Universidad Zaragoza

Trabajo Fin de Grado

Diseño e Implementación de un Simulador Distribuido de Eventos Discretos con Mecanismos de Balanceo de Carga

Design and Implementation of a Distributed Discrete Event Simulator with Load Balancing Mechanisms

Autor

Álvaro Santamaría de la Fuente

Directores

Unai Arronategui Arribalzaga

José Manuel Colom Piazuelo

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2021

AGRADECIMIENTOS

Agradezco en primer lugar, a los directores de este trabajo Unai Arronategui y José Manuel Colom y al profesor José Ángel Bañares por haberme introducido en el campo de la simulación distribuida y haber confiado en mí para la realización de este importante proyecto. Agradezco que siempre se hayan preocupado en todo momento por la evolución del trabajo y me hayan ayudado de manera inestimable para poder concluir y presentar este trabajo. Su labor docente es de un valor incalculable y se merece este reconocimiento.

En segundo lugar, también quiero agradecer a mi pareja, amigos y compañeros del grado de Ingeniería Informática, por el apoyo moral y ayudarme en muchos momentos a desconectar de la carga de trabajo diaria y poder recargar pilas para poder concluir este trabajo.

Y por último, agradecer a mi familia por su apoyo y haberme animado en los momentos más duros, no solo de este trabajo, sino también de los estudios del grado de Ingeniería Informática que, con la entrega de este documento y su posterior defensa, habré finalizado de manera satisfactoria. ¡Gracias!

RESUMEN

El análisis y comprensión del comportamiento de sistemas en el ámbito de la ingeniería ha cobrado una vital importancia en los últimos tiempos con el aumento exponencial de su tamaño y complejidad. Debido a ello, se hace muy complicado el hecho de crear prototipos o clones de un sistema para analizar su comportamiento o probar nuevas mejoras del mismo.

Para dar solución a este problema, la simulación de sistemas permite reproducir el comportamiento de un sistema dinámico mediante la interpretación de un modelo que representa de manera fidedigna su funcionamiento. En este trabajo, concretamente, se va a tratar con **sistemas de eventos discretos**, aquellos cuyo estado de funcionamiento se ve alterado en instantes puntuales de tiempo, pudiendo identificar estos instantes de una manera clara y diferenciada.

Sin embargo, para poder simular sistemas de gran tamaño y complejidad, se requiere que esta simulación y las herramientas que lo soportan sean **escalables**. Para ello, se ha abordado el desarrollo e implementación de un simulador distribuido que reproduce el comportamiento de un sistema de eventos discretos modelado por **Redes de Petri**, y además, se han incorporado mecanismos que sean capaces de redistribuir la carga de trabajo que soporta cada uno de los nodos que interviene en la simulación. Estos mecanismos son necesarios porque los nodos pueden sufrir variaciones de la **carga de trabajo** que soportan a lo largo del tiempo al tratarse de una simulación distribuida donde las particiones del modelo pueden no ser equitativas, y el reloj virtual no avance de manera coordinada. Estos mecanismos de balanceo de la carga son el aspecto innovador que aporta este trabajo al campo de la simulación distribuida conservativa, ya que la solución propuesta de redistribuir la partición de modelo que recibe un nodo de simulación durante el proceso de simulación **sin llegar a detenerla** es novedosa, a pesar de que en otros trabajos anteriores se hayan abordado soluciones respecto al balanceo de la carga en simulaciones distribuidas.

En conclusión, el simulador distribuido con mecanismos de balanceo de carga es una primera versión de un entorno de simulación capaz de realizar simulaciones de modelos de gran tamaño y complejidad sobre una arquitectura heterogénea y cuyo aspecto principal es la alta escalabilidad, dando lugar a una herramienta que facilite el análisis de los sistemas que tienen una gran relevancia en nuestra sociedad.

ACRÓNIMOS

LP: Proceso Lógico *(Logical Process)*
LEFs: *Linear Enabling Functions*
RdP: Redes de Petri
FIFO: *First-In-First-Out*
VLT: *Virtual Local Time*
LVHT: *Logical Virtual Horizontal Time*

Índice

Capítulo 1

Introducción y objetivos

1.1. Motivación y objetivos

En la actualidad, los sistemas de producción, de telecomunicaciones o logísticos tienen una alta complejidad, siendo en muchas situaciones, difíciles de comprender y de predecir cuál va a ser su comportamiento. Para ello, los sistemas se describen mediante modelos de comportamiento del sistema. Gracias a estos modelos complejos se puede simular el sistema permitiendo analizar y entender cómo se va a comportar y cuál es su naturaleza. Por todo esto, la simulación de estos modelos complejos ha cobrado una vital importancia en el mundo de la ingeniería.

Este trabajo parte de la iniciativa de desarrollar e implementar un simulador de eventos discretos, el cual se va a basar en Redes de Petri como modelo formal. La meta final de la herramienta es tener la capacidad de simular modelos de eventos discretos de gran tamaño, por lo que se va a desarrollar un simulador distribuido, en el que el modelo va a ser dividido en subredes con la intención de aprovechar al máximo la capacidad computacional de múltiples unidades de proceso al ejecutar distintos procesos de manera concurrente. Esta división del modelo va a permitir repartir la carga entre los nodos de ejecución. En simulaciones distribuidas, el grado de acoplamiento de las simulaciones distribuidas hace que el ritmo de la simulación más lenta se imponga en todos los nodos. Incluso si se consigue la mejor división inicial del modelo en subredes, la evolución de la simulación requerirá ritmos de procesamiento distintos a lo largo del tiempo para que todos los nodos avancen a la vez el tiempo simulado (reloj virtual). Por ello, los nodos pueden verse sometidos a cargas de trabajo que evolucionan con el tiempo y que interfieren con el rendimiento de las simulaciones.

El objetivo principal de este trabajo es incorporar mecanismos que sean capaces de balancear la carga que soporta un nodo, para que en todo momento su capacidad computacional sea aprovechada al máximo y se consiga un **sistema elástico**. Un

sistema elástico es capaz de operar ante un aumento significativo de la carga de trabajo, sin ver mermada su velocidad de ejecución. Esta necesidad de realizar un balanceo de carga ya ha sido abordada en trabajos previos de simulación distribuida. Sin embargo, este trabajo aporta una solución innovadora para simulaciones distribuidas conservativas, permitiendo realizar este balanceo de manera eficiente **sin necesidad de detener la simulación** de los nodos. El hecho de aportar algo innovador en este área ha resultado ser la mayor motivación para la consecución de este trabajo.

1.2. Aproximación - Punto de partida

El contexto y antecedentes de este trabajo se sitúa en los artículos de investigación presentados por los profesores Unai Arronategui, José Ángel Bañares y José Manuel Colom de la Universidad de Zaragoza[1][2][3]. En estas publicaciones se propone una metodología dirigida por el modelo para la simulación distribuida de eventos discretos basada en Redes de Petri. Se propone un lenguaje jerárquico o basado en componentes de modelado, el proceso de elaboración de Redes de Petri sin jerarquía y un compilador que genera código eficiente para la simulación de RdP. Además, en el Trabajo de Fin de Grado[4] de Sergio Herrero Barco de la Universidad de Zaragoza se realiza una primera implementación del compilador y los servicios básicos para la simulación distribuida en Java. Una vez situados los antecedentes, este trabajo consiste en el desarrollo e implementación de un simulador distribuido en el lenguaje de programación Rust que, además, incorpore mecanismos de balanceo de carga para una simulación conservativa distribuida, lo cual es realmente innovador.

Además, cabe destacar que este trabajo ha sido realizado en paralelo al Trabajo de Fin de Grado de mi compañero Hayk Kocharyan[5], y el conjunto de ambos supone el desarrollo de un sistema inicial que consiste en un entorno de simulación con arquitectura heterogénea, de alta escalabilidad y elasticidad.

1.3. Lenguaje, Herramientas y Entorno de trabajo

El simulador distribuido ha sido implementado utilizando el lenguaje de programación Rust[6]. Se ha seleccionado este lenguaje de programación debido a que puede obtener altas prestaciones de manera similar a otros lenguajes como C++[7] pero con un modelo mucho más seguro de gestión de memoria y concurrencia, ideal para el trabajo planteado. El entorno de desarrollo ha sido VSCode[8] con la extensión para Rust[9]. Para la edición de *scripts* se ha utilizado el lenguaje Ruby por su potente librería para la manipulación de ficheros de extensión json, ideal para la tarea de

adaptación del modelo compilado obtenido del compilador en Java (véase sección 4.1 para más información).
El Sistema Operativo de la máquina donde se desarrolló el proyecto fue Ubuntu 18.04 - Linux.

Para el control de versiones del proyecto se creó una organización en Github llamada *simbots-swarm*, donde se crearon distintos repositorios para mantener separadas distintas versiones del simulador, por ejemplo, una versión básica distribuida o la versión completa con balanceo de carga.

Las características de las máquinas donde se llevaron a cabo las pruebas experimentales se describen en la sección 5.2

1.4. Planificación del proyecto

Tareas	**Horas Aproximadas**
Revisión de la bibliografía publicada y TFG previos	5h
Familiarización con el lenguaje Rust	15h
Estudio y análisis del código del simulador centralizado en Rust	10h
Realización de scripts Ruby para la traducción del modelo en ficheros JSON	15h
Creación de las estructuras necesarias para la simulación distribuida	15h
Diseño e implementación de los mecanismos de sincronización entre nodos participantes en la simulación distribuida	25h
Pruebas experimentales del simulador distribuido y selección de los casos de prueba	40h
Desarrollo del algoritmo de movimiento de LEFs entre nodos	6h
Implementación del mecanismo de balanceo de carga e integración con el simulador distribuido	20h
Pruebas experimentales de los mecanismos de balanceo de carga y correción de errores	50h
Reuniones semanales con directores del TFG	40h
Redacción de esta memoria	80h
Total horas implementación y pruebas	**201 h**
Total horas	**321 h**

Capítulo 2

Definición y descripción del sistema

En esta sección se van a recordar brevemente los conceptos y terminología básica que va a ser utilizada a largo de esta memoria. Los términos definidos a continuación son la base sobre la que se sustenta este proyecto y es necesario dejar claro su significado en el contexto de este trabajo.

2.1. Simulación distribuida de eventos discretos

La **simulación** en términos de computación, consiste en la ejecución de un programa computacional (simulador) a través del cual se trata de emular el comportamiento de un sistema construyendo la secuencia de estados que sigue a lo largo del tiempo. Todos estos programas de simulación de un sistema asumen la existencia implícita o explícita de un modelo del sistema, el cual es una abstracción que retiene aquellos aspectos, elementos y relaciones que quieren ser estudiadas en su evolución temporal. Existen dos tipos de simulaciones atendiendo a las características de las variables de estado y la forma en que cambian sus valores en el tiempo:

- **Simulación de tiempo continuo**: Los cambios de estado del sistema son continuos en el tiempo. Por ejemplo, es el caso de los sistemas cuyo comportamiento está modelado por ecuaciones diferenciales

- **Simulación de tiempo discreto**: Los cambios de estado del sistema ocurren en instantes discretos de tiempo. El tiempo puede avanzar de manera regular con incrementos fijos de tiempo o de manera irregular mediante una simulación dirigida por eventos (la que se va a utilizar en este trabajo)

En este trabajo, se considera el caso de simulación de sistemas de eventos discretos. Un evento se podría definir como un suceso ubicado en instantes identificables del tiempo y que provoca la modificación de las variables que definen el estado actual

de un sistema. El comportamiento del sistema en este trabajo va a ser modelado formalmente por **redes de Petri**.

Una simulación distribuida es aquella que se ejecuta sobre computadores que no comparten reloj y se encuentran conectados a través de una red de comunicación de datos y cada uno de los computadores se encarga de simular una parte del modelo, en nuestro caso, una subred. Para la simulación consistente del modelo global, se exige que los simuladores alojados en cada uno de los computadores actúe de una manera sincronizada y coordinada, manteniendo la causalidad de las acciones independientemente del computador que las gestione. Estos entornos pueden ser un *clúster* de computadores o computadores conectados en *cloud*.

La elección de llevar a cabo una simulación distribuida en lugar de centralizada, se debe a la imposibilidad de que en una sola máquina se puedan simular modelos de grandes dimensiones debido a los grandes requerimientos de tamaño en memoria necesarios para gestionar el estado de simulación. Además esto conlleva que las prestaciones del simulador sean escalables, es decir, a más particiones del modelo y más nodos distribuidos dispongas, mayores serán las prestaciones alcanzadas para modelos grandes. Para poder llevar a cabo esta simulación distribuida, es necesario que el modelo que representa el comportamiento del sistema, sea particionado en submodelos, los cuales serán simulados de manera coordinada entre todos los nodos, ya que la ejecución de un submodelo depende de la propagación de eventos tanto del propio submodelo como del resto de particiones del modelo completo.

Esta tarea de coordinación se va a realizar mediante el intercambio de eventos a través de la red de comunicación sobre la que se conectan cada uno de los nodos encargados de realizar la simulación distribuida. Los eventos se sitúan en el tiempo y por tanto, un atributo esencial de cada evento es la etiqueta temporal que lo sitúa en la línea del tiempo. Esto conlleva la realización de dos tareas esenciales en el proceso de simulación en relación con los eventos: (1) La **ordenación de los eventos** según sus etiquetas temporales y que definirán el orden en el que serán tratados y el orden en que producirán efectos; (2) La utilización de las etiquetas de los eventos para la **determinación local del tiempo global** de la simulación del sistema, es decir, lo que se denomina sincronización de los relojes locales de simulación para inferir el tiempo global en el que se encuentra la simulación del sistema. Esta sincronización de relojes locales es fundamental ya que su valor es el que permite etiquetar los eventos nuevos generados y que deben ser enviados a los distintos computadores que contienen partes del modelo que está siendo simulado de manera distribuida.

Debido al carácter distribuido del simulador, estos eventos pueden haberse producido internamente en el mismo submodelo, o proceder de otra partición, es decir, eventos externos. Todos estos eventos se van almacenando en la lista de eventos pendientes (LEP), siendo esta lista una cola con prioridad en la que el orden viene marcado por la estampilla temporal asociada al evento. El tiempo de simulación va avanzando hasta el tiempo del evento que encabeza la LEP (el de menor estampilla), y llegado a ese instante de tiempo, se procesa el evento siendo capaz de generar nuevos eventos locales o eventos externos que son enviados a su correspondiente nodo de simulación.

Debido a que se utiliza la red de comunicación, los mensajes con eventos puedan llegar desordenados o a destiempo. Sin embargo, el entorno de simulación distribuida debe garantizar en todo momento la restricción de causalidad local para que el flujo de ejecución de la simulación sea el correcto y equivalente al de una simulación centralizada en un solo nodo. La restricción de causalidad local indica que los eventos contenidos en los mensajes entrantes a cada nodo de simulación deben ser procesados en el orden correcto de estampillas de tiempo globales de eventos. Para ello, existen dos estrategias a seguir:

- **Sincronización conservativa**: Se espera hasta que sea seguro que no van a llegar eventos de tiempo menor evitando así violar la restricción de causalidad local.

- **Sincronización optimista**: El tiempo se avanza sin tener la seguridad de que no se haya violado la restricción de causalidad. Dichas violaciones se detectan en el momento que llega un evento con etiqueta temporal cuyo valor es menor que el tiempo local de simulación (valor del reloj local). Si se detecta un evento fuera de orden durante la ejecución, se recupera el estado del sistema hasta ese evento mediante un mecanismo de *rollback*.

Debido a que una de las partes centrales de este trabajo se basa en el diseño e implementación de mecanismos de balanceo de carga, se ha decidido optar por el algoritmo de sincronización conservativo, cuya implementación será explicada posteriormente. Se ha descartado utilizar la sincronización optimista, porque esta requiere que se guarde un histórico del estado del sistema para hacer el mecanismo de *rollback* y, puesto que las particiones del modelo pueden ser redistribuidas durante el proceso de simulación por un balanceo de la carga, se necesitaría mover no solo el submodelo sino también todo lo referente al estado de esa partición, aumentando considerablemente los tiempos del proceso de balanceo de carga.

2.2. Modelo basado en Redes de Petri

Una Red de Petri es una representación formal matemática basada en un grafo bipartido con pesos y dirigido que describe la ejecución de procesos concurrentes y/o distribuidos. Es una de las herramientas más utilizadas en el modelado de sistemas de eventos discretos ya que permiten tanto la representación gráfica como matemática del modelo. Este trabajo se circunscribe a sistemas de eventos discretos que serán modelados mediante Redes de Petri Lugar/Transición, donde las variables de estado se corresponden con los lugares de la red, el valor de una variable de estado viene definido por el contenido de marcas del lugar que la representa (siempre una cantidad entera no negativa), y los cambios de estado los producen las transiciones modificando los contenidos de marcas de los lugares de entrada y de salida en valores enteros. Cuando una transición produce un cambio de estado se dice que la transición ha ocurrido o la transición ha sido disparada. Una evolución del comportamiento de la Red de Petri vendrá definida por una secuencia de disparos de transiciones a partir de un estado inicial de la red.

A la hora de simular una red de Petri, una de las tareas esenciales a implementar es la determinación de la sensibilización de una transición. Una vez que se han determinado las transiciones que están sensibilizadas se procede a su disparo y a la correspondiente actualización de los marcados de los lugares a ellas conectados. Para determinar que una transición está sensibilizada hay que ver que todos sus lugares de entrada contienen al menos una cantidad de marcas igual al peso del arco que une el lugar a la transición. Por tanto, la construcción de la lista de transiciones sensibilizadas en un ciclo de simulación, en el peor de los casos, requiere el repaso de la lista de todas las transiciones y de cada transición, sus lugares de entrada.

Esta tarea es muy pesada computacionalmente, sobre todo cuando en este trabajo estamos tratando modelos de grandes dimensiones y que por tanto tendrán un número elevado de transiciones. Pero es que además es innecesario, dado que de un ciclo a otro el número de transiciones disparadas es pequeño con relación al tamaño del modelo en los ejemplos reales, y por tanto el número de lugares que verán modificado su contenido de marcas también será pequeño. Esto quiere decir que de un ciclo a otro de simulación, el número de transiciones que verán modificadas sus condiciones de sensibilización es relativamente pequeño.

Para aprovechar esta circunstancia, en la simulación de redes de Petri vamos a utilizar un mecanismo basado en la caracterización de la sensibilización de una transición mediante una única función, denominada ***Linear Enabling Function*** (LEF) [10], que se actualiza solo si alguno de los lugares de entrada ha visto modificado

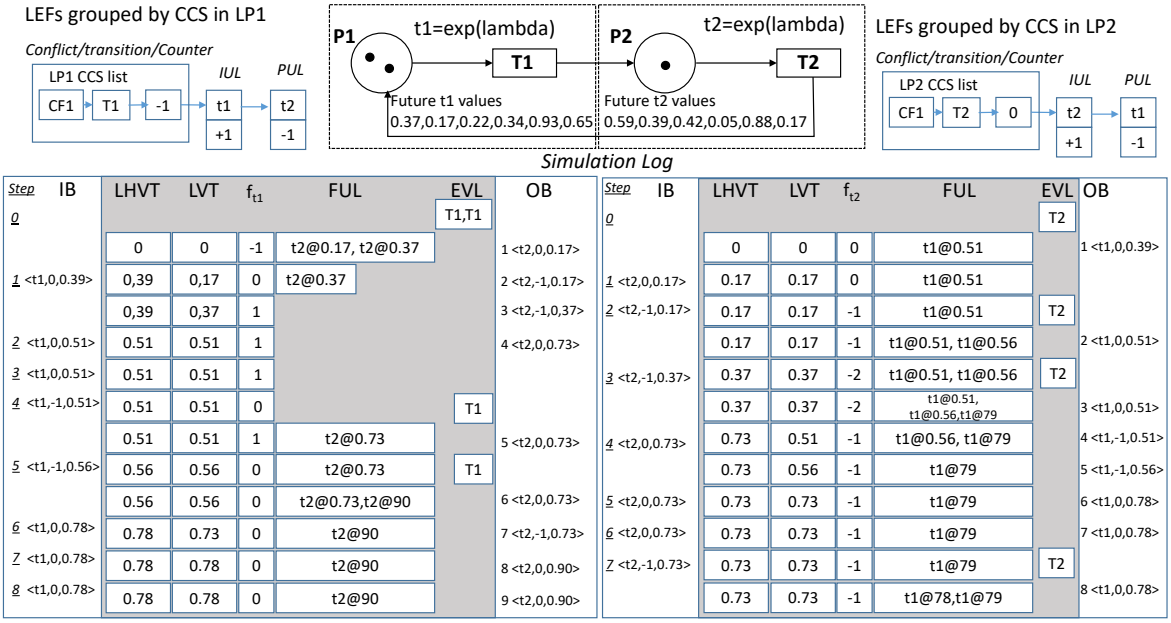

Figura 2.1: Traza de una simulación distribuida con LEFs extraída del artículo[2]

su contenido de marcas. La detección de de la lista de transiciones sensibilizadas en un ciclo se simplifica, y además se puede reducir el número de transiciones a consultar en un ciclo para ver si están sensibilizadas. Un LEF permite caracterizar cuando una transición está habilitada (un evento puede ocurrir) con una función lineal simple para el marcado. Un LEF de una transición t es una función $f_t : \mathbf{R}(\mathcal{N}, \mathbf{m_0}) \longrightarrow \mathbb{Z}$, que mapea cada marcado $\mathbf{m}$ perteneciente al conjunto de marcados alcanzables, $\mathbf{R}(\mathcal{N}, \mathbf{m_0})$, a un número entero, de tal manera, que t puede ocurrir o ser disparada en $\mathbf{m}$, si $f_t(\mathbf{m}) \leq 0$. Por ejemplo, para la transición $T1$ en la red de la figura 2.1, el LEF es: $f_{T1}(\mathbf{m}) = 1 - (\mathbf{m}[P1]), \forall \mathbf{m} \in \mathbf{R}(\mathcal{N}, \mathbf{m_0})$, donde $\mathbf{m_0}$ es la marca inicial representada. Observe que en $\mathbf{m_0}$, el valor de $f_{T1}(\mathbf{m_0}) = -1 <= 0$ y $f_{T2}(\mathbf{m_0}) = 0 <= 0$, es decir, ambas transiciones están habilitadas en $\mathbf{m_0}$. Para obtener más detalles consultar [1]. La obtención de las funciones LEF y sus valores iniciales se realiza en tiempo de compilación del modelo para su simulación. Obsérvese que la red de Petri queda codificada en una estructura de datos compuesta por los valores iniciales de cada una de las LEFs junto con las listas de constantes con las que hay que actualizar las LEFs que estén conectadas con esa transición mediante algún lugar. Por ello la migración de una subred o de un trozo de la red se reduce al envío de una estructura de datos y nunca hay que migrar código como ocurre en otros sistemas de simulación de sistemas de eventos discretos. Esta es la razón de la utilización de esta representación basada en LEFs en lugar de la especificación clásica lugar/transición.

2.3. Balanceo de carga

De manera general, el balanceo de carga es un concepto que define los mecanismos necesarios para distribuir la carga de trabajo entre varios computadores, procesos o recursos. El objetivo de estos mecanismos es conseguir que en un sistema distribuido, el trabajo sea repartido de la manera más eficiente y equitativa posible para que el funcionamiento del conjunto del sistema sea óptimo, consiguiendo un **sistema elástico**.

Los más conocidos son los que se encargan de repartir la carga en servidores web, los cuales afrontan un gran problema de escalabilidad al recibir continuamente una alta cantidad de peticiones. Estos balanceadores de carga, mediante un algoritmo, se encargan de distribuir cada una de las peticiones a los nodos del clúster de servidores para aprovechar al máximo la capacidad computacional de cada nodo y evitar *cuellos de botella*. Estos balanceadores de carga habituales en clústers web, se basan en el ritmo de recepción de eventos, utilizándose técnicas de reparto de flujos de eventos que procesan de manera independiente los datos.

Sin embargo, en el contexto de nuestro simulador distribuido, el balanceo de carga es más complejo ya que los distintos nodos deben avanzar el tiempo de simulación dependiendo de la recepción de mensajes de otros nodos, no lo hacen de manera independiente. Puede suceder que haya nodos que no avancen al mismo ritmo que otros debido a que tengan que procesar muchos más eventos que otros en el mismo tiempo de simulación, que las latencias de comunicación entre nodos sean diferentes, por lo que haya nodos que tarden más en los protocolos de coordinación de avance del tiempo con sus vecinos, que el hardware sea heterogéneo o que sufra de interferencias en la carga de trabajo. Dada la complejidad de estas causas, este trabajo se ha centrado en el diseño de mecanismos para balancear la carga de trabajo computacional sin detener la simulación, y no en las posibles causas de que la carga no se encuentre balanceada.

Además, con el fin de conseguir el máximo rendimiento del simulador distribuido sea cual sea el contexto de ejecución del simulador, cada nodo de simulación es capaz de enviar y recibir su partición del modelo o subred a otro nodo a través de la red de comunicación y fusionarlo a su modelo ya existente para conseguir una simulación más óptima en un contexto más beneficioso para la ejecución del simulador distribuido completo.

Capítulo 3

Arquitectura distribuida: Análisis y diseño

En este capítulo se van a analizar los componentes o módulos requeridos para el funcionamiento de la simulación distribuida y sus correspondientes interacciones tanto locales como externas entre los mismos, es decir, de qué se compone cada proceso de simulación y cómo es la comunicación entre dichos procesos. A continuación se presenta una figura que resume la arquitectura de componentes de la simulación distribuida y sus interacciones.

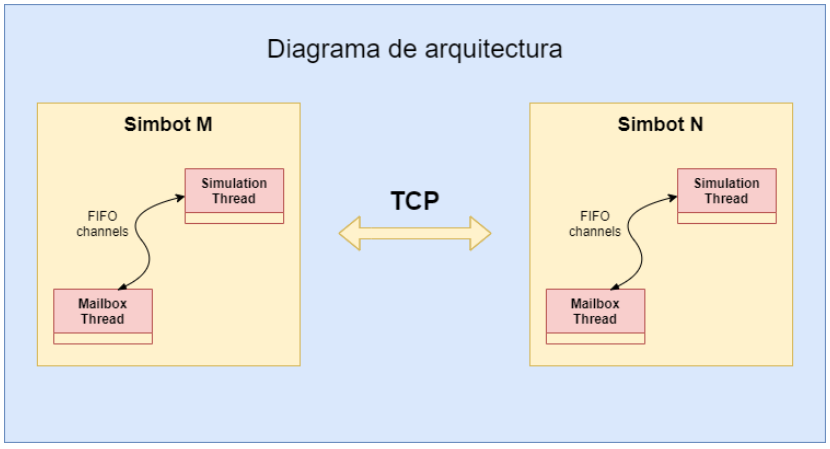

Figura 3.1: Diagrama de componentes de la arquitectura distribuida del simulador

3.1. Concepto de simbot

Cada uno de los procesos lógicos (LP) que simula una partición del modelo se denomina **simbot**. Estos simbots interactúan entre sí mediante el paso de mensajes con marcas temporales. En la figura 3.2 se presentan los componentes de un simbot y las interacciones que suceden entre ellos.

El motor de simulación es el encargado de que cada LP asegure que los eventos

generados tanto localmente como recibidos por otros simbots son procesados en el orden estricto de marcas temporales, para así garantizar que se produce la misma salida entre la simulación centralizada y distribuida. Este motor de simulación se encuentra en el primer hilo de ejecución del LP. En la figura 3.2 se corresponde con el gráfico de *Simulation Engine*, en el cual se pueden observar sus tareas principales: la interpretación del modelo con las estructuras LEFs, el avance del tiempo simulado con un reloj virtual local para cada simbot (LVT) y el tratamiento de los eventos almacenados en la lista de eventos futuros (FUL), los cuales pueden ser generados localmente o externos procedentes de simbots adyacentes. En el capítulo de implementación 4.3 se detallará en profundidad el algoritmo de simulación distribuido que realiza el tratamiento del reloj virtual y de la FUL.

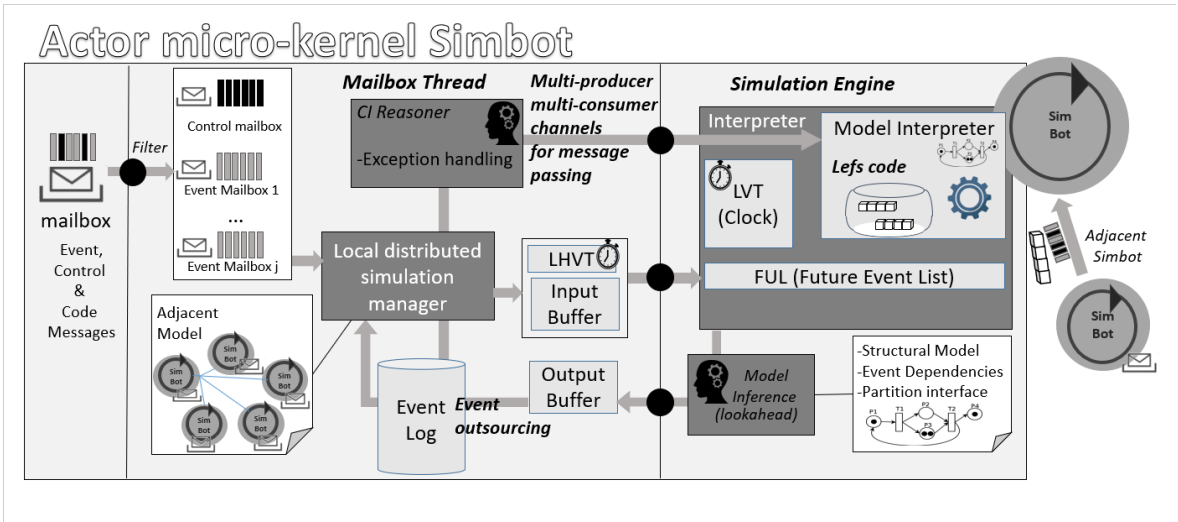

Figura 3.2: Descripción de un simbot y sus componentes

Por otro lado, cada simbot tiene una interfaz de comunicación con la que envía y recibe mensajes de simbots adyacentes y es el segundo hilo de ejecución de cada uno de los LP. Esta interfaz se ha denominado *Mailbox*, debido a que su principal función es la de un comunicador que se encarga de hacer llegar los mensajes recibidos al motor de simulación. Además, estos mensajes son filtrados según sus etiquetas antes de su envío al hilo de simulación. Toda la lógica de filtrado de mensajes y del estado del *Mailbox* será explicada en el capítulo de implementación.

3.2. Comunicación entre componentes de un simbot

Los componentes de un simbot requieren tener comunicación local entre ellos. Para ello se va a utilizar el paso de mensaje mediante canales asíncronos

multiproductor-multiconsumidor. Esto significa que ambos extremos de la comunicación, en este caso los dos hilos de ejecución que conforman el LP o simbot, van a ser capaces tanto de enviar como de recibir mensajes a través de una cola *FIFO* (First In First Out), de tal manera que ambos hilos de ejecución pueden actuar como productores o como consumidores en función de las necesidades de enviar o recibir un mensaje entre hilos.

Además este paso de mensajes se va a producir de manera asíncrona, es decir, ambos extremos de la comunicación no van a tener que esperar una respuesta para enviar un mensaje, sino que entregarán estos mensajes en el canal *FIFO* de comunicación y cuando el otro extremo esté listo absorberá el mensaje almacenado en la cola y hará un tratamiento del mismo. Esta solución es ideal porque permite que la simulación no sea interrumpida durante un ciclo de simulación, y solo adquiere los mensajes cuando una vez finalizado ese ciclo de simulación, necesita estos mensajes para proseguir con la simulación. La implementación de estos canales en Rust va a ser explicada en el Anexo B

3.3. Comunicación entre simbots

Además de comunicación entre componentes de un simbot, también se necesita la comunicación entre los propios simbots para que mediante el envío de mensajes la simulación distribuida pueda ser coordinada. Puesto que los simbots van a ser ejecutados en máquinas distintas con la posibilidad de que sus arquitecturas hardware sean heterogéneas, es necesario que esta comunicación sea realizada a través de la red y mediante un protocolo estandarizado que conozcan todas las máquinas sea cual sea su arquitectura.

El protocolo utilizado va a ser TCP[11]. Para ello cada simbot va a escuchar en un puerto específico esperando la conexión de otro simbot para establecer un socket de conexión entre ellos. Para el establecimiento de los sockets de conexión se ha trabajado con dos alternativas: Abrir y cerrar el socket de conexión cada vez que se va a enviar un mensaje, lo cual añade pérdidas de rendimiento al tener que hacer llamadas al sistema operativo o bien abrir ese socket al principio de la ejecución del simbot con todos los simbots que participan en la simulación distribuida. Esta solución última es la que se optó por llevar a cabo en el framework desarrollado en Java[4], sin embargo, debido a la gestión estricta de memoria que tiene el lenguaje Rust, mantener estas conexiones abiertas ha sido imposible de conseguir.

Finalmente se ha elegida implementar la primera aproximación propuesta, la cual además es la recomendada por la librería estándar de Rust, en la que cada vez que se va a realizar un envío, se establece la conexión creando un socket y una vez realizado el envío se cierra ese socket, quedando el extremo de recepción a la escucha de una nueva conexión entrante.

3.4. Mecanismos de balanceo

El propósito de la implementación de este mecanismo de balanceo de carga es poder redistribuir la carga de trabajo que soportan los distintos simbots que participan en la simulación distribuida. Para ello el modelo que absorben dichos simbots y que posteriormente simulan, va a ser particionado y enviado a otro simbot dinámicamente mientras se esta ejecutando una simulación.
Es importante distinguir a qué nos referimos con balanceo de carga. El balanceo de carga generalmente consiste en redistribuir la carga de trabajo sobre las máquinas disponibles para evitar que una máquina se encuentre mucho más ocupada que otra, y así aumentar las prestaciones del conjunto total. Este mecanismo suele realizarse a nivel de proceso, es decir, el proceso lógico completo compilado se mueve a otra máquina garantizando un estado seguro del sistema distribuido y, una vez completado el movimiento entero del proceso lógico, la ejecución continúa normalmente.
Sin embargo, el mecanismo de balanceo de carga que se propone consiste en realizar la distribución de la carga a **nivel de modelo** en lugar de a nivel de proceso lógico. Para una mayor comprensión de ello, se va a utilizar el término de **Simulation Process (SP)**.
Un SP es el conjunto que incluye por un lado el simbot (el proceso compilado), y por otro lado el modelo que este simbot absorbe y simula (estructura de datos que representa la Red de Petri y su estado de simulación, es decir, estructura de datos que representa los valores actuales de las LEFs y las listas de constantes a enviar como eventos etiquetados temporalmente como consecuencia del disparo de la transición). Debido a esto, el balanceo de carga propuesto se va a encargar de distribuir una parte del modelo que está siendo simulado en un simbot, enviándolo a otro simbot para que lo absorba y lo fusione con el modelo que ya disponía, y continuar la simulación de tal forma que, la simulación distribuida de estos SPs va a funcionar de una manera mucho más eficiente al haber equilibrado la carga de trabajo que soportan.
El algoritmo de balanceo de carga va a añadir un tercer hilo de ejecución a los dos que ya había inicialmente en cada simbot tal y como ya se ha descrito en la sección 3.1. Este tercer hilo, llamado ***LEFs Movement Manager***, se va a encargar de

monitorizar el hilo de simulación y avisar al hilo de simulación que hay que realizar un movimiento de LEFs de un simbot a otro. En la figura 3.3 se observa la aparición de este nuevo componente. La interacción entre el motor de simulación y este LEFs Movement Manager va a ser de manera análoga al Mailbox, mediante paso de mensajes utilizando canales asíncronos multi-productor multi-consumidor.

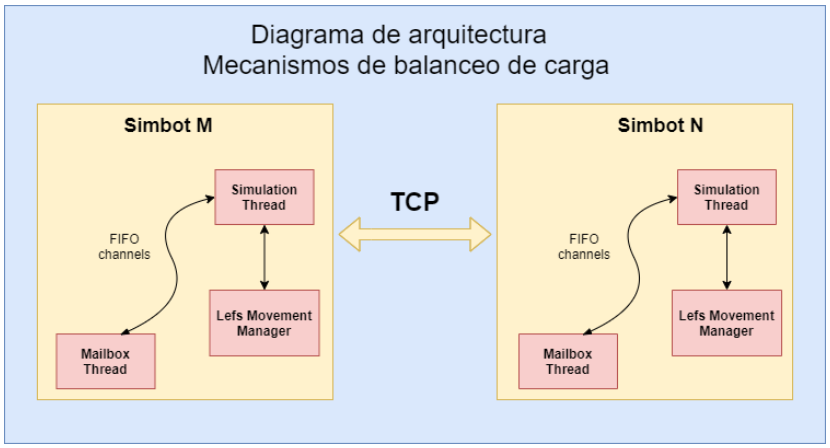

Figura 3.3: Diagrama de componentes incluyendo el nuevo componente

La decisión de balancear la carga va a ser tomada solamente teniendo en cuenta el tiempo de simulación, realizando una operación de movimiento de LEFs de manera predeterminada. Esto se debe a que se va a poner el foco sobre el mecanismo de balanceo de la carga, es decir, se ha definido un algoritmo que garantiza que el balanceo se produce de tal manera que el resultado final de la simulación no se ve alterado con respecto a la ejecución del simulador distribuido sin mecanismos de balanceo. Las decisiones de realizar estos mecanismos de balanceo no ha sido abordada en este trabajo, ya que este aspecto sería el tema de otro trabajo de la misma magnitud que este mismo.

Capítulo 4

Implementación del sistema

En este capítulo se va a profundizar en la implementación de las características propias del simulador distribuido en lenguaje Rust. Para ello, se va a seguir el desarrollo desde las fases iniciales de adaptación del modelo compilado que se produce en el compilador en Java ya existente hasta la implementación de las estructuras de datos y algoritmos que permiten el funcionamiento correcto del simulador. Se van a especificar tanto liberías y herramientas para la programación del código, como se van a explicar los algoritmos, estructuras y máquinas de estado empleadas para el desarrollo del simulador.

4.1. Adaptación del modelo compilado para Rust

Durante las fases iniciales del trabajo, se comenzó con adaptar el **modelo compilado** que se obtiene del compilador Java para las necesidades del simulador en lenguaje Rust. Este modelo compilado se obtiene en varios ficheros de extensión *.json* correspondientes a las subredes o particiones del modelo completo, los cuales son absorbidos por cada simbot cuando este se ejecuta mediante la librería *serde*[12]. Para adaptar los ficheros de subredes se ha desarrollado un **script** con ***Ruby*** que adapta el código LEFs compilado de cada subred para las estructuras de datos que absorben dicha información y la utilizan para la ejecución del motor de simulación. En este script se crean las listas de transiciones externas o se hace la asignación de máquinas para cada subred. Así, cada subred tiene su índice de subred y una dirección *ip:port* asignada que permite los envíos de eventos entre simbots a través del protocolo *TCP*. Se ha elegido el lenguaje Ruby ya que permite la manipulación de ficheros *JSON* de una manera muy rápida y eficiente mediante el módulo json[13]

4.2. Aspectos diferenciales: Simulación distribuida conservativa

4.2.1. Estructuras creadas para la simulación distribuida

El simulador distribuido respecto del simulador centralizado incorpora las siguientes estructuras de datos principales:

1. La posibilidad de que sucedan eventos exteriores, es decir, que el disparo de transiciones locales produzca eventos que se han de propagar a un simbot vecino o adyacente, el cual sin ese evento no puede proseguir con su simulación, implica que se haya creado una estructura *ExternTransition*, la cual posee el índice de transición global, la dirección *Ip:Port* en la que escucha el simbot que posee esa transición y el índice de la subred a la que pertenece. Además a la estructura LEFs que se obtiene de la compilación del modelo se le ha añadido una lista de transiciones externas en la que se recogen estas transiciones externas, para que la indexación de estas transiciones se haga en la compilación y evitar búsquedas e iteraciones sobre la lista de transiciones completa en el proceso de simulación con el objetivo de aumentar el rendimiento de la simulación.

2. Para que estos eventos exteriores sean enviados y recibidos se necesitan unas estructuras que sean serializadas y enviadas a través de la red de comunicación mediante una conexión *TCP* tal y como se ha definido en la sección 3.3. Para ello la estructura *Event* ha sido alterada para que almacene el índice de transición global en lugar del local y se ha añadido un *flag* para conocer si el evento es externo o no. Adicionalmente, se ha creado la estructura mensaje, en la cual se encapsula el evento y se añaden flags adicionales para que el Mailbox o interfaz de comunicación distribuya el mensaje hacia las colas *FIFO* que consume el hilo de simulación. Esta estructura mensaje junto con el estado del Mailbox será explicado en la sección 4.4

4.2.2. El cálculo del lookahead: Principios de la simulación conservativa

El simulador distribuido debe respetar la restricción de causalidad local que implica que todos los mensajes entrantes que contienen eventos, estos deben ser procesados en el orden correcto de estampillas de tiempo globales. Si cada LP respeta la restricción de causalidad local, se asegura también la causalidad global y, por lo tanto, que la ejecución distribuida produzca los mismos resultados que una simulación centralizada del mismo modelo. Para ello la simulación distribuida ofrece dos principales estrategias tal y

como se ha explicado en la sección 2.1. En este trabajo se ha seguido la sincronización conservativa. Para asegurar el procesado de eventos en el orden temporal correcto de simulación, se asocia una cola *FIFO* por cada simbot adyacente de otro simbot. Se dice que un simbot es adyacente si este le puede enviar un mensaje que contiene un evento con marcado temporal, es decir, cada conexión de recepción de mensajes en un simbot. El algoritmo conservativo que será explicado en profundidad en la sección 4.3, espera a que cada cola *FIFO* de recepción contenga, al menos, un mensaje. Una vez se ha cumplido esta premisa, se toma y se elimina el evento de menor estampilla de su *FIFO* para calcular el LVHT. Esta espera sobre las colas *FIFO* puede causar un interbloqueo si varios LPs quedan esperando mensajes entre sí. Para evitar este bloqueo, se produce el envío de lookaheads. Estos son mensajes de valor *NULL* que solo contienen una marca temporal sin eventos. Indican una previsión de mínimo valor de tiempo para cualquier evento futuro cuyo origen sea ese simbot adyacente. Es decir, un valor temporal que indica al resto de simbots el tiempo mínimo que pueden avanzar sus relojes virtuales hasta que ese simbot genere un evento exterior. Al no disponer de eventos no son seleccionables para ser procesados como eventos, pero sí permiten calcular el LVHT y así evitar bloqueos entre simbots. El cálculo de este valor se hace a partir del modelo en tiempo de compilación, dado que el modelo formal de Redes de Petri nos permite hacerlo. Para ello cada vez que se tiene que enviar un lookahead se hace la suma del valor de reloj virtual en ese momento y el tiempo que va a tardar en llegar una marca desde todas las transiciones que están habilitadas o sensibilizadas en ese instante de tiempo. Este último valor de tiempo es el que se obtiene en la compilación del modelo, ya que se puede predecir en todo momento los instantes discretos de tiempo simulado que tienen que pasar hasta que una marca llegue desde una transición a otra, puesto que el propio modelo nos informa de ello.

4.3. Motor de simulación: Algoritmo de simulación distribuida

Se ha implementado el **algoritmo de simulación conservativa** que garantiza la ejecución de eventos en el orden correcto de estampillas de tiempo globales y avanzar el tiempo de simulación sin necesidad de recuperar un estado anterior de simulación. Para conocer cómo se inicializa el simbot y se sincronizan los nodos de simulación, consultar el Anexo C

En la figura 4.1 se muestra el algoritmo de simulación conservativo que se lleva a cabo en el motor de simulación. Para ver su codificación en lenguaje Rust consultar el Anexo E

```
1: when Start_Synchronized() is received                          ▷ Initialize Simbot
2: VT ← 0; FUL ← {};
3: while VLT ≤ final_cycle do
4:     LVHT ⇐ check_neighbours();                                 ▷ Wait for events from every simbot
5:     initialize_simbots_to_send(lefs)
6:     simulate_one_step(LVHT);                                   ▷ Simulate until LVHT
7:     send_lookaheads(simbots_to_send);
8: end while
9: finish_simulation();
```

Figura 4.1: Algoritmo de simulación conservativo

Una vez inicializado el sistema, el motor de simulación entra en un bucle hasta que el valor de VLT (reloj local virtual) no alcanza el valor de ciclo final, que se define como un parámetro de la ejecución del simulador.

En la `linea 4` se obtiene el *LVHT* u horizonte virtual de tiempo hasta el cual es seguro que el simulador ejecute un paso de simulación(`línea 6`). Se obtiene de las colas *FIFO* que almacenan eventos procedentes de cada simbot adyacente. En este método *check_neighbours()* comprueba si hay un evento encolado de cada simbot adyacente y si no lo hay se queda a la espera de tener un evento o lookahead, para poder avanzar la simulación. Esto es debido a la sincronización conservativa, que nos impide avanzar hasta un horizonte de tiempo sin tener seguro que no vamos a recibir un evento con menor estampilla temporal. El *LVHT* por lo tanto, será el tiempo del evento o lookahead de menor estampilla de los eventos recibidos en todas las colas *FIFO*.

En la `linea 5` se obtiene una lista de todos los simbots adyacentes. Esta se obtiene del modelo (LEFS) compilado, concretamente de la lista de eventos externos.

En la `linea 6` se ejecuta el ciclo de simulación hasta el *LVHT*. En la figura 4.2 se profundiza en este método.

Por último, se envía el valor del lookahead a los simbots a los cuales no se les ha enviado un evento en el paso de simulación (`linea 7`). Si al avanzar el tiempo en el paso de simulación se alcanzase el valor del ciclo final de simulación, el bucle de simulación conservativo se finaliza y se comienza el proceso de sincronización

En la figura 4.2, se presenta en profundidad el método de la línea 4 en la figura 4.1. Se puede dividir en 3 grandes bloques:

1. En las (`lineas 4-17`), se produce el disparo de las transiciones habilitadas. Para cada transición habilitada ($t \in IUL(t')$), el algoritmo aplica inmediatamente factores de actualización *IUF* (`lineas 5-10`). Después inserta eventos en em PUL, que representan que los tokens aparecerán en lugares posteriores en el tiempo del reloj futuro. A continuación se trata la lista de PUL, de tal manera que si la transición de esa lista de eventos producidos es externa, dicho evento con su factor o constante de actualización *IUF* es enviado al simbot destinatario

```
1: procedure simulate_one_step(LVHT)
2:     while (VLT <= LVHT) do
3:         for all (t' ∈ EL) do                                          ▷ Fires enabled transitions
4:             if (f_t'(M) ≤ 0) then                                     ▷ Checks transition is enabled yet
5:                 for all (t ∈ IUL(t')) do
6:                     f_t(M) ← f_t(M) + UF(t' → t);
7:                     if (t = t' and f_t(M) ≤ 0) then                   ▷ Avoids race conditions
8:                         insert-FUL (t, 0, τ(t) + clock);
9:                     end if
10:                end for
11:                for all (t ∈ PUL(t')) do
12:                    if (t.is_extern) then
13:                        send_msg(lefs.etl.address, t, UF(t' → t), τ(t') + clock))   ▷ Send Event to Extern Transition
14:                    else
15:                        insert-FUL (t, UF(t' → t), τ(t') + clock);   ▷ Local Transition, insert into FUL
16:                    end if
17:                end for
18:                if (head-FUL.time > LVHT) then VT ← LVHT              ▷ Update Virtual Time
19:                else VT ← head-FUL.time
20:                end if
21:                while (head-FUL.time = VT) do                         ▷ Update Event List
22:                    t ← head-FUL.pt; f_t(M) := f_t(M) + head-FUL.UF;
23:                    if (f_t(M) ≤ 0) then insert(EL, t);
24:                    end if
25:                    head-FUL ← pop(FUL);
26:                end while
27:            end if
28:        end for
29:    end while
30: end procedure
```

Figura 4.2: Ciclo de Simulación hasta LVHT

(`lineas 12-13`). En caso de que el evento producido sea local a ese simbot, se añade a lista de eventos futuros local (*FUL*).

2. En las (`lineas 18-20`), se produce el avance del tiempo simulado. Si el evento de menor estampilla temporal almacenado en la *FUL* es mayor que el *LVHT* u horizonte de tiempo, el tiempo simulado (*VLT*) se actualiza al valor de *LVHT* y se para este paso de simulación. En caso contrario, se avanza el reloj virtual al tiempo de ese evento, que es inferior al *LVHT*, por lo que este paso de simulación continuará iterando.

3. Una vez avanzado el tiempo de simulación, se procesan los eventos de este *VLT* en las líneas 21-25, insertando las transiciones habilitadas en la *EL* (Enabled Transition List).

4.4. Interfaz de comunicación: Mailbox

La interfaz de comunicación o hilo *Mailbox* es un receptor de mensajes que se encarga de filtrarlos y repartirlos a los distintos búferes o colas que el motor de simulación va a consumir.

En la figura 4.3 se muestra la máquina de estados que describe el funcionamiento de esta interfaz de comunicación.

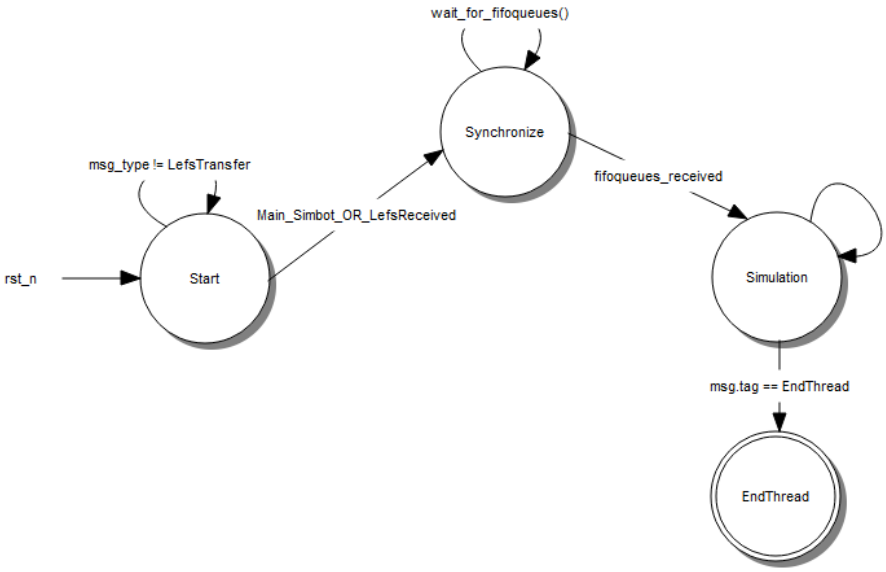

Figura 4.3: Máquina de estados del hilo Mailbox

En el comienzo de la ejecución, el *Mailbox* de un simbot espera recibir el código LEFs por parte del simbot principal. Una vez es recibido, pasa al estado de sincronización donde espera recibir las colas *FIFO* de los simbots adyacentes por parte de su motor de simulación (estas se obtienen a partir del código LEFs recibido), de ahí viene la necesidad de que la comunicación entre componentes de un simbot sean canales bidireccionales, es decir, multi-productores multi-consumidores. Después el *Mailbox* ya entra en la fase de simulación donde espera recibir mensajes de eventos o lookahead que irá filtrando hacia las colas *FIFO* de recepción en función de quien sea el emisor del mensaje.

Para más información sobre los tipos de mensajes que puede recibir el *Mailbox*, consultar el Anexo D.

4.5. Implementación de los mecanismos de balanceo: Algoritmo de movimiento de LEFs

A continuación se va a describir el algoritmo de movimiento de LEFs que ha sido añadido al simulador distribuido de eventos discretos, el cual es una modificación del algoritmo de simulación conservativa 4.1 en la que está basado el simulador distribuido

- Se añade en cada iteración, la comprobación del hilo de movimiento de LEFs (*linea 2 - listen_movement_manager()*). En caso de que haya que hacer un movimiento se ejecuta la función de movimiento de LEFs que posteriormente va a ser explicada y se sincronizan todos los simbots involucrados para continuar la simulación.

```
1: while VLT ≤ final_cycle do
2:     channel_msg ⇐ listen_movement_manager();          ▷ Check for msgs from manager thread
3:     if channel_msg.len() > 0 then                     ▷ Msg from Thread received!
4:         LVHT ⇐ lefs_movement(channel_msg, my_index);  ▷ Simbot involved in Lefs Movement Step
5:         sync_restart();
6:     else                                              ▷ Not involved in Lefs Movement Step
7:         LVHT ⇐ check_neighbours();
8:     end if
9:     simulate_one_step(LVHT);
10:    send_lookaheads(simbots_to_send);
11: end while
12: finish_simulation();
```

Figura 4.4: Algoritmo de movimiento de LEFs

– En caso de que haya que hacer el movimiento, el nuevo horizonte de tiempo va a ser calculado a partir de los mensajes almacenados en las colas de vecinos previamente antes de realizar el movimiento del LEFs, por lo que no hay que volver a comprobar esas colas como si fuese una iteración normal de simulación.

```
1: // Post: Returns the new LVHT calculated
2: // using the msgs from the dropped queues
3: procedure lefs_movement(channel_msg, my_index)
4:     simbot_receiver ⇐ channel_msg.receiver;
5:     simbot_sender ⇐ channel_msg.sender;
6:     msgs_from_queues ⇐ save_buffered_msgs();          ▷ Save all messages stored in FifoQueues
7:     if simbot_receiver == my_index then               ▷ Simbot is Receiver of the Lefs
8:         lefs ⇐ receive_lefs();                        ▷ Blocking call
9:         msgs_from_queues ⇐ receive_buffered_msgs()    ▷ Blocking call
10:    else if simbot_sender == my_index then            ▷ Simbot is Sender of the Lefs
11:        send_lefs(simbot_receiver);
12:        send_buffered_msgs(msgs_from_queues);
13:    end if
14:    // Modify socket addresses of the LEFs moved.
15: return update_addrs(simbot_sender, simbot_receiver, msgs_from_queues);
16: end procedure
```

Figura 4.5: Paso de movimiento de LEFs

Hay tres casos en los que un simbot detiene su simulación para realizar la operativa de movimiento de LEFs: (1) ser el simbot **origen** del movimiento, el cual envía su código LEFs a otro simbot, (2) ser el simbot **destinatario** de esa estructura, o (3) ser un **vecino** de esa subred movida.

Por vecino se entiende a aquel simbot que o bien es receptor de eventos externos que genera la partición de LEFs que va a ser movida (necesita modificar la dirección *ip:port* asociada a esa cola *FIFO* de entrada) o bien ser un simbot cuyo LEFs tiene una transición de salida hacia ese código LEFs que va a ser movido (necesita modificar la dirección *ip:port* de esa transición externa en su lista de transiciones externas).

Los pasos que se van a seguir para realizar el movimiento de LEFs son los siguientes:

1. Para comenzar se obtienen los índices de los simbots que van a ser emisores y receptores, los cuales además de modificar las direcciones *ip:port* previamente mencionadas, deben realizar el envío o recepción del código LEFs.

2. Antes de hacer estas operaciones, primero se almacenan los mensajes que hay almacenados en las colas *FIFO* de entrada, ya que estas van a ser modificadas, de tal manera que alguna de las colas pueda ser borrada (no espera más transiciones de ese simbot) o añadir una nueva con el nuevo simbot que recibe el LEFs si antes este último no tenía una cola asociada. Además estos mensajes son muy importantes para el cálculo del nuevo horizonte de tiempo *LVHT* en este paso de simulación (recordar que la simulación no se para, va a continuar una vez realizado el balanceo de carga).

3. Una vez almacenados, se realiza la distinción de qué tiene que hacer cada simbot en función del rol: enviar o recibir la estructura. En caso de ser un vecino simplemente deberá actualizar sus colas*FIFO* y las direcciones de sus transiciones externas (función *update_addrs()*).

4. Si tiene rol de emisor (`líneas 10 a 13`), le envía la estructura y después le envía los mensajes almacenados en las colas anteriormente para que el nuevo simbot pueda calcular el nuevo LVHT y no se pierda ningún evento que no haya sido tratado todavía al ser posterior al LVHT previo.

5. En caso de ser receptor, recibe la estructura y la lista de mensajes ordenada almacenada antes de modificar las colas*FIFO*, en sendas llamadas bloqueantes. En la recepción de la lista de mensajes buferizados, también van incluidos los eventos de la FUL que tenía ese simbot.

6. Por úlitmo, sea cual sea el rol del simbot en la operativa, modifican las direcciones *ip:port* de las colas *FIFO* y las transiciones externas y además obtienen el *LVHT* tratando los mensajes ordenados de la variable *msgs_from_queues*

Después el algoritmo distribuido de simulación continúa con su normal ejecución, no sin antes realizar la sincronización (`linea 5 -- Figura 4.4`).

En el Anexo A se muestran las máquina de estados que representan la operativa descrita anteriormente en los distintos componentes de un simbot.

Capítulo 5

Estudio experimental

5.1. Proceso de despliegue y puesta en marcha

El proceso de despliegue y puesto en marcha depende de dos tareas bien diferenciadas. Tal y como se ha hecho referencia anteriormente, un Simulation Process (*SP*) necesita de un modelo y de un LP o simbot que cargue en memoria dicho modelo y lo ejecute.

5.1.1. Compilación del modelo

Para obtener el modelo en un formato que sea asumible por el simulador, este requiere de un proceso de compilación. Este proceso se lleva a cabo mediante el *framework* desarrollado en el TFG[4]. Para ello se genera la Red de Petri que representa el modelo que se quiere simular en un fichero textual resultante de poner en marcha el proceso de compilación en lenguaje Java con los siguientes parámetros:

PARÁMETROS:
$1: nombre del fichero de salida de la RdP Textual
$2: número de lugares horizontales
$3: número de lugares verticales
$4: número de bloques

En la figura 5.1 se muestra un ejemplo de Red de Petri con las que se han ido realizando las pruebas de ejecución. En la esquina superior derecha se pueden ver los parámetros dados para la generación de esta red. El número de bloques indica que se van a necesitar 4 nodos de simulación o 4 subredes. La primera subred sería la que incluye P0,P1,t0,t1. Estas transiciones (t0 y t1) son las que sincronizan la ejecución de las 3 ramas paralelas, las cuales van a ser, cada una de ellas, una subred del modelo.

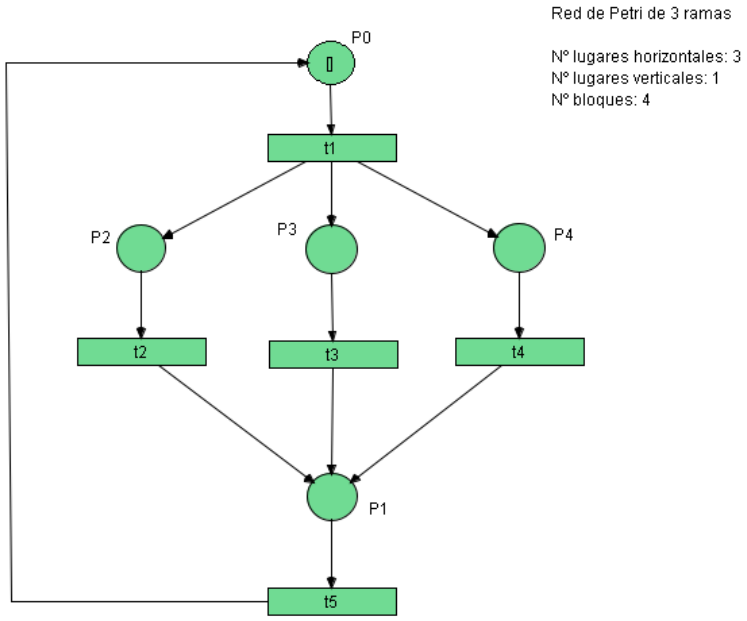

Figura 5.1: Red de Petri con ramas de ejemplo

Una vez se ha obtenido este fichero con la red textual, se genera un fichero de extensión *.json* para cada subred, el cual luego será absorbido por el simulador distribuido en Rust.

Por último, se requieren traducciones para adaptar el fichero *json* generado a las nuevas estructuras creadas mencionadas en la sección 4.2.1 para el simulador distribuido de Rust. Para ello se ejecuta el script en Ruby desarrollado al comienzo de este trabajo (explicado en la sección 4.1).

5.1.2. Ejecución del simulador distribuido

Una vez se ha obtenido el modelo compilado particionado en subredes y se ha adaptado mediante scripts para que se corresponda con las estructuras creadas para la simulación distribuida, se debe compilar el código del simbot programado en lenguaje Rust.

Para la compilación se ha utilizado la herramienta cargo[], el gestor de paquetes de Rust. Cargo permite definir perfiles de compilación de una manera similar a los niveles de optimización de compiladores como GCC. El comando ejecutado para la compilación del código de un simbot es el siguiente:

cargo build –release

Con la opción –release se compila con el profile *release* definido en el fichero de configuración *cargo.toml* del proyecto. Este perfil incluye optimizaciones para la

compilación con la opción **-o3** o **lto=true**, esta última realizando optimizaciones para el rendimiento a pesar de incrementar el tiempo de linking del programa. Este proceso de compilación genera un programa en binario que puede ser ejecutado posteriormente como cualquier ejecutable en Unix.

Una vez se tiene, por un lado el modelo compilado y adaptado para Rust, y por otro el código del simbot en Rust compilado con las opciones de optimización, se pone en marcha el simulador con el comando de ejecución básico de Unix, acompañado de los siguientes argumentos:

$$./esimc < nombre\_fichero\_modelo >< num\_ciclos >< lista\_ip : port\_nodos >$$

Donde *esimc* es el nombre del binario generado por cargo.Este comando tendrá que ser ejecutado en los distintos nodos que vayan a participar en la simulación distribuida.

5.2. Entorno de pruebas

En esta sección, se van a describir las características de los nodos que han sido utilizados para la realización de pruebas experimentales. Se distinguen 3 entornos principales dónde se han realizado las ejecuciones de este simulador.

- Ejecuciones en el **entorno de desarrollo**. Las primeras ejecuciones se realizaron sobre la misma máquina donde se desarrolló e implementó el simulador distribuido. El aspecto a tener en cuenta de estas ejecuciones era el hecho de que los nodos distribuidos se ejecutaban sobre procesos concurrentes del mismo sistema, por lo que las latencias de red eran nulas.

  Las características de esta máquina son:

  - Sistema Operativo: Ubuntu Linux 18.04 Bionic Beaver
  - Intel Core i5-7200U CPU 2 Cores 4 hilos
  - Memoria Ram: 8Gb

- Ejecuciones en un entorno ***on premise*** de la Universidad de Zaragoza. Este entorno cuenta con 48 *Raspberry Pi 4* todas conectadas entre sí a través de la red mediante la máquina *dawson.cps.unizar.es*. Las características de estas Raspberries son:

  - Sistema Operativo: Ubuntu 20.04.2 LTS Focal Fossa
  - ARM Cortex-172 con cuatro núcleos a 1,5 GHz

- Memoria Ram: 8Gb

– Ejecuciones en un entorno de ***cloud*** con *Google Cloud Platform*. Este entorno ha sido puesto en marcha por el compañero Hayk Kocharyan en su TFG[5], y ha sido utilizado al final del desarrollo de ambos trabajos para comparar prestaciones con el resto de entornos. Cabe destacar que el desarrollo del entorno *cloud* mediante Slurm ha permitido lanzar ejecuciones del simulador en un entorno de *cloud* híbrido (*on premise y cloud*) y así poder demostrar la utilidad de los mecanismos de balanceo de carga, llevando la carga de trabajo a un solo entorno evitando las altas latencias de red entre entornos, lo que suponía un cuello de botella.

5.3. Análisis de resultados

5.3.1. Análisis del rendimiento del simulador distribuido

A continuación, se van a presentar resultados del rendimiento del simulador distribuido de eventos discretos implementado en lenguaje Rust para este trabajo. Para ello, primero se van a hacer una serie de pruebas tanto en el simulador distribuido como en su versión centralizada que ya había sido implementada anteriormente en aproximaciones previas a este trabajo.

En la Tabla 5.1 muestra los resultados del simulador de eventos discretos programado en lenguaje Rust. Las Redes de Petri que han sido testeadas siguen el patrón de ramas de la figura 5.1,con variaciones en la profundidad y en el número de ramas, para las versiones tanto centralizada como distribuida del simulador programado en lenguaje Rust. El tiempo de simulación para esta serie de pruebas es de 10 millones de ciclos de simulación. El entorno de simulación donde se han realizado las pruebas es el entorno *on premise* definido anteriormente en la sección 5.2.

SimBot Simulator	#br.	trans/br.	Events	Nodes	Events / sec	Exec. Time
Centr.	2	10 000	19 998 003	1	7 522 936	2.658 sec
Distr.	2	10 000	19 998 003	3	2 609 886	7.619 sec
Centr.	2	100 000	19 999 803	1	7 529 029	2.656 sec
Distr.	2	100 000	19 999 803	3	2 203 961	4.537 sec
Centr.	7	10 000	69 988 013	1	4 407 922	17.79 sec
Distr.	7	10 000	69 988 013	8	6 652 821	10.52 sec

Tabla 5.1: Simulación distribuida vs simulación centralizada con diferentes cargas de trabajo por simbot implementados en Rust.

En las primeras dos filas se muestran los resultados de una prueba de baja carga de dos ramas y diez mil transiciones por rama de profundidad. Como se puede apreciar se obtienen mejores resultados para el simulador centralizado. Sin embargo, cuando la profundidad se ve aumentada de 10.000 a 100.000 transiciones por rama, ya se comienza a apreciar una mejora en los resultados del simulador distribuido, a pesar de que todavía no superan a los de la versión centralizada. Esto significa que, si la profundidad de las ramas continúa siendo incrementada, la versión distribuida desarrollada en este trabajo será la idónea. Finalmente, se hizo otra prueba en la que el número de ramas se ve incrementado hasta 7 ramas, buscando encontrar una situación más propicia para el uso del simulador distribuido aumentando la carga de trabajo. En este caso, sí que se observan mejores resultados en la versión distribuida ya que el tiempo real de simulación (última columna) es menor, lo cual indica que si el número de ramas que se ejecutan en paralelo se ve aumentado, el tiempo de simulación de la versión centralizada aumenta sustancialmente y, por lo tanto, el uso del simulador distribuido para este caso es más recomendable. La sexta columna muestra el número total de eventos por segundo en la simulación. Se observa, que el número de nodos es uno por rama, y un nodo adicional que contiene la subred que sincroniza al comienzo y al final la simulación del resto de subredes. Este simbot contiene un número despreciable de transiciones y no es considerado para la tasa de eventos procesados por segundo. El número de eventos por segundo que se obtiene en el último caso con 7 ramas es aproximadamente 6 652 821, lo cual está muy cerca de las prestaciones obtenidas por el simulador centralizado en el primer caso pero ejecutando un modelo considerablemente más largo.

De todos estos resultados, se puede sacar las conclusiones de que a mayor carga de trabajo, es mucho más recomendable utilizar el simulador distribuido desarrollado e implementado en este trabajo tal y como se esperaba.

Para medir de manera independiente la eficiencia del simulador distribuido desarrollado e implementado en este trabajo, se han utilizado las siguientes variables

- P: Performance o eventos por segundo (tiempo real en segundos)
- E: Densidad de eventos o eventos por segundo simulado (tiempo o ciclos de simulación, valor máximo que alcanza el VLT)
- R: Ritmo de avance de simulación (tiempo de simulación en segundos/tiempo real en segundos)

, Estas, junto a las variables de lookahead (L, calculada en cada paso de simulación) y la latencia de comunicación calculada entre los nodos de simulación (τ), se obtiene el factor de acoplamiento (λ). El uso de esta métrica para medir el rendimiento del

simulador conservativo con implementación del paso de mensajes *NULL* o lookahead fue propuesto en[14]. En este artículo se indica que un valor de λ mayor que 10 y menor que 100 son valores apropiados de este factor de acoplamiento que mide el rendimiento según la carga de cada simbot.

$$\lambda = LE/\tau P$$

El valor de la latencia en los experimentos previos es de 350 microsegundos. El factor de acoplamiento es de $\lambda = 13{,}52$ para experimentos de 2 ramas y 10.000 transiciones por rama, y de $\lambda = 117{,}49$ para 2 ramas con 100.000 transiciones por rama. Esto significa que, en el primer caso, los simbots se quedan sin carga de trabajo suficiente a simular y en el segundo caso, una excesiva carga por simbot. El valor λ puede resultar útil para interpretar resultados, pero no es suficiente para evaluar globalmente una partición de modelo. En el último caso de 7 ramas y 10.000 transiciones por rama, con un valor $\lambda = 17{,}25$, muestra un apropiado tamaño de distribución y la carga de trabajo por partición es adecuada.

Todos estos resultados de las simulaciones recogidas en esta sección han sido incluidos en el artículo anual de investigación[15] publicado recientemente.

5.3.2. Pruebas de los mecanismos de balanceo de carga

Para las pruebas de los mecanismos de balanceo de carga se buscó un escenario donde hubiese un contexto desfavorable para la ejecución de la simulación distribuida, y este se viese alterado para conseguir incrementar el rendimiento de la simulación utilizando los mecanismos de balanceo de la carga de trabajo implementados como parte central de este trabajo. Es por ello que se decidió ejecutar las pruebas en un entorno de *cloud* híbrido donde se dispuso de nodos en *Google Cloud Platform* y un nodo *on premise* siendo la alta latencia de red entre ambos entornos el cuello de botella de las simulaciones distribuidas.

El experimento realizado consistió en ejecutar una simulación con dos nodos *cloud* y un nodo *on premise*, y en un instante predeterminado de la simulación, realizar el movimiento del código lefs del nodo *on premise* a uno de los nodos en *cloud* para aglutinar toda la carga de trabajo en nodos del mismo entorno y evitar las altas latencias de red entre entornos. Este movimiento de lefs se realizó a mitad de simulación para así poder diferenciar de una manera más sencilla el rendimiento antes y después del movimiento de lefs.

En la figura 5.2 se puede observar la progresión en el tiempo de la ejecución de

la simulación descrita anteriormente. En el eje x se ha colocado la variable de tiempo real de simulación, y en el eje y se dispone de la variable de número de ciclos o tiempo de simulación. Se observa como hasta los 8 segundos aproximadamente, se sigue una misma tendencia en la evolución del ritmo de simulación, y a partir de la mitad del tiempo de simulación (eje x) se produce un incremento del rendimiento tras el balanceo de carga finalizando la simulación completa a los 11 segundos aproximadamente. Esto indica que la segunda mitad de ejecución, tras el balanceo, se reducen las latencias de comunicación y tarda tan solo 3 segundos en ejecutar el tiempo de simulación, el cual en la primera mitad con un entorno híbrido tardó en ejecutar 8 segundos.

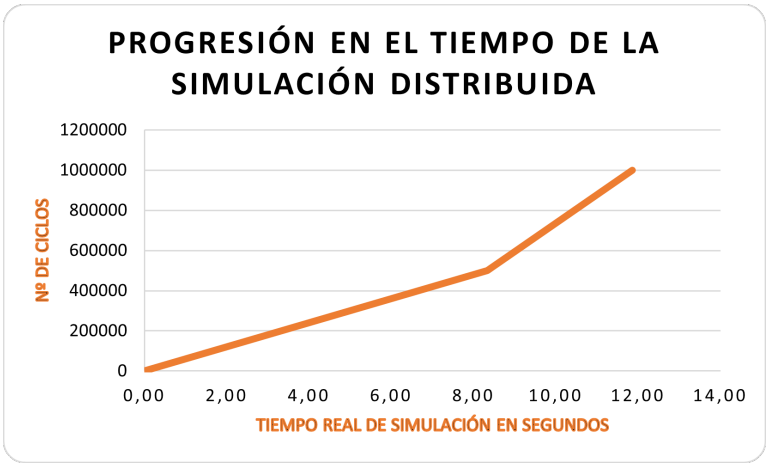

Figura 5.2: Evolución temporal de la prueba de balanceo de carga

Estos resultados ponen de manifiesto que el balanceo de carga puede incrementar el rendimiento del simulador distribuido, moviendo la carga de trabajo entre nodos para que, en todo momento, la ejecución se realice sobre un entorno idóneo.

Capítulo 6

Conclusiones

El principal objetivo que tenía la consecución de este trabajo era la implementación de un simulador distribuido que incluyese mecanismos de balanceo de carga y poder demostrar el potencial que tiene su implementación para mejorar las prestaciones de un simulador conservativo.
Viendo los resultados obtenidos, por un lado cabe destacar que el simulador distribuido desarrollado en el lenguaje Rust ya supera con creces tal y como se esperaba los valores de prestaciones obtenidos con el *framework* de simulación en Java. Además en la sección 5.3.1 ha quedado demostrado que, cuando se simulan modelos complejos y de gran tamaño, el simulador distribuido se comporta mejor que el simulador centralizado. De esta manera, queda demostrado tanto el potencial del lenguaje Rust como la escalabilidad de la simulación distribuida para programas donde se requieren altas prestaciones.Y, por otro lado, con la experimentación de los mecanismos de balanceo en un entorno de *cloud* híbrido donde existen altas latencias que limitan el rendimiento del simulador completo y una arquitectura heterogénea entre máquinas, se demuestra cómo la implementación de estos mecanismos son capaces de aumentar el rendimiento de las simulaciones conservativas, sobre todo destacando el hecho de que se haga sin detener los procesos de simulación de los nodos, y resultando eficiente, ya que los tiempos de movimientos de la carga han resultado ser insignificantes frente a la ganancia que suponen.

Por todo ello, el planteamiento inicial del trabajo ha sido cumplido con creces y mi valoración personal es que, a pesar de que se haya hecho duro en muchas ocasiones poder obtener los resultados esperados, finalmente se han conseguido con creces, siendo estos objeto de publicación en el artículo[15] y habiendo participado en la redacción de una parte del mismo. Además recientemente el artículo ha sido elegido como el mejor papel de la conferencia. Por todo ello, estoy muy satisfecho con el esfuerzo que ello ha supuesto y definitivamente ha merecido la pena el tiempo invertido.

Capítulo 7

Trabajo Futuro

Este trabajo desarrollado es una versión inicial de un proyecto que busca desarrollar un sistema completo de simulación con arquitectura heterogénea, alta escalabilidad y que pueda simular redes de tamaño considerable.

Para conseguir este hito, se proponen las siguientes tareas a realizar para el futuro:

- El simulador distribuido en la actualidad incorpora mecanismos de balanceo de carga pero **no** se ha desarrollado la lógica de la **toma de decisiones** que activen la puesta en marcha estos mecanismos de balanceo. En esta versión inicial se realizan movimientos de la carga de una manera predeterminada antes de poner en marcha el simulador, simplemente para demostrar que los mecanismos desarrollados funcionan. Una de las posibilidades que se proponen para analizar la carga de trabajo de los simbot y así decidir si hay que realizar un movimiento es centralizar el tercer hilo de ejecución *LEFs Movement Manager* para que tenga información de todos los simbots a la vez, y con los datos que este monitorice, vaya orquestando los movimientos de LEFs. Para la toma de decisiones de este monitor centralizado se podría seguir el modelo predictivo de Holt[16] que mide parámetros como la carga de CPU para determinar si hay que hacer una redistribución de la carga de trabajo.

- Otra mejora que se propone es la de implementar un **servidor centralizado de trazas** para la depuración del simulador distribuido, ya que en la actualidad, para depurar su funcionamiento, cada simbot o nodo de simulación muestra sus propias trazas locales. Esto ha sido un pequeño inconveniente a la hora de ejecutar simulaciones en el entorno de *cloud* híbrido que se está desarrollando en paralelo a este trabajo por el compañero Hayk Kocharyan, debido a que en el nodo maestro de ejecución se reciben las trazas del resto de nodos y aparecen desordenadas haciendo muy difícil la depuración sobre dicho entorno.

– También se propone mejorar los mecanismos de balanceo de carga, implementando la **partición de código LEFs** de una subred. En la actualidad, el balanceo de carga se realiza movimiento la estructura LEFs o subred que tiene un simbot. Sin embargo, al igual que se hace la partición del modelo en el proceso de compilación, se podría repetir esto mismo para que el propio simbot fuese capaz de partir su subred o código LEFs y enviar solo una parte del mismo a otro simbot. La posterior fusión del código LEFs particionado al código LEFs ya existente en un simbot ya ha sido implementado en esta versión inicial.

– Por último, se sugiere la modificación de la **asignación de *IPs***, evitando que se haga la misma en el proceso de compilación/adaptación del modelo y que esta se haga en el proceso de ejecución del simulador, consiguiendo un mayor desacoplamiento entre el proceso de compilación y el proceso de ejecución del simulador, permitiendo de esta manera, una mayor flexibilidad a la hora de poder lanzar simulaciones en paralelo o parametrizadas.

Capítulo 8

Bibliografía

[1] José Ángel Bañares and José Manuel Colom. Model and simulation engines for distributed simulation of discrete event systems. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 77–91. Springer, 2018.

[2] Unai Arronategui, José Ángel Bañares, and José Manuel Colom. Towards an architecture proposal for federation of distributed des simulators. In *GECON 2019 - International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 197–210. Springer, 2019.

[3] José Ángel Bañares y José Manuel Colom Unai Arronategui. A mde approach for modelling and distributed simulation of health systems. *17th International Conference, GECON Online*, September 15-17, 2020.

[4] Sergio Herrero Barco. Desarrollo de un framework de simulación de sistemas de eventos discretos complejos. Trabajo fin de grado, Universidad de Zaragoza, 2020.

[5] Hayk Kocharyan. Automatización del despliegue de simulaciones distribuidas en cloud híbrido. Trabajo fin de grado, Universidad de Zaragoza, 2021.

[6] Rust - el lenguaje de programación. `https://www.rust-lang.org/es`. Accessed: 2021-09-15.

[7] C++, a brief description - the c++ resources network. `https://www.cplusplus.com/info/description`. Accessed: 2021-09-19.

[8] Visual studio code - code editing. redefined. `https://code.visualstudio.com`. Accessed: 2021-09-15.

[9] Rust - visual studio marketplace. `https://marketplace.visualstudio.com/items?itemName=rust-lang.rust`. Accessed: 2021-09-07.

[10] José Luis Briz and José Manuel Colom. Implementation of weighted place/transition nets based on linear enabling functions. In *International Conference on Application and Theory of Petri Nets*, pages 99–118. Springer, 1994.

[11] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. `http://www.rfc-editor.org/rfc/rfc793.txt`.

[12] Serde. `https://serde.rs`. Accessed: 2021-09-14.

[13] Javascript object notation (json) – ruby module. `https://ruby-doc.org/stdlib-2.6.3/libdoc/json/rdoc/JSON.html`. Accessed: 2021-09-16.

[14] Andras Varga, Yasar Ahmet Sekercioglu, and Gregory K Egan. A practical efficiency criterion for the null message algorithm. In A Verbraeck and V Hlupic, editors, *Simulation in Industry: Proceedings of the 15th European Simulation Symposium (ESS 2003)*, pages 81 – 92, 2003.

[15] Paul Hodgetts, Hayk Kocharyan, Fidel Reviriego, Álvaro Santamaría, Unai Arronategui, José Ángel Bañares, and José Manuel Colom. Workload evaluation in distributed simulation of dess. *18th International Conference, GECON Online*, September 21-23, 2021.

[16] Robson Eduardo De Grande, Azzedine Boukerche, and Raed Alkharboush. Time series-oriented load prediction model and migration policies for distributed simulation systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):215–229, 2017.

[17] Bincode - a compact encoder/decoder implementation in rust. `https://github.com/bincode-org/bincode`. Accessed: 2021-09-17.

[18] Using message passing to transfer data between threads. `https://doc.rust-lang.org/book/ch16-02-message-passing.html`. Accessed: 2021-09-12.

[19] Crossbeam channel rust crate. `https://github.com/crossbeam-rs/crossbeam/tree/master/crossbeam-channel`. Accessed: 2021-09-12.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Diagramas de estado de los componentes de un simbot con mecanismos de balanceo de carga

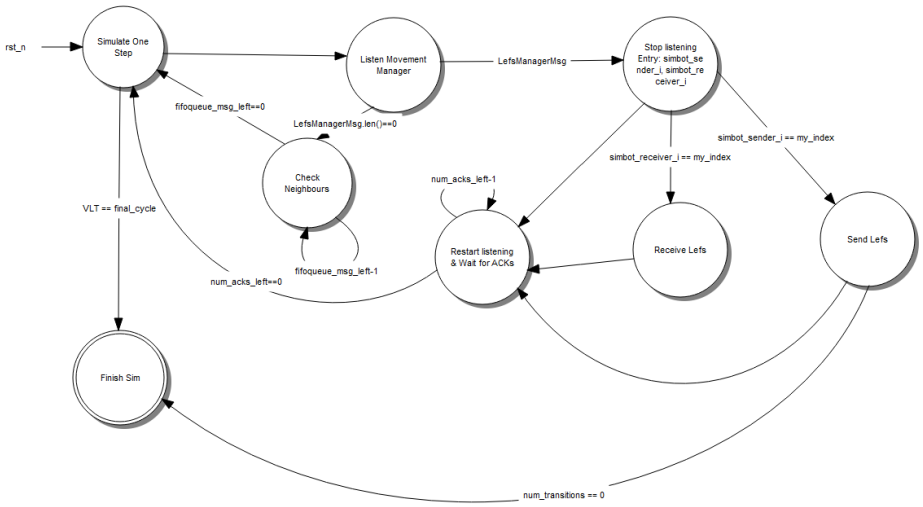

Figura A.1: Máquina de estados del hilo de simulación del algoritmo de movimiento de lefs

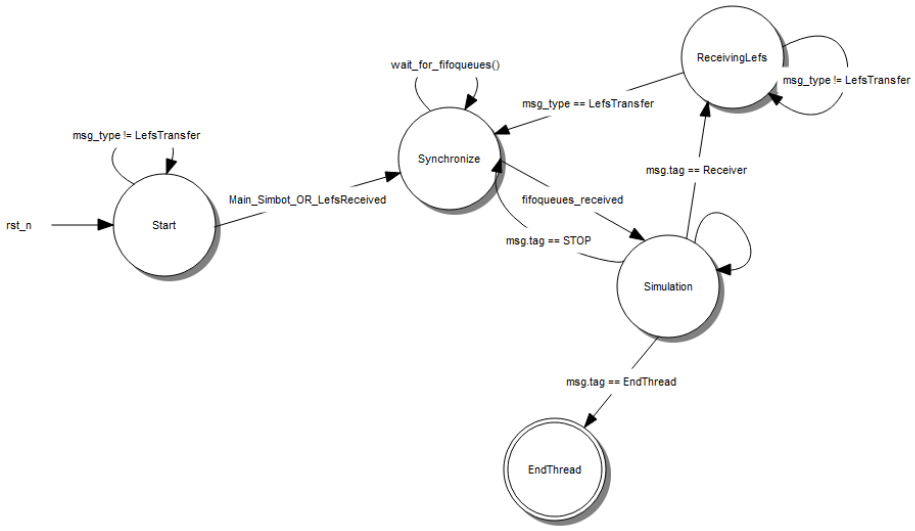

Figura A.2: Máquina de estados del hilo Mailbox del algoritmo de movimiento de lefs

Anexos B

Canales asíncronos de comunicación en Rust

Para la implementación de la comunicación entre hilos en Rust, se requiere de un intermediario para que esto se produzca. La solución más sencilla que se propone por la documentación oficial del lenguaje[18] es el uso del paso de mensajes mediante canales asíncronos. De esta manera se evita la memoria compartida entre threads, lo cual incurre en problemas de concurrencia y en el uso de mecanismo para garantizar el acceso a memoria en exclusión mutua.

En la documentación de referencia, se sugiere el uso de los canales de la librería estándar de Rust *std::mpsc*. Estos canales son multiproductor consumidor, lo que significa que solo se permite un extremo de recepción a ese canal, es decir, solo uno de los dos threads de ejecución va a ser capaz de recibir los mensajes almacenados en el búfer de ese canal. Esta solución no se adaptaba completamente a las necesidades de nuestro simbot, por lo que se buscó una alternativa más viable.

La solución elegida fue la utilización de *crossbeam channels*[19]. Es una alternativa a los canales *mspc* de la librería estandar con un mejor rendimiento con la diferencia de que son canales multiproductor multiconsumidor, lo cual permite replicar tanto el extremo de envío como el extremo de recepción. Esto último es muy importante porque permite que los dos hilos de ejecución sean capaces de recibir y enviar mensajes. Además su implementación es muy similar a la de la librería estándar al ser una adaptación mejorada de la misma.

Anexos C

Mecanismos de sincronización de los nodos distribuidos

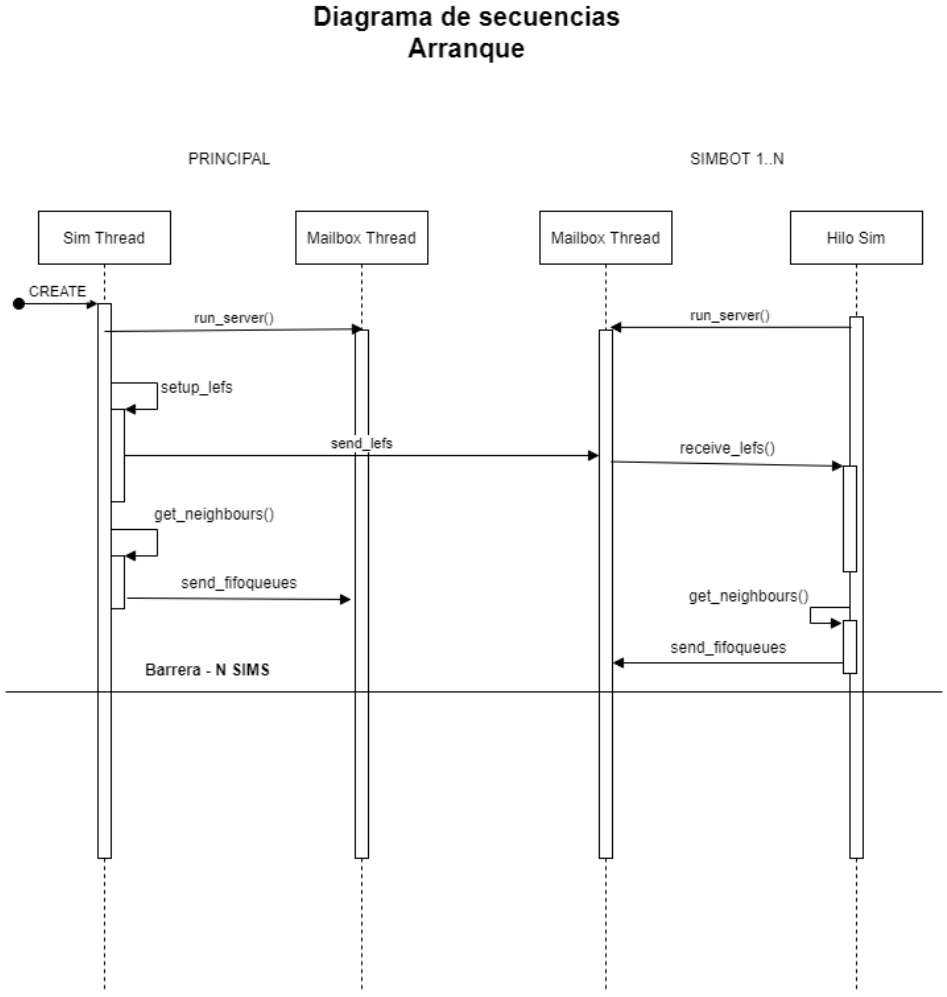

Figura C.1: Diagrama de secuencias del arranque de la simulación distribuida

Anexos D

Tipos de mensajes que se intercambian los simbots

La estructura de datos ***Message*** ha sido creada para la encapsulación de los eventos y otro tipo de mensajes, para después que estos sean serializados mediante la librería *bincode*[17], y enviados a través de la red mediante *TCP*. En el otro extremo de la comunicación, el Mailbox del simbot destinatario deserializa el mensaje y lee el flag del mensaje, para poder filtrar el mensaje y enviarlo por la cola *FIFO* de recepción correspondiente o enviar un mensaje de control al motor de simulación a través de sus canales asíncronos de comunicación.

Los **flags** de estos mensajes son los que indican qué tipo de mensaje es y los valores que pueden tomar son los siguientes:

- ***Start***: Es el mensaje que se utiliza al comienzo de la ejecución de un simbot para sincronizarse en el algoritmo de barrera distribuida. Para consultar el funcionamiento de esta sincronización, ir al Anexo C. Este es un mensaje de control que se envía por el canal asíncrono que comunica el mailbox con el motor de simulación.
- ***Event***: Mensaje que contiene un evento externo. Es el más frecuente y se almacena en las colas *FIFO* de recepción correspondiente que el motor de simulación comprueba en el bucle de simulación conservativa (`Figura 4.3 - línea 4`).
- ***Lookahead***: Mismo comportamiento que un evento externo, pero a diferencia de este, no se agrega posteriormente a la FUL para que sea procesado.
- ***Finish***: Es el mensaje de control que se utiliza al final de la ejecución del simulador, en el algoritmo de barrera distribuida para sincronizar los simbots en su finalización.

- ***EndThread***: Es el último mensaje de control que se envía. Se usa para que la interfaz de comunicación deje de recibir mensajes y finalice su hilo de ejecución de manera segura.

- ***Ping***: Es un tipo de mensaje que solo envía el simbot principal(índice 0) y solo recibe el simbot de índice 1. Es un mensaje para calcular la latencia de conexión entre dos simbots y se utiliza para las estadísticas del simbot. Se reenvía inmediatamente a su emisor sin pasar por ninguna cola de recepción ni canal asíncrono para evitar desvirtuar ese valor de la latencia.

- ***Latency***: Tipo de mensaje que envía el simbot principal al resto para que todos tengan el valor de la latencia una vez calculado y poder mostrar las métricas locales.

Además de estos mensajes, una vez incorporados los mecanismos de balanceo de carga, se han añadido los siguientes mensajes de control:

- ***Stop***: Mensaje de control que recibe el Mailbox por parte del motor de simulación indicándole que comienza un paso de movimiento de LEFs. Si recibe este mensaje de control es que se trata de un simbot vecino.

- ***Receiver***: Mensaje de control que recibe el Mailbox por parte del motor de simulación indicándole que comienza un paso de movimiento de LEFs. Si recibe este mensaje de control es que se trata del receptor del LEFs.

- ***SyncStop***: Mensaje de control que realiza la misma función que ***Start*** o ***Finish*** pero en este caso se intercambia cuando se van a sincronizar los nodos de simulación tras un balanceo de carga.

También se ha creado la estructura de datos *LefsTransfer* que encapsula un LEFs junto a la información *ip:port* del simbot emisor para su posterior serialización y envío al igual que una estructura de tipo *Message*

Anexos E

Implementación de algoritmos conservativos en Rust

A continuación se muestra un fragmento de código en lenguaje Rust que implementa el algoritmo de simulación distribuida con sincronización conservativa.

```
    // simulation loop
    while self.local_clock < final_cycle {

        // CHECK NEIGHBOURS
        // Check every FIFO queue and wait until
        // all of them are not empty.
        // Using crossbeam channel
        LVHT = check_neighbours(self, &fifoqueues_r);

        initialize_simbots_to_send(self, lefs);

        if LVHT > final_cycle{
            LVHT = final_cycle
        }
        // SIMULATION STEP UNTIL NEXT LVHT
        self.simulate_one_step(lefs, LVHT);

        send_lookaheads(self, lefs, false);

        if LVHT == final_cycle{
            break;
        }
    }
    finish_simulation(self, index, &rx_sim, &list_nodes, lefs);
```

Y en el siguiente fragmento de código se muestra el mismo algoritmo incluyendo los mecanismos de balanceo de carga. Observe las diferencias:

```
    // simulation loop
    while self.local_clock < final_cycle {
        match listen_lefs_movement_manager(&rx_lefs)?{
            Some(msg) => {
                // LEFS MOVEMENT STEP
                let (lvht_new, fifoqueues_r_new) = lefs_movement(
                    self, msg, index, &list_nodes, &mut fifoqueues_r,
                    lefs, &fifoq_channel_sender, &rx_lefst
                )?;

                LVHT = lvht_new;
                fifoqueues_r = fifoqueues_r_new;
                sync_restart( index, &rx_sim, &list_nodes)?;
            },
            // NO LEFS MOVEMENT STEP!
            // CHECK NEIGHBOURS
            // Check every FIFO queue and wait until
            // all of them are not empty.
            // Using crossbeam channel
            None => LVHT = check_neighbours(self, &fifoqueues_r)
        }

        initialize_simbots_to_send(self, lefs);

        if LVHT > final_cycle{
            LVHT = final_cycle
        }

        // SIMULATION STEP UNTIL NEXT LVHT
        self.simulate_one_step(lefs, LVHT, &tx_sim)?;

        send_lookaheads(self, lefs, false);
        if LVHT == final_cycle{
            break;
        }
    }
    finish_simulation(self, index, &rx_sim, &list_nodes, lefs);
```