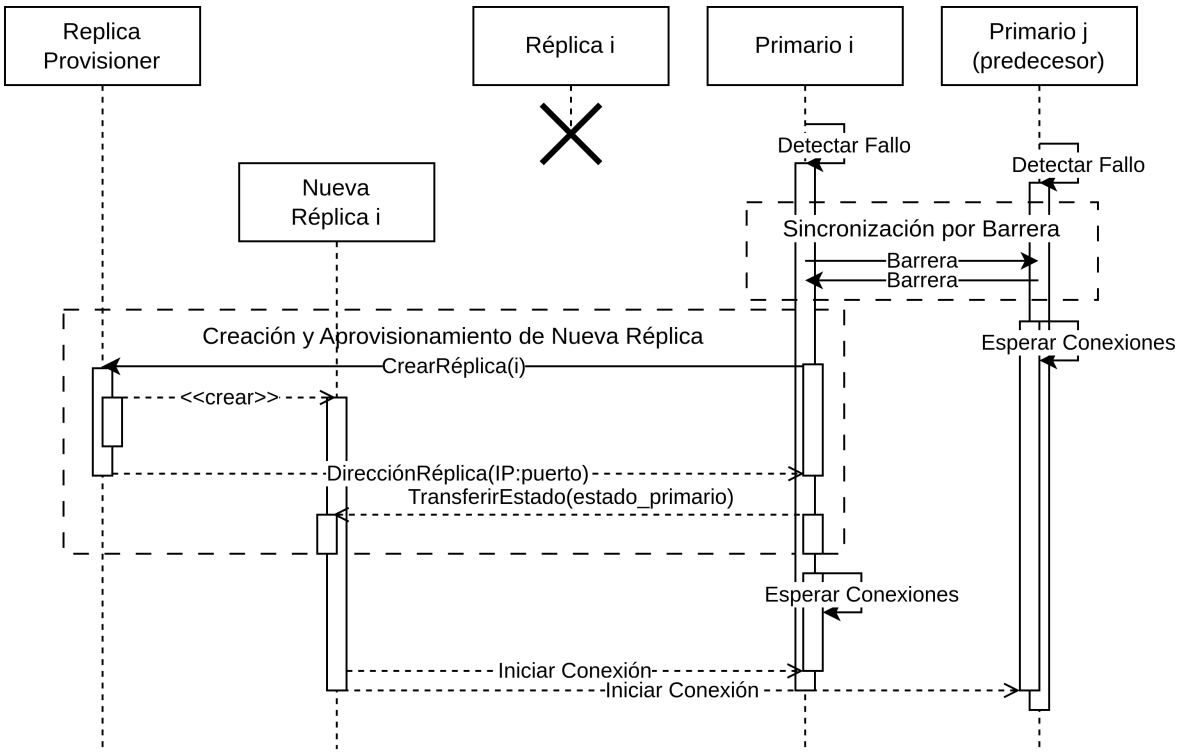

Figura 4.3: Diagrama de Secuencia de Recuperación de Fallo de Réplica

Diagrama de secuencia de la interacción entre los *simbots* de una simulación y el *Replica Provisioner* al caer una réplica. Se detalla la sincronización entre vecinos, además de la petición de una nueva réplica y la transmisión de su estado.

4.3.1. Fallo de la Réplica

Cuando el *simbot* réplica de una subred falla, es necesario incorporar una nueva réplica a la simulación. Primero, el *simbot* primario notifica al proveedor de réplicas de que la réplica ha caído para que aprovisione una nueva réplica. A continuación, el primario recibe la dirección de red dónde el nuevo *simbot* está esperando la transmisión del estado. En este punto, los vecinos predecesores de la subred se sincronizan y aplazan el envío de eventos hasta que la nueva réplica esté disponible. Después, el primario copia su estado en el nuevo *simbot*, convirtiéndolo en su réplica. Por último, la réplica inicia sus conexiones y retoma la simulación en el mismo punto que el primario. El resto de *simbots* que requieren una conexión con la réplica (primario y predecesores) esperan hasta que esta conexión se establezca. El diagrama de secuencia de la Figura 4.3 muestra la interacción descrita.

4.3.2. Fallo del Primario y Promoción de la Réplica

Cuando el *simbot* primario falla, la réplica sigue el procedimiento de promoción a primario para alcanzar el mismo estado en el que se encontraba el primario fallido. Primero, el *simbot* cambia su rol de tolerancia a fallos, de réplica a primario, y notifica a sus conexiones existentes. A continuación, inicia las conexiones con aquellos vecinos con los que no tenía conexiones establecidas previamente, es decir, los *simbots* primario y réplica de sus vecinos sucesores. Por último, el nuevo *simbot* primario tiene que retomar la consistencia con el anterior.

Para retomar la consistencia del primario antiguo se han de tomar diferentes acciones dependiendo de que caso se dé. Utilizando la información del registro de estado del primario antiguo (véase Sección 4.1.1), se resuelven las distintas situaciones de la siguiente manera:

- Si $M^R < M^P$ el nuevo primario va más retrasado que el antiguo y conforme genera todos los eventos ignora su envío, hasta llegar al evento con identificador M^P (Caso 1).
- Si $M^R > M^P$ el nuevo primario va más avanzado que el antiguo e inmediatamente envía todos los mensajes pendientes en Q^R (Caso 2).
- Si $M^R = M^P$ el nuevo primario y el antiguo van a la par y no se debe tomar ninguna acción(Caso 3).

La Figura 4.4 muestra el proceso promoción de una réplica al caer su primario y la interacción con sus vecinos. Independientemente de qué procedimiento se ejecuta para retomar la consistencia, en cuanto el nuevo primario está disponible tras la promoción, debe seguir los pasos de la Sección 4.3.1 para incorporar una nueva réplica a la simulación.

4.4. Diseño del *Simbot*

La simulación ejecutada por el conjunto de *simbots* se puede ejecutar en dos modos. El *simbot* ha sido diseñado en dos fases bien definidas: la primera, que abarca la simulación sencilla (sin tolerancia a fallos), y la segunda, que amplía su funcionalidad con la tolerancia a fallos diseñada en este trabajo.

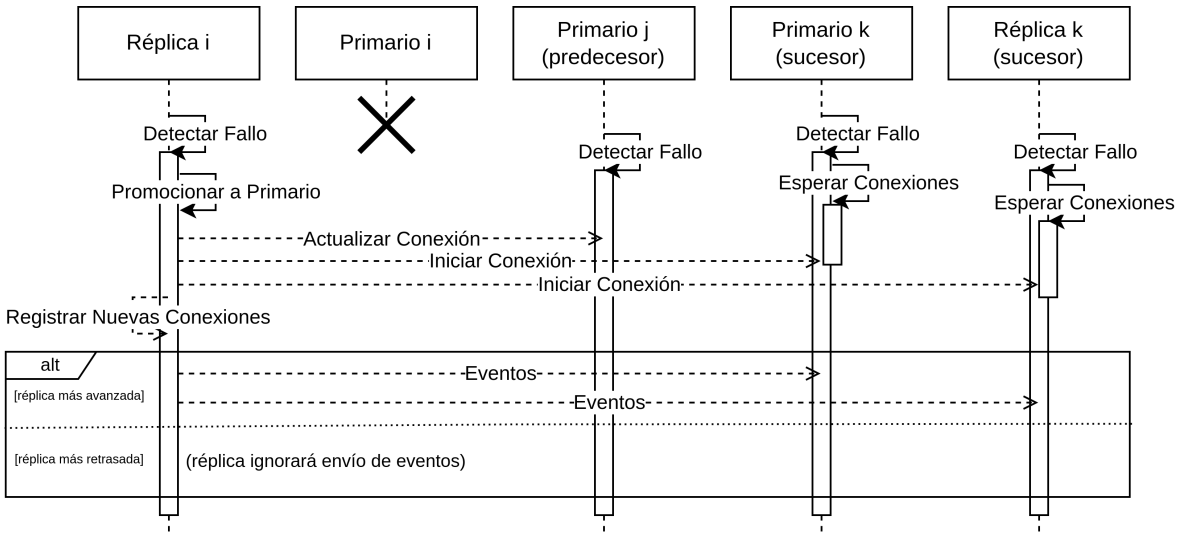

Figura 4.4: Diagrama de Secuencia de Promoción de Réplica tras Fallo del Primario

Diagrama de secuencia de la interacción entre los *simbots* de una simulación al caer un primario. Se detalla la sincronización entre vecinos y el proceso de promoción de la réplica a nuevo primario.

4.4.1. Diseño del *Simbot* para Simulación Sencilla

El nuevo diseño del *simbot* resuelve los problemas descritos en la Sección 3.4. En resumen, la gestión de conexiones y la arquitectura del *simbot* no soporta su expansión para los mecanismos de tolerancia a fallos. Esta sección describe los detalles más relevantes del diseño sin tolerancia a fallos, para comprender las modificaciones posteriores. El diseño del *simbot*, el cual no está relacionado con la tolerancia a fallos, ha requerido gran cantidad de trabajo y planificación. Debido a su importancia, los aspectos cuya mención no se ha considerado esencial son detallados en el Anexo B.

En el *simbot* existen dos hilos (*threads*) de ejecución: uno dedicado al motor de simulación (*Simulation Engine*) y otro al receptor de mensajes (*Mailbox*). Los dos hilos de ejecución están comunicados mediante canales de mensajes bloqueantes. Esta división permite al hilo de ejecución simular a máxima velocidad, únicamente bloqueándose cuando se ha simulado hasta el horizonte temporal y no existen eventos externos por procesar. Para abstraer la ejecución del motor de simulación de la sincronización entre los componentes internos y el resto de los vecinos, se ha establecido el componente principal *Simbot*, encargado de controlar el flujo de ejecución y coordinar los *threads* de simulación y de comunicación.

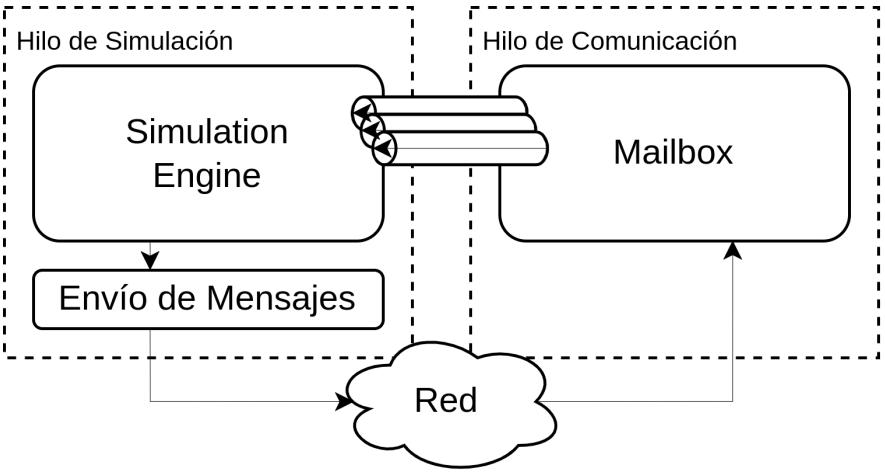

Figura 4.5: Arquitectura Original del *Simbot*

Arquitectura del diseño original del *simbot*. Existen dos componentes principales (el *Simulation Engine* y el *Mailbox*), ejecutados en dos hilos diferentes. Existe un módulo auxiliar para realizar los envíos a otros *simbots* en conexiones TCP diferentes.

Con el objetivo de poder establecer una única conexión entre cada par de vecinos y utilizar esta conexión para el envío y recepción de mensajes, se diseña un gestor de conexiones compartido entre el resto de componentes, el *Connection Manager*, el cual identifica la conexión TCP de cada vecino con su índice de subred, abstrayendo la información de red de la lógica de simulación. Tener una conexión permanente por cada vecino significa que se necesita un *listener* que permita atender a varias conexiones simultáneamente. El *Network Message Receiver* es el encargado de recibir nuevas conexiones y mensajes de conexiones ya establecidas.

La arquitectura del diseño original del *simbot* se muestra en la Figura 4.5. Por otro lado, la arquitectura del diseño nuevo para la simulación sin tolerancia a fallos se describe en la Figura 4.6. Ambos diseños producen el mismo resultado en la ejecución sencilla de la simulación, pero el nuevo mejora ampliamente el rendimiento de la simulación.

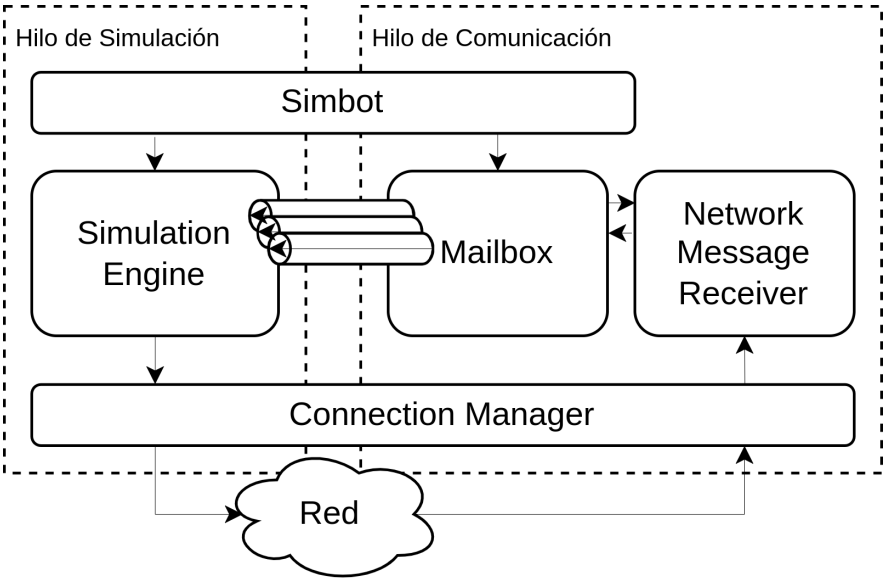

Figura 4.6: Arquitectura del *Simbot* para Simulación Sencilla

Arquitectura del nuevo diseño del *simbot*. El componente principal (*Simbot*) coordina la ejecución del *Simulation Engine* y el *Mailbox* en diferentes hilos. Las conexiones TCP por cada uno de los vecinos son gestionadas por el *Connection Manager* y el *Network Message Receiver* asiste con la recepción de nuevas conexiones y mensajes de la simulación. El resto de componentes y estructuras de datos auxiliares se han dejado fuera del esquema por simplicidad.

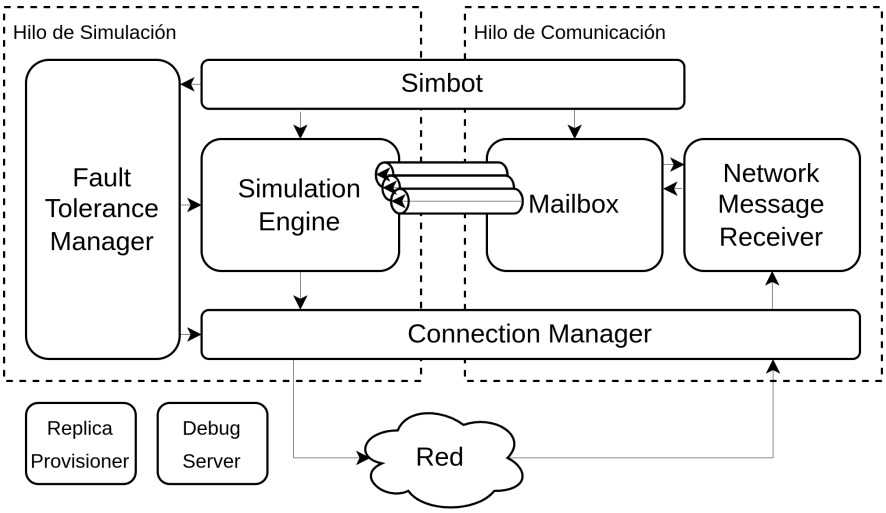

Figura 4.7: Arquitectura del *Simbot* para Simulación Tolerante a Fallos

Arquitectura del *simbot* para simulación con tolerancia a fallos. El componente principal (*Simbot*) interactúa, aparte de con el *Simulation Engine* y el *Mailbox*, con el *Fault Tolerance Manager* para gestionar los aspectos de la tolerancia a fallos. Los componentes del *Connection Manager* y el *Network Message Receiver* funcionan de igual manera que en el diseño para simulación sencilla. El *Replica Provisioner* y el *Debug Server* son servicios auxiliares externos utilizados por el *simbot*.

4.4.2. Diseño del *Simbot* para Tolerancia a Fallos

El simulador tolerante a fallos se ha basado en el nuevo diseño para simulación sencilla. El diseño busca abstraer los componentes y funcionalidad relacionados con la tolerancia a fallos con el objetivo de permitir un cambio efectivo entre ambos modos de simulación. La nueva arquitectura se apoya extensivamente en los componentes diseñados con anterioridad, en especial, el *Connection Manager* y el *Network Message Receiver*. Además, añade un nuevo componente, el *Fault Tolerance Manager*, el cual se ocupa de gestionar la detección y recuperación de fallos, al igual que el registro de los eventos necesarios para conservar la consistencia. La Figura 4.7 muestra la arquitectura principal del *simbot* con tolerancia a fallos.

En el diseño se ha contado con dos servicios auxiliares centralizados: el servidor de depuración (*Debug Server*) y el proveedor de réplicas (*Replica Provisioner*). El diseño e implementación de ambos servicios se detalla en el Anexo D.

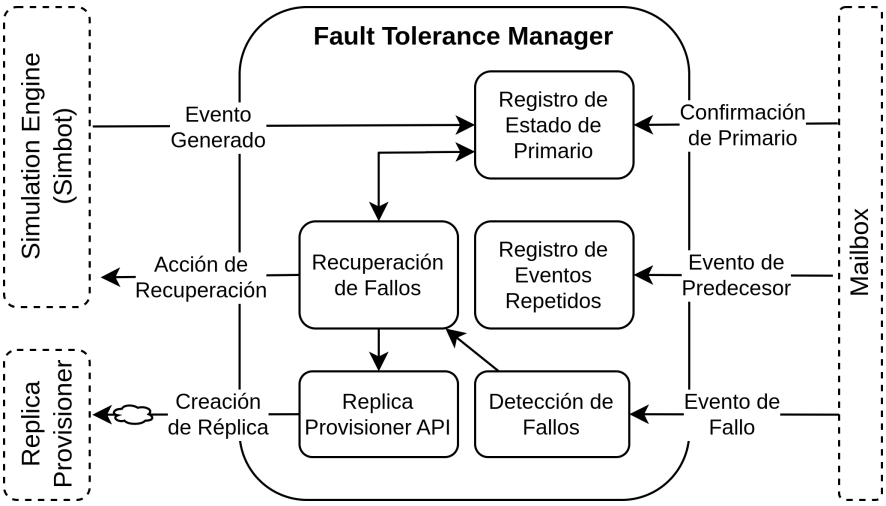

Figura 4.8: Componentes del *Fault Tolerance Manager*

Representación de los componentes del *Fault Tolerance Manager* y sus interacciones.

El *Fault Tolerance Manager* gestiona los aspectos de tolerancia a fallos de la simulación. Su funcionalidad recoge la detección y tratamiento de fallos, la sincronización y recuperación ante el fallo de otro *simbot*, la comunicación con el *Replica Provisioner* y la gestión de la consistencia. Las siguientes secciones describen las diferentes funcionalidades. La Figura 4.8 muestra los componentes del gestor y las interacciones entre componentes, tanto internos como externos. Los componentes englobados por el *Fault Tolerance Manager* siguen siendo entidades distintas agrupadas bajo la misma estructura.

Registro de Estado del *Simbot* Primario

El registro de la diferencia del estado entre el primario y la réplica de una misma subred del modelo se lleva a cabo por este componente del *Fault Tolerance Manager*, el cual ejecuta los procedimientos del Algoritmo 1 en el *simbot* réplica. El *Mailbox* redirige las confirmaciones del primario recibidas y el *Simulation Engine*, los eventos generados por la réplica. Este registro es utilizado durante las acciones de recuperación de fallos para garantizar la consistencia de la simulación.

Registro de Eventos Repetidos

La problemática descrita en la Sección 4.1.1 relativa a la posible recepción de eventos repetidos tras un fallo se resuelve mediante este componente, el cual ignora los mensajes repetidos en caso de que ocurran. El *Mailbox* registra cada mensaje antes de procesarlo para asegurar que no es repetido.

Detección de Fallos

La detección de fallos de otros *simbots* es llevada a cabo por el *Network Message Receiver*, al recibir un evento de cerrado de conexión TCP o al fallar en la recepción de un evento, o por el *Simulation Engine*, al fallar en el envío de un evento. El fallo es notificado por el *Mailbox* a través de una canal al *Fault Tolerance Manager*, el cual gestiona el fallo cuando el estado de la simulación lo permite.

Recuperación de Fallos

Cuando el *Simbot* se encuentra en un estado desde el que puede recuperar un posible fallo, si ha ocurrido un fallo inicia los procesos de recuperación descritos en la Sección 4.3. Como parte de la recuperación de fallos, en el caso de que se necesite una nueva réplica, se utiliza el *Replica Provisioner API* para contactar con el *Replica Provisioner*.

Capítulo 5

Implementación, Experimentación y Resultados

5.1. Aspectos de Implementación

El lenguaje de programación utilizado para el desarrollo del *simbot* es Rust [26, 29]. Los principales beneficios de Rust son su eficiencia, su seguridad en memoria forzada por su sistema de *ownership* y *borrowing*, y su seguridad en concurrencia. Todos los componentes del proyecto han sido desarrollado para ejecutarse en máquinas Linux. De las 8786 líneas de código Rust del proyecto, se han escrito 8374, las cuales forman 3 aplicaciones distintas (`simbot`, `debug_server` y `replica_provisioner`). Además, se han desarrollado numerosos *scripts* en Python (229 líneas) y Bash (441 líneas) para automatizar experimentación.

La descripción completa de las características de la implementación del *simbot* se encuentra en el Anexo C y la implementación de los servicios auxiliares en el Anexo D. En esta sección se enumeran las librerías externas utilizadas más relevantes en la implementación del *simbot*.

La comunicación entre los componentes internos del *simbot* se realiza utilizando los canales de mensajes de la librería `crossbeam_channel`. Para sincronizar el acceso a las estructuras de datos compartidas se han envuelto utilizando los componentes `RwLock` (*lock* de un escritor y múltiples lectores) y `Arc` (*Atomically Reference Counted*) de la librería estándar `std::sync`. Para acceder a un objeto `D`, que se almacena como `Arc<RwLock<D>>`, se ejecutan las funciones `write()` y `read()` que devuelven una referencia mutable o no, respectivamente, al dato `D`.

El `NetworkMessageReceiver` (el *listener* de conexiones asíncrono) está implementado utilizando la librería `mio`. Se utiliza el `mio::Poll`, el cual, a su vez, utiliza la API de Linux *epoll*. El `Poll` gestiona la notificación de eventos en las conexiones registradas y permite la lectura de los mensajes recibidos. Los *streams* de datos no-bloqueantes `mio::net::TcpStream` son gestionados en memoria compartida por el `ConnectionManager` para que ambos *threads* puedan acceder a ellos. Tanto los mensajes enviados entre *simbots* y el resto de componentes, como la representación del modelo en LEF, han sido codificados utilizando la librería de serialización binaria `bincode`.

5.2. Entorno de Experimentación

Durante el desarrollo de la aplicación se ha realizado una validación continua de los cambios implementados. Utilizando la funcionalidad de *tests* de la herramienta Cargo, se han diseñado pruebas unitarias y de integración y se han ejecutado al introducir cada nueva funcionalidad. Junto a las pruebas de integración durante el desarrollo, la experimentación final se ha basado en el despliegue del *simbot* en distintos entornos y la simulación de varios modelos compilados previamente. Mientras que los *simbots* de las pruebas de integración se han ejecutado como procesos independientes en la misma máquina, los experimentos posteriores se han realizado en un clúster de Raspberry Pi.

Debido a que la capacidad computacional del ordenador personal utilizado en el desarrollo no es suficiente para realizar pruebas y obtener tiempos de ejecución, se ha decidido utilizar un clúster de Raspberry Pi para ejecutar las pruebas. Se ha contado con 20 Raspberry Pi 4 Modelo B con un procesador Broadcom `BCM2711` con una frecuencia base de 1.5 GHz, 4 *cores*, 8 GB de memoria RAM y Ubuntu Server `20.04` con Rust instalado mediante el *script* de instalación [1].

[1] `https://sh.rustup.rs`

5.3. Experimentos y Resultados

Con el objetivo de demostrar la efectividad del simulador implementado, se han ejecutado las diferentes versiones del simulador y se han recopilado métricas de tiempo de ejecución. Cabe destacar que, se han realizado experimentos adicionales para comprobar la corrección de la simulación y de los mecanismos de tolerancia a fallos (véase Anexo E).

Las versiones del simulador utilizadas han sido: el simulador distribuido original [6] (`base`), la nueva versión sin tolerancia a fallos (`sencillo`) y la versión tolerante a fallos (`fallos`). Las métricas han sido obtenidas mediante temporizadores utilizando el módulo adicional `simulation_metrics` y comunicadas al final de la simulación al *Debug Server*. El tiempo de ejecución de la simulación se muestra en segundos mientras que la duración predeterminada de una simulación se especifica en *simseconds* o ciclos de simulación.

5.3.1. Mejora frente a Implementación Base

Se ha realizado un experimento sencillo para evaluar la mejora en el tiempo de ejecución frente a la implementación `base`. Se ha medido el tiempo de ejecución de simulaciones con diferentes duraciones medidas en ciclos (*simseconds*). El modelo simulado tiene 1 transición por rama, es decir, apenas tiene carga de simulación; con el objetivo de analizar la diferencia de la operativa de comunicación. Esta ejecución no es comparable a la simulación de un modelo real.

Los resultados obtenidos (ver Cuadro 5.1) muestran que la nueva versión del simulador (`sencillo`) mejora significativamente el tiempo de ejecución en comparación con la versión `base`, en un factor de 21 para una simulación de 100.000 ciclos. Esta mejora se atribuye a la optimización del código, las estructuras de datos utilizadas y el uso de conexiones TCP permanentes entre nodos.

Ciclo Final (*simseconds*)	**Tiempo de Ejecución** (segundos)		
	`base`[6]	`sencillo`	`fallos`
20	0.0131	0.0067	0.0108
1000	0.461	0.162	0.253
10000	20.437	1.643	2.478
100000	372.88	17.655	22.751

Cuadro 5.1: Comparación Tiempo de Ejecución de Distintas Versiones del Simulador
Comparación de tiempo de ejecución total (en segundos) de las versiones `base`, `sencillo` y `fallos` del simulador en simulaciones con diferente ciclo final (en *simseconds*)

5.3.2. Latencia de Simulación con Tolerancia a Fallos

Ejecución con Carga Mínima de Simulación

Una simulación distribuida es más rentable que una centralizada cuando la carga de simulación es suficientemente grande como para que el factor de acoplamiento [25] es $\lambda > 100$. Para calcular la carga de simulación mínima para que una ejecución de nuestro modelo sea rentable, debemos obtener el valor del *lookahead* L.

- El simulador es capaz de procesar $P = 7522936$ [4] eventos por segundo.
- La latencia de red del entorno de simulación es $\tau = 350$ microsegundos.
- La densidad de eventos del modelo es $E = 1$ evento por *simsecond*.

$$\lambda = \frac{L \times E}{\tau \times P} \tag{5.1}$$

$$100 >= \frac{L \times E}{350 \times 10^{-6} \times 7{,}5 \times 10^{6}} \tag{5.2}$$

$$L \times E >= 100 \times 350 \times 10^{-6} \times 7{,}5 \times 10^{6} = 262500 \tag{5.3}$$

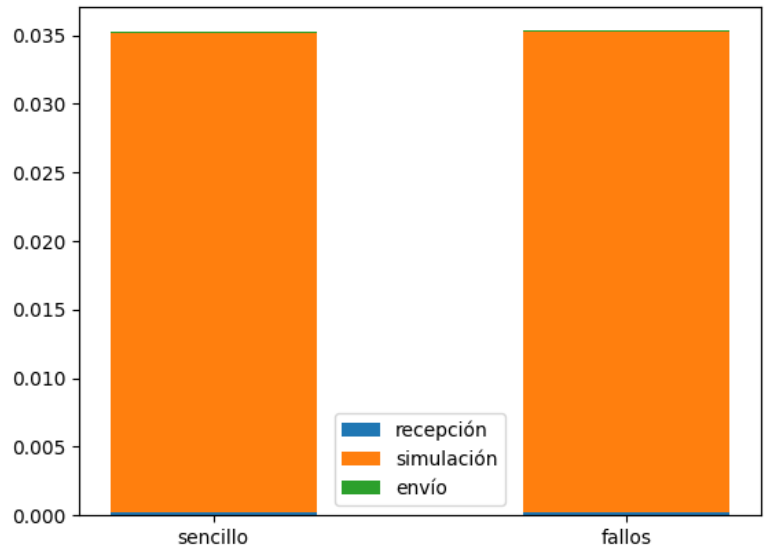

Figura 5.1: Tiempo de Ejecución de la Simulación con Carga Mínima (0.035 s.)
Tiempo de ejecución medio de cada fase de la simulación (recepción, simulación y envío) de la versión `sencillo` y `fallos`. La carga de simulación del modelo es la mínima para ser rentable frente a la simulación distribuida.

Mediante estos cálculos, se obtiene que $L \times E$ debe ser superior a 262500. El valor del *lookahead* L representa el número de *simseconds* que el simulador puede ejecutar antes de esperar futuros eventos de sus predecesores. Dado que la densidad de eventos E de nuestro modelo es 1, un ciclo de simulación debe durar al menos 262500 *simseconds* que, con una frecuencia de $7{,}5 \times 10^6$ *simseconds* por segundo, significa una carga de simulación de 0.035 segundos.

La Figura 5.1 muestra el tiempo de ejecución medio de cada fase de la simulación con la carga calculada. La diferencia entre la versión sencilla y la tolerante a fallos es despreciable (0,269 %) con la carga mínima, siendo mucho menor en situaciones reales. Por ello, podemos afirmar que los mecanismos de tolerancia a fallos no añaden una latencia significativa a la simulación.

Ejecución sin Carga de Simulación

En situaciones reales, la versión tolerante a fallos no aumenta el tiempo de ejecución frente a la versión sencilla. Para evaluar el impacto de los mecanismos de tolerancia a fallos en la comunicación de los *simbots*, se analizan los resultados de las ejecuciones del modelo sin carga de simulación. Mediante análisis detallado del tiempo asociado a cada fase de la simulación (véase Figura 5.2), se observa que solo la fase de envío experimenta un incremento.

Este aumento se debe al envío de eventos a múltiples *simbots* en lugar de un único sucesor, como ocurre en la versión sencilla. En concreto, el crecimiento es atribuible a las consecutivas llamadas al sistema para realizar el envío TCP. En este caso extremo analizado, el aumento en el tiempo de ejecución de la versión tolerante a fallos del simulador es de tan solo un 28 % (ver Cuadro 5.1).

Es importante destacar que el registro del estado no afecta al rendimiento de la réplica, ya que esta no retrasa su ejecución, comparada con el primario. Además, el desacoplamiento de la ejecución entre el primario y la réplica contribuye a que el tiempo de recepción se mantenga igual, es decir, sin esperas adicionales para eventos de sucesores. Este experimento confirma que la tolerancia a fallos no se transfiere a la simulación en general, sino solo a la fase de envío.

5.3.3. Tiempo de Recuperación de Fallos

Los experimentos realizados (véase Figura E.7) revelan que el tiempo medio necesario para recuperarse de un fallo es de 0.4 segundos. La mayor parte de este tiempo (99 %) se atribuye a la espera de la respuesta del *Replica Provisioner* y a la copia del estado a la nueva réplica. En el entorno de experimentación, el tiempo de espera al *Replica Provisioner* es prolongado debido a su método de inicio del proceso de la nueva réplica utilizando SSH. En contraste, en simulaciones de modelos grandes y densos, la copia del estado se convierte en el parámetro más influyente en el tiempo de recuperación.

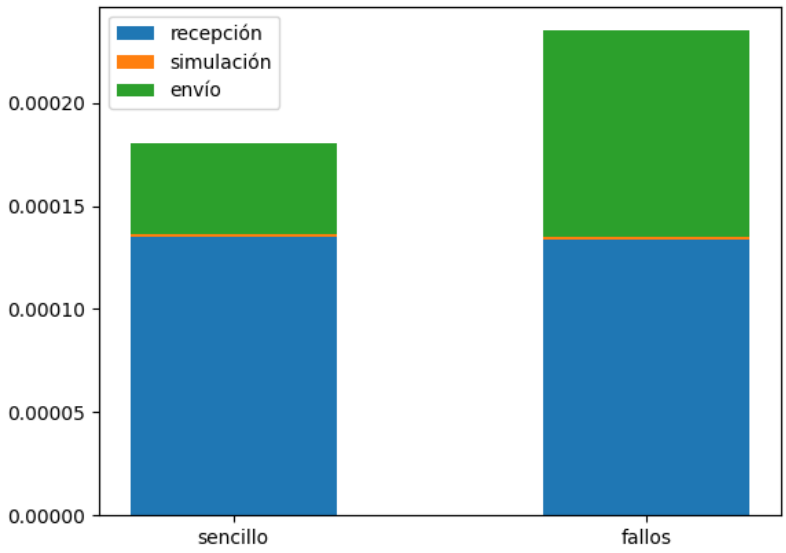

Figura 5.2: Tiempo de Ejecución de la Simulación sin Carga

Tiempo de ejecución medio de cada fase de la simulación (recepción, simulación y envío) de la versión `sencillo` y `fallos`. No existe carga de simulación para observar la influencia de la tolerancia a fallos en la comunicación.

El tiempo de recuperación se considera como el umbral del tiempo total de ejecución, a partir del cual se vuelve rentable emplear la tolerancia a fallos. En caso contrario, reiniciar la simulación resulta más eficiente. Este tiempo varía en cada simulación, ya que depende del tamaño y la estructura del modelo simulado, del entorno de ejecución y de la red, así como del mecanismo de aprovisionamiento de réplicas y de la detección de fallos. En una simulación y entorno nuevos, idealmente, se deberían realizar pruebas iniciales con fallos para estimar el tiempo requerido para recuperarse de un fallo.

Capítulo 6

Conclusiones

Este proyecto de investigación se fundamenta en el desarrollo de la tolerancia a fallos de un simulador distribuido de sistemas de eventos discretos. El objetivo ha sido abordar el compromiso entre prestaciones y tolerancia a fallos del sistema mediante la propuesta de un diseño innovador que preserve el rendimiento en ausencia de fallos.

El modelo de tolerancia a fallos se basa en la replicación, adaptada específicamente para la simulación conservativa. Se ha demostrado que utilizando los mensajes y tiempos propios de la simulación, es posible conservar la consistencia del sistema. Además, el desacoplamiento de la ejecución de las réplicas reduce la cantidad de mensajes y sincronizaciones necesarias, logrando mínima latencia adicional. El simulador se apoya en los mecanismos de detección y recuperación para tolerar los fallos, y en el proveedor de réplicas para incorporar nuevos nodos a la simulación.

Este enfoque garantiza rendimiento óptimo del simulador en su operativa normal, es decir, en ausencia de fallos. El coste de tiempo adicional se limita al envío de eventos para persistir el estado en las réplicas, sin necesidad de sincronización adicional. Como se ha identificado, este coste resulta despreciable en simulaciones largas y de gran tamaño. La efectividad de las estrategias de tolerancia a fallos implementadas se ha verificado a través de experimentos en los que la simulación ha tolerado múltiples fallos.

Adicionalmente, se ha logrado optimizar la implementación del simulador mediante la gestión eficiente de conexiones y estructuras de datos, lo que ha reducido significativamente el tiempo de ejecución. Este resultado adicional del trabajo contribuye en gran medida el proyecto de investigación global sobre el desarrollo del simulador.

En conjunto, el diseño propuesto aborda con éxito el desafío de obtener un sistema tolerante a fallos al mismo tiempo que se preserva las prestaciones de la simulación, cumpliendo ampliamente con los objetivos establecidos. Es importante destacar la relevancia de este trabajo en el ámbito de la tolerancia a fallos para simulación distribuida, ya que mantener un rendimiento óptimo en un entorno distribuido, a pesar de la ocurrencia de fallos, es fundamental para la ejecución simulaciones largas y costosas.

El proyecto sienta una base sólida para futuras investigaciones y mejoras en la tolerancia a fallos de la simulación distribuida. En particular, en el ámbito de nuestro simulador distribuido, se abren numerosas vías de continuación. Principalmente, se considera la incorporación de réplicas adicionales a la simulación. Esta modificación, junto a la implementación de un envío *multicast* fiable que anule el coste de tiempo de envío adicional por cada copia, permite mantener la latencia adicional igual, independientemente del número de copias. Esta posibilidad mejora significativamente las prestaciones de la tolerancia a fallos frente a técnicas de replicación general, donde la carga de la replicación es multiplicativa por cada copia. Todas las propuestas de continuación analizadas tras la finalización del proyecto son recogidas en el Anexo F para facilitar la tarea de retomar el proyecto a futuros estudiantes.

El desarrollo del Trabajo de Fin de Grado ha requerido una gran dedicación de tiempo y esfuerzo. El Anexo G proporciona un desglose detallado de las fases y el tiempo asignado a cada tarea. Impulsado por mi pasión por el tema y los buenos resultados obtenidos, he abordado este trabajo con una alta motivación. A lo largo del proyecto, he superado desafíos significativos, como la necesidad de reemplazar la implementación del simulador y enfrentar momentos de depuración intensiva.

Las experiencias vividas durante el transcurso del trabajo me han brindado valiosas lecciones. He adquirido la capacidad de asumir la responsabilidad en la toma de decisiones importantes y desarrollar habilidades para adaptarme a los cambios y desafíos inesperados. Además, he aprendido a valorar la importancia de la claridad y organización durante el desarrollo de un sistema, para fomentar su comprensión y ampliación. Estas lecciones han contribuido significativamente a mi crecimiento personal y profesional, representando un valioso aprendizaje durante la etapa final de mi formación en Ingeniería Informática.

Capítulo 7

Bibliografía

[1] José Ángel Bañares y José Manuel Colom. «Model and simulation engines for distributed simulation of discrete event systems. [On International Conference on the Economics of Grids, Clouds, Systems, and Services]». En: (2018), págs. 77-91.

[2] José Ángel Bañares Unai Arronategui y José Manuel Colom. «Towards an architecture proposal for federation of distributed des simulators [On GECON 2019 - International Conference on the Economics of Grids, Clouds, Systems, and Services]». En: (2019), págs. 197-210.

[3] José Manuel Colom José Ángel Bañares y Unai Arronategui. «A modern approach for modelling and distributed simulation of health systems [On 17th International Conference, GECON Online]». En: (2020).

[4] Paul Hodgetts et al. «Workload Evaluation in Distributed Simulation of DESs». En: *Economics of Grids, Clouds, Systems, and Services: 18th International Conference, GECON 2021, Virtual Event, September 21–23, 2021, Proceedings 18*. Springer. 2021, págs. 3-16.

[5] Sergio Herrero Barco. «Desarrollo de un framework de simulación de sistemas de eventos discretos complejos, Universidad de Zaragoza». En: (2020).

[6] Álvaro Santamaría de la Fuente. «Diseño e Implementación de un Simulador Distribuido de Eventos Discretos con Mecanismos de Balanceo de Carga, Universidad de Zaragoza». En: (2021).

[7] Fidel Reviriego Navarro. «Diseño e Implementación de un Simulador Distribuido de Alta Escala de Sistemas de Eventos Discretos, Universidad de Zaragoza». En: (2022).

[8] Richard M Fujimoto. *Parallel and distributed simulation systems*. Vol. 300. Citeseer, 2000.

[9] Tobias Kiesling. «Fault-tolerant distributed simulation: a position paper». En: *Universität der Bundeswehr München, Institut für technische Informatik* (2003).

[10] Rachid Guerraoui y André Schiper. «Fault-tolerance by replication in distributed systems». En: *Reliable Software Technologies—Ada-Europe'96: 1996 Ada-Europe International Conference on Reliable Software Technologies Montreux, Switzerland, June 10–14, 1996 Proceedings 1*. Springer. 1996, págs. 38-57.

[11] Ifeanyi P Egwutuoha et al. «A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems». En: *The Journal of Supercomputing* 65 (2013), págs. 1302-1326.

[12] Brian M Oki y Barbara H Liskov. «Viewstamped replication: A new primary copy method to support highly-available distributed systems». En: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. 1988, págs. 8-17.

[13] Leslie Lamport. «The part-time parliament». En: *ACM Trans. on Computer Systems* 16 (1998), págs. 133-169.

[14] Leslie Lamport. «Paxos made simple». En: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), págs. 51-58.

[15] Diego Ongaro y John Ousterhout. «The raft consensus algorithm». En: *Lecture Notes CS* 190 (2015), pág. 2022.

[16] Om P Damani y Vijay K Garg. «Fault-tolerant distributed simulation». En: *ACM SIGSIM Simulation Digest* 28.1 (1998), págs. 38-45.

[17] Divyakant Agrawal y Jonathan R. Agre. «Recovering from multiple process failures in the time warp mechanism». En: *IEEE transactions on computers* 41.12 (1992), págs. 1504-1514.

[18] Divyakant Agrawal y Jonathan R Agre. «Replicated objects in time warp simulations». En: *Proceedings of the 24th conference on Winter simulation*. 1992, págs. 657-664.

[19] Gabriele D'Angelo, Stefano Ferretti y Moreno Marzolla. «Fault tolerant adaptive parallel and distributed simulation through functional replication». En: *Simulation Modelling Practice and Theory* 93 (2019), págs. 192-207.

[20] David Jefferson et al. «Time warp operating system». En: *Proceedings of the eleventh ACM Symposium on Operating systems principles*. 1987, págs. 77-93.

[21] Luciano Bononi et al. «ARTIS: a parallel and distributed simulation middleware for performance evaluation». En: *Computer and Information Sciences-ISCIS 2004: 19th International Symposium, Kemer-Antalya, Turkey, October 27-29, 2004. Proceedings 19*. Springer. 2004, págs. 627-637.

[22] Gabriele D'Angelo et al. «Fault-tolerant adaptive parallel and distributed simulation». En: *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE. 2016, págs. 37-44.

[23] Judith S Dahmann. «High level architecture for simulation». En: *Proceedings First International Workshop on Distributed Interactive Simulation and Real Time Applications*. IEEE. 1997, págs. 9-14.

[24] Johannes Lüthi, Clemens Berchtold y R v Landeghem. «Concepts for dependable distributed discrete event simulation.» En: *ESM*. 2000, págs. 59-66.

[25] András Varga, Yasar Ahmet Sekercioglu y Gregory K Egan. «A practical efficiency criterion for the null message algorithm». En: *European Simulation Symposium 2003*. SCS Europe Publishing House. 2003, págs. 81-92.

[26] «Rust Programming Language». En: (2023). URL: `https://www.rust-lang.org/`.

[27] *The Rust Programming Language: Understanding Ownership*. 2023. URL: `https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html`.

[28] *The Rust Programming Language: Concurrency*. 2023. URL: `https://doc.rust-lang.org/stable/book/ch16-00-concurrency.html`.

[29] *The Rust Programming Language*. 2023. URL: `https://doc.rust-lang.org/stable/book/`.

[30] Kurt Vanmechelen, Silas De Munck y Jan Broeckhove. «Conservative distributed discrete-event simulation on the Amazon EC2 cloud: An evaluation of time synchronization protocol performance and cost efficiency». En: *Simulation Modelling Practice and Theory* 34 (2013), págs. 126-143.

[31] James L Peterson. «Petri nets». En: *ACM Computing Surveys (CSUR)* 9.3 (1977), págs. 223-252.

Anexos

Anexo A

Simulación Distribuida

Este Anexo describe la simulación distribuida de un sistema de eventos discretos con estrategia conservativa. Se utiliza como referencia en el documento principal y sirve para comprender el punto de partida del trabajo.

A.1. Conceptos Generales

La definición y la terminología definida por Vanmechelen *et al.* [30] sobre la simulación de sistemas de eventos discretos sirve como una buena introducción de los conceptos y es la siguiente:

> Una simulación es una representación de un sistema físico que evoluciona en el tiempo. Este sistema físico es modelado por un proceso lógico (LP). Un LP consta de una serie de entidades que completan tareas o procedimientos y que interactúan entre sí intercambiando mensajes. Estos mensajes se representan como eventos para el LP. El estado de las entidades cambia con el tiempo, lo que provoca una evolución en el sistema. El proceso de estos cambios acumulados es impulsado por el avance del tiempo virtual (VT) en la simulación.

> En el modelo discreto de progresión del tiempo, los cambios de estado solo pueden ocurrir en ciertos puntos discretos en el tiempo. Una simulación de eventos discretos (DES) avanza en el tiempo de simulación hasta el momento de ejecución de la próxima acción, también llamada evento. Un evento tiene un tiempo de disparo asociado, que indica el momento en el tiempo de la simulación en que se producirá el evento. La ejecución de un evento puede crear nuevos eventos y la simulación completa termina cuando se han procesado todos los eventos o cuando la simulación alcanza un tiempo de finalización configurado previamente.

El artículo de Vanmechelen, junto a libro de Fujimoto [8], son muy buenos puntos de partida para comprender la simulación distribuida.

En nuestro simulador, para modelar formalmente el comportamiento del sistema a simular, se utilizan las redes de Petri [31]. Las redes de Petri son representaciones matemáticas que permiten expresar un sistema a eventos concurrentes mediante componentes llamados lugares, transiciones y marcas. Las redes de Petri han sido elegidas debido a su sencillez y generalidad para modelar sistemas diferentes y permiten modelar sistemas concurrentes y distribuidos fácilmente.

Dos de los objetivos del simulador es soportar modelos de gran tamaño y ser escalable. Debido a estos dos propósitos se selecciona la simulación distribuida frente a una centralizada. Una única máquina no puede simular modelos de gran tamaño, ya que el almacenamiento de los modelos en memoria es inviable. Para facilitar la escalabilidad, el modelo debe de ser dividido y distribuido en los distintos nodos. Para ello, la red de Petri que representa el comportamiento del sistema, se define de manera jerárquica, para después dividirla en distintas subredes que se pueden distribuir entre los nodos participantes y ser simulados coordinadamente.

Un *simbot* es cada uno de los nodos distribuidos que conforman la simulación. Los *simbots* son nodos independientes que no comparten reloj y conectados por una red de comunicación. Cada *simbot* ejecuta una partición de la red de Petri del modelo completo. Estas particiones se denominan subredes. La subred asociada a cada *simbot* contiene transiciones de entrada y de salida que, respectivamente, originan o están destinadas a otras subredes. Los *simbots* que participan en la simulación están únicamente conectados a otros *simbots* por la red de comunicación. El conjunto de todos los *simbots* del simulador completan la red de Petri que modela DES.

Al ser un sistema distribuido, los distintos *simbots* de la simulación avanzan independientemente, lo que requiere cierta coordinación para simular el modelo. Los eventos generados por la simulación en cada uno de los *simbots* se utilizan como mecanismo de sincronización entre todos los *simbots*, ya que se intercambian eventos entre ellos y estos poseen una etiqueta temporal de cuando han sido generados. Estas etiquetas sirven para determinar el tiempo global de la simulación, y para ordenar los eventos a su llegada y ejecutarlos en la secuencia correcta.

La simulación sigue una estrategia conservativa para determinar hasta que punto del horizonte avanzar la ejecución antes de continuar procesando otros eventos entrantes. Esto significa que un *simbot* espera hasta que sea seguro que no van a llegar eventos con una etiqueta temporal menor al tiempo de simulación. La estrategia de simulación conservativa garantiza la restricción de causalidad local, la cual indica que los eventos entrantes a un *simbot* siempre serán procesados por el *simbot* en el orden correcto de tiempo, establecido por la etiqueta temporal. Para ello, un *simbot* se bloquea hasta recibir un evento con una estampilla temporal que le permita procesar el evento recibido o, si se puede asegurar que no va a llegar un evento con menor estampilla temporal, los eventos que tienen una estampilla menor. En cada paso de la ejecución, se simula hasta el LVHT, el horizonte temporal de la simulación.

Por el contrario, la estrategia optimista de simulación avanza el tiempo de simulación sin seguridad de haber violado la restricción de causalidad. En el momento en el que se recibe un evento con una etiqueta temporal menor al tiempo local de simulación, se recupera el estado del sistema hasta ese evento mediante un mecanismo de *rollback*. Esta estrategia no es utiliza en la simulación y se ha optado por el uso de la conservativa.

La evolución del comportamiento del sistema viene definida por el disparo de las transiciones y el estado actual, determinado los valores de los lugares de la red. Para definir una transición y determinar si puede ser disparada, se utiliza una función llamada *Linear Enabling Function* (LEF). El uso de este mecanismo reduce el tiempo de computación para determinar que transiciones están sensibilizadas y el tamaño de la estructura de datos que representa la subred del sistema.

Al utilizar la estrategia de simulación conservativa, en algún punto los *simbots* esperarán eventos de otros *simbots* antes de continuar con la simulación. Esto puede causar un interbloqueo de *simbots* esperando entre sí. Para evitarlo, se introducen los *lookaheads*, mensajes de evento nulos que contienen la estampilla temporal del mínimo valor de tiempo para cualquier evento futuro procedente desde el *simbot* remitente. De esta manera se puede calcular el horizonte temporal cuando existe ausencia de eventos.

A.2. Partición del Modelo

La red de Petri de la Figura A.1 es una red sencilla con 3 lugares denotados con círculos y 3 transiciones denotadas con rectángulos. Se propone la división del diagrama para distribuirla en 3 *simbots* diferentes. Dado el *simbot* C se define como predecesor el *simbot* B, que simula la subred que contiene transiciones de salida a la subred de C; y, como sucesor, el *simbot* A, que simula la subred que contiene transiciones de entrada desde C.

La simulación de la red en Figura A.1 de manera distribuida es ineficiente, ya que cada transición se ejecuta de manera secuencial. Por otro lado, la ejecución de la red de Petri simple de la Figura A.2, se puede ejecutar de manera eficiente en distribuido, donde subredes B y C simulan en concurrencia las transiciones T_1 y T_2, respectivamente. La subred A sincroniza las subredes B y C en la transición de entrada T_4 y genera marcas que son enviadas como eventos en las transiciones T_3 y T_5. En esta red de Petri, las subredes B y C son vecinos predecesores y sucesores de A.

La definición de predecesores y sucesores se puede extender a una situación más habitual como la descrita en la Figura A.3. El número de predecesores y sucesores puede variar de 1 a más, dependiendo de la subred simulada por el *simbot*. Además, un *simbot* puede tener vecinos que son tanto sus predecesores como sucesores. El *simbot* central recibe eventos de sus predecesores y envía los eventos externos a sus sucesores.

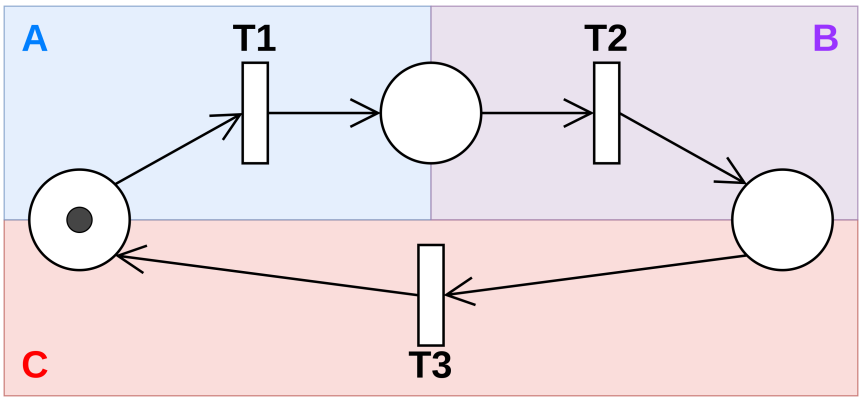

Figura A.1: Partición de una Red de Petri Básica

La partición de una red de Petri básica en la que el modelo se divide en subredes que son asignadas a cada uno de los *simbots*. La simulación del modelo de la figura sería secuencial, donde cada *simbot* tiene un vecino predecesor y otro sucesor.

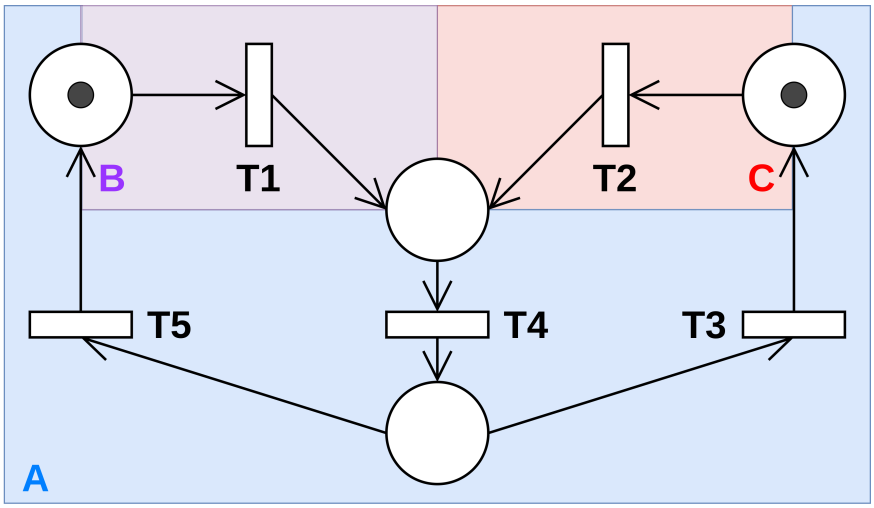

Figura A.2: Partición de una Red de Petri con Concurrencia

La partición de una red de Petri donde las subredes B y C se ejecutan de manera concurrente, favoreciendo la ejecución distribuida.

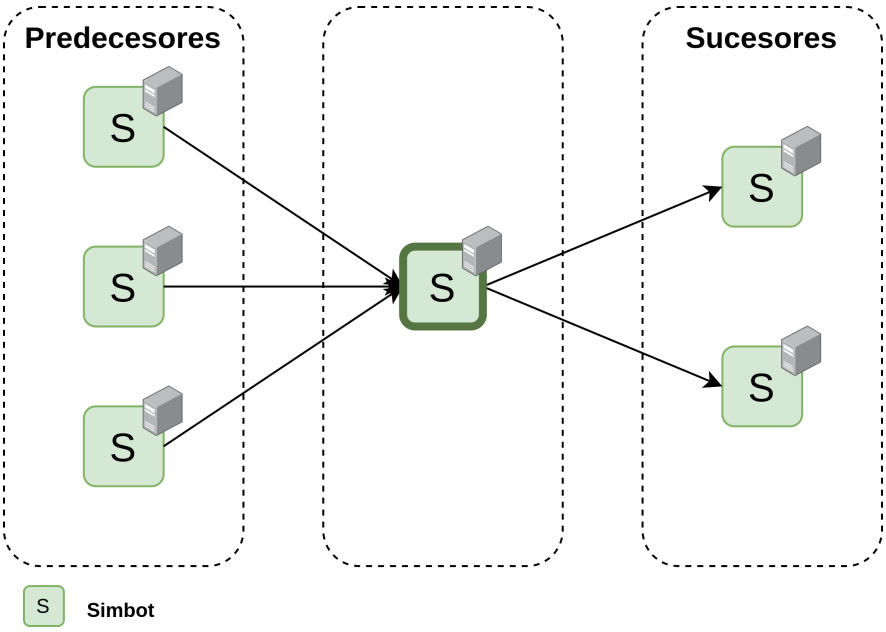

Figura A.3: Estructura de *Simbots* Vecinos

Estructura y nomenclatura de los vecinos del *simbot* S_i. Sus predecesores se denominan S_{i-1} y sus sucesores S_{i+1}. Las flechas muestran el flujo de eventos de la simulación.

A.3. Ciclo de Simulación del *Simbot*

La evolución del estado interno del *simbot* depende de la ejecución del resto de *simbots*, ya que recibe mensajes de sus predecesores y este envía mensajes a sus sucesores. Entre estos mensajes se encuentran los eventos que modifican el estado actual de la simulación, es decir, los disparos de transiciones que hacen avanzar las marcas de la red de Petri.

Al iniciar la ejecución de un *simbot*, este toma la partición del modelo correspondiente a su subred como un conjunto de LEF. Cada *simbot* contiene una cola de prioridad de eventos. Esta cola se ordena de menor a mayor por las estampillas temporales de dichos eventos, lo que permite conocer cuál de los eventos siguientes a procesar es el más cercano al tiempo actual de simulación. En esta cola se reciben tanto los eventos generados por un mismo *simbot* como los eventos generados por los predecesores y enviados al *simbot*.

Tras inicializar el *simbot* con el modelo se ejecuta un paso tras otro de la simulación hasta llegar al ciclo final indicado. Un paso o ciclo de la simulación realiza los siguientes pasos:

1. Se recibe un mensaje por cada subred predecesora. Estos mensajes pueden ser eventos o *lookaheads*. El receptor de los mensajes se bloquea en cada uno de los canales individuales procedentes del hilo de comunicación.
2. Los eventos recibidos se añaden a la cola de eventos y los *lookaheads* se utilizan para determinar el LVHT, que se establece como el tiempo menor de todos los mensajes recibidos en la iteración.
3. Si el primer evento de la cola, que esta ordenada por tiempo, tiene un tiempo menor al LVHT, se establece el tiempo del evento como el nuevo tiempo local. En el caso de que no exista ningún evento o el primero de la cola tenga un tiempo mayor al LVHT, no se simulará nada en este paso, saltando al paso 6.
4. Se procesan todos los eventos con un tiempo igual al tiempo local, que es el tiempo del primer evento de la cola. Al procesar un evento se actualizan los valores de los LEF, lo que puede producir que algunas transiciones se activen.
5. Las transiciones activadas se disparan, lo que genera nuevos eventos. Los eventos que corresponden a transiciones en la subred local son añadidos a la cola de eventos. Los eventos correspondientes a transiciones externas, son añadidos a la lista de envío.
6. Por último, se envía un mensaje por cada subred sucesora. Si a cierta subred le corresponde un evento, se envía, en el caso contrario se le envía un *lookahead* con el tiempo futuro mínimo.

Anexo B

Diseño del *Simbot*

El diseño original del simulador distribuido ha sido heredado de pasados trabajos realizados por otros alumnos. En este anexo se presenta el diseño original y sus carencias, las cuales han motivado un rediseño previo a la implementación de los mecanismos de tolerancia a fallos. Además, se detallan las características del nuevo diseño y de sus principales mejoras, excluyendo el diseño de la tolerancia a fallos, la cual se detalla en el documento principal.

B.1. Análisis del Diseño Original

El diseño original del simulador distribuido [6] es un primer prototipo que se centra en separar la lógica de la comunicación y de la simulación. Se divide el *simbot* en dos hilos de ejecución (*threads*): el de simulación y el de comunicación (*mailbox*). El hilo de ejecución se dedica a ejecutar la simulación y el hilo de comunicación, de hacer llegar a la simulación los mensajes recibidos. Los dos *threads* están conectados mediante una serie de canales, uno por cada predecesor, donde reciben los eventos y *lookaheads* de la subred asociada; y varios canales de control, para enviar mensajes relativos a otros aspectos de la simulación. La estructura de los componentes del diseño se muestra en la Figura B.1.

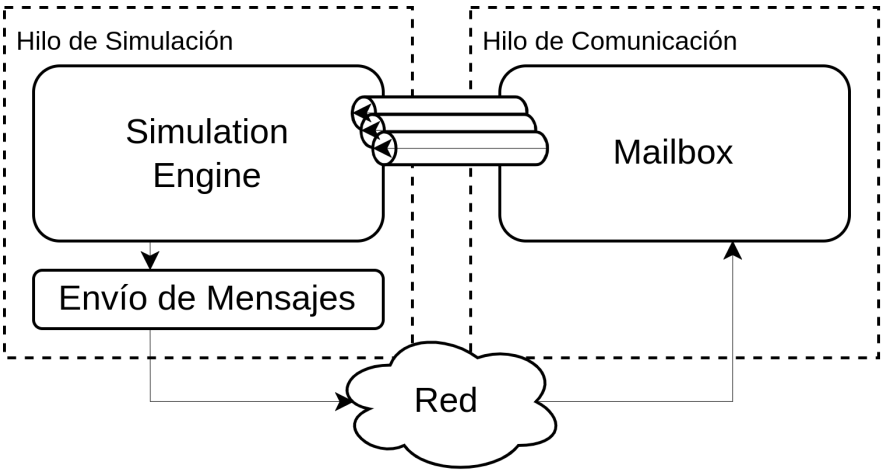

Figura B.1: Arquitectura Original del *Simbot*

Arquitectura del diseño original del *simbot*. Existen dos componentes principales (el *Simulation Engine* y el *Mailbox*), ejecutados en dos hilos diferentes. Existe un módulo auxiliar para realizar los envíos a otros *simbots* en conexiones TCP diferentes.

El hilo de simulación está compuesto por el motor de simulación (*Simulation Engine*), el cual recibe eventos y *lookaheads* de los canales de comunicación respectivos a los predecesores. Estos canales son bloqueantes por lo que, cuando un *simbot* decide recibir mensajes, se bloquea hasta obtener uno por cada uno de sus predecesores.

La función del hilo de comunicación inicialmente es sencilla. Su componente principal es el *Mailbox*, el encargado de recibir mensajes a través de la red de comunicación, procedentes de los otros *simbots*, y redistribuirlos a los distintos canales dirigidos al hilo de simulación. Además, el hilo de comunicación procesa los mensajes no relativos a la simulación; los cuales son pocos en el diseño original, pero aumentan al introducir complejidad.

El *Mailbox* realiza una escucha TCP en la dirección de red asignada en el arranque del *simbot* y acepta una conexión tras otra. Por cada conexión, lee los mensajes enviados, los trata de acuerdo a la lógica del *simbot*, redirigiendo a la simulación los eventos y *lookaheads* a través de los canales de comunicación. Al enviar un evento a un *simbot* vecino, el hilo de simulación utiliza un módulo que crea una conexión TCP, envía el mensaje y cierra la conexión.

B.1.1. Discusión

En cuanto al envío y recepción de mensajes, el método utilizado resulta poco eficiente debido a que la creación de una conexión TCP por cada mensaje a enviar puede incrementar mucho el tiempo de envío. Además, este diseño impide aprovechar características de TCP como la notificación a un extremo de la conexión que el otro extremo se ha cerrado, o en relación con este proyecto, que el otro nodo ha fallado. También impide la identificación del emisor del mensaje a través de la dirección y puerto en el *socket* TCP, ya que las conexiones de salida se realizan con un puerto aleatorio elegido por el sistema al realizar la llamada *connect* de TCP.

Además, las estructuras de datos utilizadas en la simulación son confusas y, en ocasiones, poco eficientes. Por ejemplo, el tipo `Message` se representa como un `struct` de Rust en vez de un `enum`. Representaciones de datos como la de los vecinos de un *simbot* (que son únicos), podrían ser representados como un conjunto (`HashSet`) en vez de como una lista (`Vec`), creando tipos más naturales.

El diseño inicial aborda la problemática del transporte de LEF de un *simbot* a otro. Este mecanismo resulta útil para la tolerancia a fallos, pero se encuentra tan arraigado al diseño original que se decide descartar por una nueva implementación que además incluya la posibilidad de transferir el estado entero de un *simbot*, no solo el modelo ejecutado.

Inicialmente, las direcciones de red de los nodos que ejecutan las diferentes particiones del modelo se encuentran dentro de la definición del los LEF de la subred. Como resultado, cambiar un el *simbot* ejecutando una subred concreta o tener a varios *simbots* ejecutando la misma no resulta sencillo. Lo mismo ocurre con las direcciones de los vecinos, que en una simulación con tolerancia a fallos, pueden cambiar.

Además, se asume que los vecinos predecesores de un *simbot* son los mismos que los sucesores de este. Esta estructuración de las subredes vecinas es correcta en los modelos simples evaluados en anteriores versiones, pero no se generaliza a la totalidad de los casos posibles, donde las particiones de la red de Petri son más complejas. Esto es un error severo de implementación que imposibilita la ejecución de modelos complejos y más cercanos a los reales.

En un desarrollo paralelo del simulador [7] se mejoran los *lookaheads* enviados por los *simbots*, pero el diseño se mantiene muy similar. Existe una primera aproximación a establecer conexiones permanentes entre vecinos durante la totalidad de la ejecución, ya que el hilo de comunicación escucha en varios *sockets* TCP simultáneamente. Pese a ello, los envíos se siguen realizando en conexiones individuales, por lo que no resuelve el problema de las conexiones TCP no persistentes. Esta implementación paralela no resuelve ninguno de los problemas introducidos por el diseño anterior.

La parte del *simbot* que ejecuta la simulación está descompuesta en diferentes módulos y estructuras de datos, pero, en general, la implementación del simulador es poco modular y no facilita la integración de mecanismos adicionales para asistir a la ejecución del modelo. Además, las estructuras elegidas para representar sus tipos de datos incitan a una implementación propensa a errores y con dificultad de depuración.

Se identifica varios puntos de mejora: la gestión de conexiones TCP, los mecanismos de envío y recepción de mensajes, la representación de los tipos de datos utilizados por la simulación, la separación del modelo a simular de en que nodo se está ejecutando y como se comunica con otros nodos, la modularización del código y su documentación.

B.2. Diseño

Debido a que los diseños anteriores imponían diversas restricciones al ampliar la funcionalidad del *simbot*, principalmente el manejo de conexiones entre vecinos, se ha optado por un realizar un rediseño. En esta sección se detalla los nuevos componentes y sus interacciones entre ellos, al igual que las ventajas que estas modificaciones implican. El objetivo del nuevo diseño es ejecutar la simulación de la misma manera, modificando los componentes alrededor de la simulación para permitir introducir otras mejoras. La posibilidad de habilitar la tolerancia a fallos en la simulación con facilidad se ha tenido en cuenta durante el diseño.

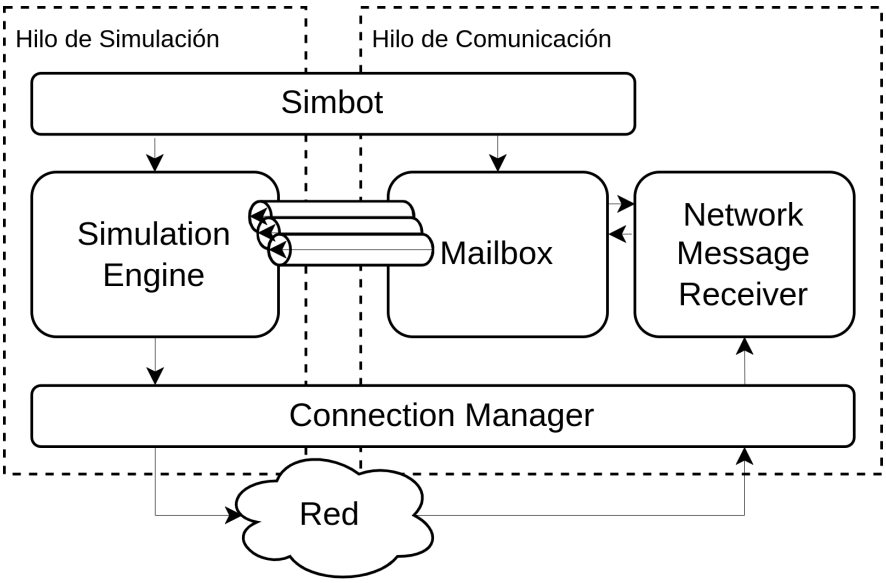

Figura B.2: Arquitectura Nueva del *Simbot*

Arquitectura del nuevo diseño del *simbot*. El componente principal (*Simbot*) coordina la ejecución del *Simulation Engine* y el *Mailbox* en diferentes hilos. Las conexiones TCP por cada uno de los vecinos son gestionadas por el *Connection Manager* y el *Network Message Receiver* asiste con la recepción de nuevas conexiones y mensajes de la simulación. El resto de componentes y estructuras de datos auxiliares se han dejado fuera del esquema por simplicidad.

La estructura general del *simbot* se muestra en la Figura B.2. El componente *Simbot* es el principal, encargado de dirigir y coordinar la ejecución de la simulación y el hilo de comunicación. El *Simulation Engine* se ha reducido a únicamente los datos y funcionalidad relativa a la ejecución de la simulación del modelo, suprimiendo todos los aspectos de comunicación. El hilo de comunicación se ha dividido en dos componentes diferentes que funcionan complementariamente: *Mailbox* y *Network Message Receiver*. El *Network Message Receiver* abstrae la funcionalidad de recibir mensajes de otros nodos de la red. El *Mailbox* es el encargado de procesar los mensajes de acuerdo a la lógica del *simbot*. Por último, el *Connection Manager* abstrae las conexiones TCP del simulador y permite el envío y recepción de mensajes con otros *simbots* sin preocuparse de la red subyacente.

B.2.1. Estructuras de Datos

Existen múltiples estructuras de datos en los anteriores diseños relacionadas con la simulación, entre ellas la lista de eventos, los LEF y las transiciones. Otras ellas, definidas de manera menos explícita se han creado para reducir la complejidad de la información con la que trabaja el *simbot*.

LEF

El diseño de los LEF, que representan las particiones del modelo a simular, se ha mantenido igual salvo dos pequeños detalles. Primero, se han eliminado todas las referencias a la dirección de red del nodo donde se está ejecutando el modelo y se ha sustituido por el índice de subred. El índice de subred es el identificador de la partición que está siendo simulada, y de esta manera se puede cambiar fácilmente que nodo está ejecutando la partición. Segundo, se ha incluido la información sobre el índice de subred de los vecinos predecesores, es decir aquellos que tienen transiciones de salida hacia el modelo simulado. Esta modificación solventa el problema en el que se asumía que los vecinos predecesores eran iguales a los sucesores, posiblemente provocando fallos en la simulación de modelos en los que estos no son iguales.

Vecinos

Se ha extraído la gestión de los vecinos a un módulo propio. La separación de la funcionalidad permite aislar el comportamiento y tratamiento de las relaciones con los vecinos de un *simbot*.

Canales de Mensajes para Vecinos Predecesores

Los *Predecessor Neighbors Channels* se han definido como una estructura con los canales que transportan los mensajes procesados en el hilo de comunicación y son recibidos por la simulación. Se ha incluido funcionalidad para clonar los mensajes dentro de una serie de canales, para restaurar canales con una lista de mensajes, incluso si estos canales tienen mensajes anteriores; y para eliminar mensajes concretos. Esta funcionalidad resulta útil en el modo de tolerancia a fallos.

B.2.2. Receptor de Mensajes

El receptor de mensajes (*Network Message Receiver*) es un componente del *simbot* que escucha en múltiples conexiones TCP y recibe los mensajes en demanda del *Mailbox*. El *Mailbox*, se mantiene como la parte central del hilo de comunicación, el cual ejecuta un bucle que procesa todos mensajes recibidos por el *Network Message Receiver* de acuerdo a la lógica del simulador.

El *Network Message Receiver* tiene las siguientes características:

- Recibe mensajes de nuevas conexiones.
- Registra conexiones para empezar a recibir sus mensajes.
- Recibe mensajes de conexiones ya establecidas.
- Inválida conexiones cuando estas fallan.
- La recepción de mensajes se hace de manera no-bloqueante y a demanda del *Mailbox*, sin perder mensajes.
- Identifica mediante el índice de subred el nodo que ha enviado un mensaje a través de una conexión.
- Gestiona mensajes de *simbots* como de posibles nodos adicionales de la aplicación.
- Es modular y puede ser cambiado por cualquier otra implementación cuando sea necesario.

El registro de conexiones se hace de distintas maneras. Al iniciarse, el receptor de mensajes registrará todas las conexiones ya creadas. Mientras se esté ejecutando, registrará todas las nuevas conexiones que lo soliciten. Por último, registrará a demanda las conexiones iniciadas por otros componentes del *simbot*.

B.2.3. Gestión de Conexiones TCP

Se requiere un sistema con una única conexión TCP por cada vecino con el que se necesita comunicar, lo que facilita la identificación del remitente los mensajes y la detección de fallos cuando se cierre la conexión TCP. Al igual que mantener una única conexión por *simbot* vecino simplifica y optimiza la comunicación entre nodos, también facilita la abstracción de los envíos y recepción de mensajes de las conexiones subyacentes.

El gestor de conexiones (*Connection Manager*) identifica las conexiones mediante el índice de subred correspondiente. Utilizando el índice de subred, un *simbot* accede a la dirección de red asociada a su vecino y a la conexión TCP, si esta existe. Además, permite la eliminación o actualización de estos datos. El *Connection Manager* implementa la inicialización de las conexiones con los *simbots* vecinos necesarios para comenzar la simulación. Por último, el gestor de conexiones abstrae el envío y recepción de mensajes dado un índice de subred.

Las conexiones TCP son bidireccionales, por lo que cuando un nodo se conecta a todos sus vecinos utiliza las conexiones creadas tanto para enviar como para recibir mensajes. Esto conlleva que las conexiones TCP van a ser accedidas simultáneamente por los dos hilos de ejecución: por el de comunicación, al recibir mensajes, y por el de simulación, al enviarlos. Por ello, es necesario incluir mecanismos de sincronización para acceder a los datos compartidos por ambos hilos.

Envío y Recepción de Mensajes

La funcionalidad de envío y recepción de mensajes de la simulación, previamente parte de cada módulo que la necesitaba, se ha agrupado en el gestor de conexiones. De esta manera, se puede enviar o recibir mensajes a través de cierta conexión especificando únicamente el índice de subred del vecino deseado. Además, se ha dividido la funcionalidad de envío de mensajes de control de la de eventos y *lookaheads* para permitir la especificación de un tratamiento adicional a este tipo de mensajes.

Inicialización de conexiones

Las conexiones entre vecinos tienen que inicializarse al comienzo de la simulación. Se ha optado por conectarse únicamente a los vecinos, en vez de a todos los *simbots* involucrados en la simulación, para permitir la escalabilidad del sistema. Al comenzar su ejecución, cada *simbot* recibe la información de quienes son sus vecinos predecesores y sucesores, y en que dirección de red se encuentran escuchando por nuevas conexiones y mensajes. Se necesita realizar una única conexión por cada pareja de vecinos, y por ello establecer una manera determinante de decidir que vecino inicia la conexión TCP, y que vecino la escucha y la acepta.

Algorithm 2 Inicialización de Conexiones de un *Simbot* s_i

$vecinosPorConectar \leftarrow \{s_j \in N_i | j < i\}$
for por cada vecino $s_j \in vecinosPorConectar$ **do** ▷ Fase 1: Conexión
    Conectarse a s_j
end for
$vecinosPorEscuchar \leftarrow \{s_j \in N_i | j > i\}$
for por cada vecino $s_j \in vecinosPorEscuchar$ **do** ▷ Fase 2: Escucha
    Escuchar nueva conexión
end for

Para cada vecino, sea predecesor o sucesor, un *simbot* iniciará la conexión TCP con aquellos que tengan un índice de subred menor al propio. Por otro lado, recibirá las conexiones correspondientes a las subredes con índice mayor. De esta manera, los nodos se coordinan de manera distribuida para no obtener conexiones duplicadas entre vecinos. Como los índices de subred son únicos, definidos en la compilación del modelo, no es posible que falten o existan conexiones repetidas. Sea $S = \{s_0, s_1, \ldots, s_n\}$ el conjunto de todos los *simbots* en la simulación, los nodos predecesores a un *simbot* dado $s_i \in S$ son $P_i \subset S$ y los nodos sucesores son $U_i \subset S$. El conjunto de vecinos de s_1 es $N_i = P_i \cup U_i$. El Algoritmo 2 especifica el proceso que sigue un *simbot* para conectarse a sus vecinos.

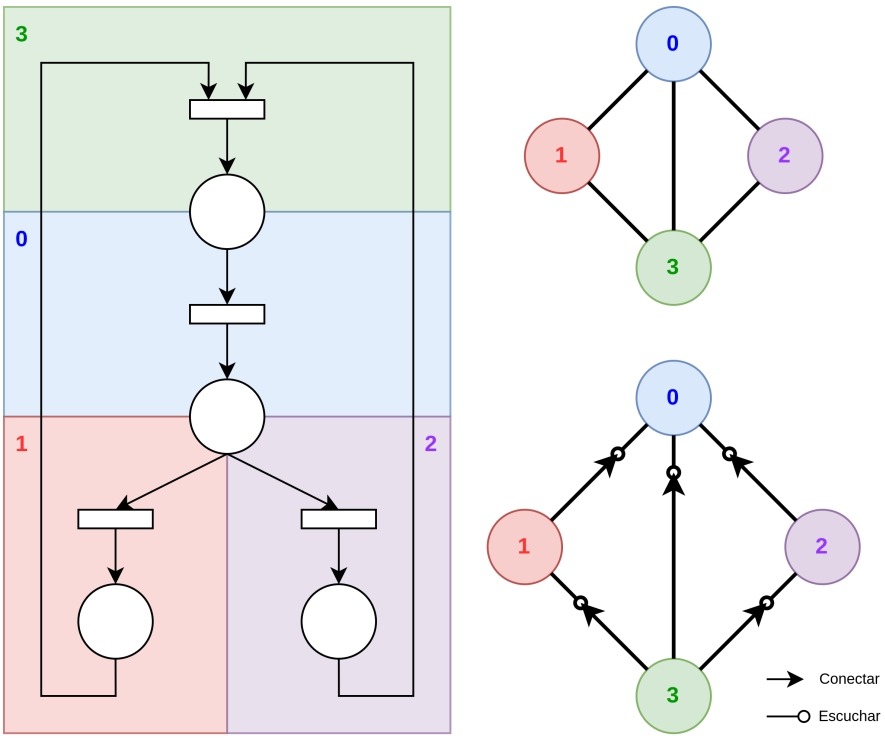

Figura B.3: Inicialización de Conexiones

Visualización del algoritmo de inicialización de conexiones ejecutada por el *Connection Manager*. Izquierda: Red de Petri de ejemplo dividida en 4 subredes, donde cada una tiene asignada un índice de subred. Arriba-Derecha: Representación de los *simbots* de las diferentes subredes y las conexiones necesarias entre los vecinos. Abajo-Derecha: Designación de las acciones determinadas por el algoritmo de que nodo escucha y cuál se conecta al iniciar las conexiones.

Se tiene en cuenta que se van a diseñar mecanismos que introducirán *simbots* a la simulación sin necesidad de sincronizar a todos los otros *simbots*. Para que el proceso de inicialización de conexiones sea homogéneo entre todos los casos, se decide separar la primera fase (la conexión) de la segunda (la escucha). El *Connection Manager* realizará la primera fase, mientras que el *Network Message Receiver*, que permite que otros *simbots* se conecten a él una vez ha sido iniciado, realizará la segunda.

El algoritmo de inicialización de conexiones entre los vecinos garantiza que no existe un bloqueo entre los nodos y que eventualmente todos los nodos se conectarán a ya aquellos que necesitan. Esto es porque la relación de vecinos es mutua y los índices de subred no son repetidos. Por ello, para una pareja de vecinos uno siempre realizará la conexión al otro.

Anexo C

Implementación del *Simbot*

Este anexo describe los detalles de implementación de los componentes más importantes del *simbot* involucrados en la simulación en modo sencillo y de tolerancia a fallos. La implementación se ha desarrollado en dos fases: la primera, adaptando la implementación original a los nuevos componentes encargados de manejar las conexiones TCP, y la segunda, la integración de los mecanismos de tolerancia a fallos. Pese a que el diseño nuevo implica la reimplementación de muchos de los módulos del *simbot*, se ha realizado una mejora incremental para poder asegurar la corrección de la aplicación. En este anexo se incluyen detalles relativos a la implementación en Rust del simulador para permitir futuros trabajos basados en este navegar el código y entender el funcionamiento del proyecto.

C.1. Comunicación Interna del *Simbot*

C.1.1. Envío de Mensajes entre Componentes del *Simbot*

El envío de mensajes entre ambos *threads* se realiza utilizando los canales de la librería `crossbeam_channel`. Al crear un canal se debe especificar el tipo de dato concreto a enviar y se obtiene un `Sender` y un `Receiver`, los cuales se distribuyen a su correspondiente *thread*.

C.1.2. Sincronización de Estructuras de Datos Compartidas

Varias de las estructuras de datos implementadas deben de ser compartidas entre ambos hilos de ejecución (simulación y comunicación). El lenguaje Rust obliga a las estructuras de datos compartidas entre *threads* a implementar los *trait* `Send` y `Sync`. Para permitir que estas estructuras se compartan, se envuelven en los tipos `std::sync::Arc`, una referencia que implementa `Send`, y `std::sync::RwLock`, un *lock* de un escritor y varios lectores que implementa `Sync`. De esta manera se asegura que los *thread* que desean acceder a los datos subyacentes primero deben obtener el *lock*.

C.2. Estructuras de Datos

C.2.1. LEF

El avance de otros proyectos relacionados con el compilador de modelos ha producido en un cambio en la representación de los LEF que consume el simulador. Previamente, un *simbot* tomaba la definición de su partición del modelo como un fichero formato JSON. Ahora, el modelo es transferido al *simbot* como un `struct` del lenguaje Rust. De igual manera, su serialización y de-serialización se realiza utilizando la librería `bincode`, en vez de `serde_json`. Además, porque los LEF son la parte central de la simulación, se han cambiado los nombres de los tipos y de las funciones asociadas a ellos para fomentar su comprensión por un desarrollador no tan familiarizado con la simulación.

C.2.2. Vecinos. `Neighbors`

Los vecinos se han estructurado como un conjunto de índices de subred con una relación asociada: `Predecessor`, `Successor`, o `BothPredecessorSuccessor`. Esta estructura se obtiene de los LEF al inicio de la simulación y permite realizar operaciones poco costosas durante la ejecución.

C.2.3. Canales de Mensajes para Vecinos Predecesores. `PredecessorNeighborsChannels`

Los canales de mensajes entre el `Mailbox` y el `SimulationEngine` están implementados con la librería `crossbeam_channel`. Debido a que la implementación no permite el clonado de los mensajes dentro de los canales, se ha expandido la funcionalidad desarrollando una colección de métodos que lo permiten. Este mecanismo se basa en la lectura de todos los mensajes de un canal hasta vaciarlo, su clonado y su re-envío a través del mismo canal. Para evitar que otros componentes de la simulación accedan a estos canales mientras están siendo modificados, se ha envuelto el `PredecessorNeighborsChannels` con un `RwLock` (*read-write lock*) de la librería `std::sync`.

C.2.4. Mensajes. `Message`

Previamente, los mensajes utilizaban una estructura de datos con uno de sus campos denotando la variante de mensaje a la que pertenecía. Además, el tipo de datos del mensaje contenía campos dedicados a la información que algunos de las variantes de los mensajes incluían. Esta estructura es poco escalable, ya que en cuanto se incluyen más tipos de mensajes que contienen información diferente la cantidad de campos de la estructura se vuelve difícil de manejar y desaprovecha mucho espacio en memoria.

Por las razones expuestas se reestructura el tipo mensaje y se hace uso de la noción de un *enum* en Rust. Los *enum* permiten definir un tipo de datos con múltiples variantes y cada una de ellas contiene un tipo de datos propio. El tamaño en memoria de este tipo es igual al tamaño de la variante más grande. Por ello, las estructuras de datos contenidas en el *enum* `Messages` se han mantenido pequeñas. Los datos que se necesitan enviar entre *simbots* que ocupan considerablemente más tamaño se envían individualmente, utilizando un tipo separado. Esta modificación ha permitido la generalización del tipo mensaje y la inclusión de múltiples variantes correspondientes a mensajes de control (conexiones, caídas, tolerancia a fallos) y depuración.

C.3. Motor de Simulación. `SimulationEngine`

Las estructuras de datos del `SimulationEngine` como la `EventList`, `Lefs` y `Transition`, han sido agrupadas bajo un mismo módulo padre, `simulation`. Además, se ha cambiado el nombre de las estructuras para fomentar su entendimiento. La implementación de la simulación es muy similar a la utilizada en otros trabajos, exceptuando los arreglos a ciertos errores y la inclusión del identificador de mensaje.

El identificador de mensaje acompaña a cada mensaje es generado por el *Simulation Engine* y es utilizado por el *Fault Tolerance Manager* para conservar la consistencia de la simulación, registrando el estado del primario mediante sus confirmaciones e invalidando los eventos repetidos. Su implementación es básica: un contador que se incrementa y se asigna su valor a cada evento generado. Para enviar esta información, se incluye un campo con el identificador en los eventos y *lookaheads*.

C.4. Distinción de Primario y Réplicas

Al inicio de la simulación cada nodo es asignado una partición de la red de Petri del modelo a simular, la cual es contenida en la estructura de datos `LEF`. Los LEF contienen la lista de transiciones de la subred a simular y llevan asignado un índice único (`SubnetIndex`). Este índice se utiliza en diferentes partes de la simulación para abstraer la identificación de los vecinos del nodo donde están siendo simulados, entre otros. En el modo de simulación sencillo solo existe un único nodo simulando una subred, pero los mecanismos de tolerancia a fallos duplican la cantidad de nodos, haciendo necesaria la distinción entre primario y réplica.

El modo sencillo utiliza los números enteros positivos $(0, \infty)$ para representar índices de subred. En el modo de tolerancia a fallos se mantienen estos índices para designar los *simbots* primarios. Dado el índice de subred de un primario P, el índice de subred R de la réplica que simula esta subred es $R = -|P| - 1$. La operación inversa para obtener el índice de la subred de un primario dado su réplica es $P = +|R| - 1$. A la réplica del primario de la subred 0 le corresponde el índice -1, al 1 el -2, al 2 el -3, y así sucesivamente. Se generaliza como que el índice de subred del primario n tiene una réplica con índice de subred $-(n + 1)$.

Se ha definido un *enum* `FaultToleranceRole` para abstraer la diferenciación de primario y réplica del `SubnetIndex`. Los dos valores del *enum* son `Primary` y `Replica`, y son utilizados en toda la implementación para distinguir los *simbots* que deben ejecutar acciones propias a cada uno de los roles.

C.4.1. Creación del *Simbot*

La creación de la estructura principal `Simbot` también difiere dependiendo de que rol ejecuta. Esto se debe a que el *simbot* réplica recibe múltiples estructuras de datos del primario ya inicializadas, las cuales contienen el estado de este, y se deben integrar en el nuevo *simbot*. Por el otro lado, el *simbot* primario debe inicializar estas estructuras utilizando la representación del modelo de los LEF.

C.5. Receptor de Mensajes. `NetworkMessageReceiver`

El módulo `NetworkMessageReceiver` sustituye a un simple `TcpListener` de la librería `std::net`. Pese a que el diseño del nuevo `NetworkMessageReceiver` proporciona mucha más funcionalidad que el *listener* original, implica una implementación compleja.

Para recibir los mensajes procedentes de la red, se ha utilizado un servicio de notificaciones de eventos, y de entrada y salida (I/O) no-bloqueante. La librería `mio` ofrece abstracción del sistema operativo para implementar estas operaciones a bajo nivel. `Mio` ofrece una estructura `Poll` en la que se registran los objetos que generan eventos, en este caso los *TcpStream*. Cuando eventos de lectura o escritura ocurren, el `Poll` se desbloquea y recorre la lista de eventos recibidos identificados por un *token* único asociado a cada conexión TCP de la que se pueden leer los mensajes. Esta técnica permite la escucha general en todos los *sockets* y seleccionar el que ha recibido un evento para realizar la lectura.

Al inicio de su ejecución, el *Network Message Receiver* registra en el `Poll` un único *listener* asíncrono y no-bloqueante en la dirección de red conocida por todos los nodos. A continuación, todas las conexiones entrantes al *listener* son tratadas en función del primer mensaje enviado. Esto fuerza a todos los nodos que desean iniciar una conexión permanente con el *Network Message Receiver* a enviar un mensaje de tipo `InitConnection` indicando el índice de subred del *simbot* remitente. Una vez se ha recibido la petición de iniciar una conexión permanente, el *Network Message Receiver* registra la nueva conexión asignándole un identificador (*Token*) asociado a su índice de subred. A partir de este momento, todos los mensajes a través de esta conexión serán identificados por el *token* y tratados de manera acorde por el *Network Message Receiver*.

Un evento del `Poll` puede ser *writable* o *readable*. Los eventos que interesan al hilo de comunicación son los entrantes *readable*, es decir, los mensajes enviados desde otros *simbots*. Un evento puede notificar al `Poll` la llegada de uno o varios mensajes, o incluso el cierre del otro extremo de la comunicación. Este comportamiento precisa de leer el contenido del *socket* entero y después de-serializar los mensajes individualmente, los cuales varían de tamaño. Debido a ello, se ha implementado una función de recepción de una cantidad variable de mensajes *stream* de lectura, asociada a un *socket* no-bloqueante (véase C.6.2).

C.6. Gestión de Conexiones TCP. `ConnectionManager`

Se ha utilizado el protocolo de comunicación TCP para establecer conexiones persistentes y fiables entre los vecinos que requieren intercambiar mensajes. Debido a que se ha optado por mantener una única conexión por pareja de *simbots* vecinos, se ha creado la estructura de datos compartida `ConnectionManager`, la cual gestiona las conexiones utilizadas para envío y recepción de mensajes entre vecinos. El *Connection Manager* almacena los índices de subred (`SubnetIndex`), las direcciones de red (`SocketAddr`), las conexiones con otros nodos y las relaciones entre todos ellos.

En Rust, las conexiones TCP están representadas por el objeto `TcpStream` de la librería `std::net`, los cuales permiten la transmisión de datos, mediante lectura o escritura. Al utilizar la librería `mio` desde el *Network Message Receiver*, los *streams* han sido sustituidos por `mio::net::TcpStream`. El gestor de conexiones identifica las conexiones por su índice de subred (y rol de tolerancia a fallos, si necesario) y además, ofrece métodos para poder consultar, insertar, actualizar y eliminar estos campos.

El hilo de comunicación, al recibir un mensaje en el *Network Message Receiver*, necesita acceder a la conexión (el `TcpStream`) y, de manera simultanea, el hilo de simulación envía mensajes de eventos y *lookaheads* a sus vecinos, identificando las conexiones con su índice de subred. Por ello, el *Connection Manager* debe permitir el acceso concurrente. Se ha utilizado el `RwLock` (*read-write lock*) de la librería `std::sync` para sincronizar los diferentes componentes de la simulación accediendo al *Connection Manager*.

C.6.1. Inicialización de conexiones

Un método del *Connection Manager* implementa el algoritmo de inicialización de conexiones (Sección B.2.3). Las conexiones TCP en Rust son representadas con el tipo `TcpStream`. Son una representación del *socket* subyacente en el que se escribe y lee al enviar y recibir mensajes, respectivamente. Existen múltiples implementaciones del `TcpStream`, pero se utiliza la implementación de `mio::net` la cual es reconocida por `Poll` del *Network Message Receiver*.

Un *simbot* se conecta a sus vecinos utilizando la misma dirección y puerto que le han sido asignados, de esta manera pueden ser reconocidos por los otros *simbots*. La implementación de *sockets* TCP de `mio::net` en Rust no permite varias conexiones simultáneas desde la misma pareja IP-puerto, pese a que TCP sí que lo permite. Por ello se ha utilizado la implementación de `Socket` de la librería `socket2`, la cual permite la modificación de las *flags* del sistema operativo `SO_REUSEADDR` y `SO_REUSEPORT`. Después de fijar ambas *flags*, se convierte el *socket* a `mio::net::TcpStream`. De igual manera, se utiliza la llamada `setsockopt` con el valor `sockopt::ReusePort` de la librería `nix::sys::socket` para permitir repetir el puerto de escucha en el *Network Message Receiver*.

Al iniciar la conexión con otro *simbot*, puede ser que este no este escuchando todavía. Como las conexiones iniciales con los vecinos son críticas para comenzar la simulación, si la llamada a `connect()` falla, se vuelve a intentar tras esperar un tiempo arbitrario, hasta obtener una conexión exitosamente.

Debido a que los *simbots* réplica entran a la simulación con todas las conexiones inicializadas, deben establecer la conexión con todos aquellos *simbots* con los que se desean comunicar. De igual manera que los primarios utilizan un método del *Connection Manager* para realizar la conexión, mientras que los receptores reciben la conexión en el *Network Message Receiver*.

C.6.2. Envío y Recepción de Mensajes

El envío y recepción de mensajes se ha abstraído utilizando el gestor de conexiones para recuperar el *I/O stream* asociado al *socket* TCP de la conexión y realizar las operaciones de escritura y lectura correspondientes.

Los métodos del gestor de conexiones utilizan las funciones definidas en el módulo `connection_manager::send_receive`. Estas funciones son genéricas y permiten enviar y recibir distintos tipos de datos y, aunque son utilizadas principalmente para enviar datos del tipo `Message`, también se utilizan para enviar LEF y direcciones de red, entre otros.

Debido a que la llamada a la función de recibir mensajes puede darse mientras un mensaje se está transmitiendo y a que los *streams* no-bloqueantes de la librería `mio` no permiten esperar a que haya un mensaje entero en el *buffer* del *socket*, la función de recibir devuelve un número variable de mensajes. De esta manera se asegura que no se leen mensajes incompletos que no se podrán completar en la siguiente llamada a la función de recibir.

C.7. Sincronización de Vecinos

La sincronización entre los *simbots* es un aspecto esencial del correcto funcionamiento de la simulación. Por ello, se han implementado dos mecanismos que permiten a los vecinos coordinarse para ejecutar acciones: una barrera y una espera de inicialización de conexiones.

C.7.1. Barrera TCP

Se ha implementado una barrera selectiva que permite elegir con que *simbots* de la simulación sincronizarse. Hace uso de las conexiones TCP del *Connection Manager*, lo que implica que estas conexiones deben haber sido establecidas previamente a la ejecución de la barrera. La función que ejecuta la barrera en cada nodo se desarrolla en 2 pasos. Primero, se envía un mensaje `Barrier` a cada uno de los vecinos involucrados. Seguidamente, se bloquea hasta obtener un mensaje `Barrier` de cada uno de los vecinos involucrados.

C.7.2. Espera a Conexiones Inicializadas

La ejecución de la simulación depende de la existencia de ciertas conexiones con los vecinos de un *simbot* definidas en la Figura 4.1. Es necesario que en el periodo de inicialización y tras la recuperación de un fallo cada *simbot* se bloquee hasta tener todas las conexiones necesarias. Se ha implementado la *struct* `WaitConnections` que permite a un *simbot* dado esperar a que todas las conexiones se hayan establecido. Se utiliza un `WaitConnectionsNotificator` para notificar desde el *thread* de comunicación al `WaitConnections` de las conexiones entrantes. Mientras, el *thread* de simulación del *simbot* espera utilizando un `crossbeam_channel`. Al obtener todas las conexiones, se desbloquea el *thread* enviando un mensaje a través del canal.

C.8. Registros de Tolerancia a Fallos

C.8.1. Invalidación de Mensajes Repetidos. `FaultToleranceEventRegister`

El `FaultToleranceEventRegister` implementa un registro, el cual almacena el último mensaje recibido por un *simbot*, sea evento, *lookahead* o confirmación. Cuando un *simbot* registra un nuevo mensaje, este es comparado con el almacenado previamente, y si coinciden no es procesado por el *Mailbox*. Cada *simbot* tiene una entrada en el registro por cada predecesor y, si es un *simbot* réplica, para su primario.

C.8.2. Diferencia de Estado entre Primario y Réplica. `FaultToleranceReplicaStateTracker`

El `FaultToleranceReplicaStateTracker` implementa el Algoritmo 1 para mantener el registro del estado del primario y ejecutar la recuperación de la consistencia en el caso de que este falle. Este *struct* utiliza dos `usize` para almacenar los identificadores M^P y M^R, y utiliza una cola `VecDeque` para Q^R. El *struct* implementa ambos procedimientos del algoritmo y realiza las comprobaciones necesarias. Durante el desarrollo se ha utilizado este mecanismo para detectar situaciones de pérdida de mensajes.

C.9. Detección de Fallos

La detección de fallos se fundamenta en el uso de las conexiones TCP establecidas al inicio de la simulación. Al enviar o recibir un mensaje, si el *simbot* en el otro extremo de la conexión ha fallado, la operación falla y se asume que este ha fallado. Por otro lado, la estructura `mio::Poll` utilizada en el *Network Message Receiver* implementa la notificación asíncrona de eventos en los *socket* TCP. La API de Linux *epoll* notifica cuando el extremo contrario de la conexión ha sido cerrada utilizando el *flag* `EPOLLHUP` y se recibe como un evento `read_closed` en el `Poll`. Debido a que todas las conexiones de un *simbot* generaran el evento `read_closed` cuando el nodo al otro extremo falle, un *simbot* cualquiera detectará el fallo de todas los nodos con los que haya establecido una conexión: sus vecinos predecesores y sucesores, y su réplica o primario.

C.10. Recuperación de Fallos

Los procedimientos de recuperación de fallos diseñados en la Sección 4.3 han sido implementados como parte del `FaultToleranceManager`. El `Simbot` en ejecución comprueba la detección de fallos antes y después de enviar y recibir eventos, y en caso de su existencia, actúa para recuperarlo. Al detectar un fallo, se registra para decidir que acciones tomar en función de que *simbot* ha fallado. En caso de que ambos primario y réplica hayan fallado, se notifica el fallo fatal y se aborta la ejecución. Las funciones de recuperación de fallos utilizan los métodos de sincronización diseñados para asegurar el orden correcto de ejecución de las acciones.

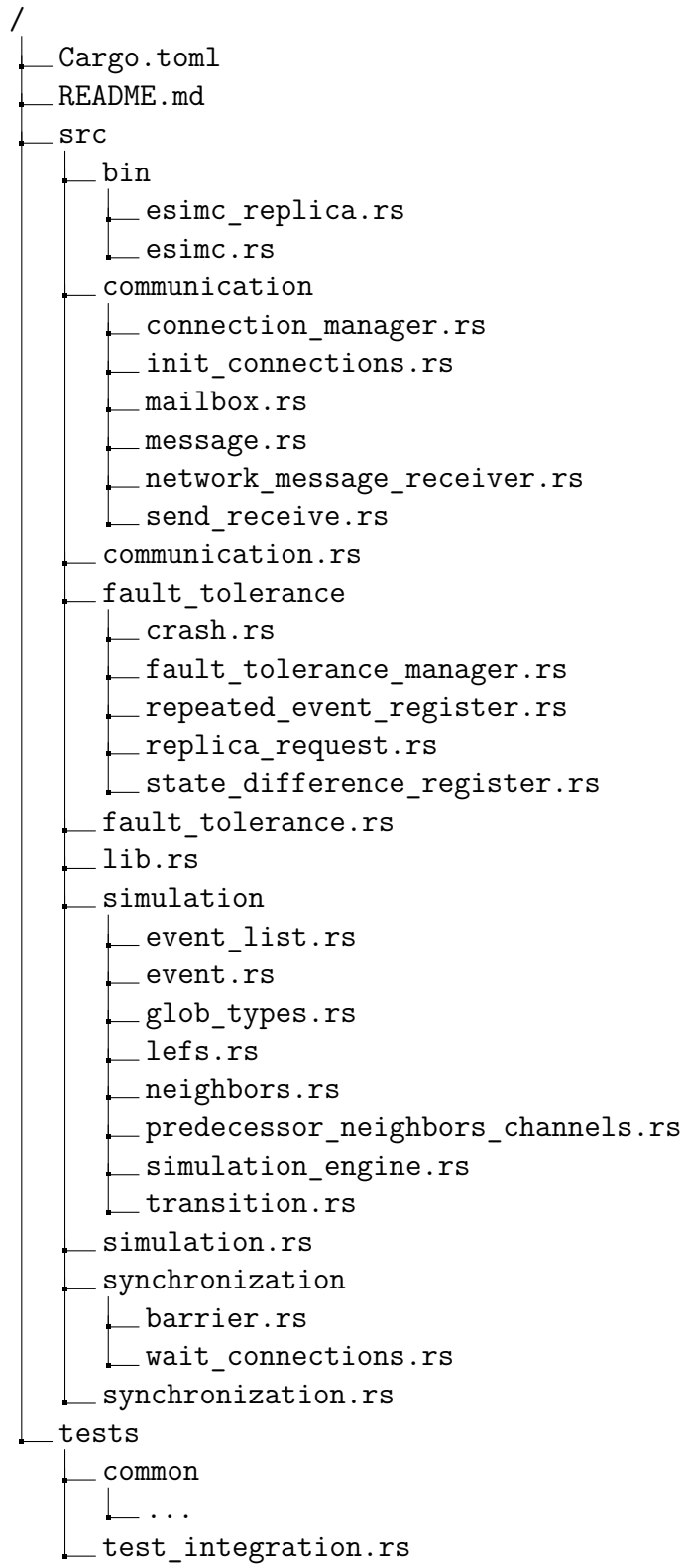

Figura C.1: Árbol de Directorios del Proyecto `simbot`

Anexo D

Servicios Auxiliares Centralizados

La funcionalidad que requiere el simulador distribuido se extiende más allá del *simbot*. Varios sistemas complementarios han sido desarrollados en otros trabajos paralelos con el objetivo de enriquecer el uso del simulador. El compilador del modelo es un componente principal de la fase previa a la ejecución de la simulación, donde un modelo definido como una red de Petri se divide en subredes y luego transforma en los LEF correspondientes. El sistema de información que supervisa la simulación permite visualizar los resultados y asiste en la toma de decisiones.

Durante el desarrollo de este trabajo, dos servicios auxiliares se han desarrollado para asistir a la ejecución de la simulación. El servicio de depuración permite registrar (*log*) mensajes de todos componentes de la simulación de manera centralizada, para asistir con la depuración del código y de la simulación, en general. El proveedor de réplicas proporciona a la simulación con *simbots* preparados para ser incluidos dentro de la simulación y empezar a ejecutar.

Ambos servicios han sido desarrollados específicamente para este proyecto, pero teniendo en mente su posible uso por parte de otros proyectos si fuese necesario. Han sido implementados en Rust para facilitar su compatibilidad con el *simbot*. El código para ambos proyectos reside en el mismo repositorio [1] de GitHub que el *simbot*. Para la organización del código se ha utilizado un *workspace* de Cargo, el cual permite definir múltiples *crates* en un mismo directorio.

[1] `https://github.com/simbots-swarm/simbot_fault_tolerance`

D.1. Servicio de Depuración Centralizado

El servicio de depuración o *Debug Server* se idea inicialmente como un servidor al que todos los *simbots* pueden enviar *logs* para que sean procesados de manera centralizada y sacados por pantalla por un mismo proceso. Este uso se puede extender a otros servicios los cuales recogen y analizan los *logs*. El servicio resulta muy útil durante el desarrollo, ya que facilita la consulta de los *logs* para depurar la aplicación ante fallos de ejecución. Para ejecutar el servidor se ha creado un binario dentro del *crate* del *Debug Server* que dada la dirección de red donde va a escuchar, crea el servidor y lo ejecuta.

D.1.1. Conexión al *Debug Server*

Cada *simbot* que desea conectarse al *Debug Server* utiliza la API de la librería `DebugServerAPI`. Este componente está presente en todas las estructuras del simulador que necesitan conectarse al servidor. Para inicializarse, únicamente se necesita la dirección donde el servidor está escuchando.

La API está programada en Rust de manera que el *simbot* puede importar la librería y utilizar las funciones para realizar las peticiones al *Debug Server* mediante tipos de mensajes preestablecidos, diferentes a los de la simulación. Cada petición utiliza una conexión TCP nueva. Esto es una gran desventaja, pero se ha optado por este diseño debido a su simplicidad. Si en el futuro el *Debug Server* cobra más importancia y se necesita una conexión permanente, se debería implementar un *listener* de varias conexiones (similar al *Network Message Receiver*) y una manera de compartir la conexión entre todos los módulos del *simbot* intentando acceder al `DebugServerAPI`.

D.1.2. Mensajes de *Log*

La funcionalidad principal del *Debug Server* es la recolección de *logs* centralizada y su análisis. Los clientes del *Debug Server*, principalmente los *simbots*, utilizan la API definida para comunicarse con el servidor. El *Debug Server* permite diferentes niveles de *log*: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL` e `INTERNAL`. Utilizando el nivel establecido por el *Debug Server* se determina si el *log* se saca por pantalla o no.

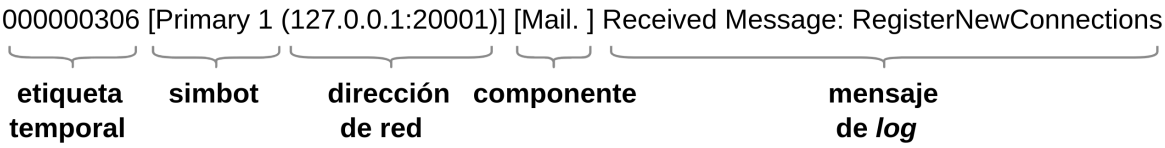

Figura D.1: Mensaje de *Log* del *Debug Server*

Información del mensaje de *log* sacado por pantalla por el *Debug Server*.

La información mostrada por los mensajes de *log* en la versión actual del servidor se expone en la Figura D.1. El primer campo es la marca de tiempo (*timestamp*) del momento en el que se ha generado el *log*, en microsegundos. El segundo campo es el número identificador de la subred y el rol de tolerancia a fallos del *simbot* que genera el *log*. El tercero es la dirección de red del *simbot*. El cuarto, la abreviatura del componente que ha generado el mensaje, por ejemplo el *Mailbox* o el *Connection Manager*. Y por último, se muestra el mensaje *log* enviado. Mientras que el identificador del *simbot* es detectado por el *Debug Server* en el envío, el resto de campos son enviados por el *simbot* como una cadena de caracteres.

D.1.3. Fichero de *Log*

De manera predeterminada, el *Debug Server* saca por pantalla los *logs* de la simulación. Utilizando una configuración adicional, se puede notificar al *Debug Server* de incluir los *logs* en un fichero aparte, permitiendo su análisis posterior. Mediante la automatización de la creación de distintos ficheros de *log* se puede obtener una ejecución más apta para la depuración y menos manual.

D.1.4. Mapa de *Simbots*

Una de las funcionalidades adicionales del *Debug Server* permite recoger el identificador y la dirección de red de un *simbot* y mantenerla actualizada durante toda la ejecución de la simulación, en la que un *simbot* puede cambiar de máquina o ser cambiado de rol de tolerancia a fallos. Para preservar la información actualiza, cada *simbot* comunica su identificador y su dirección de red con la primera conexión al *DebugServer*. Además, tras un cambio, un *simbot* debe retransmitir la nueva información.

D.1.5. Simulación de Fallo

En los experimentos iniciales con la tolerancia a fallos el *Debug Server* se utiliza para notificar a los *simbots* de cuándo simular un fallo. Depende del entorno de ejecución de la simulación, el fallo se deberá realizar de una manera diferente. En el entorno de ejecución local se ha experimentado con *simbots* como *threads* y como procesos.

El *Debug Server* puede simular el fallo de un proceso ejecutando el comando `kill -9 <PID>` con el que se mata el proceso. Se necesita el PID (*Process IDentifier*), el cual es transmitido en el mensaje de actualización de información de un *simbot* junto a su identificador.

Para simular el fallo de un *thread* el *Debug Server* manda un mensaje al *Network Message Receiver* del *simbot* que se desea parar y este, al recibirlo, acaba su ejecución abruptamente con fallo.

Simular el fallo de manera centralizada permite determinar en que tiempo de la simulación o tras que condiciones se debe simular el fallo, obteniendo más facilidad para realizar la experimentación. En la última versión, el *DebugServer* es inicializado con la lista de *simbots* que deben fallar y el *timestamp* en el que se ejecutará el fallo. Cuando un mensaje del *simbot* que se quiere que falle es recibido, si tiene el *timestamp* correspondiente, se toma la acción que produce el fallo.

D.1.6. Notificación de Fallo Fatal

Por último, el *Debug Server* se utiliza para transmitir a todos los *simbots* la ocurrencia de un fallo fatal. Cuando un fallo fatal ocurre en la simulación, no se puede recuperar, por lo que todos los *simbots* deben de ser notificados y abortar la simulación. Cuando un *simbot* detecta un fallo fatal, lo notifica al *Debug Server* y este notifica a todos los otros nodos de la simulación, los cuales conoce su dirección.

El mensaje de notificación de un fallo fatal es recibido por el *Network Message Receiver* de todos los *simbot*. Como se utiliza un tipo de mensaje distinto, el *Network Message Receiver* debe detectar que es un mensaje del *Debug Server* y leer el mensaje como otro tipo. Este mecanismo permite el envío de muchos otros tipos de mensajes al *simbot* y puede ser utilizado en el futuro para implementar otros mecanismos.

D.1.7. Implementación

La implementación del *Debug Server* es un simple servidor TCP que escucha conexiones y trata de manera síncrona las peticiones. El *crate* utiliza la librería estándar de Rust `std::net` para tratar las conexiones TCP.

Interacción con *Debug Server*

Existen dos maneras de interactuar con el *Debug Server*: la API implementada en Rust o los binarios ejecutables. La API permite la interacción de un *crate* de Rust con un *Debug Server* que se está ejecutando. Por otro lado, los binarios permiten conectar mediante un comando con el *Debug Server* para enviarle ciertos mensajes de control. La API está pensada para la interacción desde el *simbot*, mientras que los binarios, para la interacción en las pruebas automatizadas.

Lista de Mensajes

La lista de mensajes que utiliza el *Debug Server* para comunicarse es la siguiente:

- `DebugLog { network_address: SocketAddr, timestamp: usize, thread_header: String, message: String }`: Mensaje de *log* enviado por el *simbot* con la información que desea que el *Debug Server* registre.
- `DebugSubnetNotification { address: SocketAddr, simbot_header: String, pid : u32 }`: Mensaje de actualización de dirección de red, de identificador del *simbot* y de PID.
- `DebugCrash`: Mensaje enviado por el *Debug Server* a un *simbot* para que simule un fallo.
- `FatalCrash { crashed_subnet: usize }`: Mensaje enviado por el *Debug Server* a un *simbot* para notificarle de un fallo fatal en la simulación.
- `Shutdown`: Mensaje enviado al *Debug Server* para que termine su ejecución.
- `OpenOutputFile { filename: String }`: Mensaje enviado al *Debug Server* para que inicie el *log* en el fichero `filename`.
- `CloseOutputFile`: Mensaje enviado al *Debug Server* para que cese el *log* a un fichero.

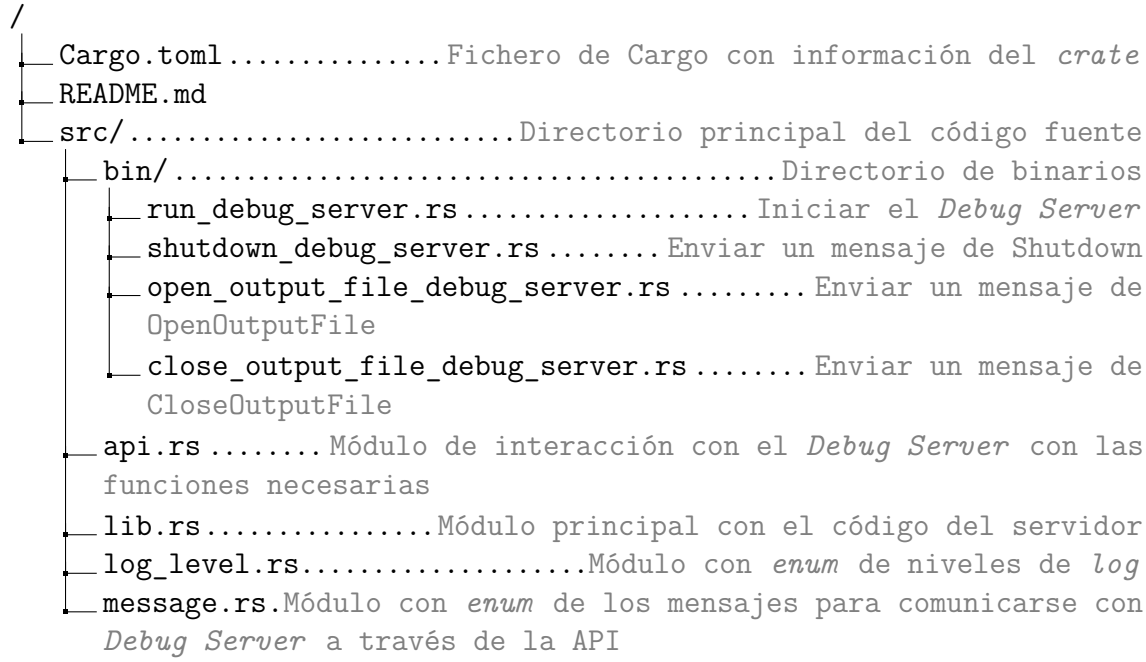

Figura D.2: Árbol de Directorios del Proyecto `debug-server`

Estructura de Ficheros

La Figura D.2 muestra la estructura de ficheros del *crate* `debug_server`, con todos los módulos que lo forman.

D.2. Servicio Proveedor de Réplicas

El proveedor de réplicas o *Réplica Provisioner* es un servicio que inicializa un *simbot* y lo prepara bajo demanda de otros *simbots* de la simulación. El *Replica Provisioner* es propio al entorno de ejecución de la simulación, es decir, debe implementar las técnicas específicas para crear un nuevo nodo dependiendo del entorno. Para ejecutar el servidor se ha creado un binario dentro del *crate* del *Replica Provisioner* que dada la dirección de red donde va a escuchar, crea el servidor y lo ejecuta.

D.2.1. Conexión al *Replica Provisioner*

El método de conexión al *Replica Provisioner* es el mismo que el del *Debug Server* (Sección D.1.1). Se utiliza la API de la librería `ReplicaProvisionerAPI`, también programada en Rust y con su propio tipo de mensajes.

D.2.2. Creación de un *Simbot*

El proveedor de réplicas debe ser capaz de crear *simbots* para cualquier entorno de ejecución de la simulación.

- **Entorno de Desarrollo Local** (*Thread*): Para crear un *simbot* desde un nuevo *thread* el *Replica Provisioner* puede crear la estructura `Simbot` desde Rust y ejecutarla o ejecutar el binario del *simbot* directamente. Ya que el proveedor de réplicas puede ser implementado en varios lenguajes y no debe tener dependencias de la librería principal, se opta por la segunda opción.
- **Entorno de Desarrollo Local** (Proceso): De la misma manera que para un *thread*, el proceso ejecuta el comando del binario del *simbot* de manera asíncrona. El proceso se asemeja más a un sistema distribuido al igual que permite simular fallos matando el proceso sin intervención de la aplicación.
- **Entorno de Experimentación** (Vagrant VMs): El *Replica Provisioner* se pone en contacto con una máquina virtual a través de SSH y ejecuta el comando ya compilado que inicia una nueva réplica.

D.2.3. Proceso de Inicialización de una Réplica

El proveedor de réplicas debe ofrecer a los *simbots* de la simulación la misma interfaz para realizar peticiones y recibir respuestas independientemente del entorno donde se ejecute. De igual manera, el proceso de creación de una réplica es siempre el mismo. La Figura D.3 muestra la interacción entre un primario y el proveedor de réplicas para la creación de una nueva réplica.

D.2.4. Implementación

La implementación del *Replica Provisioner* es un simple servidor TCP que escucha conexiones y trata de manera síncrona las peticiones. El *crate* utiliza la librería estándar de Rust `std::net` para tratar las conexiones TCP.

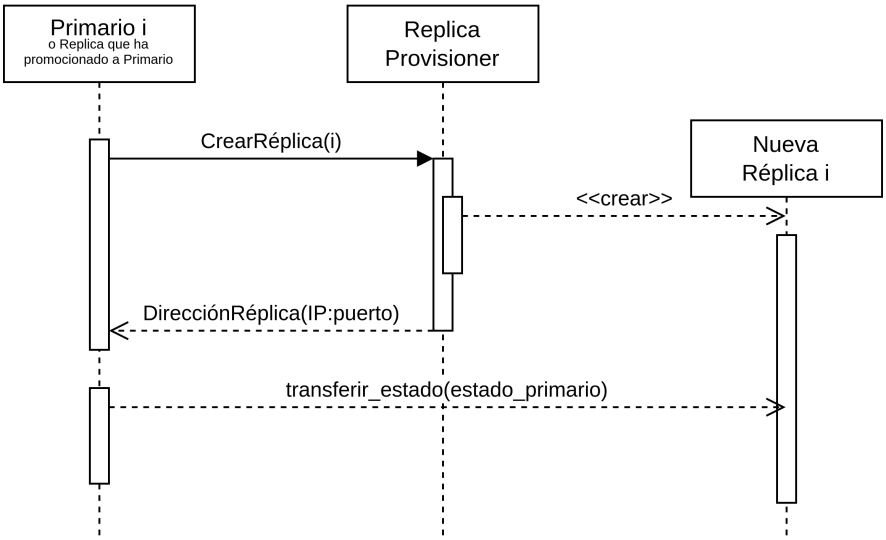

Figura D.3: Diagrama de Secuencia de Petición de Réplica

Diagrama de secuencia de la interacción entre el *simbot* primario y el *Replica Provisioner* al solicitar una nueva réplica y la transmisión de su estado.

Diferentes Proveedores de Réplicas

El módulo `provider` define un *trait* (`ReplicaProvisionerProvider`) el cual es implementado por todos los submódulos de este. Las estructuras que implementan este *trait* deben de ser capaces de ejecutar réplicas a demanda utilizando un mismo método llamado `create_replica()`. El `Provider` concreto debe de controlar que réplicas siguen activas, cuando se han agotado y el paso de argumentos a la nueva réplica.

Lista de Mensajes

La lista de mensajes que utiliza el *Replica Provisioner* para comunicarse es la siguiente:

- `CreateReplica { primary_subnet: usize }`: Mensaje enviado al *Replica Provisioner* para la creación de una réplica a demanda.
- `NotifyReplicaAddress { replica_address: SocketAddr }`: Mensaje enviado al *simbot* para comunicarle la dirección de una nueva réplica.
- `Shutdown`: Mensaje enviado al *Replica Provisioner* para que termine su ejecución.

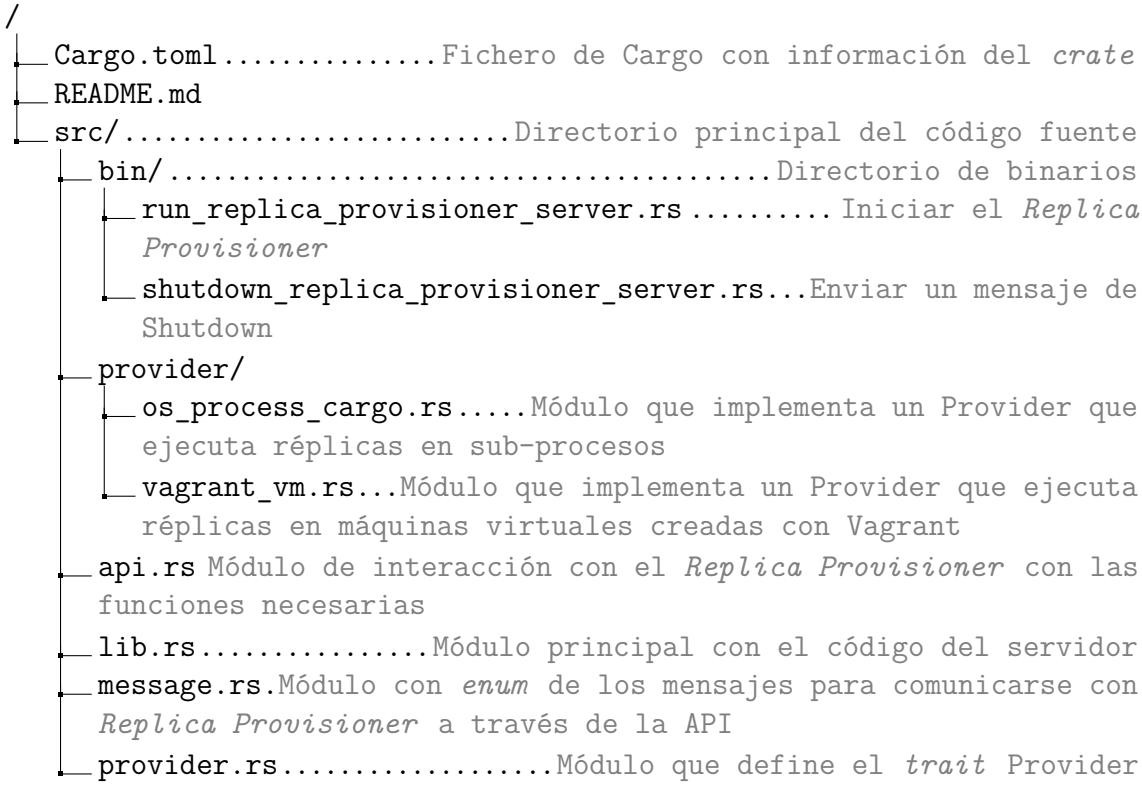

Figura D.4: Árbol de Directorios del Proyecto `replica-provisioner`

Estructura de Ficheros

La Figura D.4 muestra la estructura de ficheros del *crate* `replica_provisioner`, con todos los módulos que lo forman.

Anexo E

Resultados de Experimentos Adicionales

Este Anexo incluye los resultados de los experimentos complementarios sobre la corrección y tolerancia a fallos del simulador implementado.

E.1. *Simulation Metrics*

La Figura E.1 muestra un ejemplo de la salida del *Debug Server* con los resultados finales de un nodo en una simulación.

E.2. Tolerancia a Fallos del Simulador

Mediante este experimento se prueba el correcto funcionamiento de los mecanismos de tolerancia a fallos. Se ha comprobado que la simulación termina correctamente y que todos los *simbots* vecinos implicados en la recuperación del fallo se sincronizan correctamente mediante las trazas enviadas al *Debug Server*. Se han comprobado las situaciones relacionadas con fallos mostradas a continuación.

E.2.1. Fallo de Réplica

Las trazas de la Figura E.2 muestran la detección y recuperación del fallo de una réplica, mediante la petición de una nueva réplica.

E.2.2. Fallo de Primario

Las trazas de la Figura E.3 muestran la detección y recuperación del fallo de un primario, mediante la promoción de la antigua réplica a nuevo primario.

E.2.3. Fallo Simultaneo de Dos Subredes

La trazas de la Figura E.4 muestran la detección y recuperación del fallo de dos *simbots* en dos subredes diferentes.

E.2.4. Fallo Fatal (Primario y Réplica)

Las trazas de la Figura E.5 muestran la detección y el aborto de la simulación por parte de todos los *simbots*.

E.3. Corrección de la Simulación

Es esencial probar que los mecanismos de tolerancia a fallos implementados no afectan al resultado final de la simulación y que la consistencia entre primario y réplica se mantiene tras los fallos. Un error en el registro del estado del primario o una copia incorrecta del modelo a una nueva réplica incorporada puede afectar al desarrollo de la simulación y a sus resultados.

Para probar que los resultados son los mismos, se ha recogido la lista de transiciones disparadas de la simulación junto a los tiempos en los que han sido disparadas. Las transiciones disparadas por las diferentes versiones del simulador desarrolladas en este proyecto han sido comparadas con la versión desarrollada y probada más recientemente [6]. Se ha probado la corrección del simulador en modo sencillo y en tolerancia a fallos; este último, ejecuciones con y sin fallos. Todas las versiones han obtenido la lista de transiciones disparadas de la Figura E.6.

E.4. Tiempo de Recuperación de Fallo

La Figura E.7 muestra la salida de los resultados de un nodo primario de la subred 1 tras recuperar un fallo de la réplica de su misma subred.

```
[Primary 0 (192.168.1.30:51807)] [Simbot]
----------------- RESULTS for 0 -------------------
NUMBER OF FIRED TRANSITIONS: 66667
Total Cycles: 100000
Total Simulation Time (seconds): 255.124364143 seconds
Performance (Events per Second):    P=391.96570008479824
Event Density (Events per Cycle):   E=1
Relative Speed (Cycles per Second): R=P/E=391.96570008479824
---------- EXTENDED SIMULATION METRICS++ ----------
No. Cycles:                 100000
Simulation Duration:        255.12436 s
Cycle Duration Avg.:        0.00274 s
Sim Step Duration Avg.:     0.00211 s    (82.29%)
Send Duration Avg.:         0.28099 ms    (10.55%)
 |- Lock Wait Avg:          0.00033 ms    (0.12%)
 |- Register Avg:           0.00053 ms    (0.23%)
 |- Connection Avg:         0.27684 ms    (98.38%)
Receive Duration Avg.:      0.15313 ms    (5.46%)
Send Error Count:           0
Receive Error Count:        0
Fatal Error Count:          0
Recovery Duration:          0.00000 ms
 |- Sync                    0.00000 ms
 |- Replica Creation        0.00000 ms
 |- Promotion               0.00000 ms
Latency:                    AVG: 0.000343941 s
                            MED: 0.000337516 s
--------------- END RESULTS for 0 -----------------
```

Figura E.1: Ejemplo de Métricas de Experimentos

Ejemplo de métricas recogidas por los *simbots* y enviadas al final de la simulación al *Debug Server*. Estas métricas han sido utilizadas en los experimentos.

```
[Primary 0] [MsgRcv] Connection closed: SubnetIndex(-2)
[Primary 0] [MsgRcv] Crash Detected SubnetIndex(-2)
[Primary 0] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(-2))
[Primary 1] [MsgRcv] Connection closed: SubnetIndex(-2)
[Primary 1] [MsgRcv] Crash Detected SubnetIndex(-2)
[Primary 1] [FTMan.] Crash of Type ReplicaCrash
[Primary 0] [Wait. ] Wait Requested for {SubnetIndex(1)}
[Primary 0] [Wait. ] Unblocked Successfully
[Primary 0] [Wait. ] Wait Requested for ALL NEIGHBORS
[Primary 1] [Simbot] Requesting Replica Creation
[Replica Provisioner] Replica Child Process for 127.0.0.1:20004 Joined.
[Replica Provisioner] Replica 127.0.0.1:20004 join and added to pool
[Replica Provisioner] Thread created for replica 127.0.0.1:20004
[Primary 1] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica 1] [Simbot] Simbot Initiated
[Replica 1] [Conn. ] Connection initiated SubnetIndex(0)
[Replica 1] [Conn. ] Connection initiated SubnetIndex(1)
[Replica 1] [Mail. ] Received Message: RegisterNewConnections
[Replica 1] [Simbot] Simulated cycle SimulatedClock(9)
[Primary 1] [MsgRcv] Connection initiated SubnetIndex(-2)
[Primary 0] [MsgRcv] Connection initiated SubnetIndex(-2)
[Primary 0] [Wait. ] Unblocked Successfully
[Primary 1] [Wait. ] Unblocked Successfully
```

Figura E.2: Traza de Recuperación de Fallo de Réplica

Lista de mensajes de depuración enviados por los *simbots* involucrados con la recuperación del fallo de la réplica simulando la subred 1

```
[Primary 2] [MsgRcv] Connection closed: SubnetIndex(0)
[Primary 2] [MsgRcv] Crash Detected SubnetIndex(0)
[Primary 1] [MsgRcv] Connection closed: SubnetIndex(0)
[Primary 1] [MsgRcv] Crash Detected SubnetIndex(0)
[Primary 2] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(0))
[Primary 1] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(0))
[Primary 1] [Wait. ] Wait Requested for {SubnetIndex(0)}
[Primary 2] [Wait. ] Wait Requested for {SubnetIndex(0)}
[Replica 0] [MsgRcv] Connection closed: SubnetIndex(0)
[Replica 0] [MsgRcv] Crash Detected SubnetIndex(0)
[Replica 2] [MsgRcv] Connection closed: SubnetIndex(0)
[Replica 2] [MsgRcv] Crash Detected SubnetIndex(0)
[Replica 2] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(0))
[Replica 1] [MsgRcv] Connection closed: SubnetIndex(0)
[Replica 1] [MsgRcv] Crash Detected SubnetIndex(0)
[Replica 2] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica 1] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(0))
[Replica 1] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica 0] [FTMan.] Crash of Type PrimaryCrash
[Replica 0] [Simbot] Initiating Primary Promotion Procedure
[Primary 0] [Simbot] Promoted to Primary
[Primary 0] [Simbot] Connection updated: SubnetIndex(1)
[Primary 0] [Simbot] Connection updated: SubnetIndex(2)
[Primary 0] [Conn. ] Connection initiated: SubnetIndex(-3)
[Primary 0] [Conn. ] Connection initiated: SubnetIndex(-2)
[Primary 1] [Wait. ] Unblocked Successfully
[Primary 2] [Wait. ] Unblocked Successfully
```

Figura E.3: Traza de Recuperación de Fallo de Primario

Lista de mensajes de depuración enviados por los *simbots* involucrados con la recuperación del fallo del primario simulando la subred 0

```
[Primary 2] [MsgRcv] Crash Detected SubnetIndex(-1)
[Primary 0] [MsgRcv] Crash Detected SubnetIndex(-1)
[Primary 0] [MsgRcv] Connection closed: SubnetIndex(-1)
[Primary 2] [MsgRcv] Connection closed: SubnetIndex(-1)
[Primary 0] [FTMan.] Crash of Type ReplicaCrash
[Primary 2] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(-1))
[Primary 2] [Wait. ] Wait Requested for {SubnetIndex(0)}
[Primary 2] [Wait. ] Unblocked Successfully
[Primary 2] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica Provisioner] Replica Child Process for 127.0.0.1:20006 Joined.
[Replica Provisioner] Replica 127.0.0.1:20006 join and added to pool
[Replica Provisioner] Thread created for replica 127.0.0.1:20006
[Primary 1] [MsgRcv] Crash Detected SubnetIndex(-1)
[Primary 1] [MsgRcv] Connection closed: SubnetIndex(-1)
[Primary 1] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(-1))
[Primary 1] [Wait. ] Wait Requested for {SubnetIndex(0)}
[Primary 1] [Wait. ] Unblocked Successfully
[Primary 1] [Wait. ] Wait Requested for ALL NEIGHBORS
[Primary 0] [Simbot] Requesting Replica Creation
[Primary 0] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica 0] [Simbot] First connection to Debug Server
[Replica 0] [Conn. ] Connection initiated: SubnetIndex(1)
[Replica 0] [Conn. ] Connection initiated: SubnetIndex(2)
[Replica 0] [Conn. ] Connection initiated: SubnetIndex(0)
[Primary 1] [Wait. ] Wait Unblock: Sending ConnectionsReady to unblock
[Replica 0] [Simbot] Initiating Simulation Period
[Primary 2] [Wait. ] Wait Unblock: Sending ConnectionsReady to unblock
[Primary 1] [MsgRcv] Connection initiated: SubnetIndex(-1)
[Primary 2] [MsgRcv] Connection initiated: SubnetIndex(-1)
[Primary 1] [Wait. ] Unblocked Successfully
[Primary 2] [Wait. ] Unblocked Successfully
[Primary 0] [Wait. ] Wait Unblock: Sending ConnectionsReady to unblock
[Primary 0] [MsgRcv] Connection initiated: SubnetIndex(-1)
[Primary 0] [Wait. ] Unblocked Successfully
[Primary 0] [MsgRcv] Crash Detected SubnetIndex(-2)
[Primary 1] [MsgRcv] Crash Detected SubnetIndex(-2)
[Replica Provisioner] Replica Child Process for 127.0.0.1:20004 Joined.
[Replica Provisioner] Replica 127.0.0.1:20004 join and added to pool
[Primary 1] [MsgRcv] Connection closed: SubnetIndex(-2)
[Replica Provisioner] Thread created for replica 127.0.0.1:20004
[Primary 1] [FTMan.] Crash of Type ReplicaCrash
[Primary 0] [MsgRcv] Connection closed: SubnetIndex(-2)
[Primary 0] [FTMan.] Crash of Type SuccessorCrash(SubnetIndex(-2))
[Primary 0] [Wait. ] Wait Requested for {SubnetIndex(1)}
[Primary 0] [Wait. ] Unblocked Successfully
[Primary 0] [Wait. ] Wait Requested for ALL NEIGHBORS
[Primary 1] [Simbot] Requesting Replica Creation
[Primary 1] [Wait. ] Wait Requested for ALL NEIGHBORS
[Replica 1] [Simbot] First connection to Debug Server
[Replica 1] [Conn. ] Connection initiated: SubnetIndex(0)
[Replica 1] [Conn. ] Connection initiated: SubnetIndex(1)
[Replica 1] [Simbot] Initiating Simulation Period
[Primary 1] [Wait. ] Wait Unblock: Sending ConnectionsReady to unblock
[Primary 0] [Wait. ] Wait Unblock: Sending ConnectionsReady to unblock
[Primary 0] [MsgRcv] Connection initiated: SubnetIndex(-2)
[Primary 1] [MsgRcv] Connection initiated: SubnetIndex(-2)
[Primary 0] [Wait. ] Unblocked Successfully
[Primary 1] [Wait. ] Unblocked Successfully
```

Figura E.4: Traza de Notificación de Fallo en Dos Subredes

Lista de mensajes de depuración enviados por los *simbots* involucrados con la recuperación de dos fallos simultáneos en 2 subredes

```
[Primary 1 ] [MsgRcv] Connection closed: SubnetIndex(0)
[Primary 1 ] [MsgRcv] Crash Detected SubnetIndex(0)
[Primary 1 ] [MsgRcv] Connection closed: SubnetIndex(-1)
[Primary 1 ] [MsgRcv] Crash Detected SubnetIndex(-1)
[Primary 1 ] [FTMan.] Crash of Type FatalCrash(SubnetIndex(-1))
[Primary 1 ] [Mail. ] Received: FatalCrash (SubnetIndex(0))
[Replica 2 ] [Mail. ] Received: FatalCrash (SubnetIndex(0))
[Primary 1 ] [Mail. ] Handling Crash: FatalCrash(SubnetIndex(0))
[Primary 2 ] [Mail. ] Received: FatalCrash (SubnetIndex(0))
[Primary 2 ] [Mail. ] Handling Crash: FatalCrash(SubnetIndex(0))
[Replica 1 ] [Mail. ] Received: FatalCrash (SubnetIndex(0))
[Replica 1 ] [Mail. ] Handling Crash: FatalCrash(SubnetIndex(0))
[Primary 2 ] [Simbot] Exiting simulation. Fatal Error Ocurred
[Replica 2 ] [Mail. ] Handling Crash: FatalCrash(SubnetIndex(0))
[Replica 1 ] [Simbot] Exiting simulation. Fatal Error Ocurred
[Primary 1 ] [Simbot] Exiting simulation. Fatal Error Ocurred
[Replica 2 ] [Simbot] Exiting simulation. Fatal Error Ocurred
```

Figura E.5: Traza de Notificación de Fallo Fatal

Lista de mensajes de depuración enviados por los *simbots* al detectar un falo fatal en la subred 0

```
SUBNET 0 : TIME:  0  -->   TRANSITION:  0
SUBNET 0 : TIME:  2  -->   TRANSITION:  1
SUBNET 0 : TIME:  3  -->   TRANSITION:  0
SUBNET 0 : TIME:  5  -->   TRANSITION:  1
SUBNET 0 : TIME:  6  -->   TRANSITION:  0
SUBNET 0 : TIME:  8  -->   TRANSITION:  1
SUBNET 0 : TIME:  9  -->   TRANSITION:  0
SUBNET 0 : TIME:  11  -->   TRANSITION:  1
SUBNET 0 : TIME:  12  -->   TRANSITION:  0
SUBNET 0 : TIME:  14  -->   TRANSITION:  1
SUBNET 0 : TIME:  15  -->   TRANSITION:  0
SUBNET 0 : TIME:  17  -->   TRANSITION:  1
SUBNET 0 : TIME:  18  -->   TRANSITION:  0
SUBNET 0 : TIME:  20  -->   TRANSITION:  1
SUBNET 1 : TIME:  1  -->   TRANSITION:  0
SUBNET 1 : TIME:  4  -->   TRANSITION:  0
SUBNET 1 : TIME:  7  -->   TRANSITION:  0
SUBNET 1 : TIME:  10  -->   TRANSITION:  0
SUBNET 1 : TIME:  13  -->   TRANSITION:  0
SUBNET 1 : TIME:  16  -->   TRANSITION:  0
SUBNET 1 : TIME:  19  -->   TRANSITION:  0
SUBNET 2 : TIME:  1  -->   TRANSITION:  0
SUBNET 2 : TIME:  4  -->   TRANSITION:  0
SUBNET 2 : TIME:  7  -->   TRANSITION:  0
SUBNET 2 : TIME:  10  -->   TRANSITION:  0
SUBNET 2 : TIME:  13  -->   TRANSITION:  0
SUBNET 2 : TIME:  16  -->   TRANSITION:  0
SUBNET 2 : TIME:  19  -->   TRANSITION:  0
```

Figura E.6: Traza de Transiciones Disparadas

Lista de transiciones disparadas por la simulación de un modelo con un ciclo final de 20

```
----------------- RESULTS for 1 -------------------
SUBNET : 1  --> NUMBER OF FIRED TRANSITIONS: 50
SUBNET : 1  --> SIMULATED TIME in cicles: 150
SUBNET : 1  --> SIMULATION REAL TIME: 3.570704969 seconds
SUBNET : 1  --> EVENTS PER SECOND 14.002837096340345
---------- EXTENDED SIMULATION METRICS++ ----------
No. Cycles:                 150
Simulation Duration:        3.57070 s
Cycle Duration Avg.:        0.45766 ms
Sim Step Duration Avg.:     0.00151 ms    (0.01%)
Send Duration Avg.:         0.08792 ms    (0.43%)
 |- Lock Wait Avg:           0.00028 ms    (0.36%)
 |- Register Avg:            0.00015 ms    (0.20%)
 |- Connection Avg:          0.08598 ms    (98.58%)
Receive Duration Avg.:      0.12021 ms    (12.62%)
Send Error Count:           0
Receive Error Count:        1
Fatal Error Count:          0
Recovery Duration:          0.42178 s
 |- Sync                    0.00154 s
 |- Replica Creation        0.41980 s
 |- Promotion               0.00000 ms
-------------------------------------------------
```

Figura E.7: Resultados de Simulación Mostrando Tiempo de Recuperación

Resultados finales de simulación de un fallo de la réplica de la subred 1. Los resultados son los del primario de la subred 1.

Anexo F

Propuestas de Continuación

Durante el desarrollo del estudio, diseño e implementación de la tolerancia a fallos en el simulador, se han detectado múltiples puntos de mejora y continuación del proyecto. Este anexo describe las propuestas en detalle con la esperanza de que este proyecto se tome como punto de partida de trabajos futuros.

F.1. Múltiples Réplicas

Una cuestión básica en el diseño de la solución de tolerancia a fallos ha sido el nivel de replicación utilizado (véase Sección 4.1). Debido a que el proyecto se trataba de un primer prototipo donde se primaba la sencillez del diseño y cumplir los objetivos propuestos, y que el trabajo ha requerido un gran esfuerzo adicional al planificado, se ha optado por una única réplica.

El aumento del nivel de replicación no es trivial, pero durante el diseño de la replicación se ha tenido en cuenta esta propuesta y se ha intentado crear una base que se pueda aprovechar en futuras extensiones de la tolerancia a fallos. Una replicación con N-copias del *simbot* permite tolerar $N - 1$ fallos en cualquier subred simulada.

El diseño se debe modificar incluyendo el envío de los eventos a todas las réplicas de los sucesores, al igual que el envío, por parte del primario, de las confirmaciones a todas sus réplicas. Otras alternativas pueden cambiar el modo de distribución de los eventos y confirmaciones para liberar al primario de la responsabilidad de realizar tantos envíos y ocupar a las réplicas con esa tarea. Los mecanismos implementados en la actualidad deberían satisfacer los requisitos de consistencia actuales, pero se recomienda una revisión más exhaustiva.

Los procedimientos para recuperar un fallo del primario se deben adaptar ante el aumento de las réplicas. En concreto, se requiere un algoritmo de elección de líder para seleccionar que réplica es promocionada a primario tras el fallo. Una solución puede ser la sincronización de las réplicas para seleccionar cuál de ellas tiene un estado más avanzado y que esta sea la que promocione a primario. Además, el resto de réplicas deben actualizar el registro del estado con el de este nuevo primario para conservar la consistencia en adelante.

F.2. Uso de *Multicast*

El envío del mismo evento junto a la confirmación por parte de *simbot* primario a sus sucesores y réplica se realiza de manera secuencial en todas las conexiones TCP. Esta operativa triplica el tiempo de envío de un evento, lo que el uso de múltiples réplicas añadiría mucha latencia a la simulación. El uso de un *multicast* puede reducir considerablemente, incluso hacer transparente, la diferencia de tiempo de ejecución de la simulación sencilla frente a la tolerante, ya que esta diferencia de tiempo se debe en mayor parte al incremento del número de envíos al final del ciclo de simulación.

El uso del protocolo TCP se fundamenta en su fiabilidad para enviar mensajes. Pese a ello, no es el más eficiente para enviar mensajes a múltiples receptores. El protocolo IP tiene una funcionalidad *multicast* que permite que envíos únicos sean difundidos a múltiples receptores. IP no proporciona la fiabilidad de TCP. Por ello, se debería usar un protocolo *Multicast* fiable para enviar los eventos a los sucesores y sus réplicas.

F.3. Divergencia de Primario y Réplicas

En el diseño actual la ejecución de la simulación del primario y réplica puede divergir sin ningún límite (véase Sección 4.1.1), es decir, si un *simbot* no avanza, el otro puede seguir simulando sin ninguna restricción. En entornos *cloud* y en simulaciones grandes, donde dos máquinas pueden tener un rendimiento muy diferente, dos *simbots* pueden ir muy desacoplados produciendo que, en el caso de que la réplica vaya muy avanzada, su registro de consistencia se desborde en memoria produciendo un error no deseado.

Una solución puede basarse en el cálculo dinámico de la divergencia entre los *simbots* primario y réplica. Si el primario se ralentiza demasiado, la réplica puede parar temporalmente su ejecución. Como una propuesta adicional, se puede considerar que si la réplica se ralentiza y el tiempo que tardaría el primario en pedir una nueva réplica y copiarla su estado actual es menor al que le costaría a la réplica en alcanzar al primario, cancelar la réplica desacoplada e introducir una nueva.

F.4. Aprovechamiento de Espera a Réplica Nueva

La operativa actual de recuperación ante el fallo de un *simbot* réplica sincroniza los primarios con los predecesores y luego realiza la petición de una nueva réplica (véase Sección 4.1.2). Dependiendo del entorno de ejecución de la simulación el aprovisionamiento de un nuevo *simbot* puede tardar desde segundos a minutos. Durante este periodo primarios y predecesores pueden seguir simulando, ya que la transmisión del estado actual no se realiza hasta que el nuevo *simbot* esta disponible. La implementación de estos mecanismos require la notificación asíncrona al primario y a los predecesores de que la nueva réplica ya está disponible, lo que añade gran complejidad al diseño y flujo de ejecución del *simbot*.

Una solución alternativa que puede mitigar el problema se basa en que el *Replica Provisioner* aprovisione ciertas máquinas en adelanto y estas sean configuradas como réplicas en cuanto se realice una petición, reduciendo el tiempo que tarda en tener un *simbot* disponible. Esta solución aumenta el coste de la simulación, ya que se deben tener máquinas adicionales inutilizadas durante todo momento.

F.5. Estudio de Confirmaciones por Ciclo

La invalidación de mensajes duplicados como consecuencia del modelo de consistencia conlleva almacenar los mensajes recibidos (véase Sección 4.1.1). Si cada confirmación es enviada tras un único envío a los sucesores, solo es necesario utilizar un registro para almacenar el último mensaje de cada predecesor. Por el contrario, si las confirmaciones se envían cada N envíos a los sucesores, se necesitan almacenar los N últimos mensajes.

Existe un compromiso entre cuantos mensajes almacenar por cada predecesor, incrementando el uso de memoria, frente a la frecuencia con la que se envían las confirmaciones. Se ha optado por enviar confirmaciones tras cada envío por simplicidad de la operativa, pero el envío de confirmaciones menos frecuentes puede descongestionar la red de comunicación en ciertas simulaciones. Una exploración más detallada de la selección del número de ciclos por confirmación en un entorno más propenso a fallos que el supuesto, puede resultar en un aumento de las prestaciones de la simulación.

F.6. Optimización de Memoria Compartida

Tras el análisis de los resultados sobre el tiempo de ejecución del *simbot* se concluye que la fase de envío de eventos comprende una gran parte de este periodo. Se asume que la fase de envío de eventos del *Simulation Engine* se prolonga durante más tiempo debido a los bloqueos y sincronización necesarios para acceder a la estructura de datos compartida *Connection Manager*. Esta estructura se envuelve en un `std::sync::RwLock` (*read/write lock*) para permitir múltiples lectores y un único escritor accediendo a los datos. El uso de *lock*, junto a la referencia `Arc`, es necesario debido a la restricción de Rust de que todo dato que se accede de manera concurrente debe implementar los *trait* `Send` y `Sync`. En el caso del *simbot* los datos compartidos del *Connection Manager* son las conexiones (`TcpStreams`), las cuales permiten lectura y escritura simultánea, ya que son conexiones TCP. Además, el `RwLock` bloquea la estructura de datos completa, a diferencia de las conexiones individuales que son requeridas al realizar envíos y recepciones de mensajes.

Las restricciones de Rust impiden el acceso más rápido y sin bloqueos innecesarios a las conexiones individuales durante la simulación durante la espera a que los datos compartidos se liberen. Se recomienda encarecidamente la manipulación de los mecanismos de bloqueo en acceso a estructuras de datos compartidas. Las opciones exploradas incluyen la implementación insegura (término de Rust, *unsafe*) de los *trait* `Send` y `Sync` para el *Connection Manager*, lo que evita la necesidad de obtener el *lock* para acceder a los datos. Además, se debe implementar un *lock* propio (probablemente un `std::sync::Mutex`) que bloquee la estructura cuando se estén realizando modificaciones que sí que son mutables, por ejemplo, añadir o eliminar una conexión.

F.7. Detección de Fallos Activa

La detección de fallos implementada en el *simbot* se basa en la notificación del cierre de la conexión TCP por parte del *Network Message Receiver*. Debido a que el *simbot* no realiza ninguna acción, sino que monitoriza la red de comunicación para detectar el fallo de otro *simbot*, esta detección se puede denominar pasiva. Pese a que el uso de la detección pasiva no se ha traducido en largos intervalos de tiempo entre el fallo y su detección, la detección activa puede aumentar la precisión de la detección y reducir el tiempo de recuperación. Por ejemplo, el uso de latidos periódicos entre *simbots* puede permitir una reducción del tiempo de detección a cambio de los inconvenientes de otro *thread* que se ocupa de esta tarea.

F.8. Tolerancia a Fallos de Servicios Auxiliares

La tolerancia a fallos de los servicios auxiliares (el *Debug Server* y el *Replica Provisioner*) es trivial, ya que estos servicios no tienen estado o este puede ser almacenarse en memoria persistente sin suponer un coste elevado. La replicación de estos servicios puede resultar de gran utilidad en entornos de producción donde la tolerancia a fallos y la alta disponibilidad de la solución completa sea imprescindible.

Anexo G

Planificación y Esfuerzos Dedicados

El proyecto se ha realizado durante el año académico 2022-23, con su finalización en Junio de 2023. Inicialmente, se ha realizado la introducción a la simulación distribuida, al proyecto de investigación y al lenguaje de programación Rust. A continuación, se ha desarrollado el análisis y diseño iniciales de la tolerancia a fallos. Tras obtener un diseño refinado, se ha realizado el diseño del *simbot*, su implementación y la integración de los mecanismos de tolerancia a fallos. Por último, se han realizado los experimentos y analizado los resultados del proyecto. La memoria se ha desarrollado de manera continua durante la duración del proyecto.

Durante el desarrollo del proyecto nos encontramos con un problema imprevisto: la arquitectura e implementación base del *simbot* era incapaz de soportar la integración de los mecanismos de tolerancia a fallos. Este contratiempo ha demorado la finalización del proyecto, ya que ha requerido la el rediseño e implementación del *simbot* tras el diseño de la tolerancia a fallos. Además, el proyecto se ha realizado de manera simultánea al desarrollo de la actividad laboral y a otro proyecto de investigación, lo que ha disminuido en mayor medida la disponibilidad de tiempo a dedicar al proyecto.

Pese a la disponibilidad reducida, todo el tiempo restante ha sido dedicado al trabajo con un alto grado de compromiso y motivación. Las reuniones con los directores se han realizado con regularidad, semanalmente, y en ellas se presentaba el trabajo realizado, se aclaraban dudas y se proponían nuevos hitos para la semana. El control de horas aproximadas dedicadas al proyecto se muestra en el Cuadro G.1. Además, el Cuadro G.2 muestra un diagrama de Gantt del proyecto.

Tarea Desarrollada	**Tiempo** (horas)
Revisión de la bibliografía	10
Introducción al lenguaje Rust	15
Introducción al código del simulador	20
Análisis de fallos y diseño de la solución	50
Diseño, implementación y depuración del *simbot*	110
Implementación de tolerancia a fallos en el *simbot*	50
Experimentación	75
Reuniones	45
Redacción de la memoria	60
Horas totales	**435 horas**

Cuadro G.1: Control de Horas Dedicadas

TAREA	JULIO-AGOSTO				SEPTIEMBRE-OCTUBRE				NOVIEMBRE-DICIEMBRE				ENERO-FEBRERO				MARZO-ABRIL				MAYO-JUNIO			
	1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8	1-2	3-4	5-6	7-8
Introducción a Rust																								
Revisión de Bibliografía																								
Introducción al Simulador																								
Análisis de Fallos																								
Diseño de Tolerancia a Fallos																								
Implementación del Simbot																								
Depuración del Simbot																								
Implementación de Tolerancia a Fallos																								
Depuración de Toleranica a Fallos																								
Experimentación																								
Redacción de Memoria																								
Reuniones																								

Cuadro G.2: Diagrama de Gantt del Proyecto