Universidad Zaragoza

Trabajo Fin de Máster

Diseño e implementación de un simulador distribuido de alta escala de sistemas de eventos discretos

Design and Implementation of a Large Scale Simulator of Discrete Event Systems

Autor

Fidel Reviriego Navarro

Director

Unai Arronategui Arribalzaga

Codirector

José Ángel Bañares Bañares

Universidad de Zaragoza / Escuela de Ingeniería y Arquitectura
Máster Universitario en Ingeniería Informática

2021

RESUMEN

En la actualidad los sistemas tienen cada vez un mayor tamaño y complejidad que requiere la utilización de modelos que describan su comportamiento. La solución a este problema consiste en la simulación de sistemas para reproducir el comportamiento de un sistema dinámico mediante la interpretación de un modelo. En este trabajo se utilizan sistemas de eventos discretos que se caracterizan porque su estado de funcionamiento se ve alterado en instantes puntuales de tiempo.

Sin embargo, para poder simular sistemas de gran tamaño y complejidad es necesario que la simulación y las herramientas que se utilicen sean escalables. Para lograrlo se realiza el desarrollo e implementación de un simulador distribuido de sistemas de eventos discretos modelados por redes de Petri. El simulador utiliza el método de sincronización conservativo mediante el uso de los mensajes de lookahead.

Debido a que los modelos a simular tienen un gran tamaño el simulador debe tener unas prestaciones adecuadas al realizar simulaciones de alta escala. El principal mecanismo del que dependen sus prestaciones es del lookahead, el cual es de vital importancia y depende del modelo a simular. En el trabajo se realiza un análisis profundo de la gestión del lookahead, tanto de su cálculo como de cuándo y de qué forma se debe realizar la transmisión del mismo. Se utilizan dos métodos diferentes: cálculo dinámico de lookahead con envío al terminar de simular y cálculo con vector de lookahead con solicitud de lookahead. En los dos casos se analiza para qué tipos de simulaciones son más adecuados y se realizan pruebas en un entorno *on premise* de la Universidad de Zaragoza.

Con la mejora del rendimiento que se consigue con los métodos anteriores se pueden simular modelos de mayor tamaño. No obstante, no es suficiente para modelos de gran escala que necesiten el uso de un número elevado de máquinas. Para solucionarlo se propone una solución innovadora en este campo que se ha llamado simulación a nivel de regiones. Consiste en dividir la simulación en regiones y propagar de una forma rápida la información de una región a otra para acelerar la simulación en modelos de alta escala. Una región sería un agrupamiento de procesos de simulación, los cuales se ejecutan cada uno en una máquina.

En conclusión, el simulador distribuido desarrollado es una siguiente versión al que había antes del trabajo, realizado por el grupo de investigación COSMOS. Con este trabajo se ha logrado mejorar las prestaciones del simulador distribuido ofreciendo diferentes mecanismos de gestión del lookahead que presentan un mayor o menor rendimiento en función del modelo a simular. También se ha presentado el diseño de un método para permitir simulaciones de alta escala que tengan un coste temporal aceptable. Además, se ha colaborado en la publicación de un artículo de investigación que ha sido elegido como mejor artículo de la conferencia en la GECON.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Motivación

En la actualidad los sistemas tienen una alta complejidad lo que requiere la utilización de modelos que describan su comportamiento. Como herramienta para el diseño y desarrollo de estos sistemas se utiliza la simulación de sistemas de eventos discretos de grandes dimensiones modelados con redes de Petri. La simulación no es posible realizarla de forma centralizada por el gran tamaño de estos modelos, haciendo necesario la utilización de motores de simulación distribuidos para comprender y analizar su comportamiento. Así se consiguen mejores prestaciones al aprovechar la concurrencia real del procesamiento.
La simulación distribuida de sistemas de eventos discretos es una estrategia conocida pero la gestión de eventos con estampillas temporales que viajan por la red introduce dificultades ante el retraso de eventos o la llegada desordenada de eventos. Es un problema de sincronización en el que la simulación distribuida se debe realizar en un orden correcto de tiempo. Se pueden diferenciar dos estrategias de sincronización: conservadora y optimista. Está claro que la optimista limita mucho por la gestión del almacenamiento de las historias de la simulación hasta el estado en el que se detecta un evento perdido de cara a recuperar un estado pasado seguro. Sobre todo penaliza o impide completamente la introducción de mecanismos de balanceo de carga en la arquitectura distribuida y el balanceo de las partes de modelo a simular en cada uno de los procesos de simulación (SP). Esto hace que a priori se descarten las técnicas optimistas y se adopte como estrategia la simulación conservadora.

1.2. Objetivos y alcance

El objetivo del proyecto es la construcción de un simulador distribuido con estrategia de gestión de los eventos conservadora, explotando los mecanismos de mejora de las prestaciones de la simulación conservadora. La estrategia conservadora requiere que no existan eventos etiquetados para un tiempo anterior al tiempo al que se quiere avanzar el reloj en un proceso de simulación o simbot. Esta táctica de aseguramiento está basada en el intercambio de información entre los procesos de simulación acerca de hasta qué valor de tiempo se pueden avanzar los relojes teniendo en cuenta los eventos que producirán y enviarán en un futuro los distintos procesos de simulación.
El cálculo de estas predicciones se puede hacer porque de una manera implícita o explícita podemos referirnos al modelo de redes de Petri para obtener esta información. La obtención de esta información es lo que se llamará cálculo del lookahead. Este cálculo es crucial para introducir paralelismo en la simulación, cuyo rendimiento depende principalmente del lookahead. En modelos de gran tamaño el lookahead es la clave para que su simulación se pueda realizar en un tiempo razonable.

En este proyecto se analizan algunas de las estrategias de cálculo y envío de lookahead

y se aporta una solución innovadora para la simulación de alta escala, que se ha llamado simulación a nivel de regiones.

1.3. Contexto

Este proyecto ha recibido una de las becas "Prácticas con TFM del I3A" y se ha realizado en el grupo de investigación Computer Science for Complex System Modeling (COSMOS), cuya labor se centra en el desarrollo de sistemas distribuidos complejos.
El contexto y antecedentes de este trabajo parte de los artículos de investigación presentados por los profesores Unai Arronategui, José Ángel Bañares y José Manuel Colom de la Universidad de Zaragoza [2, 4]. En estos artículos 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 redes de Petri.
Mediante diferentes proyectos de varios estudiantes como el Trabajo de Fin de Grado de Sergio Herrero Barco [5] de la Universidad de Zaragoza, se realizó una primera implementación del compilador y los servicios básicos para la simulación distribuida en Java. En este punto se disponía de una solución ad hoc para una simulación distribuida para una red específica utilizada para comenzar a probar el sistema.
Después en las prácticas del máster que hice antes de este trabajo realicé un simulador distribuido en Java más completo, permitiendo simular redes de Petri de grafos marcados. Este simulador estaba diseñado para redes de Petri de pequeño tamaño, y la gestión del lookahead era muy básica y de bajo rendimiento ya que no se había analizado.
Una vez situados los antecedentes, este trabajo consiste en el desarrollo e implementación de un simulador distribuido en el lenguaje de programación Rust [6], con un análisis profundo de la gestión del lookahead con el objetivo de simular redes de alta escala.
Además, este trabajo ha sido realizado en paralelo al Trabajo de Fin de Grado de Álvaro Santamaría [7], del cuál se ha utilizado una parte de su simulador implementado en Rust como código base.

1.4. Herramientas y tecnologías usadas en el proyecto

El simulador distribuido ha sido implementado en el lenguaje de programación Rust. El uso de este lenguaje era un requisito impuesto por el grupo de investigación en el que se realizó el trabajo. El motivo por el que en su momento se seleccionó este lenguaje de programación es debido a que permite obtener unas elevadas prestaciones, similares a las que se obtienen en otros lenguajes como C++. Sin embargo, Rust tiene un modelo más seguro de gestión de memoria y concurrencia lo que favorece el desarrollo del simulador distribuido.
Como entorno de desarrollo se ha utilizado Visual Studio Code [8] con la extensión de Rust [9].

Se ha utilizado GitHub para el control de versiones y compartir el código mediante una organización que se creó llamada simbots-swarm. En esta organización hay múltiples repositorios para mantener separadas distintas versiones del simulador, junto a otros trabajos del grupo de investigación relacionados con el simulador distribuido.

1.5. Estructura del documento

La estructura de la memoria se basa en las fases en las que se ha dividido el desarrollo del proyecto, empezando por las secciones de introducción y estado del arte donde se explica el contexto del problema a resolver, los principales objetivos y la situación actual.
El capítulo 3 trata sobre el análisis y el diseño del sistema, explicando las diferentes posibilidades que hay y las decisiones tomadas.
El capítulo 4 es sobre la implementación realizada y el capítulo 5 explica cómo se ha validado el sistema y los resultados obtenidos.
Por último, el capítulo 6 contiene las conclusiones, el posible trabajo futuro que puede realizarse y una valoración personal sobre el trabajo realizado.

En el anexo de este documento aparecen algunos aspectos más detallados y otras opciones de diseño que se analizaron y fueron descartadas por determinados motivos.

2. Simulación distribuida de sistemas de eventos discretos

2.1. Los principios de la simulación distribuida y los problemas fundamentales

La simulación computacional es el funcionamiento de un programa computacional (simulador) que representa el comportamiento a lo largo del tiempo de otro sistema. Utiliza modelos matemáticos como sistema de representación para estudiar la evolución temporal de determinados elementos y sus relaciones. En este trabajo se utilizan redes de Petri para modelar el comportamiento del sistema.
Según la forma en que cambien las variables de estado a lo largo del tiempo se puede diferenciar entre simulación de tiempo continuo y discreto. Si es de tiempo continuo puede haber cambios en todos los puntos durante la simulación mientras que si es de tiempo discreto solo ocurren en tiempos discretos de tiempo. En este último caso el tiempo puede avanzar de forma escalonada con incrementos fijos de tiempo o de manera no regular si la simulación está dirigida por eventos.
En este trabajo se utiliza la simulación de sistemas de eventos discretos, donde un evento es un suceso que provoca una modificación de las variables de estado que representan el estado actual del sistema. Estos eventos están etiquetados con una marca temporal que permite identificar el tiempo en que es procesado.

La simulación distribuida es la ejecución de simulaciones sobre computadores que no comparten reloj y se comunican a través de una red de comunicación de datos. En este trabajo cada computador que simula una parte del modelo se llama proceso de simulación (SP) [1] o simbot, y lo que simulan es una subred. Para lograr una simulación global correcta del modelo cada SP debe procesar los eventos en el orden correcto de estampilla de tiempo global, por lo que debe actuar de forma sincronizada respetando la causalidad global.
La decisión de realizar simulaciones distribuidas en lugar de centralizadas se debe a que para modelos de gran tamaño no es posible realizarlo en una sola máquina por el tamaño en memoria requerido, necesario para representar el estado. Otro aspecto es el tiempo de la simulación que al utilizar múltiples máquinas permite reducirlo obteniendo mejores prestaciones.

Al utilizar la red de comunicación los mensajes con eventos pueden llegar desordenados o no llegar en el tiempo esperado. La simulación distribuida debe garantizar la restricción de causalidad local para que la ejecución sea equivalente a la correspondiente simulación centralizada en un solo computador. Los eventos contenidos en los

[1] Utilizamos en la memoria SP en lugar del tradicional LP (Proceso lógico), porque SP implica un motor de simulación y la partición del código. La diferencia principal es que un LP es compilado y por lo tanto sólo es posible balancear la carga moviendo todo el LP.

mensajes entrantes deben ser procesados en el orden correcto de estampillas de tiempo globales de eventos, pudiendo utilizar dos estrategias:

- **Sincronización conservadora**: Evitar violar la restricción de causalidad local. Se espera hasta que sea seguro que no van a llegar eventos con tiempo menor para no procesar eventos con estampilla de tiempo fuera de orden.
- **Sincronización optimista**: El tiempo avanza permitiendo violaciones de causalidad local. Durante la ejecución al detectar eventos procesados fuera de orden se recupera el estado del sistema hasta ese evento mediante un mecanismo de rollback.

En este trabajo se opta por la sincronización conservadora debido a que se van a utilizar modelos de gran tamaño. Se ha descartado utilizar la sincronización optimista por el elevado consumo de memoria que puede tener al guardar la evolución del estado del sistema, necesario para recuperar un estado si se detecta un evento procesado fuera de orden. Además, en otros trabajos del grupo de investigación también se utiliza la sincronización conservadora para aspectos como el balanceo de carga.
La sincronización conservadora tiene distintas soluciones como la evitación de bloqueo con mensajes vacíos o nulos, la detección de bloqueos y recuperación, o algoritmos síncronos. La solución más conocida es la utilización de mensajes nulos que es la que se utiliza en este trabajo.

2.2. Sincronización conservadora con evitación de bloqueo (lookahead)

En la sincronización conservadora un SP solo puede procesar eventos seguros, es decir, eventos hasta un tiempo de simulación para los cuales se ha garantizado que el SP no recibirá eventos externos con marca de tiempo menor al tiempo de simulación actual (reloj local, LVT). Además, todos los eventos (internos y externos) deben ser procesados en orden cronológico. Esto garantiza que el flujo de mensajes producido por un SP esté en orden cronológico. También es necesario preservar el orden de los mensajes enviados entre los SP (estructura FIFO) para garantizar que no puedan llegar mensajes fuera de orden cronológico [1].
La solución consiste en que un SP antes de simular hasta un cierto tiempo debe tener información de los SPs que tengan relación con él. Para ello debe esperar un mensaje de estos SPs que tengan información del siguiente evento que le van a enviar y en qué tiempo. Se puede producir un problema de bloqueo mutuo si un SP está esperando a que llegue un mensaje de otro SP pero este SP ya está bloqueado porque también está esperando algún mensaje. La figura 1 muestra este problema en el caso de 3 SPs.

Para solucionar este problema se utilizan mensajes nulos que son mensajes que envían los SPs y no llevan contenido pero tienen una marca de tiempo que indica el tiempo mínimo hasta el cual ese SP no va a enviar ningún evento. Estos mensajes son los valores de lookahead (L), que se caracterizan por ser mensajes de control que transmiten información temporal.

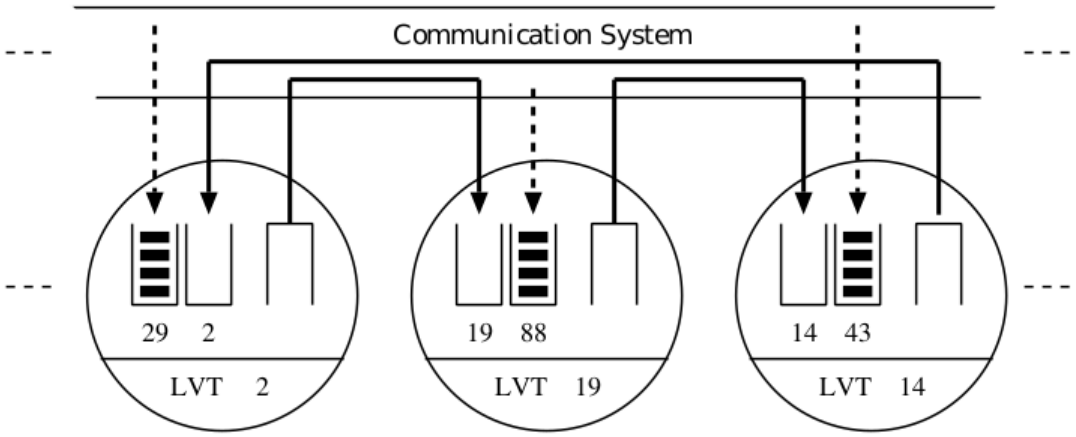


Figura 1: Bloqueo mutuo en simulación conservadora (extraída de [1])

Existen diferentes estrategias para reducir la sobrecarga causada por la transmisión de mensajes nulos [3], ya que la comunicación SP a SP en simulación distribuida tiene un coste importante. Estas estrategias consisten en que o los SP envían los mensajes nulos en un determinado momento o los SP solicitan mensajes nulos (On-demand Null-Message Sending, REQ).
En el primer caso, se pueden enviar los mensajes nulos cada vez que se avanza el tiempo (Standard Chandy-Misra-Bryant protocol, STD), cuando expire un cierto temporizador (Timeout-based Null-message Sending, TIM) o cuando un SP no tenga nada para simular y vaya a entrar en un estado de parada (Deadlock Avoidance Null-Message Sending, BLO), entre otros casos. En el artículo citado se analizan otras opciones aunque existen otras alternativas que son menos habituales.
La variedad de alternativas se debe a la búsqueda de un equilibrio entre tener valores actualizados de los otros SP sin sobrecargar la red con mensajes nulos que provoquen una pérdida de prestaciones.
Otra opción es el protocolo de simulación ideal (Ideal Simulation Protocol, ISP) que utiliza una traza de una ejecución de simulación previa para determinar si un SP puede avanzar con seguridad. Es la ejecución ideal al no tener sobrecarga de sincronización por lo que sirve como referencia para comparar simuladores.

2.3. Modelo basado en redes de Petri

Hay una gran variedad de tipos de redes de Petri según su aplicación en problemas específicos. Existen distintas clasificaciones para ellas pudiendo diferenciar a alto nivel entre redes de Petri clásicas y las extensiones que se han ido creando. Las redes de Petri clásicas son un grafo dirigido con dos tipos de nodos: lugares y transiciones. Entre los nodos están los arcos dirigidos que unen las transiciones con los lugares y viceversa. Los lugares tienen asociados elementos llamados tokens o marcas que definen el estado de la red y sirven para establecer las condiciones de disparo de una transición. Dentro de estas redes hay distintas variantes según las restricciones que se apliquen a la red.
Estas redes de Petri no se pueden aplicar en algunos problemas por lo que fueron surgiendo extensiones. La más común es la de tiempo aunque hay otras como las jerárqui-

cas y las coloreadas. Con las redes de Petri clásicas no es posible manejar tiempo por lo que se agrega al modelo el concepto de tiempo para poder describir el comportamiento temporal del sistema. Dos de las clases de redes de Petri temporizadas más habituales son las redes de Petri de tiempo determinista y redes de Petri de tiempo estocástico. Estos tiempos hacen referencia al tiempo de disparo de las transiciones.

En este trabajo las redes de Petri que se van a utilizar son binarias y grafos marcados en los que cada lugar tiene exactamente un arco de entrada y un arco de salida. Por tanto no tienen conflictos en el disparo de transiciones. En este tipo de redes solo hay un tipo de ejecución con unas ciertas proporciones. El motivo de usar grafos marcados es porque hay un campo lo suficientemente amplio de workflows que utilizan grafos marcados, junto a que el cálculo del lookahead es más sencillo, y en una primera aproximación al problema se ha elegido esta restricción. Como trabajo futuro se podría ampliar las redes de Petri permitidas incluyendo tiempos estocásticos o conflictos, entre otros aspectos.

La utilización de redes de Petri para representar el modelo a simular [2] permite mejorar el rendimiento al utilizar un mecanismo basado en la caracterización de la sensibilización de una transición mediante una única función, denominada Linear Enabling Function (LEF) [10]. En el anexo I se explica este mecanismo. De esta manera en la simulación se utiliza una representación basada en LEFs en lugar de la especificación clásica de lugar/transición.
El uso de LEFs se debe a que el coste computacional teórico de una simulación centralizada es lineal respecto al tamaño de la red, mientras que con la representación lugar/transición el coste es exponencial. Por tanto el uso de redes de Petri con LEFs es vital para poder simular redes de gran tamaño.

En una simulación distribuida, una red de Petri se divide en subredes que se simulan en diferentes SPs. Estos SPs están conectados a otros SPs según las subredes que simulan, de forma que las salidas de transiciones de una subred pueden estar conectadas a transiciones de otros SPs. La transición que tenga algún lugar de salida externo, hacia otro SP, se denomina transición de salida mientras que la transición que tenga ese lugar como entrada, una entrada que venga del exterior, se llama transición de entrada.
Por ejemplo en la figura 2 se puede ver una red de Petri y la red equivalente en tres subredes. En la subred 0 (central) la transición t0 sería una transición de salida porque tiene dos lugares de salida que son lugares de entrada de la transición t1 de la subred 1 (izquierda) y de la transición t2 de la subred 2 (derecha), que son transiciones de entrada.

El concepto de lookahead en redes de Petri es el mínimo tiempo en que una transición de salida va a generar una marca. Esto quiere decir el tiempo en que como mínimo los lugares de salida de una transición de salida van a tener una marca, y por tanto el tiempo en que los correspondientes lugares de entrada de los SP destino van a recibir una marca.

Para ilustrar los algoritmos propuestos, utilizaremos la red de Petri de la figura 3, que es una red formada por un número de subredes secuenciales que tienen sincronizado el

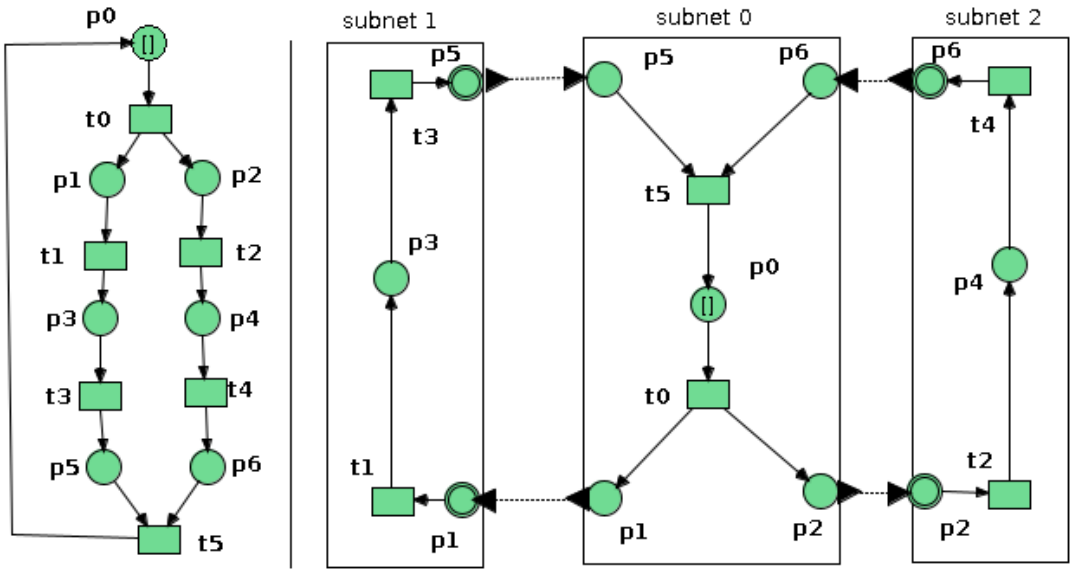


Figura 2: Red de Petri (izquierda) y su división en tres subredes (derecha).

principio y final de su ejecución. Esta red está formada por múltiples ramas secuenciales que en la simulación distribuida se dividen en subredes de forma que cada proceso simula una de estas subredes. Además una de las subredes está compuesta por las dos transiciones de sincronización que son la primera y la última de la figura. En la figura se muestra una red con h ramas que serán subredes, con v transiciones en cada una de las ramas.
Se utiliza este tipo de red de Petri porque es un grafo marcado que tiene una fácil distribución en subredes y permite detectar de una forma rápida posibles fallos en la simulación.

Sin embargo, si se utilizan otras redes de Petri su análisis es más complejo. Respecto a las transiciones de entrada, se deben tener en cuenta todas de la subred puesto que su comportamiento depende de esos factores. Si hay varias transiciones de entrada, algo común es tener varios posibles caminos (secuencias de disparos de transiciones) en vez de un solo camino. Se deben analizar todos los caminos para determinar cual es que se debe tener en cuenta a la hora de obtener información precisa del valor de lookahead.
En este caso, al analizar una subred de Petri se deben valorar los caminos posibles y según el valor del LEF de las transiciones que se analicen, determinar cuál es el camino que se debe elegir. Si una transición T_i tiene X entradas, tiene por tanto X caminos que se deben analizar. Analizar un camino consiste en calcular el tiempo en que como mínimo generará una marca en su lugar final, el que está conectado a la transición T_i. Una vez analizados los X caminos, en función del valor de LEF se elegirá cual es el camino que se debe tener en cuenta. Si el valor de la estructura LEF es Z, es decir precisa de Z marcas, el camino que se toma como referencia es el Z-ésimo camino, teniendo todos los caminos ordenados de menor a mayor tiempo.
Por ejemplo, si una transición tiene tres caminos posibles (tres lugares de entrada) de tiempos [2,3,5], si el valor de su estructura LEF es dos (precisa de dos marcas), el tiempo a considerar es 3. Se debe a que al necesitar dos marcas en el tiempo 2 como

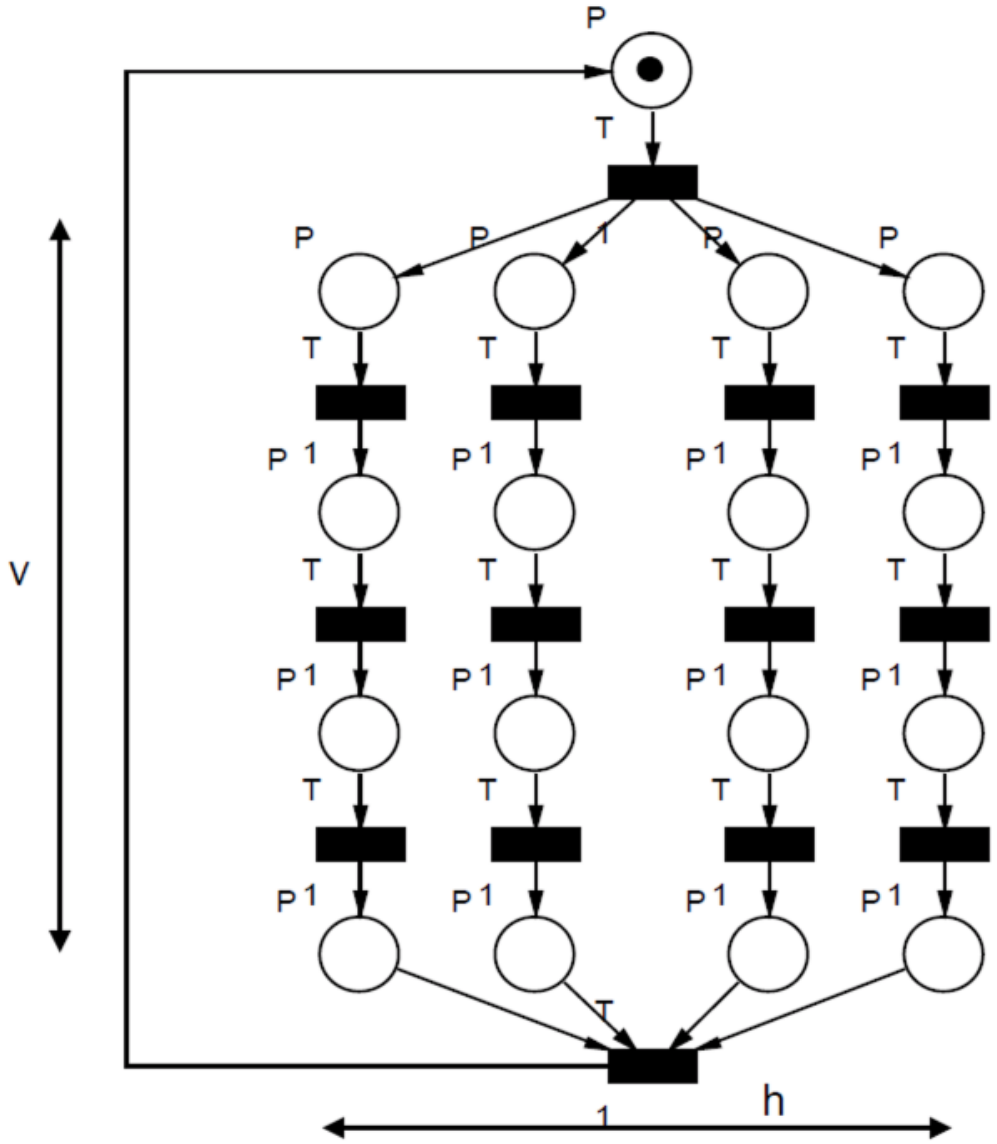


Figura 3: Red de Petri de ramas sincronizadas.

mucho habrá recibido una marca lo que es insuficiente, mientras que en el tiempo 5 podría haber recibido tres marcas que son más de las necesarias. Por tanto hasta el tiempo 3 como mínimo no habrá recibido las marcas suficientes como para estar sensibilizada.
El objetivo de este cálculo es obtener el lookahead de una transición de salida pero puede requerir el cálculo del tiempo para otras transiciones dado que dentro de un camino pueden surgir otros caminos.
Incluso aunque solo haya una transición de entrada, por lo que solo hay un camino en la subred, si esta tiene varios lugares de entrada pertenecientes al interfaz de la subred se deben tener en cuenta todos ellos puesto que es información que se recibe de otras subredes.

Otro caso es si la subred de Petri tiene varias transiciones de salida, lo que afecta

al cálculo de lookahead ya que es necesario calcular el valor para cada transición de salida. De esta manera esos valores se enviaran a las subredes destino. Esto es que para una cierta transición el lookahead es para un tiempo X pero para la otra transición el tiempo puede ser diferente y es necesario calcular el tiempo mínimo en el que cada una de esas transiciones generarán una marca. Si tiene varias transiciones de salida, al realizar el cálculo el coste computacional es mayor pero es necesario para obtener una información precisa.

3. Análisis y diseño de un simulador distribuido que incorpore mecanismos de lookahead

3.1. Proceso de simulación (SP) de subredes de Petri

Un proceso de simulación simula una partición del modelo y también se le denomina simbot, que es como se le llamó en trabajos anteriores a este proyecto. Estos SPs o simbots interactúan entre sí mediante el paso de mensajes con marcas temporales. En la figura 4 se presentan los componentes de un simbot y las interacciones que se producen entre ellos.

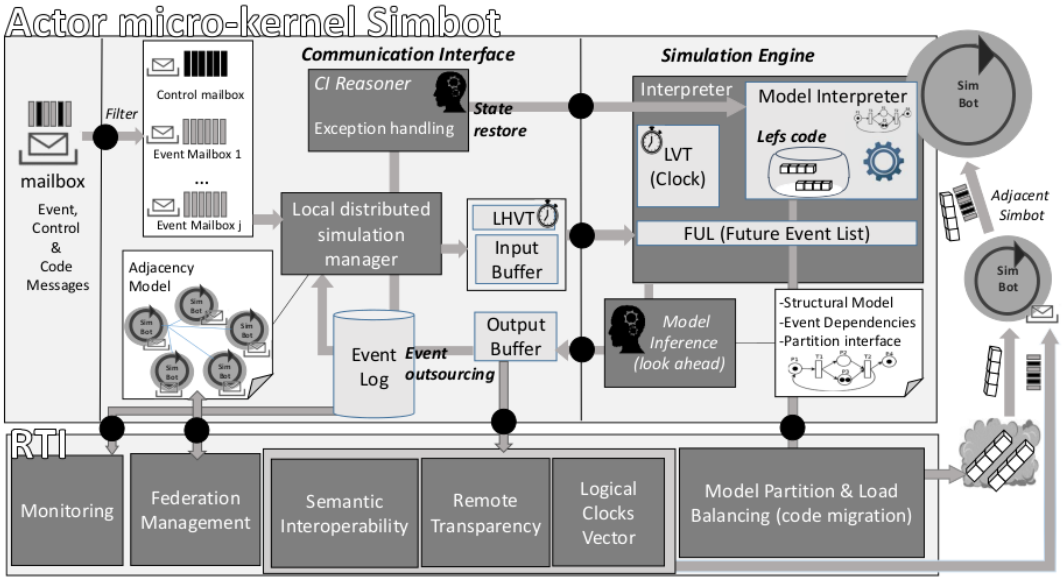


Figura 4: Descripción de un simbot y sus componentes (extraída de [2]).

El motor de simulación se asegura de que los eventos generados localmente y los recibidos de otros simbots son procesados en el orden correcto según sus marcas temporales, garantizando que el resultado es correcto. Se ejecuta como un hilo y sus tareas principales son: interpretación del modelo con las estructuras LEFs, avance del tiempo simulado mediante un reloj virtual local para cada simbot (LVT) y tratamiento de los eventos almacenados en la lista de eventos futuros (FUL), que pueden ser eventos internos o externos recibidos de otros simbots adyacentes.
Cada simbot tiene otro hilo de ejecución, denominado Mailbox, que gestiona la comunicación de forma que envía y recibe mensajes de otros simbots. Además se encarga de comunicar los mensajes recibidos al hilo del motor de simulación.

Antes de iniciar la simulación se realiza una fase de compilación con el compilador

creado en trabajos anteriores. Su tarea consiste en transformar la red de Petri a simular en el modelo que necesita el simulador, aplicando una serie de transformaciones como la obtención de las funciones LEF y sus valores iniciales.

Para la sincronización conservativa cada SP contiene una cola FIFO de mensajes entrantes para cada uno de los SP que le envían eventos, llamados predecesores. Cada cola de entrada Q_i tiene una marca de tiempo $T(Q_i)$ que contiene el tiempo del último mensaje recibido. El mínimo de estas colas $min_i(T(Q_i))$ es el horizonte de tiempo virtual lógico (Local Horizon Virtual Time, LHVT), también llamado marca de tiempo de límite inferior (Lower Bound Time Stamp, LBTS) en [3], hasta el cual se puede simular de forma segura, disparando eventos generados por el propio SP y aquellos recibidos de otros SP cuyo tiempo sea menor o igual a LHVT. Simular hasta LHVT se considera realizar un **intervalo seguro de simulación**, el cual se compone de diferentes pasos de simulación.
Un intervalo seguro de simulación consiste en realizar la simulación desde el ciclo actual de reloj LVT hasta el horizonte de tiempo LHVT. En cada ciclo, se realiza un paso de simulación en el que se disparan las transiciones que estén sensibilizadas y se procesan los eventos etiquetados con el tiempo de simulación en curso (LVT). Cuando no hay nada que simular, ese paso de simulación termina y se actualiza LVT con el tiempo del primer evento en la lista de eventos futuros (FUL) y se repite el proceso. Cuando LVT es mayor que LHVT se termina el intervalo seguro de simulación.

Para realizar un intervalo seguro de simulación, se debe tener información de todos los SP predecesores. Para cada predecesor se debe tener algún evento en Q_i y si no se tiene, el SP debe esperar a recibir evento en Q_i o el valor de lookahead.
En este trabajo se utilizan dos técnicas de envío de lookahead: al finalizar el intervalo seguro de simulación y solicitar el lookahead. Para la primera, al acabar el intervalo seguro de simulación se calcula el valor del lookahead y se envía a los SP sucesores, que son aquellos a los que le envía eventos. Tras ello se vuelve al inicio del bucle de simulación, comprobando el estado de las colas Q_i. La otra opción es que en lugar de que un SP envíe el lookahead cuando termine de simular un intervalo seguro de simulación, sea necesario hacer una solicitud/respuesta. Un SP cuando necesite información de otro SP le envía una petición tras la cual el SP destino obtendrá su lookahead y lo enviará al SP que lo solicitó.
Se utilizan estos dos algoritmos (que se explican en la siguiente sección) para poder comparar el rendimiento en ambos casos y ser capaces de determinar cual es más apropiado. Además, al principio se realizó el envío de lookahead al acabar de simular porque su diseño es más sencillo pero al pensar en simulaciones de gran tamaño se optó por la solicitud de lookahead. Se debe a que se reduce el número de mensajes transmitidos ya que aunque para cada lookahead se envían dos mensajes (solicitud y respuesta), solamente se envían cuando es necesario, mientras que en el otro caso se enviaba independientemente de si se necesitaba o no. La reducción de mensajes que circulan por la red de comunicación es vital especialmente en simulaciones de alta escala para evitar sobrecargar la red empeorando las prestaciones.

En la figura 5 se muestra una descripción general de la arquitectura SP.

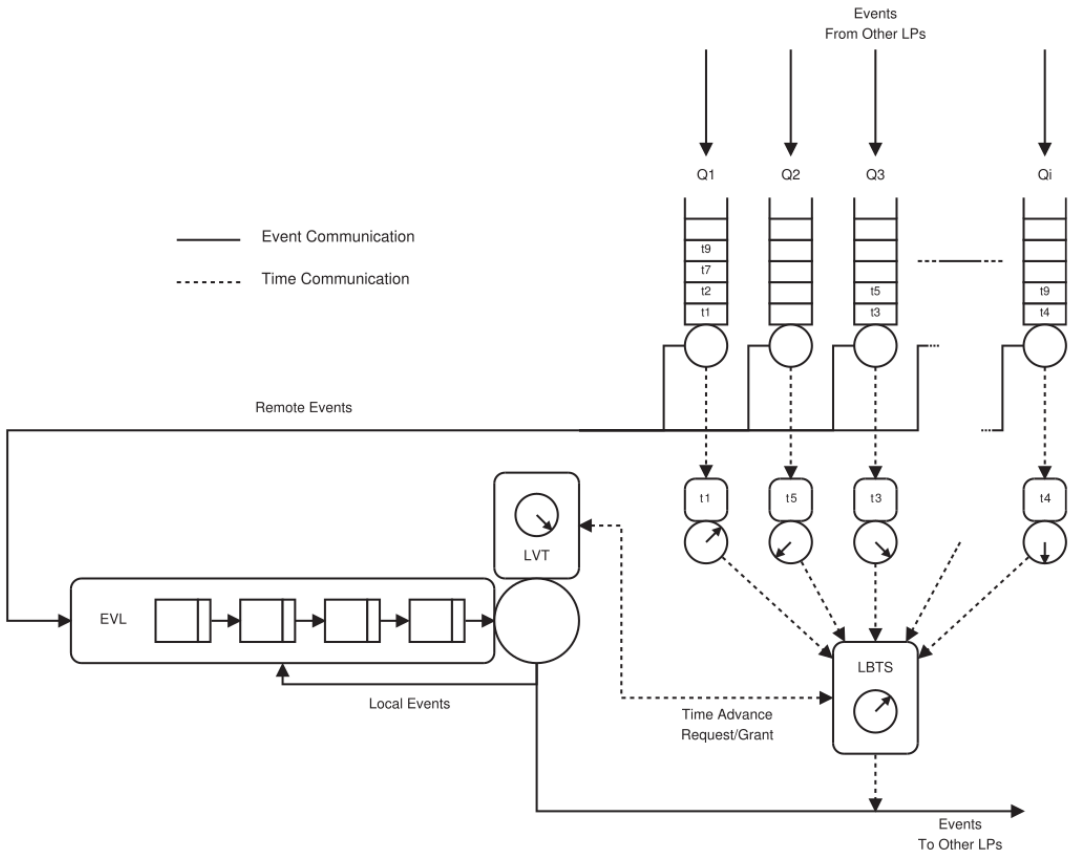


Figura 5: Arquitectura SP (extraída de [3])

3.1.1. Algoritmo de simulación

El algoritmo de simulación distribuido utilizado está basado en el publicado en [2] que se muestra en la figura 6.

En la figura 7 se muestra con mayor detalle el algoritmo implementado en el que el envío de lookahead se produce al final de un intervalo seguro de simulación.

En el algoritmo se cambia ligeramente la notación utilizada con los siguientes términos: enabled_tr (lista de transiciones sensibilizadas), SP_{pre} (SP predecesores) y SP_{post} (SP sucesores).
Entre los cambios realizados está que para cada SP_{pre} además de poner a cero el tiempo recibido en adj, se marca que no hay evento en una nueva estructura de datos llamada adj_new, que indica si hay evento sin procesar (1) o no (0) (líneas 7-9). Se realiza una barrera entre los SP para que todos estén preparados antes de empezar a simular (línea 10). Finalmente para cada SP en SP_{post} se calcula y envía su lookahead (líneas 11-13).

Después ya comienza la simulación hasta el ciclo final que se quiera simular (línea 16). En este bucle primero se recibe un evento de otro SP (operación bloqueante) y se actualiza adj con el tiempo recibido y adj_new indicando que hay evento (líneas 17-18). Si es un evento y no un mensaje nulo se añade el evento a FUL (líneas 19-20). Luego se comprueba si se ha recibido evento de todos los SP_{pre}, esto es que para cada adj su marca de tiempo sea mayor que el reloj local LVT, o que sea igual pero adj_new

```
1: when Start() is received
2: VT ← 0; FUL ← {};
3: for all (t ∈ PUL_ext) do
4:     Adj[t] ← 0;
5:     t ! < 0, lookahead(t) >
6: end for
7: for all (t ∈ LEFs) do
8:     if (f_t(M) ≤ 0) then insert(EL, t);
9:     end if
10: end for
11:
12: when Event(t,UF,ts) is received
13: Adj[t] ← ts;
14: insert-FUL (t, UF, ts);
15: if allReceived(Adj) then
16:     LVTH=min(Adj);
17:     Simulate(ts);
18:     for all (t ∈ PUL_ext) do
19:         if (t ∈ FUL) then
20:             t ! < UF, ts >; remove-FUL(t);
21:         else
22:             t ! < 0, lookahead(t) >;
23:         end if
24:     end for
25: end if
```

Figura 6: Esquema de algoritmo genérico de simulación distribuida (extraída de [2])

indique que hay evento sin procesar (línea 21). Si se cumple, se obtiene el mínimo de los valores de adj y se guarda en LHVT (línea 22), que es el horizonte de tiempo hasta el que se puede simular.
Ahora se cambia el valor de adj_new a cero para los valores de adj que sean iguales a LHVT para indicar que esos eventos se van a procesar (líneas 23-26).
Se hace una comprobación de que el horizonte de tiempo no supere al tiempo máximo a simular (líneas 27-28), y si lo supera se modifica con el tiempo máximo y tras acabar la siguiente simulación terminará.
Realiza un intervalo seguro de simulación hasta el horizonte de tiempo (línea 29) y después envía los eventos a los SP_{post} (línea 30). Esto consiste en enviar todos los eventos exteriores generados en el intervalo seguro de simulación anterior. Además, si alguno de los SP_{post} no tenían ningún evento exterior, se calcula y envía su lookahead. Como alternativa se puede enviar siempre el lookahead en este caso, tanto si se envía evento exterior como si no. Esta versión tendría una mayor información a costa de transmitir un mayor número de mensajes, obteniendo una red con más carga. Ambas versiones son correctas y pueden obtener mejor resultado que la otra en función de la red de Petri que se simule, aunque en este proyecto se ha optado por la primera alternativa la cual es más simple y tiene menor número de mensajes transmitidos.

Por último, finaliza la simulación (línea 33) sincronizando los SP y agrupando los logs en el SP principal.

```
1:  # When the SP starts
2:   enabled_tr ← {}; FUL ← {};
3:   for all (t ∈ LEFs) do
4:      if (f_t(M) ≤ 0) then insert(enabled_tr, t);
5:      end if
6:   end for
7:   for all (t ∈ SP_pre) do
8:      adj[t] ← 0; adj_new[t] ← 0;
9:   end for
10: barrierSimbots();
11: for all (t ∈ SP_post) do
12:    t ! < 0, lookahead(t) >
13: end for
14:
15: # Simulation algorithm
16: while (LVT <= final_time) do
17:    receive_event();
18:    adj[t] ← ts;  adj_new[t] ← 1;
19:    if (is_event()) then insert-FUL(t, UF, ts);
20:    end if
21:    if (allReceived()) then
22:       LHVT=minAdj();
23:       for all (t ∈ adj) do
24:          if (adj[t] == LHVT) then adj_new[t] ← 0;
25:          end if
26:       end for
27:       if (LHVT > ai_final_time) then LHVT=ai_final_time;
28:       end if
29:       simulate_interval();
30:       send_events();
31:    end if
32: end while
33: finish_simulation();
```

Figura 7: Algoritmo de simulación distribuida con envío de lookahead al finalizar un intervalo seguro de simulación.

El intervalo seguro de simulación (línea 29) consiste en una serie de pasos de simulación, siendo un paso un ciclo. En cada paso se disparan las transición sensibilizadas, se comprueba si se puede avanzar el tiempo y se procesan los eventos. El avance de tiempo consiste:

- Si no hay eventos en FUL: LVT = LHVT y acaba el intervalo seguro de simulación.
- Si el primer evento en FUL tiene tiempo t $>$ LVT:
  - t $>$ LHVT: LVT = LHVT y acaba el intervalo seguro de simulación.
  - t $\leq$ LHVT: LVT = t, acaba ese paso y comienza otro paso de simulación.

Es necesario utilizar las estructuras adj y adj_new ya que solo con adj no es suficiente. Con adj se almacena la información del último tiempo recibido de los SP_{pre}. Sin embargo, si no se utiliza adj_new para indicar cuando hay eventos sin procesar, el algoritmo podría no funcionar.
Por ejemplo, dado SP_i que solo tiene a SP_j como predecesor, SP_i podría recibir un mensaje nulo con el valor de lookahead 10, asignando adj[j]=10. Obtendría que LHVT=10, y realizaría un intervalo seguro de simulación. Al acabar, esperaría a recibir otro mensaje de SP_j, que podría ser un mensaje nulo con un tiempo mayor al último recibido o un evento. En caso de que fuese un evento, su etiqueta de tiempo podría ser 10 porque ese es el valor de lookahead que previamente envió SP_j. Al tener tiempo 10, el valor de adj[j] no cambia, por lo que en ese caso con adj no se conoce si hay información que se puede procesar o no. Por tanto, es necesario la estructura adj_new para indicar que en esos casos hay información a tratar y poder obtener LHVT para realizar un intervalo seguro de simulación.

En el anexo II se muestra la traza de una simulación distribuida para una red de Petri con tres SPs.

El algoritmo de simulación distribuida con **solicitud de lookahead** es similar al anterior. En la figura 8 se puede ver el algoritmo. La diferencia está en que al final del intervalo seguro de simulación no se envía el lookahead sino que es el hilo de Mailbox el que al recibir una petición de lookahead calcula el valor y lo envía. Además, antes de realizar un intervalo seguro de simulación se verifica que se dispone de información para calcular el nuevo LHVT (línea 17). En caso de no tener información suficiente para algún SP vecino, se solicita su lookahead y se espera la respuesta. En esta función también se comprueba si hay eventos recibidos del exterior y se guardan en la lista de eventos.
La condición de información suficiente para un SP vecino se satisface si se dispone de un último tiempo de ese SP mayor estricto a LVT. También es válido un tiempo igual a LVT siempre que haya pendiente por procesar un evento de ese SP para el tiempo LVT, ya que permitiría continuar con la simulación aunque el nuevo LHVT seguiría siendo el mismo, igual a LVT. Este sería un caso similar al explicado antes en el otro algoritmo con el uso de las estructuras adj y adj_new, que aquí no son necesarias.

3.2. Mejora del lookahead explotando el modelo

El lookahead aporta la información de hasta qué ciclo de simulación como mínimo no se va a enviar ningún evento. Es una información imprescindible en la simulación ya que cuando un SP quiere calcular un nuevo LHVT, al recibir un mensaje nulo puede utilizar esa información para extender su valor de LHVT y así eliminar el bloqueo. De esta manera conoce hasta cuando puede avanzar su reloj de simulación, mediante la gestión de las colas Q_i. Al extraer estos mensajes de las colas en caso de que sean nulos no se guardan en la lista de eventos FUL ya que no tienen información.
Al poder obtener el horizonte temporal, es posible realizar un intervalo seguro de simulación en el cual se va avanzando el reloj. El incremento del reloj local es seguro

```
1:  # When the SP starts
2:   enabled_tr ← {}; FUL ← {};
3:   for all (t ∈ LEFs) do
4:      if (f_t(M) ≤ 0) then insert(enabled_tr, t);
5:      end if
6:   end for
7:   for all (t ∈ SP_pre) do
8:      adj[t] ← 0; adj_new[t] ← 0;
9:   end for
10: barrierSimbots();
11: for all (t ∈ SP_post) do
12:    t ! < 0, lookahead(t) >
13: end for
14:
15: # Simulation algorithm
16: while (LVT <= final_time) do
17:    check_neighbours();
18:    LHVT=minAdj();
19:    for all (t ∈ adj) do
20:       if (adj[t] == LHVT) then adj_new[t] ← 0;
21:       end if
22:    end for
23:    if (LHVT > ai_final_time) then LHVT=ai_final_time;
24:    end if
25:    simulate_interval();
26: end while
27: finish_simulation();
```

Figura 8: Algoritmo de simulación distribuida con solicitud de lookahead.

siempre que LVT $\leq$ LHVT, en caso contrario se establece que LVT = LHVT, ya que solo se puede avanzar hasta LHVT como máximo. De este modo se garantiza que no habrá eventos que lleguen fuera de orden temporal y por tanto la simulación es correcta, al respetar la restricción de causalidad local.

El lookahead se calcula utilizando la red de Petri del modelo a simular estimando el tiempo mínimo en el que enviará un evento. Su cálculo se puede dividir en dos fases: compilación y simulación. En la primera fase se obtiene un precálculo del modelo de red de Petri y en la segunda fase se utiliza esa información junto al estado de la red de Petri para obtener el valor de lookahead. El cálculo se explica en la siguiente sección.

El tráfico de lookahead está determinado por las relaciones de vecindad entre los SP. Un SP envía estos valores únicamente a aquellos SP que sean sus sucesores ya que dependen de su información del estado de la red. Los sucesores son aquellos que tienen lugares cuyo marcado los determina el SP predecesor. Del mismo modo, un SP solo

recibe tráfico de lookahead de sus SP predecesores, ya que no requiere información del resto de SP.

Es importante destacar que el lookahead depende del modelo y es crucial para introducir paralelismo en la ejecución de la simulación. La falta de información para extraer lookahead de forma precisa es una de las mayores dificultades de los algoritmos conservativos. Una de las características más importante de este trabajo es explotar la información estructural del modelo para poder obtener información de los lookahead.
Cuanto mayor sea el valor del lookahead mejores prestaciones se conseguirán en la simulación ya que un punto crítico es la espera de un SP a recibir mensajes de otros para continuar la simulación. Así cuando un SP consulta la información de sus predecesores, si le falta al menos la información de uno de ellos debe parar la simulación hasta recibir ese dato, reduciendo las prestaciones.
De esta forma si el valor del lookahead es pequeño, el intervalo seguro de simulación acabará antes y se comprobará con mayor frecuencia el estado de los predecesores, con un mayor número de paradas. Sin embargo, si el valor es grande, se simulará un mayor número de ciclos en el intervalo seguro de simulación. Así se obtiene un menor número de comprobaciones del estado de los vecinos y por tanto menos puntos de parada de la simulación.

Dada una subred simulándose en un SP, la obtención del mayor valor de lookahead que enviar a los sucesores dependerá de la información estructural de la subred de Petri en el SP y de los eventos recibidos por los predecesores. Esto quiere decir que el tiempo mínimo en el que puede enviar un evento está en función del tiempo en el que un predecesor le enviará un evento al SP. Tras producirse este envío, se dispararán las transiciones correspondientes en la subred de Petri para al final enviar un evento exterior en un determinado ciclo, estimado con el valor de lookahead. Este es mejor valor ya que si depende de alguna transición de la propia subred, el lookahead será menor.

La vigencia del valor de lookahead es hasta ese intervalo seguro de simulación ya que solo se puede incrementar. Es cierto que el verdadero lookahead podría cambiar de un ciclo a otro, en el caso de que dependa de la información de algún predecesor. Por ejemplo, en un ciclo T se podría obtener un cierto valor de lookahead que depende de la información de mensajes nulos de algún predecesor. En el ciclo T+1 esa información se podría actualizar incrementándose por lo que el valor real del lookahead sería mayor.
En este proyecto se ha optado por mantener vigente el valor de lookahead hasta el intervalo seguro de simulación porque es la solución más simple y además en principio en otro caso se generaría un mayor número de mensajes que podría terminar reduciendo las prestaciones de la simulación.
Otras posibles opciones a valorar serían tener un equilibrio entre tener un lookahead muy actualizado sin sobrecargar el tráfico de red de mensajes. En el caso extremo estaría calcular y enviar el lookahead en cada disparo de transición consiguiendo un valor actualizado a costa de sobrecargar la red de comunicación, con el coste implicado. Como trabajo futuro sería adecuado realizar un análisis de este problema para encontrar el equilibrio entre valor actualizado y tráfico de red que permita tener mejores prestaciones.

En relación a lo anterior, la actualización del lookahead puede no ser necesaria en redes de Petri deterministas, al menos hasta que el reloj local alcance el valor del lookahead recibido.
En el caso de redes con conflictos, que se permitirán en versiones posteriores de este proyecto, cuando se produzca un conflicto habría que actualizar la información de lookahead con los nuevos valores obtenidos según el resultado del conflicto. Además, en estas redes ya no se puede explotar la información del modelo en la fase de compilación como en las redes deterministas. En caso de hacerlo el resultado sería demasiado pesimista ya que se tomaría la cota mínima que sería el camino más largo, dado que en compilación no se conoce el resultado del conflicto y para asegurar la simulación tomaría el de mayor tiempo.

Existen múltiples estrategias para el envío de mensajes nulos como se explica en el capítulo 2. En este proyecto se comparan dos de ellas: el envío de mensajes nulos cada vez que finaliza un intervalo seguro de simulación (similar al protocolo Chandy-Misra-Bryant con ligeros cambios) y la solicitud de lookahead.
La elección final es la solicitud de lookahead porque como se explica en el capítulo 5 las prestaciones obtenidas son mejores para las redes utilizadas, que están pensadas para simulaciones de gran tamaño.

3.3. Métodos de cálculo de lookahead

El cálculo de lookahead tiene dos fases: compilación y simulación. En la primera fase se obtiene un precálculo del modelo de la red de Petri que se utiliza en la segunda fase de simulación junto al estado de la red de Petri en ese momento para obtener el valor del lookahead. En la fase de simulación se han diseñado e implementado dos formas diferentes de calcular el lookahead. Cada variante tiene sus ventajas y desventajas que después se explicarán.

3.3.1. Cálculo en compilación

En la fase de compilación se obtiene para todas las transiciones de salida el tiempo que va a costar llegar una marca desde cada transición, llamado tiempo de marca (TM). El pseudocódigo se puede ver en la figura 9.
Se propuso una implementación recursiva inicial, pero los tiempos de ejecución hicieron necesario replantear el algoritmo de forma iterativa. Para cada transición de salida se calculan los tiempos de marca. Para ello se lleva un registro de las transiciones que hay que consultar junto a sus valores de tiempo de marca hasta esas transiciones (línea 7) y de las transiciones que ya se han visto (línea 8).
Mientras que haya transiciones por calcular (líneas 9-19), se obtiene una transición (línea 10), se marca como vista (línea 11) y se calcula y almacena su tiempo como el tiempo hasta esa transición más su duración de disparo (línea 12-13).
Después se obtienen las transiciones correspondientes a sus lugares de entrada, es decir las transiciones anteriores (líneas 14-18). Si esas transiciones no se han visto ya y además pertenecen a la subred (línea 15) se guardan para visitarlas después.

```
1:  # For each output transition:
2:  # Iterative function that calculates the time from the
3:  # transition given to all the previous subnet transitions.
4:  func time_to_marks(trans_output):
5:      marks_time ← {}
6:      #  List of {transition, time to mark} of transitions to watch
7:      trans_pending ← {trans_output,0}
8:      trans_seen ← {}
9:      while (trans_pending not empty) do
10:         {trans, time} ← trans_pending
11:         trans_seen ← {trans}
12:         TM ← time + trans.time
13:         marks_time ← {trans,TM}
14:         for all (trans_pre ∈ prev_transitions(trans)) do
15:             if (is_in_subnet(trans_pre) and (trans_pre not in trans_seen)) then
16:                 trans_pending ← {trans_pre,mark_time}
17:             end if
18:         end for
19:     end while
20:     return marks_time
21: end func
```

Figura 9: Cálculo del tiempo de marca en la fase de compilación de la red de Petri.

Así se repite este procedimiento hasta que no hay más transiciones anteriores, es decir, se llega a los lugares de entrada de la subred cuyas marcas provienen de otras subredes. Estos tiempos se almacenan en un mapa con el identificador local de la transición como clave y el tiempo como valor. Tras calcular los tiempos se devuelven para guardarlos para esa transición (línea 20).

En esta fase se realizan otros cálculos para utilizarlos en la fase de simulación. Uno de ellos es la obtención de la lista de transiciones de entrada que se consigue mirando qué transiciones de la subred de Petri son de entrada. El otro es la lista de transiciones predecesoras para cada una de las transiciones. Para ello se mira todas las transiciones y en cada una se obtiene cuales son las transiciones siguientes (Projected Updating List, PUL). Después para cada transición sucesora de la lista PUL se actualiza esa transición estableciendo la transición actual como su predecesora.
Esta información se utiliza en la siguiente fase del cálculo de lookahead.

Ejemplo
En la figura 10 hay una subred de Petri que tiene una transición de salida (t3) y dos transiciones de entrada (t0 y t1). Además, t2 tiene dos lugares de entrada (p1 y p3), ambos internos.
La duración de disparo de t1 es de 5 unidades de tiempo y de 1 unidad de tiempo para t0, t2 y t3.

Los tiempos de marca obtenidos en compilación para la transición de salida t3 aparecen marcados en la figura en color azul y son: {t3: 1, t2: 2, t1: 7, t0: 3}.

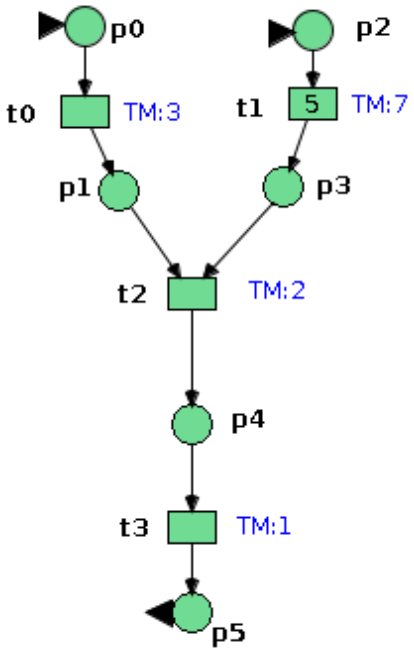


Figura 10: Subred de Petri con cálculo de tiempos de marca (TM).

3.3.2. Cálculo en simulación

Se han desarrollado dos métodos para el cálculo del lookahead en simulación. Primero se realizó un método que se ha llamado cálculo dinámico del lookahead cuya principal ventaja es que se puede utilizar aunque las subredes de Petri de los SPs cambien durante la ejecución. De este modo es posible utilizar mecanismos de balanceo de carga como los propuestos en [7], consiguiendo un simulador más completo. El otro método es mediante el cálculo de un vector de lookahead que no permite el balanceo de carga pero a cambio es más rápido, permitiendo simulaciones con mejor rendimiento.
Hay otras posibles soluciones, cada una con sus ventajas y desventajas, como la que se propuso en un inicio que se explica en el anexo III. Esta solución tiene un rendimiento intermedio entre los otros dos métodos pero se descartó porque tiene más restricciones en los modelos a simular.

3.3.2.1. Método 1: cálculo dinámico de lookahead

Este método esta diseñado para que se utilice enviando el lookahead al final del intervalo seguro de simulación. Para utilizar la solicitud de lookahead sería necesario realizar ciertos cambios, aunque se podría hacer sin excesivo trabajo.
Es importante destacar que este método, en la versión actual, tiene la limitación de que las redes de Petri utilizadas deben ser redes binarias sin conflicto. Además, las transiciones de entrada solo pueden tener un lugar de entrada, es decir, no se permiten varios lugares de entrada de la subred en la misma transición. Tampoco se permite

que una transición tenga un lugar de entrada de la subred y otros lugares de entrada internos.

Este método consiste en el algoritmo de la figura 11 cuyo objetivo es encontrar la transición que tras simular un determinado número de ciclos pueda hacer que la transición de salida genere un evento, y por tanto se envíe al SP destino. Para ello se realiza un recorrido de la subred de Petri empezando por la transición de salida y continuando por las transiciones anteriores, las predecesoras.
Primero se inicializan las variables (líneas 2-3) para llevar un registro de las transiciones que se han visto (*trans_seen*), las que están por mirar (*trans_pending*), los tiempos desde cada transición (*trans_times*), las transiciones pendientes de obtener su tiempo (*trans_pending_time*) y si la subred solo tiene un camino o no (*one_way*). Los tiempos desde cada transición (*trans_times*) consiste en que para cada una de las transiciones de la subred que se consultan se almacena el tiempo mínimo que costaría que esa transición enviase una marca a la siguiente (o siguientes) transición. Este tiempo está compuesto por el tiempo que tardaría la propia transición en estar sensibilizada con el estado de la red actual, sumado al valor de su duración de disparo.
Se utiliza dos listas de transiciones pendientes porque la primera es para mirar todas las transiciones (*trans_pending*) pero para la mayoría de ellas no se obtendrá su tiempo porque hará falta información de otras transiciones. Por tanto hace falta otra lista de transiciones que no se han obtenido su tiempo (*trans_pending_time*) para calcular su tiempo después.
Además, se utiliza una variable para conocer si en toda la subred de Petri solo existe un camino (*one_way*) ya que en caso afirmativo se aplican algunas optimizaciones para mejorar las prestaciones del algoritmo. Estas optimizaciones, que se explicarán después, están pensadas para las redes que se utilizan en el proyecto de ramas secuenciales.

La primera optimización que se aplica (líneas 4-6) consiste en que si no hay eventos en FUL y solo hay una transición de entrada el tiempo es el de la transición de entrada. Este tiempo sería el último valor recibido para esa transición, que es cuando podría estar sensibilizada, sumado a su tiempo de marca. Este último valor se ha obtenido en la fase de compilación y es el tiempo desde que esa transición se dispara y hasta que se genera una marca en la transición de salida.
Esta mejora está enfocada para las redes simples usadas en el proyecto de ramas secuenciales de manera que para esos casos no es necesario recorrer las transiciones de la subred de Petri.

Mientras que haya transiciones por calcular (líneas 7-32), se obtiene una transición y se marca como vista (línea 8) y se trata la transición:

- **La transición es de entrada** (líneas 9-13): hay que utilizar la información de lookahead recibida para esa transición. El tiempo de ese camino hasta esa transición sería el último valor de lookahead recibido para esa transición sumado a su duración de disparo (línea 13). Ese tiempo se guarda para utilizarlo después. Este tiempo es la estimación del ciclo en el que como mínimo la transición podría estar sensibilizada, en caso de que reciba un evento del correspondiente SP, más la duración de su disparo.

```
1:  func get_lookahead(trans_output):
2:    trans_seen ← {};  trans_pending ← trans_output;  trans_times ← {}
3:    trans_pending_time ← {};  one_way = true
4:    if (no_events() and num_transition_input() = 1) then  // Optimization
5:      return get_ext_time(trans) + time_mark()
6:    end if
7:    while (trans_pending not empty) do
8:      {trans} ← trans_pending;  trans_seen ← {trans}
9:      if (is_entry(trans)) then
10:       if (one_way) then  // Optimization
11:         return get_ext_time(trans) + time_mark()
12:       end if
13:       trans_times[trans] ← get_ext_time(trans) + trans.firing_time
14:     else then
15:       lef_val ← get_lef_value(trans)
16:       if (num_events_FUL(trans) >= lef_val) then
17:         if (one_way) then  // Optimization
18:           return time_event_FUL(trans, lef_val) + time_mark()
19:         end if
20:         trans_times[trans] ← time_event_FUL(trans, lef_val) + trans.firing_time
21:       else then
22:         trans_pending_time ← trans;  one_way = (len(trans.pre) == 1)
23:         for all (trans_pre ∈ trans.pre) do
24:           if (trans_pre not in trans_seen) then
25:             trans_pending ← trans_pre
26:           else if (trans_pre not in trans_times) then  // Cycle
27:             trans_times[trans_pre] ← LVT + trans_pre.firing_time
28:           end if
29:         end for
30:       end if
31:     end if
32:   end while
33:   while (trans_pending_time not empty) do
34:     {trans} ← trans_pending_time;  times_pre ← {}
35:     for all (trans_pre ∈ trans.pre) do
36:       if (trans_pre in trans_times) then
37:         times_pre ← trans_times[trans_pre]
38:       else then
39:         break
40:       end if
41:     end for
42:     if (len(times_pre) = len(trans.pre)) then
43:       sort(times_pre);  trans_times[trans] = times_pre[trans.lef-1] + trans.firing_time
44:     else then
45:       trans_pending_time ← trans
46:     end if
47:   end while
48:   return trans_times[trans_output]
49: end func
```

Figura 11: Cálculo del lookahead del método dinámico.

Si la subred solo tiene un camino (líneas 10-12), se aplica la optimización y directamente se devuelve el tiempo como el último valor de lookahead recibido para la transición junto a su tiempo de marca. En otro caso (la red tiene varios caminos) no se puede devolver el tiempo y acabar el cálculo porque puede ser necesario otras acciones para obtener un tiempo preciso.

- **La transición estará sensibilizada en un tiempo futuro** (líneas 15-20): si la transición no es de entrada hay que comprobar si con los eventos pendientes en FUL estará sensibilizada en un ciclo futuro. Primero se obtiene cuantas marcas necesita para estar sensibilizada, es decir, su valor de LEF (línea 15). Después se comprueba si con los eventos en FUL se puede sensibilizar la transición (línea 16). Al ser redes binarias cada evento dirigido a la transición quita una marca al LEF (en redes de Petri sería añadir una marca a un lugar de entrada).

  Si hay tantos eventos como el valor de LEF, se obtiene y se guarda su tiempo (línea 20) y en otro caso se hace otra comprobación para transiciones que no estarán sensibilizadas con los eventos en FUL. Se admite que haya más eventos que el valor de LEF, para permitir redes con múltiples disparos de una transición en distintos tiempos. Por ejemplo, una transición t_i con tiempo de disparo de 3 unidades de tiempo podría dispararse en el tiempo T=1 y T=3. En T=3, en FUL habría dos eventos generados por esa transición, uno para T=4 y otro para T=6.

  El tiempo que se guarda es el valor de la duración de disparo más el tiempo del evento que sensibilice la transición que se consulta. Esto es el evento que al procesar permita que se pueda disparar la transición. Por ejemplo, si LEF=2 se toma el tiempo del evento de la segunda marca, es decir, el segundo evento de FUL dirigido a la transición. Estos eventos están ordenados de menor a mayor tiempo.

  Igual que en el caso anterior se aplica una optimización (líneas 17-19) de manera que si la subred solo tiene un camino se devuelve el tiempo del evento que sensibiliza la transición sumado a su tiempo de marca. Así se mejora el rendimiento al juntar el cálculo estático obtenido en compilación para el tiempo de marca con el cálculo dinámico que mira el estado de la red.

- **La transición no estará sensibilizada en un tiempo futuro** (líneas 21-30): en este caso es necesario obtener la información de las transición predecesoras. Primero se guarda la referencia a la transición para completar su información más adelante y se actualiza la variable *one_way* en función de si solo hay un camino (solo hay una transición predecesora) o no (línea 21). Después se obtienen las transiciones correspondientes a sus lugares de entrada, es decir, las transiciones anteriores (líneas 23-29). Si esas transiciones no se han visto ya (línea 24) se guardan para visitarlas después (línea 25).

  Si ya se han visto, si todavía no se ha obtenido su tiempo (línea 26) se calcula su tiempo obteniendo la cota mínima (línea 27). Al obtener las transiciones predecesoras, si ya se han visto entonces hay un ciclo en la red de Petri de forma que hay un camino de transiciones que empiezan y acaban en un mismo lugar. Otra

posibilidad es que sea un conflicto de forma que dos o más transiciones comparten un mismo lugar, aunque en este trabajo no se permiten redes con conflictos.

En el caso de un ciclo, hay que guardar el tiempo de ese camino obteniendo su cota mínima, que es el valor LVT más el tiempo de disparo de la transición. De esta manera lo que se está haciendo es eliminar los ciclos para poder calcular el valor del lookahead, ya que en caso de no hacerlo se produciría un bloqueo. Este cálculo solo se realiza si el valor del camino de esa transición no se ha obtenido ya, porque si ya lo tiene se puede utilizar ese tiempo.

En este punto ya se han tratado todas las transiciones pero solo tienen información completa aquellas que son de entrada o que con los eventos en FUL se conoce que estarán sensibilizadas. Cabe decir que no es necesario consultar todas las transiciones de la subred, ya que se van mirando las transiciones anteriores y en el caso de que se encuentra una transición sensibilizada en un tiempo futuro no se tratan sus transiciones predecesoras.
Ahora se completa la información de las transiciones pendientes (líneas 33-47). Mientras que haya transiciones pendientes se obtiene una transición y se inicializa un registro de los tiempos de sus predecesoras (línea 34). Este registro puede tener un solo tiempo (si solo tiene una transición anterior) o varios tiempos.
Para cada transición predecesora (líneas 35-41) se guardan sus correspondientes tiempos. Si alguna predecesora no tiene información, se sale del bucle (línea 39).
Si se tiene toda la información disponible (línea 42), entonces se ordenan los tiempos de las predecesoras de menor a mayor tiempo y se obtiene y almacena su tiempo (línea 43). Este tiempo es la duración de su disparo sumado al valor de la predecesora que podría sensibilizar la transición. Para obtenerlo se utiliza su valor de LEF actual menos uno debido a que el índice empieza en cero. Por ejemplo, si LEF=2 el tiempo es el de la predecesora que quite la segunda marca.
Si no se tiene la información de todas las predecesoras, se marca la transición que se está consultando como pendiente (línea 45), para completarla más adelante. Esto puede ocurrir al consultar una transición t_i si tiene una predecesora t_j cuya información aún no está disponible.

Finalmente se devuelve el valor obtenido correspondiente a la transición de salida (línea 48).

Es necesario mirar si hay eventos pendientes de procesar en FUL porque cuando se calcula el lookahead ya se ha simulado hasta el horizonte temporal. Por tanto no hay transiciones sensibilizadas ya que en caso de haberlas ya se habrían disparado, por lo que hay que comprobar si hay eventos pendientes para tiempos de simulación mayores a LVT.

Hay un caso especial que es al inicio de la simulación. En esta fase, antes de simular se calculan y envían los valores de lookahead para los SP sucesores. En este punto no hay ningún evento en FUL pero hay transiciones sensibilizadas por el marcado inicial. El pseudocódigo de este cálculo se puede ver en la figura 12.
Su funcionamiento es similar al caso general, pero en vez de consultar FUL se mira si está sensibilizada la transición (línea 12). También se utiliza el valor de LVT en lugar

```
1:  func get_initial_lookahead(trans_output):
2:      trans_seen ← {}; trans_pending ← trans_output; trans_times ← {}
3:      trans_pending_time ← {}; one_way = true
4:      while (trans_pending not empty) do
5:          {trans} ← trans_pending; trans_seen ← {trans}
6:          if (is_entry(trans)) then
7:              if (one_way) then  // Optimization
8:                  return LVT + time_mark()
9:              end if
10:             trans_times[trans] ← LVT + trans.firing_time
11:         else then
12:             if (is_enabled(trans)) then
13:                 if (one_way) then  // Optimization
14:                     return LVT + time_mark()
15:                 end if
16:                 trans_times[trans] ← LVT + trans.firing_time
17:             else then
18:                 trans_pending_time ← trans; one_way = (len(trans.pre) == 1)
19:                 for all (trans_pre ∈ trans.pre) do
20:                     if (trans_pre not in trans_seen) then
21:                         trans_pending ← trans_pre
22:                     else if (trans_pre not in trans_times) then  // Cycle
23:                         trans_times[trans_pre] ← LVT + trans_pre.firing_time
24:                     end if
25:                 end for
26:             end if
27:         end if
28:     end while
29:     while (trans_pending_time not empty) do
30:         {trans} ← trans_pending_time; times_pre ← {}
31:         for all (trans_pre ∈ trans.pre) do
32:             if (trans_pre in trans_times) then
33:                 times_pre ← trans_times[trans_pre]
34:             else then
35:                 break
36:             end if
37:         end for
38:         if (len(times_pre) = len(trans.pre)) then
39:             sort(times_pre)
40:             trans_times[trans] = times_pre[trans.lef-1] + trans.firing_time
41:         else then
42:             trans_pending_time ← trans
43:         end if
44:     end while
45:     return trans_times[trans_output]
46: end func
```

Figura 12: Cálculo del lookahead del método dinámico: fase inicial.

del evento en FUL (línea 16) y del último valor de lookahead recibido (línea 10), ya que en esta fase aún no se ha obtenido información de lookahead de los SP predecesores. Por tanto se asume que en el ciclo actual, LVT, podría llegar algún evento.

En el algoritmo hay algunas optimizaciones enfocadas para redes simples que en un futuro se podrían ampliar para otros casos. Ahora lo que se realiza es que si la subred de Petri solo tiene un camino posible una parte del tiempo de lookahead se puede obtener utilizando el tiempo de marca. Esta técnica se podría aplicar en varios niveles con una visión recursiva, desde una transición hasta otra transición posterior. El camino entre estas dos transiciones debe ser único de forma que por ejemplo si entre una transición t1 y t4 solo hay un camino y en t4 hay varias salidas, de t1 a t4 se puede utilizar el tiempo precalculado en compilación. En t4 habría que analizar en ejecución cual es el camino que se debe tener en cuenta y después se podría volver a utilizar el tiempo obtenido en compilación si desde la siguiente transición a t4 hasta otra transición tX solo hay un camino.
Esta técnica se podría aplicar recursivamente hasta llegar a la transición de salida para evitar cálculos innecesarios de las duraciones de disparo entre dos transiciones que solo tienen un único camino.

Ejemplo: subred de Petri con dos caminos
En la figura 13 hay una subred de Petri que tiene una transición de salida (t3) y dos transiciones de entrada (t0 y t1). Además, t2 tiene dos lugares de entrada (p1 y p3), ambos internos.

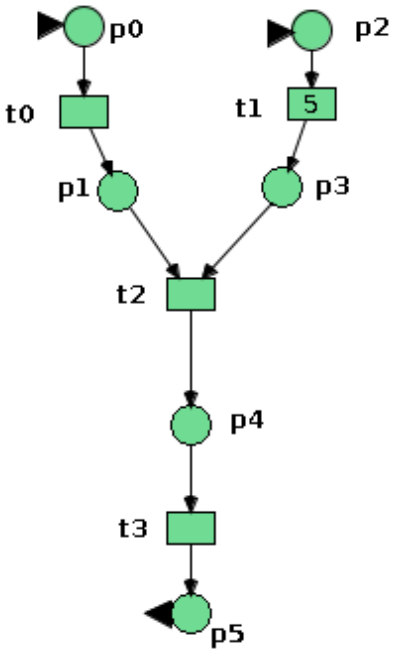


Figura 13: Subred de Petri con una transición de salida (t3), dos de entrada (t0 y t1) y una interna (t2).

La duración de disparo de t1 es de 5 unidades de tiempo y de 1 unidad de tiempo para t0, t2 y t3.

Si en un determinado momento de la simulación se calculase el lookahead con esta red en el estado que aparece en la figura (con todos los lugares sin marcas y sin eventos en FUL), el valor que se obtendría sería: L(t0)+3 o L(t1)+7. Dónde L(tX) es el último valor de lookahead recibido para tX.
Este valor se obtiene mediante la aplicación del algoritmo. Primero se miraría t3, que no se podría obtener su información por lo que se miraría t2. Del mismo modo se miraría t0 y t1. Para t0 se obtendría un tiempo de L(t0)+1, y para t1 L(t1)+5.
El siguiente paso sería completar la información de t2. Se tomarían los dos tiempos anteriores y se ordenarían de menor a mayor tiempo. Si por ejemplo, L(t0)=2 y L(t1)=0 el resultado sería: 3, 5. Como el valor de LEF de t2 es 2, es decir, requiere dos marcas, es la segunda marca la que podría sensibilizar la transición. Por tanto se toma el valor de 5 (que viene del camino de t1) porque en el tiempo 3 podría recibir una marca pero hasta que no reciba otra en el tiempo 5 no se podría disparar. La transición t2 toma el valor de 5+1 y lo guarda.
Finalmente, t3 recupera la información de su predecesor t2 que es 6. A este valor suma su duración de disparo que es 1 y devuelve como lookahead el valor 7.

Otro posible caso sería si el valor de LEF de t2 fuese 1. El proceso sería el mismo pero al requerir solo una marca tomaría el menor valor de sus predecesores, t0 y t1.

Ejemplo: subred de Petri con ciclo
En la figura 14 hay una subred de Petri similar a la del ejemplo anterior pero que tiene un ciclo en las transiciones t1 y t2. El ciclo se produce si la transición t1 está sensibilizada y se dispara. Después si la transición t2 está sensibilizada y se dispara, como uno de sus lugares de salida es p10 que es entrada de t1, se vuelve a la transición t1.
Al calcular el valor de lookahead, en la transición t2 se obtendría que para el camino de t0 su valor sería L(t0)+1. Respecto al camino de t1, como hay un ciclo habría que mirar las dos transiciones predecesoras de t1: t10 y t2. Para t10 su valor sería L(t10)+1, mientras que para t2 como ya se ha visto antes porque es un ciclo, su tiempo sería LVT+1.
Como t1 solo necesita una marca para estar sensibilizada, se toma el menor de los tiempos anteriores: LVT+1. Este tiempo es la cota mínima ya que en la red de Petri se puede observar que es el lugar p2 el que no tiene marca, y por tanto se debería tomar el valor de su camino L(t10)+1. Sin embargo, al utilizar LEF no se conoce, o al menos no de una forma sencilla, qué lugares son los que ya tienen marca y cuales faltan. Por tanto lo que se quiere saber es cual podría enviar una marca en el menor tiempo, aunque el tiempo que se obtenga sea una cota mínima que puede no ser igual al tiempo real.

3.3.2.2. Método 2: cálculo con vector de lookahead

Este método está diseñado para usarse junto a la solicitud de lookahead entre los SPs. Consiste en que dada una transición de salida en una subred de Petri, se dispone de un

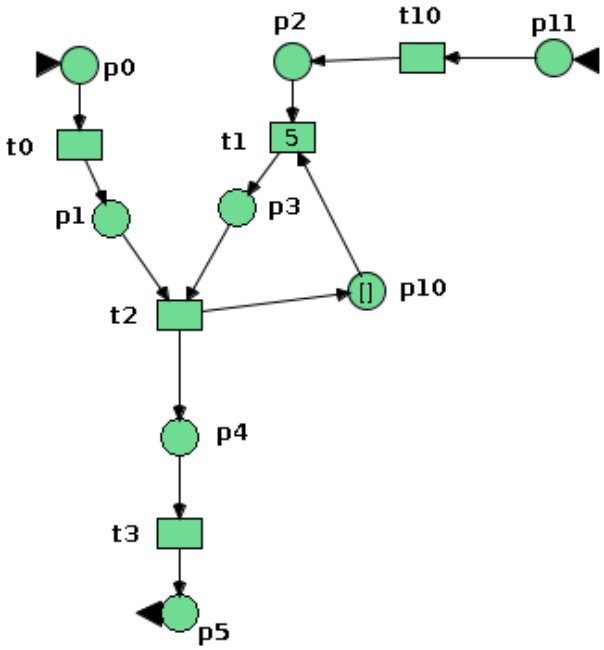


Figura 14: Subred de Petri con un ciclo.

vector de valores de lookahead con un valor para cada camino posible. Estos valores indican el número de ciclos que va a tardar como mínimo en producirse una marca siguiendo esos caminos.
Este vector va actualizando sus valores en cada disparo de transición, reduciendo el tiempo de los valores según el tiempo que tarda el disparo de esa transición. Para conocer la relación entre transiciones y caminos, cada transición tiene una variable para indicar a qué camino o caminos pertenece. Estas relaciones se asignan en la fase de compilación, iterando desde la transición de salida hacia sus transiciones anteriores. Si no hay ninguna marca en el camino, su valor de lookahead es la suma de las transiciones del camino.
Este método tiene un coste menor que el anterior, consiguiendo reducir el tiempo de simulación, pero el otro método permite simular casos más genéricos.

Se utiliza junto a la solicitud de lookahead de forma que cuando un SP quiere obtener el lookahead de otro SP, le envía un mensaje solicitándolo (request) y espera la respuesta. Un SP al recibir un mensaje request, obtiene el valor actual del vector de lookahead que ya está calculado y lo envía. De esta forma le comunica que como mínimo hasta dentro de esos ciclos no le va a enviar nada.
Se envía de forma inmediata para que el SP solicitante no tenga que esperar más tiempo del necesario, evitando que se retrase la simulación. Además, en caso de que el lookahead a enviar sea igual al último enviado, no se envía hasta que tenga un nuevo valor. Esto puede ocurrir ya que un SP puede ir más rápido que otro, o incluso por temas de latencia de la red de comunicación, el valor de lookahead puede no estar actualizado en el momento en que se recibe la solicitud. De esta forma se reduce el número de mensajes ya que si se enviase el mismo valor se volvería a repetir la solicitud hasta la obtención de un lookahead que permita continuar la simulación al solicitante.

Cuando un SP lee su vector de lookahead lo que hace es obtener el valor menor, el camino mínimo, para garantizar el tiempo mínimo en el que se podría enviar una marca.

Ejemplo: subred de Petri con dos caminos
En la figura 15 se muestra la misma subred que en el ejemplo del método 1. Aparecen tres estados diferentes de simulación y el vector de lookahead en cada caso. Estos estados no representan un avance temporal (no ocurren uno después de otro), son tres estados independientes para explicar el funcionamiento del método de cálculo.

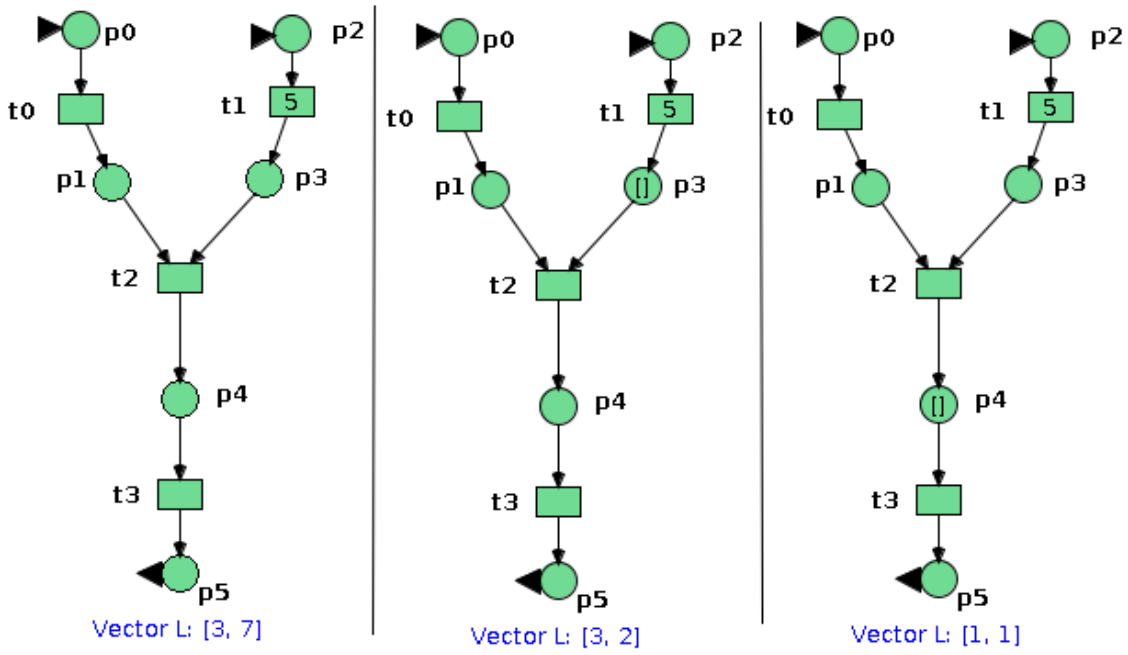


Figura 15: Subred de Petri con cálculo de vector de lookahead en diferentes estados de simulación.

La transición t2 tiene dos caminos dado que tiene como entradas dos lugares que son lugares de salida de las transiciones t0 y t1. En el estado de la izquierda no hay ninguna marca en la subred y los valores son la suma de los tiempos de disparo de las transiciones. El valor es [3,7] que sería el tiempo de marca de las transiciones de entrada. El primer valor corresponde al camino de la izquierda, que está formado por las transiciones: t3, t2, t0. El segundo valor es del camino de la derecha formado por las transiciones: t3, t2, t1.
Para el estado del centro de la figura el valor del vector de lookahead es [3,2] porque el lugar p3 tiene una marca. El primer camino sigue igual por lo que su tiempo no ha cambiado pero en el segundo sí que ha cambiado porque se ha restado al tiempo anterior el disparo de la transición t1.
En el estado de la derecha el vector de lookahead es [1,1] porque p4 tiene una marca, haciendo que para los dos caminos solo falte el disparo de la transición t3. Si en este estado se disparase t3, el vector de lookahead se actualizaría con los valores iniciales de la suma de tiempos ([3,7]) o con los valores correspondientes en caso de que hubiese más marcas en los caminos. En este último caso se tendría en cuenta el primer lugar

que tuviese marca, empezando a mirar transiciones desde la transición de salida hacia sus predecesoras.

3.4. Modificaciones para escalabilidad

Las simulaciones de modelos de un gran tamaño pueden requerir el uso de un número elevado de máquinas en las que se ejecuten los SPs de la simulación. Para simulaciones de alta escala el número de SPs podría ser de cientos o miles, utilizando entornos de despliegue como cloud.
En este tipo de simulación no se pueden utilizar los mecanismos desarrollados hasta la fecha porque no se diseñaron para funcionar en un entorno de gran tamaño. Cuando surgió la idea de simulador distribuido no se disponía del mismo número de máquinas que ahora ni de su potencia. Aunque se pueden cambiar algunas partes de los métodos y aplicar ciertas optimizaciones, como las explicadas en las secciones anteriores cambiando la gestión de lookahead, es necesario algún otro mecanismo para que la simulación de alta escala pueda ser viable.
Hasta la realización de este trabajo no se ha aportado ninguna gran contribución en este aspecto. En este proyecto se propone una solución para la simulación distribuida de alta escala. A continuación se va a presentar el problema que se produce y la solución propuesta.

Siguiendo el ejemplo de las redes de Petri de ramas secuenciales, si en cada rama hay millones de transiciones pueden ser necesarios múltiples SPs que simulen cada uno una parte de las transiciones de esa rama. Se debe a que podría no caber en memoria un número tan elevado de LEFs, haciendo necesaria su división en subredes. La figura 16 muestra una red de Petri de ramas secuenciales con un gran número de transiciones en cada rama, junto a los SPs que se obtendrían en la simulación de cada una de las ramas y como estarían conectados.
El problema que se muestra en la figura es que si se requieren X SPs para simular una rama, siendo X un número elevado, el SP X no recibe la información de SPs anteriores como SP 1 hasta que no ha pasado su información por todos los SPs, siguiendo un orden secuencial. En la simulación distribuida cada SP recibe información de su SP anterior, que es el que está conectado con él por el modelo obtenido con las subredes de Petri. Sin embargo, el disponer de información de otros SPs que sean anteriores aunque no tengan una relación directa puede ser útil para conocer si puede avanzar la simulación aunque su SP anterior no se lo haya confirmado.

La solución que se propone está inspirada en el funcionamiento del protocolo BGP [11] y se ha llamado simulación a nivel de regiones. Su objetivo es tener la información de los SPs anteriores, de manera rápida para poder propagar de una región a otra región y conseguir acelerar la simulación en modelos de alta escala con un número elevado de SPs.
Una región es simplemente un agrupamiento de un cierto número de SPs, que pueden ser de tres tipos:

- SP entrada: SP que simula una subred y recibe información de otra región.

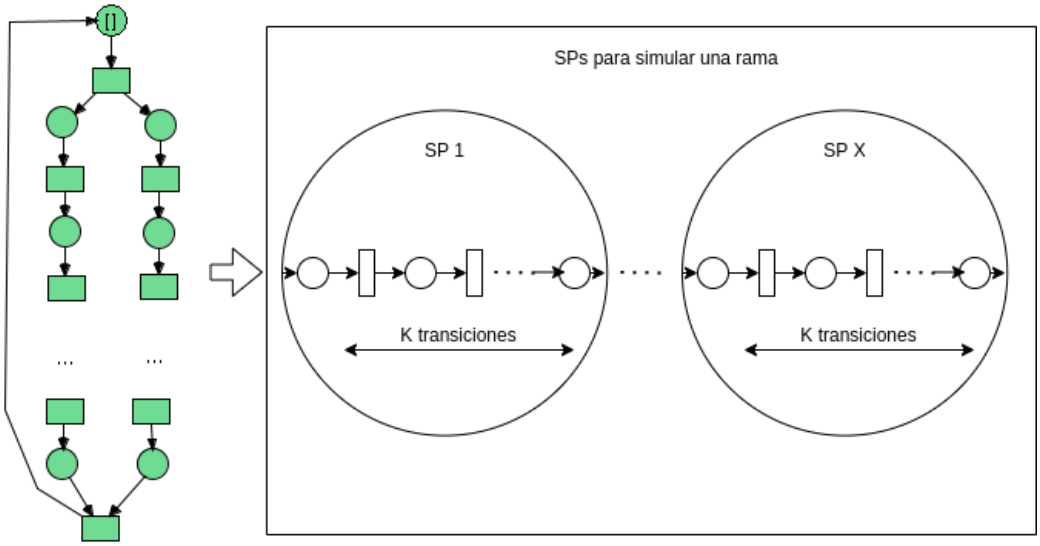


Figura 16: Red de Petri de ramas secuenciales (izquierda) y los SPs que se utilizarían para simular una de las ramas con un gran número de transiciones (derecha).

- SP intermedio: SP que simula una subred.
- SP salida: SP que simula una subred y recopila información de su región para transmitirla a otra región.

Así una simulación tiene una estructura jerárquica con diferentes regiones, cada una con varios SPs que se ejecutan en distintas máquinas. La gestión de lookahead es mediante solicitud, añadiendo una capa más para que el SP salida conozca los valores de lookahead de los SPs de la región. El SP salida tiene un componente llamado tabla de lookahead que consiste en una tabla con la información de lookahead de los SPs de la región. Su formato es que para cada SP que es predecesor y pertenece a la región, tiene la información de cuál es su SP anterior y su valor de lookahead, es decir hasta dentro de cuántas unidades de tiempo no va a enviar una marca. Además, cada lookahead tiene una etiqueta para saber si es un valor real (es el tiempo exacto en que se va a producir una marca) o hipotético (ese es el tiempo suponiendo que ese SP reciba una marca en su transición de entrada en ese ciclo de simulación).
La figura 17 muestra la arquitectura de esta solución.
De esta manera cuando por ejemplo el SP entrada tiene nueva información que puede propagar a su siguiente SP (SP intermedio 1) el SP salida puede disponer también de esa información. Si no se utiliza este método el SP salida tendría que esperar a que la información llegue a SP intermedio 1, se propague a SP intermedio 2 y finalmente le llegue a él. En este caso tendría que esperar más tiempo al sumarse los tiempos de procesamiento de los SPs y las latencias de la red de comunicación, empeorando las prestaciones de la simulación distribuida. En este ejemplo habrían sido necesarias tres comunicaciones en la red para ir de SP entrada a SP salida pero según el modelo puede ser mayor, mientras que con la solución propuesta solo se realiza una comunicación.

La figura 18 muestra el protocolo de comunicación de lookahead. Al igual que antes un SP solicita el lookahead al SP anterior y este le responde. Además, ahora también

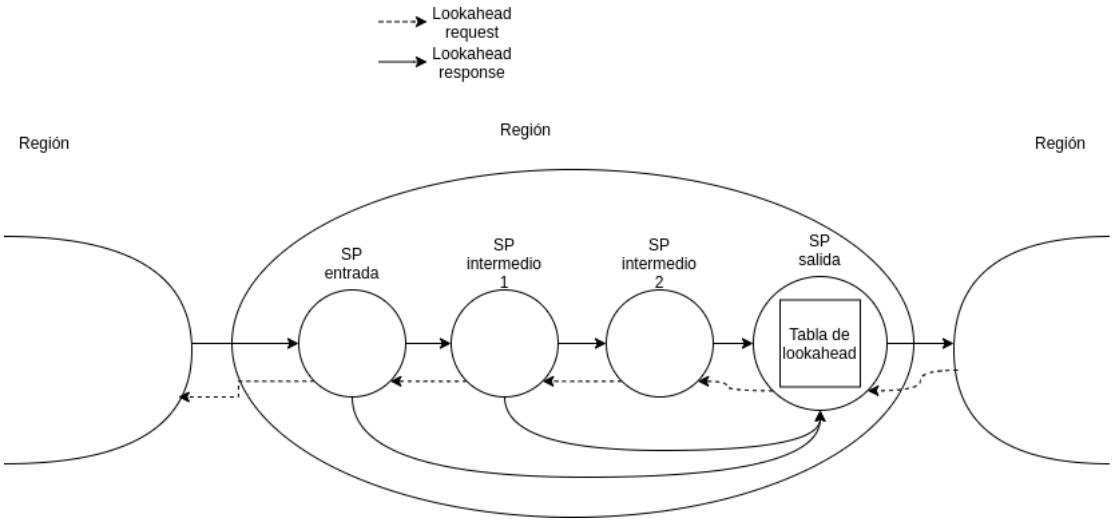


Figura 17: Arquitectura de simulación a nivel de regiones para una rama de una red de Petri de ramas secuenciales.

envía ese valor al SP salida para que disponga de la misma información que los SPs de la región.
Otra opción sería no enviarle el lookahead hasta que sea el SP salida el que realice una solicitud. Esto se realizaría cuando le soliciten el lookahead a él desde un SP entrada de otra región, cuya respuesta sería solicitar lookahead a su región y tras obtener todas las respuestas responder a la otra región. Esta opción tendría un tiempo más preciso pero un mayor coste de tiempo en obtener la respuesta, por lo que se ha optado por la primera opción.

Los SPs de salida tienen que gestionar la información recibida de la región actualizando la tabla de lookahead con los valores recibidos. Cuando reciben una petición de lookahead (desde otra región) deben aplicar un algoritmo de selección de camino utilizando la tabla lookahead. Este algoritmo consiste en iterar desde el propio SP salida hacia los SP predecesores hasta encontrar un valor de lookahead real, es decir, un camino que tenga alguna marca. En el recorrido se va sumando los tiempos de lookahead de cada unos de los SPs que sean hipotéticos, porque dependen de la información del SP anterior por lo que se consulta después a ese SP anterior. Finalmente se envía el tiempo obtenido al siguiente SP.
Además, en caso de que hubiese múltiples caminos (en las redes de Petri de ramas secuenciales no hay pero en otras puede haber), se toma el camino mínimo para tener el valor cuanto antes y continuar con la simulación.

La división en regiones y los roles de cada SP se deben realizar en la fase de compilación, mediante estrategias que deben ser correctamente analizadas para hacer un reparto adecuado que permita optimizar la simulación. Estas estrategias no se han definido porque son un aspecto importante a la par que complicado de analizar, que se podría hacer en otro trabajo.

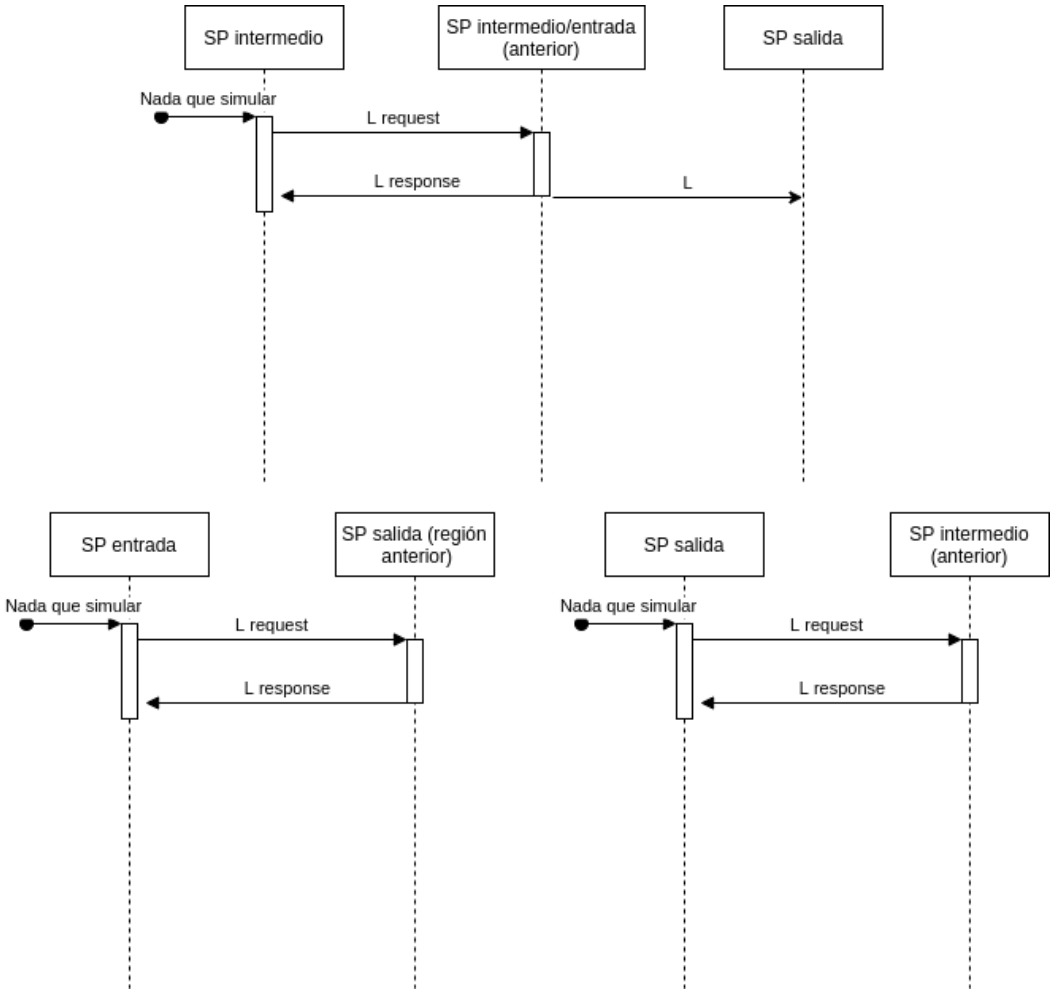


Figura 18: Diagramas de secuencia de solicitud y comunicación de lookahead entre los diferentes SPs de una simulación a nivel de regiones.

4. Implementación de los mecanismos de lookahead a incorporar al SP

La implementación del simulador distribuido desarrollado se realiza en el lenguaje de programación Rust. Como código base se utiliza el simulador del trabajo de un compañero del grupo de investigación, que se centra en mecanismos de balanceo de carga [7].
Una gran parte de este código se mantiene intacto, añadiendo los nuevos aspectos que se han realizado en este trabajo. Ahora se van a explicar las características más importantes de las cuales se hará hincapié en las novedades introducidas en el trabajo. También se comentarán en un alto nivel los atributos más importantes que ya estaban, los cuales se puede consultar más detalladamente en [7].

Cada SP tiene dos hilos de ejecución, hilo de simulación y Mailbox, que se comunican entre ellos mediante canales bidireccionales, es decir, multi-productores multi-consumidores utilizando la librería de Rust *crossbeam_channel* [12].
Al comienzo de la simulación el hilo Mailbox crea un servidor TCP y el hilo de simulación inicia las conexiones TCP con el resto de SPs con los que tenga comunicación. Estas conexiones están abiertas durante toda la simulación de forma que el servidor TCP de cada SP comprueba si alguna de sus conexiones abiertas tiene algún mensaje y realiza su tratamiento. La librería utilizada es *mio* [13] porque la transmisión de los mensajes es más rápida que con la librería estándar que se utilizó en primer lugar.
Al recibir un mensaje el hilo Mailbox comprueba de qué tipo es y se lo envía al hilo de simulación mediante los canales de comunicación. Los mensajes pueden ser de diferentes tipos ya sea para iniciar, acabar o realizar la simulación. Los más importantes son los de la simulación que pueden ser de los siguientes tipos:

- Event: mensaje que contiene un evento externo.
- RequestL: solicitud de lookahead.
- ResponseL: envío del valor de lookahead en respuesta a una solicitud previa.

Para el segundo método de cálculo de lookahead en simulación (vector de lookahead) se utilizan dos estructuras de datos adicionales para dar soporte a la gestión de lookahead. Una es para almacenar el valor actualizado de lookahead y la otra para llevar un registro de los últimos valores de lookahead enviados a los SPs adyacentes, y si hay una petición de lookahead pendiente de ellos o no. Ambas están compartidas entre los dos hilos en memoria compartida y su acceso es en exclusión mutua mediante el mecanismo mutex.
La primera de ellas contiene el valor de lookahead relativo, es decir, cuántas unidades de tiempo o ciclos de simulación va a tardar como mínimo en enviar una marca. Este valor lo va actualizando el hilo de simulación cuando el lookahead va cambiando, que es al disparar las transiciones o al recibir información del exterior.

La segunda estructura de datos es un mapa con el identificador del SP como clave y una variable de tipo *LookaheadRequest* como valor. Esta variable tiene dos elementos: el último tiempo (lookahead) enviado a ese SP, y si hay una petición pendiente o no.
El hilo Mailbox al recibir un mensaje de tipo RequestL obtiene el valor de lookahead actual, mediante la variable compartida, y lo compara con el último valor enviado. Este dato lo obtiene de la estructura de datos compartida consultando la variable LookaheadRequest. En caso de que sea mayor, envía el lookahead en un mensaje de tipo ResponseL. En caso contrario, marca la petición como pendiente para que el hilo de simulación la tenga en vigilancia y cuando disponga de un valor de lookahead mayor responda a esa petición, y actualice esa petición como no pendiente.

En cuanto a los tiempos de marca obtenidos en compilación, se guardan en un mapa con el identificador de la transición como clave y el tiempo de marca como valor. El uso de mapas se debe a que se quiere consultar los valores con un coste temporal bajo, que es O(1).
Además, también se ha modificado el compilador que está desarrollado en Java para obtener la información de la lista de transiciones de entrada para cada SP, la lista de transiciones predecesoras para cada una de las transiciones, y los tiempos de marca de las transiciones de salida. Estos cambios se han realizado en Java porque el compilador está desarrollado en ese lenguaje, aunque hay un compilador escrito en Rust realizado por otros compañeros del grupo de investigación que estaba en desarrollo durante la realización del trabajo.

En el anexo está el código implementado de los principales algoritmos. El código del algoritmo de simulación distribuida está en el anexo IV, el cálculo de lookahead en compilación en el anexo V y el cálculo dinámico de lookahead en simulación en el anexo VI.

5. Metodología para la prueba del prototipo y resultados experimentales

En este capítulo se explica como se pone en ejecución una simulación distribuida indicando los pasos que se realizan. Después se explica el entorno utilizado y las pruebas realizadas junto a sus resultados.

5.1. Proceso de despliegue

El lanzamiento de una simulación requiere de dos tareas: compilación y ejecución de la simulación. Primero se obtiene un modelo en la compilación que el SP carga en memoria para comenzar a simular.

5.1.1. Compilación del modelo

El modelo debe tener un formato asumible por el simulador, el cual se obtiene en el proceso de compilación. Este proceso utiliza el framework desarrollado en [5] y modificado en este trabajo con la adición para la implementación del cálculo de lookahead en compilación.
Las redes de Petri que se simulan en este trabajo son de ramas secuenciales como las de la figura 3 y se crean mediante un programa en Java. Este programa genera un fichero textual con la red de Petri definida en sus parámetros. Estos parámetros son: nombre del fichero de salida, número de ramas secuenciales (h) y número de transiciones en cada rama (v).
Este fichero tiene definida la red de Petri mediante un lenguaje de descripción de redes de Petri. Este lenguaje ya estaba en [5] para definir un modelo de red de Petri en formato textual.
Una vez obtenida la red de Petri en formato textual, se ejecuta el compilador. Su salida es una serie de ficheros que contienen las subredes en formato JSON con la información de los LEFs. Estos ficheros después serán absorbidos por el simulador distribuido en Rust.

5.1.2. Ejecución del simulador distribuido

Una vez obtenidas las subredes de Petri, ya se puede iniciar la simulación distribuida. Antes de la primera ejecución hay que compilar programa en Rust utilizando la herramienta cargo [14], el gestor de paquetes de Rust. El comando ejecutado para la compilación del código es:

```
cargo build --release
```

La opción –release es para que utilice una configuración definida en el fichero de configuración (cargo.toml). Esta configuración cuenta con optimizaciones como la opción

-o3 o lto=true, para aumentar el rendimiento del programa en binario que produce como salida.

Después se deben iniciar todos los SPs en las diferentes máquinas en las que se ejecuten de forma que el SP principal, el de índice cero por defecto definido en los ficheros de red, envía los LEFs a los otros SPs. De esta manera solo es necesario que estén los ficheros de las subredes en la máquina del SP principal. Este movimiento de LEFs se desarrolló en [7].
El comando para iniciar la simulación de un SP es:

```
./esimc <nombre_fichero_modelo><num_ciclos><lista_ip:port_SPs>
```

Siendo esimc el nombre del programa, num_ciclos el número de ciclos a simular y lista_ip la lista de SPs que se simulan en las diferentes máquinas (un SP por máquina).

De este modo se realiza la simulación distribuida y una vez que finaliza se muestran las trazas de la misma.

5.2. Entorno de pruebas

Para el estudio experimental se ha utilizado un entorno *on premise* de la Universidad de Zaragoza. Este entorno está compuesto de 48 máquinas Raspberry Pi 4 situadas en el laboratorio 1.03b de la Escuela de Ingeniería y Arquitectura. Sus características son las siguientes:

- Sistema operativo: Ubuntu 20.04.2 LTS Focal Fossa
- Procesador: ARM Cortex-172 con cuatro núcleos a 1,5 GHz
- Memoria RAM: 8 GB
- Red: 1Gbit/s

Se ha utilizado este entorno para realizar pruebas comparando las versiones de Java y Rust, y en cada lenguaje también se ha comparado entre la versión de simulación centralizada y distribuida. El número de máquinas utilizadas para la simulación distribuida ha ido variando según el modelo a simular y la etapa de desarrollo del proyecto. El mayor número de máquinas utilizadas ha sido de 8.

Cabe destacar que se han ido realizando pruebas a lo largo de todo el desarrollo del código. Sin embargo, solo se incluyen en esta memoria algunas de las principales que se han realizado, para demostrar el funcionamiento y escalabilidad conseguida al aumentar el número de máquinas o el tamaño del modelo.

5.3. Experimentos y resultados

En los experimentos realizados uno de los objetivos es comprobar a partir de qué tamaño la simulación distribuida presenta mejor rendimiento que la centralizada. Otro punto importante es comparar las versiones desarrolladas con diferentes algoritmos y cómo se comportan al ir aumentando el tamaño del modelo y el número de SPs.

La Tabla 1 muestra los resultados del simulador implementado en Rust. Esta tabla se publicó en la GECON 2021 [15], en un artículo en el que colaboré junto a otros miembros del grupo de investigación, y sus datos sirven como referencia. En la tabla se muestran los resultados al simular con una versión centralizada y distribuida un modelo de red de Petri de ramas secuenciales (como la figura 3). El tiempo de simulación para estas pruebas es de 10 millones de ciclos de simulación. La segunda columna (#br.) indica el número de ramas del modelo, la tercera (trans/br.) el número de transiciones que hay en cada una de las ramas, y la cuarta columna es el número de eventos totales que se han procesado. La siguiente columna indica el número de SPs (nodos) utilizados, ya que la red de Petri se divide en una subred para cada una de las ramas y otra subred con las transiciones de sincronización de las ramas. Cada una de las subredes se simula en un SP distinto, y cada SP se ejecuta en una máquina diferente. Las últimas dos columnas indican el resultado obtenido: el número de eventos procesados por segundo y el tiempo de ejecución de la simulación. El número de eventos procesados por segundo es la suma de todos los SPs excepto el que simula la subred de sincronización, ya que tiene un número despreciable de transiciones. Estos valores se utilizan para comparar los resultados obtenidos.

SP Simulator	#br.	trans/br.	Events	Nodes	Events / sec	Exec. Time
Centr.	2	10 000	19 998 003	1	7 522 936	2.658s
Distr.	2	10 000	19 998 003	3	2 609 886	7.619s
Centr.	2	100 000	19 999 803	1	7 529 029	2.656s
Distr.	2	100 000	19 999 803	3	4 407 922	4.537s
Centr.	7	10 000	69 988 013	1	3 909 926	17.79s
Distr.	7	10 000	69 988 013	8	6 652 821	10.52s

Tabla 1: Simulación distribuida vs simulación centralizada con diferentes cargas de trabajo por SP implementados en Rust. Lookahead: cálculo dinámico y envío al finalizar de simular.

Se han realizado tres simulaciones diferentes: red de Petri con dos ramas de diez mil transiciones por rama, red de Petri con dos ramas de cien mil transiciones por rama, y red de Petri con siete ramas de diez mil transiciones por rama. En los tres casos se ha ejecutado la versión centralizada y distribuida.
La primera observación que se puede hacer es que la versión centralizada presenta mejores resultados que la distribuida para los dos primeros modelos. En estos casos los modelos que se simulan en los SPs no tienen suficiente carga de trabajo y la sobrecarga añadida a la simulación distribuida no merece la pena. De todos modos se aprecia una mejora al pasar de una profundidad de rama de diez mil a cien mil transiciones. Para el último modelo la versión distribuida es más rápida que la centralizada debido a que tienen una mayor carga de trabajo en los SPs, obteniendo mejores resultados al distribuir el trabajo en los SPs en lugar de hacerlo en una sola máquina.
Como conclusión se puede extraer que como se esperaba al comienzo del proyecto, al aumentar el tamaño de los modelos la versión distribuida incrementa su rendimiento mientras que la centralizada lo empeora. Por tanto en estos casos es preferible el uso del simulador distribuido.

Otro dato a tener en cuenta es que el segundo modelo (cien mil transiciones por rama) tarda menos tiempo en simular que el primer modelo (diez mil transiciones por rama). Se debe a que el ciclo final de simulación es el mismo por lo que procesan el mismo número de eventos totales. El segundo modelo al tener mayor tamaño de rama, puede simular un mayor tiempo antes de tener que parar, transmitir mensajes por la red de comunicación y esperar la respuesta. El primer modelo al tener un menor tamaño de rama transmite un mayor número de mensajes haciendo que su simulación sea más lenta.

Estos datos son utilizando la gestión de lookahead con el algoritmo de cálculo dinámico y el envío cuando un SP no tiene nada que simular, es decir al final de un intervalo seguro de simulación. Además, en estas pruebas se utilizó comunicación TCP entre los SPs en la que para cada transmisión de un mensaje se creaba una nueva conexión TCP y después se cerraba. Esto fue claramente un error, que hace que el rendimiento sea algo peor.

Cabe decir que el cambio de lenguaje de programación de Java, que se usaba en el simulador de trabajos anteriores, a Rust es fundamental. Por ejemplo, la simulación del tercer modelo de siete ramas en Rust tarda 10.52s mientras que el mismo modelo en Java 43.4s, un tiempo cuatro veces mayor. Los datos de experimentos en Java se obtuvieron en las prácticas del máster.

En la Tabla 2 se muestran los resultados para la versión final del sistema. La comunicación sigue siendo TCP pero cada SP utiliza una única conexión con otro SP para la transmisión de mensajes que se cierra al finalizar la simulación. El algoritmo de cálculo de lookahead es el de vector de lookahead y su gestión es mediante solicitud cuando un SP requiere el valor un SP vecino.

SP Simulator	#br.	trans/br.	Events	Nodes	Events / sec	**Exec. Time**	Exec. Time (GECON)
Centr.	2	10 000	19 998 003	1	7 522 936	**2.658s**	2.658s
Distr.	2	10 000	19 998 003	3	5 437 474	**3.68s**	7.619s
Centr.	2	100 000	19 999 803	1	7 529 029	**2.656s**	2.656s
Distr.	2	100 000	19 999 803	3	8 354 348	**2.39s**	4.537s
Centr.	7	10 000	69 988 013	1	3 909 926	**17.79s**	17.79s
Distr.	7	10 000	69 988 013	8	8 281 665	**8.45s**	10.52s

Tabla 2: Simulación distribuida vs simulación centralizada con diferentes cargas de trabajo por SP implementados en Rust. Lookaehad: cálculo con vector de lookahead y solicitud de lookahead.

El formato de la tabla es el mismo añadiendo una última columna con los tiempos de ejecución anteriores. Los resultados del simulador centralizado son los mismos pero el distribuido presenta una mejora notable. En el primer modelo su tiempo de ejecución se ha reducido aproximadamente a la mitad, pasando de 7.62s a 3.68s.

El comportamiento es similar en el segundo modelo, pasando de 4.53s a 2.39s. Además, con este nuevo tiempo ya se consigue mejorar el resultado que se obtiene en el centralizado.
En el último modelo la ganancia es menor, consiguiendo una velocidad aproximadamente un 25 % mayor. La ganancia es menor porque las siete ramas tienen una carga de trabajo similar y al solicitar el lookahead lo hacen al mismo tiempo. El SP que simula la subred de Petri que tiene las transiciones de sincronización tarda más tiempo al tener que responder un mayor número de solicitudes, haciendo que los SP de las ramas estén más tiempo esperando la respuesta.
En este caso se puede observar que modificando los cálculos que se aplican se puede mejorar el rendimiento del simulador, pero depende en gran medida del modelo que se simule. Con este modelo el SP que simula la subred de sincronización tiene el problema de que es un "cuello de botella". Para mejorar el rendimiento sería necesario intentar modificar el modelo (cuando sea posible) para evitar estos problemas.

Además, se han utilizado más variables para lograr medir el rendimiento del simulador distribuido, que son las siguientes:

- P: rendimiento, eventos por segundo.
- E: densidad de eventos, eventos por segundo simulado (ciclos de simulación, valor máximo que alcanza LVT).
- R: velocidad de avance de la simulación (tiempo de simulación en segundos / tiempo real en segundos).
- L: lookahead, calculado en cada intervalo seguro de simulación.
- τ: latencia de comunicación entre los SPs.

Con estas variables se puede obtener el factor de acoplamiento (λ), con la siguiente ecuación:

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

El uso de esta métrica fue propuesto en [16] para medir el rendimiento del simulador conservativo según la carga de trabajo de cada SP. En el artículo se indica que si λ es menor que 10 es demasiado pequeño, mientras que si es mayor que 100 casi siempre es suficiente.

En las pruebas realizadas para la segunda versión del simulador la latencia obtenida es de 42 microsegundos, obteniendo un factor de acoplamiento de λ = 67.43 para el primer modelo, de dos ramas con diez mil transiciones por rama. El segundo modelo con dos ramas y cien mil transiciones por rama tiene λ = 665.49, y el tercer modelo con siete ramas de diez mil transiciones λ = 226.11.
Los datos muestran que el primer modelo se acerca a tener un valor adecuado (mayor a 100) pero no lo consigue, debido a que los SP no tienen suficiente carga de trabajo. Los otros dos tienen valores mejores que indican que tienen una carga de trabajo adecuada. Cabe decir que el valor λ puede ser útil para la interpretación de los resultados pero requiere otros análisis para evaluar de una forma más global el rendimiento del simulador.

Con estos resultados se puede concluir que ha merecido la pena el desarrollo de la segunda versión del simulador con la solicitud de lookahead entre otros aspectos. Gracias a esta versión se han mejorado los resultados de la primera versión que se publicó en GECON en 2021.
Respecto a la escalabilidad, con la última versión se ha dado un paso más para permitir la simulación de modelos de mayor tamaño. La principal aportación es la reducción del número de mensajes transmitidos por la red de comunicación, mediante la solicitud de lookahead cuando un SP necesite esa información. De este modo se evita saturar la red, que podría ocurrir con la primera versión del simulador para un modelo de gran escala, haciendo posible el planteamiento de la simulación a nivel de regiones. Este planteamiento añade una capa de complejidad más al simulador, haciendo que aumente el número de mensajes que se transmiten.
Por último, habría sido apropiado verificar el funcionamiento de las regiones y comprobar hasta qué punto permiten la simulación a alta escala. Por la falta de tiempo al realizar el proyecto, queda como trabajo futuro.

6. Conclusiones

El objetivo que tenía este trabajo era el desarrollo de un simulador distribuido que permitiera simular modelos de alta escala manteniendo unas prestaciones adecuadas. Con los resultados obtenidos, cabe decir que se ha obtenido un simulador más eficiente que el que había antes del trabajo. Gracias al exhaustivo análisis del problema de la gestión del lookahead, se ha conseguido profundizar en su problemática pudiendo desarrollar diferentes algoritmos que presentan un mayor o menor rendimiento en función del modelo a simular. Se debe a que la gestión del lookahead es el punto clave en la simulación distribuida para conseguir buenas prestaciones.
Además, se ha presentado el diseño de una mejora para la simulación de alta escala mediante la simulación a nivel de regiones. De este modo se podrían simular modelos más grandes con un mayor número de SPs con un coste menor que en las simulaciones distribuidas tradicionales. Este aspecto es importante ya que cuando diversos autores desarrollaron las simulaciones distribuidas, sus algoritmos no estaban pensados para modelos de alta escala, debido a las limitaciones de máquinas que había entonces pudiendo utilizar un menor número de máquinas.

Por tanto, en este trabajo se ha logrado dar un paso más en el desarrollo de un simulador distribuido completo y exhaustivo que el grupo de investigación COSMOS trabaja para conseguir. Además, en el trabajo se ha colaborado en un artículo de investigación publicado en GECON [15] que ha sido elegido como el mejor artículo de la conferencia.

6.1. Trabajo futuro

Este trabajo proporciona una base para que en proyectos posteriores se pueda desarrollar un sistema completo de simulación. Algunas de las tareas que se podrían realizar en el futuro son:

- Implementar y realizar pruebas de la simulación a nivel de regiones. En este trabajo se ha realizado el diseño de este aspecto enfocado a la simulación de alta escala pero por falta de tiempo no se ha llegado a implementar. El siguiente paso sería añadir su implementación al código desarrollado y hacer experimentos para comprobar su validez y comparar el rendimiento obtenido respecto a no usarlo.

- Permitir la simulación de modelos más complejos. Se podrían ampliar las redes de Petri permitidas con aspectos como el uso de tiempos estocásticos, los cuales no se utilizan ahora ya que los tiempos usados son deterministas. Los tiempos estocásticos requieren un mayor análisis ya que al contrario que ahora no se puede utilizar la información de tiempos de marca obtenida en la fase de compilación. De este modo al obtener el lookahead habría que obtener una cota mínima, impidiendo encontrar un tiempo preciso como se consigue ahora.

  También se podrían incluir los conflictos, cuyo tratamiento requiere un mayor análisis que el que se ha realizado en el trabajo. En la versión actual se puede

conocer el camino que se va a seguir y por tanto es posible obtener el tiempo utilizando información de compilación. Sin embargo, si la red de Petri tiene algún conflicto no se conoce su resolución en tiempo estático. Es necesario conocer la resolución del conflicto en tiempo de ejecución que dependerá de la política de resolución de conflictos, las cuales existen múltiples alternativas. Por ejemplo, una de las políticas más sencillas es disparar la transición que más tiempo lleve sin ser disparada. Cabe destacar que con el algoritmo desarrollado de cálculo de lookahead dinámico en tiempo de simulación se proporciona una base para permitir la gestión de los conflictos. Por tanto con ligeras modificaciones ya se podrían incluir los conflictos, mientras que con otros algoritmos como el otro desarrollado de vector de lookahead no es posible y habría que replantear el algoritmo.

- Otro punto sería realizar un análisis de la frecuencia con la que se debería actualizar el lookahead. Consistiría en encontrar el equilibrio entre valor actualizado y tráfico de red de comunicación que permita tener mejores prestaciones, para conseguir simulaciones más eficientes.

- Un aspecto muy importante sería añadir un depurador de redes de Petri en la simulación. Al simular redes de Petri de cada vez un tamaño mayor, su verificación es un tema que puede ser complejo y requerir bastante tiempo. Un depurador facilitaría el trabajo, con opciones como pueden ser el simular hasta un cierto ciclo de simulación y mostrar los valores de las variables.

- Otros aspectos de mayor ambición con un objetivo a más largo plazo, como la tolerancia a fallos y extensiones de otros trabajos del grupo de investigación como los que se explican en las secciones de trabajo futuro en [7, 17].

6.2. Esfuerzos dedicados

En la Tabla 3 se muestra el tiempo invertido en las diferentes partes del proyecto. Este TFM se inició en junio de 2021 aunque los dos meses anteriores (abril y mayo de 2021) fueron de prácticas académicas en la misma línea de trabajo, realizando el simulador distribuido en Java. Estas prácticas duraron en torno a 225 horas, que con las de este trabajo hacen un total de aproximadamente 675 horas dedicadas al ámbito de la simulación distribuida.

6.3. Evaluación personal

Tras haber concluido el proyecto y evaluando el trabajo realizado, valoro este proyecto como una experiencia positiva. En mi formación durante los años del grado y el tiempo del máster he estudiado varios sistemas distribuidos y me he enfrentado a trabajos de un cierto tamaño y complejidad. Sin embargo, este trabajo ha supuesto un reto mayor de lo que me esperaba debido a que las simulaciones distribuidas tienen una dificultad de una escala mayor a lo que había estudiado y trabajado hasta ahora.

Tarea	Horas
Revisión de la bibliografía publicada y trabajos previos	15
Aprendizaje del lenguaje Rust	30
Familiarización con el código base	10
Diseño e implementación de los mecanismos de lookahead	100
Diseño de la simulación a nivel de regiones	50
Depuración y pruebas	100
Reuniones	55
Memoria	90
Total	450

Tabla 3: Dedicación de horas.

En general estoy satisfecho con el trabajo realizado a pesar de que he tenido momentos de cierta dificultad, especialmente con la problemática del lookahead que me llevó más tiempo del esperado.

El trabajo que se proponen en el grupo de investigación es ambicioso y me alegro de haber podido aportar una parte, para dar un paso más hacia su objetivo. Sin duda recomendaría a otros estudiantes que realicen algún trabajo en algún grupo de investigación durante el grado o máster porque es una experiencia diferente que todos los estudiantes deberíamos adquirir.

7. Bibliografía

[1] Ferscha, A.: Parallel and Distributed Simulation of Discrete Event Systems. Handbook of Parallel and Distributed Computing, McGraw-Hill (1995).

[2] Arronategui, U., Bañares, J.Á., Colom, J.M.: Towards an architecture proposal for federation of distributed DES simulators. In: GECON 2019 - International Conference on the Economics of Grids, Clouds, Systems, and Services. pp. 197-110. Springer (2019).

[3] Vanmechelen, K., De Munck, S., Broeckhove, J.: Conservative distributed discrete-event simulation on the amazon ec2 cloud: An evaluation of time synchronization protocol performance and cost efficiency. Simulation Modelling Practice and Theory 34, 126–143 (2013).

[4] Bañares, J.Á., Colom, J.M.: Model and simulation engines for distributed simulation of discrete event systems. In: International Conference on the Economics of Grids, Clouds, Systems, and Services. pp. 77-91. Springer (2018).

[5] Herrero Barco, S.: Desarrollo de un Framework de Simulación de Sistemas de Eventos Discretos Complejos. Trabajo Fin de Grado Ingeniería Informática, Zaragoza (2020).

[6] Lenguaje de programación Rust. https://www.rust-lang.org/es. Accessed: 2021-11-26.

[7] Santamaría de la Fuente, A.: Diseño e Implementación de un Simulador Distribuido de Eventos Discretos con Mecanismos de Balanceo de Carga. Trabajo Fin de Grado Ingeniería Informática, Zaragoza (2021).

[8] Entorno de desarrollo Visual Studio Code. https://code.visualstudio.com/. Accessed: 2021-11-26.

[9] Extensión de Visual Studio Code para el lenguaje de programación Rust. https://marketplace.visualstudio.com/items?itemName=rust-lang.rust. Accessed: 2021-11-26.

[10] Briz, J.L., Colom, J.M.: Implementation of weighted place/transition nets based on linear enabling functions. In:International Conference on Application and Theory of Petri Nets, pp. 99–118. Springer (1994).

[11] Protocolo BGP. https://www.techtarget.com/searchnetworking/feature/BGP-tutorial-The-routing-protocol-that-makes-the-Internet-work. Accessed: 2021-11-26.

[12] Librería crossbeam_channel de canales multiproductor-multiconsumidor. https://docs.rs/crossbeam-channel/0.5.1/crossbeam_channel/. Accessed: 2021-11-26.

[13] Librería mio de comunicación de red rápida y de bajo nivel. https://docs.rs/mio/0.8.0/mio/. Accessed: 2021-11-26.

[14] Cargo: Gestor de paquetes de Rust. https://crates.io/. Accessed: 2021-11-26.

[15] Hodgetts, P., Kocharyan, H., Reviriego, F., Santamaría, Á., Arronategui, U., Bañares, J.Á., Colom, J.M.: Workload Evaluation in Distributed Simulation of DESs. In: GECON 2021.

[16] 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.

[17] Kocharyan, H.: Automatización del despliegue de simulaciones distribuidas en cloud híbrido. Trabajo Fin de Grado Ingeniería Informática, Zaragoza (2021).

8. Anexos

I. Función lineal de sensibilización de una transición (Linear Enabling Function, LEF)

Las funciones lineales de sensibilización de una transición o LEFs (Linear Enabling Function of a Transition) permiten caracterizar cuando una transición está sensibilizada con una simple función lineal dependiente del marcado.
Una LEF de una transición t es una función f_t:$\mathbf{R}(N,m_0) \rightarrow Z$ que hace corresponder a cada marcado alcanzable de la red de Petri, $m \in \mathbf{R}(N,m_0)$, un entero de manera que la transición t está sensibilizada si y solo si $f_t(m) \leq 0$. Por ejemplo, para la transición T2 de la red de la figura 19 su LEF es: $f_{T2}(m) = 2-(m[A]+m[D]), \forall m \in R(N,m_0)$, donde m_0 es el marcado inicial. Por tanto, $f_{T2}(m_0) = 2-(m_0[A]+m_0[D]) = 2-1 = 1$, es decir, la transición T2 no está sensibilizada en el marcado inicial. Si observamos la transición T1, su LEF es: $f_{T1}(m) = 1-(m[A])$, con $f_{T1}(m_0) = 0$, por lo que está sensibilizada en el marcado inicial.

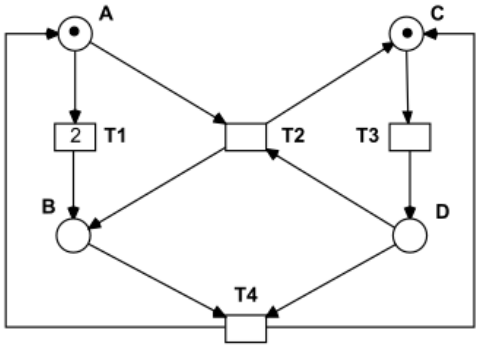
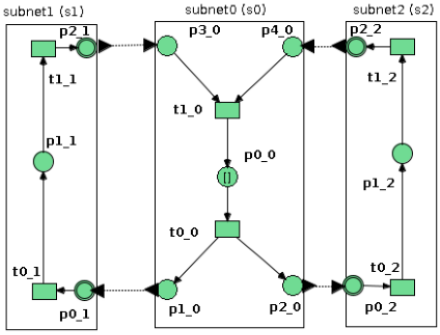


Figura 19: Red de Petri (extraída de [4]).

Para una información más detallada consultar [2, 4]

II. Ejemplo de traza de la simulación

En la figura 20 se muestra la traza de la simulación de una red de Petri compuesta por dos ramas de dos transiciones cada una (s1 y s2). La simulación tendría tres SPs, uno para cada rama y otro para la parte de sincronización de ramas (s0). Los tiempos de disparo de todas las transiciones son de una unidad de tiempo.
IB (Input Buffer) son los mensajes recibidos, OB (Output Buffer) son los mensajes que se envían y external_evs son los eventos generados cuyo destino es otro SP pero que aún no se han enviado.
No se muestra la traza de la subred 2 porque es idéntica a la de la subred 1, con los nombres de transiciones y SP destino correspondientes. Además, se muestra la traza hasta que finaliza un ciclo. El último intervalo (5) es idéntico al primer intervalo pero con mayores tiempos.



	Intervalo		Subred0			lef	lef				
		disparo	IB	LHVT	LVT	t0_0	t1_0	enabled_tr	FUL	external_evs	OB
Paso inicial				0	0	0	2	t0_0@0			s1<null;1> s2<null;1>
recibir	1		s1<null;2> s2<null;2>	0	0	0	2	t0_0@0			
LHVT				2	0	0	2	t0_0@0			
simula				2	0	0	2	t0_0@0			
disparo		t0_0,0		2	2	1	2			s1<t0_1;-1;1> s2<t0_2;-1;1>	
Envio				2	2	1	2				s1<t0_1;-1;1> s2<t0_2;-1;1>
recibir	2		s1<null;3> s2<null;3>	2	2	1	2				
LHVT				3	2	1	2				
simula				3	3	1	2				
disparo				3	3	1	2				
Envio				3	3	1	2				s1<null;5> s2<null;5>
recibir	3		s1<null;3> s2<null;3>	3	3	1	2				
LHVT				3	3	1	2				
simula				3	3	1	2				
disparo				3	3	1	2				
Envio				3	3	1	2				s1<null;5> s2<null;5>
recibir	4		<t1_0;-1; 3><t1_0;-1; 3>	3	3	1	2				
LHVT				3	3	1	2		<t1_0;-1;3> <t1_0;-1;3>		
simula				3	3	1	0	t1_0@3			
disparo		t1_0,3		3	3	1	2		<t0_0,-1,4>		
Envio				3	3	1	2		<t0_0,-1,4>		s1<null;5> s2<null;5>
recibir	5		s1<null;7> s2<null;7>	3	3	1	2		<t0_0,-1,4>		
LHVT				7	3	1	2		<t0_0,-1,4>		
simula				7	4	0	2	t0_0@4			
disparo		t0_0,4		7	4	1	2			s1<t0_1;-1;5> s2<t0_2;-1;5>	
Envio				7	7	1	2				s1<t0_1;-1;5> s2<t0_2;-1;5>

Intervalo		Subred1			lef	lef				
	disparo	IB	LHVT	LVT	t0_1	t1_1	enabled_tr	FUL	external_evs	OB
			0	0	1	1				
1		s0<null;1>	0	0	1	1				s0<null;2>
			1	0	1	1				
			1	0	1	1				
			1	1	1	1				
			1	1	1	1				s0<null;3>
2		<t0_1;-1;1>	1	1	1	1				
			1	1	0	1		<t0_1;-1;1>		
			1	1	0	1	t0_1@1			
	t0_1,1		1	1	1	1		<t1_1;-1;2>		
			1	1	1	1		<t1_1;-1;2>		s0<null;3>
3		s0<null;5>	1	1	1	1		<t1_1;-1;2>		
			5	1	1	1		<t1_1;-1;2>		
			5	2	1	0	t1_1@2			
	t1_1,2		5	2	1	1			s0<t1_0;-1;3>	
			5	5	1	1				s0<t1_0;-1;3>
4		s0<null;5>	5	5	1	1				
			5	5	1	1				
			5	5	1	1				
			5	5	1	1				s0<null;7>
5		s0<null;5>	5	5	1	1				
			5	5	1	1				
			5	5	1	1				
			5	5	1	1				<null;7>

Figura 20: Traza de simulación distribuida en una red de Petri de dos ramas de dos transiciones cada una.

III. Algoritmo lookahead para redes simples (versión anterior)

La primera versión del algoritmo de lookahead que se diseño e implementó estaba enfocada para redes simples, concretamente para redes de ramas secuenciales. Este algoritmo es más sencillo y sus prestaciones son ligeramente mejores al algoritmo utilizado en el proyecto para las redes muy simples. Sin embargo, tiene más restricciones ya que no permite la existencia de ciclos en la red de Petri a simular. Además, en caso de redes con varios caminos posibles, este algoritmo no es eficiente dado que toma el camino de peor tiempo sin considerar el estado de la red.
El problema de este algoritmo es que no explota la información de la estructura del modelo para conocer de que transiciones depende otra transición. Esto ocurre cuando una transición tiene más de un lugar de entrada, ya que con este algoritmo se calcula el tiempo menor, es decir, si hay dos caminos para llegar a una transición, aquí se devolvería siempre el de menor tiempo.
Esto es mejorable ya que se debe explotar la información de la estructura del modelo para conocer cual es el camino a tener en cuenta. Por ejemplo, si una transición tiene dos caminos, uno que tarda 4 unidades de tiempo en quitar una marca al LEF y otro que tarda 6 unidades tiempo, solo se tomará el de 4 unidade cuando LEF=1. Si LEF=2, hay que tomar el camino de 6 unidades porque en 4 unidades de tiempo al LEF se le quitará una marca pero aún necesita otra, que llegará más tarde.
Por tanto hay que tener en cuenta cual es el estado de la red para poder obtener información precisa de lookahead.

Cabe destacar que el cálculo en compilación es similar en ambos algoritmos salvo que en la versión anterior se obtienen los tiempos de marca ordenados de menor a mayor tiempo. En cambio el algoritmo de simulación es muy diferente.
El cálculo en dos fases es el siguiente.

III.1. Cálculo en simulación

El lookahead se calcula en el algoritmo de simulación distribuida tras finalizar el intervalo seguro de simulación. Una vez que acaba de simular para un horizonte LHVT dado, se produce el envío de los eventos exteriores, y para los SPs a los que no se envía evento, se calcula y envía su lookahead.

El pseudocódigo se puede ver en la figura 21.

Esta versión tiene la limitación de que las redes de Petri utilizadas deben ser redes binarias sin conflicto. Además, las transiciones de entrada solo pueden tener un lugar de entrada, es decir, no se permiten varios lugares de entrada de la subred en la misma transición. Tampoco se permite que una transición tenga un lugar de entrada de la subred y otros lugares de entrada internos.

Para obtener el valor del lookahead de una transición de salida, tomando como valor inicial un valor elevado MAX (línea 2), se recorre la lista de tiempos de esa transición de salida (líneas 3-22). La lista contiene relaciones de identificador de transición y

```
1:  func get_lookahead(trans):
2:      min ← MAX
3:      for all ([trans_id,TM] ∈ trans.times) do
4:          if (min <= (LVT + TM)) then
5:              break
6:          end if
7:
8:          if (is_entry(trans_id)) then
9:              time ← get_ext_time(trans_id) + TM
10:             if (time < min) then
11:                 min ← time
12:             end if
13:         else then
14:             lef_val ← get_lef_value(trans_id)
15:             if (num_events_FUL(trans_id) >= lef_val) then
16:                 time ← time_event_FUL(trans_id, lef_val) + TM
17:                 if (time < min) then
18:                     min ← time
19:                 end if
20:             end if
21:         end if
22:     end for
23:     return min
24: end func
```

Figura 21: Cálculo de la segunda parte del lookahead en la simulación de la red.

tiempo. Esa lista se calculó en compilación y está ordenada de menor a mayor tiempo. Para cada valor en la lista, hace las siguientes tareas:

- **Comprobar obtención del camino mínimo** (líneas 4-6): mientras que el candidato a valor mínimo encontrado hasta el momento sea mayor que el tiempo que se podría obtener con otra transición (min $>$ LVT + TM), se debe seguir iterando para encontrar el valor mínimo real. Por tanto, en el momento en que LVT sumado al valor de TM de la transición a mirar sea igual o mayor que el candidato mínimo actual (min $\leq$ LVT + TM), no es necesario seguir iterando. Esto se debe a que los valores de TM aumentan por iterar las transiciones en orden de tiempo, obteniendo unos valores de tiempo finales mayores que el que ya se ha obtenido.

  La comparación se realiza tomando el menor tiempo posible que se podría obtener en la transición actual: LVT + TM. Este tiempo podría ser mayor, ya que para tomar el valor de LVT debería ser una transición de entrada cuya última información recibida de lookahead fuese igual a LVT. En cualquier otro caso, el tiempo que se obtendría sería X + TM, donde X $>$ LVT.

  Esta comprobación de parada se realiza al principio del bucle, que es cuando es necesaria. En caso de hacerla al final, sería menos eficiente porque haría las

comprobaciones de esa transición aunque no aportan información relevante.

- **Tratamiento de la transición** (líneas 8-21): la transición tiene un tratamiento especial para el caso de que sea de entrada (tiene un lugar de entrada que es interfaz de entrada de la subred), y en caso contrario tiene otro procedimiento.

  La transición es de entrada (líneas 8-12): hay que utilizar la información de lookahead recibida para esa transición. El tiempo de ese camino sería el último valor de lookahead recibido para esa transición sumado a TM (línea 9). En caso de que sea el menor encontrado hasta el momento, se guarda su valor (líneas 10-12). Este tiempo es la estimación del ciclo en el que como mínimo la transición podría estar sensibilizada, en caso de que reciba un evento del correspondiente SP, más el valor de TM.

  La transición estará sensibilizada en un tiempo futuro (líneas 13-21): si la transición no es de entrada hay que comprobar si con los eventos pendientes en FUL estará sensibilizada en un ciclo futuro. Primero se obtiene cuantas marcas necesita para estar sensibilizada, es decir, su valor de LEF (línea 14). Después se comprueba si con los eventos en FUL se puede sensibilizar la transición (línea 15). Al ser redes binarias cada evento dirigido a la transición quita una marca al LEF (en redes de Petri sería añadir una marca a un lugar de entrada).

  Si hay tantos eventos como el valor de LEF, se obtiene su tiempo y en otro caso se deja de mirar esa transición y se consulta la siguiente. Se admite que haya más eventos que el valor de LEF, para permitir redes con múltiples disparos de una transición en distintos tiempos. Por ejemplo, una transición t_i con tiempo de disparo de 3 unidades de tiempo podría dispararse en el tiempo T=1 y T=3. En T=3, en FUL habría dos eventos generados por esa transición, uno para T=4 y otro para T=6.

  Si la transición estará sensibilizada con los eventos en FUL, se obtiene su tiempo (línea 16) y si es el menor encontrado se guarda su valor (líneas 17-19). El tiempo es el valor de TM más el tiempo del evento que sensibilice la transición que se consulta. Esto es el evento que al procesar permita que se pueda disparar la transición. Por ejemplo, si LEF=2 se toma el tiempo del evento de la segunda marca, es decir, el segundo evento de FUL dirigido a la transición. Estos eventos están ordenados de menor a mayor tiempo.

Finalmente se devuelve el valor mínimo obtenido (línea 23).

Al estar ordenada la lista de tiempos e iterar hasta tener garantía de encontrar el camino mínimo, siempre se devuelve el **tiempo mínimo**, garantizando la **simulación conservadora**.

Es necesario mirar si hay eventos pendientes de procesar en FUL porque cuando se calcula el lookahead ya se ha simulado hasta el horizonte temporal. Por tanto no hay transiciones sensibilizadas ya que en caso de haberlas ya se habrían disparado, por lo que hay que comprobar si hay eventos pendientes para tiempos de simulación mayores a LVT.

Hay un caso especial que es al inicio de la simulación. En esta fase, antes de simular se calculan y envían los valores de lookahead para los SPs sucesores. En este punto no hay ningún evento en FUL pero hay transiciones sensibilizadas por el marcado inicial. El pseudocódigo de este cálculo se puede ver en la figura 22.

```
1: func get_lookahead_initial(trans):
2:     for all ([trans_id,TM] ∈ trans.times) do
3:         if (is_enabled(trans_id) or is_entry(trans_id)) then
4:             return LVT + TM;
5:         end if
6:     end for
7: end func
```

Figura 22: Cálculo de la segunda parte del lookahead en la simulación de la red: fase inicial.

Su funcionamiento consiste en iterar la lista de tiempos de marca, mirando si la transición que se comprueba está sensibilizada o es de entrada (línea 3). Si se cumple alguna condición, ese es el valor mínimo (línea 4).
Este cálculo se debe a que las siguientes transiciones que se miren tendrán un tiempo de marca igual o mayor, por lo que el camino mínimo ya se ha obtenido. En esta fase aún no se ha obtenido información de lookahead de los SPs predecesores por lo que se asume que en el ciclo actual, LVT, podría llegar algún evento.

Ejemplo
Siguiendo el ejemplo de compilación de la figura 10, en ese caso en el que no hay ninguna marca, el valor de lookahead que se obtendría sería: lookahead(t0) + 3. Este valor es el último valor de lookahead recibido para la transición t0 más su tiempo de marca, que es 3 unidades de tiempo. Su obtención se debe a que al aplicar el algoritmo no hay ninguna transición sensibilizada con los eventos en FUL. Al seguir iterando se obtiene que t0 es de entrada por lo que se guarda su tiempo como el mínimo hasta el momento. Después se comprueba t1 (como su TM es mayor se comprueba después), se obtiene su tiempo porque es de entrada pero como no es el menor encontrado no se guarda.

En este ejemplo se aprecia que el resultado no es óptimo ya que el verdadero valor de lookahead debería ser el de t1: lookahead(t1)+7. Excepto que lookahead(t0)+3 sea mayor que ese valor, en cuyo caso sería el de t0.
No se obtiene ese valor porque con este algoritmo no se tratan los valores de LEF para conocer cuantas marcas requiere, por lo que se toma el camino mínimo para garantizar su ejecución, aunque en casos como este es demasiado pesimista.

IV. Implementación del algoritmo de simulación distribuida

En la figura 23 se puede ver el código en Rust del algoritmo de simulación distribuida utilizando la solicitud de lookahead.

```rust
// Initial lookahead calculation and update
update_initial_lookahead(self,lefs);

self.initialize_tcp_connection(lefs)?;
// Para cada SP siguiente inicializar self.l_requests
self.initialize_simbots_requests(lefs);

// STEP 1: SYNC ALL NODES --> BARRIER
barrier_sync(self, index, rx2, &list_nodes)?;
println!("Barrier ended");

let latency = ping(self, index, rx3, &list_nodes)?;
println!("LATENCY is : {}", latency);
//println!("SINCRONIZADOS");

println!("VT initial: {}", self.local_clock.0);
println!("event list start simulation: {:?}", self.event_list);

// simulation loop
let now = Instant::now(); //get init real time for becnhmark

let mut LVHT = self.local_clock;
let mut lookahead = 0;

while self.local_clock < final_cycle {
    let old_fired_trans = self.fired_transitions.len();
    let old_LVHT = LVHT;

    // Check if have enough information from the neighbors
    // to advance LHVT.
    // Otherwise, it gets any pending event if there is or
    // requests L and waits for the respone
    // Using crossbeam channel
    LVHT = check_neighbours(self, &mut fifoqueues_r, lefs);

    println!("VT: {}; LHVT: {}", self.local_clock.0, LVHT.0);

    // Update lookahead variable to display results
    if LVHT.0 - old_LVHT.0 > 0 {
        lookahead = LVHT.0 - old_LVHT.0;
    }

    if LVHT > final_cycle{
        LVHT = final_cycle
    }

    self.simulate_one_step(lefs, LVHT);


    if LVHT == final_cycle{
        break;
    }
}
let seconds_elapsed = now.elapsed().as_secs_f64();
finish_simulation(self, index, rx5, &list_nodes, lefs);
```

Figura 23: Implementación en Rust del algoritmo de simulación distribuida (versión con solicitud de lookahead).

V. Implementación del cálculo de lookahead en compilación (tiempos de marca)

La obtención de los tiempos de marca para el cálculo de lookahead en la fase de compilación está implementado en el lenguaje de programación Java. En la figura 24 se puede ver el código. Se realiza en Java porque el compilador está desarrollado en este lenguaje.

```
public Listatiempos dame_tiempohastamarcas (int codtrans,int codsubred){
    marcastiempo = new Listatiempos();
    // Tiempo hasta la transicion actual
    int timeToMark = 0;
    // Tiempo de marca de la transicion actual
    int timeMark = 0;

    LinkedHashSet<Integer[]> trans_pendientes = new LinkedHashSet<Integer[]>();
    // Lista de transiciones ya vistas
    LinkedHashSet<Integer> trans_vistas = new LinkedHashSet<Integer>();

    // Añadir la transicion de salida
    trans_pendientes.add(new Integer[]{codtrans,timeToMark});

    int li_i,li_j,li_cod_lugar,li_transicion, codtranssubred;
    while (!trans_pendientes.isEmpty()) {
        // Obtener transicion junto a su tiempo de marca
        Integer[] trans = trans_pendientes.iterator().next();
        codtrans = trans[0];
        timeToMark = trans[1];
        // Eliminar transicion de la lista de pendientes
        trans_pendientes.remove(trans);

        // Guardar la transicion como vista
        trans_vistas.add(codtrans);

        // Calcular el valor de lo que cuesta llegar desde la transicion de salida
        // hasta la actual más el tiempo de disparo de esta transicion
        timeMark = timeToMark + ia_transiciones[codtrans].ii_tiempo;
        codtranssubred = dame_cod_trans_subred(codtrans,codsubred);
        // relleno el vector de tiempos con el tiempo desde la transicion
        // de salida hasta la actual, que es li_dev
        marcastiempo.inserta(codtranssubred, timeMark);

        // Obtener las transiciones de entrada de los lugares de entrada,
        // Para cada lugar de entrada
        for (li_i=0;(li_i<ia_transiciones[codtrans].ia_pre.length) &&
                    (ia_transiciones[codtrans].ia_pre[li_i][0]!=ERROR);li_i++){
            li_cod_lugar =ia_transiciones[codtrans].ia_pre[li_i][0];
            // para cada una de sus transiciones de entrada
            for (li_j=0;(li_j<ia_lugares[li_cod_lugar].ia_pre.length) &&
                        (ia_lugares[li_cod_lugar].ia_pre[li_j][0]!=ERROR);li_j++){
                li_transicion=ia_lugares[li_cod_lugar].ia_pre[li_j][0];
                // Calcular tiempo solo si no se ha visto ya la transicion
                if (!trans_vistas.contains(li_transicion)) {
                    trans_pendientes.add(new Integer[]{li_transicion,timeMark});
                }
            }
        }
    }
    return marcastiempo;
}
```

Figura 24: Implementación en Java del cálculo de lookahead en compilación (tiempos de marca).

VI. Implementación del cálculo dinámico de lookahead en simulación

En la figura 25 y 26 se puede ver el código en Rust del cálculo dinámico de lookahead en simulación.

```rust
pub fn lookahead(&mut self, LVT: SimulatedClock, event_list: &BinaryHeap<Event>) -> SimulatedClock {
    for transition in self.transition_list.iter() {
        if transition.is_output
        {
            // Lista transiciones vistas (indices locales)
            let mut trans_seen = HashSet::new(); //Vec<LocalTransIndex> = Vec::new();
            // Lista (pila) de transiciones pendientes de comprobar (indices locales)
            let mut trans_pending: Vec<LocalTransIndex> = Vec::new();
            // Cola de transiciones pendientes de obtener su tiempo (indices locales)
            let mut trans_pending_time: VecDeque<LocalTransIndex> = VecDeque::new();
            trans_pending.push(transition.ind_local);
            // Mapa clave valor con el tiempo de las transiciones (tiempo de los caminos hasta esas transiciones)
            let mut trans_times = HashMap::new();

            let mut one_way = true;       // Bool indicando si solo hay un camino o tiene bifurcaciones

            // Optimizacion para redes de ramas: si no hay eventos y solo una transicion de entrada
            // el tiempo depende de la transicion de entrada
            if event_list.len() == 0 && self.transition_input_list.len() == 1 {
                let trans = self.transition_input_list.get(0).unwrap();
                let time_marks = &self.transition_list[transition.ind_local.0].times.il_tiempos; // Tiempos de marca
                let tm = *time_marks.get(&trans.0.to_string()).unwrap() as usize;  // Tiempo de marca de la transicion
                return SimulatedClock(self.transition_list[trans.0].ext_time.0 + tm);
            }

            // Recorrer las transiciones pendientes
            while !trans_pending.is_empty() {
                let trans = trans_pending.pop().unwrap();   // Indice local de la transicion
                trans_seen.insert(trans.0);

                // Comprobar si es de entrada
                if self.transition_list[trans.0].is_input {
                    // Si solo hay un camino el tiempo es el obtenido en compilacion
                    if one_way {
                        let time_marks = &self.transition_list[transition.ind_local.0].times.il_tiempos; // Tiempos de marca
                        let tm = *time_marks.get(&trans.0.to_string()).unwrap() as usize;  // Tiempo de marca de la transicion
                        return SimulatedClock(self.transition_list[trans.0].ext_time.0 + tm);
                    }
                    trans_times.insert(trans.0, self.transition_list[trans.0].ext_time.0 + self.transition_list[trans.0].firing_time.0);
                } else {
                    // Comprobar si en el futuro estara sensibilizada con los eventos en FUL
                    // Valor de LEF
                    let lef_val = self.transition_list[trans.0].lef_value.0 as usize;
                    // Eventos dirigidos a la transicion
                    let events = self.events_ful(trans, event_list);
                    if events.len() >= lef_val {
                        // Estara sensibilizada: guardar su tiempo
                        let event_time = events[lef_val-1].time.0;  // Tiempo del evento que sensibilizara la transicion
                        // Si solo hay un camino el tiempo es el obtenido en compilacion
                        if one_way {
                            let time_marks = &self.transition_list[transition.ind_local.0].times.il_tiempos; // Tiempos de marca
                            let tm = *time_marks.get(&trans.0.to_string()).unwrap() as usize;  // Tiempo de marca de la transicion
                            return SimulatedClock(event_time + tm);
                        }
```

Figura 25: Implementación en Rust del cálculo dinámico de lookahead en simulación (parte 1).

```
                    trans_times.insert(trans.0, event_time + self.transition_list[trans.0].firing_time.0);
                } else {
                    // No estara sensibilizada
                    trans_pending_time.push_back(trans);     // Marcar para obtener su tiempo despues
                    // Recorrer transiciones predecesoras
                    for trans_pre in self.transition_list[trans.0].pre.iter() {
                        if !trans_seen.contains(&trans_pre.0) {
                            // La transicion no se ha visto: marcar para comprobar despues
                            trans_pending.push(*trans_pre);
                        } else if !trans_times.contains_key(&trans_pre.0) {
                            // Ciclo: obtener su tiempo como la cota inferior para avanzar el calculo de lookahead
                            trans_times.insert(trans_pre.0, LVT.0 + self.transition_list[trans_pre.0].firing_time.0);
                        }
                    }
                    if self.transition_list[trans.0].pre.len() > 1 {
                        // Hay mas de 1 camino
                        one_way = false;
                    }
                }
            }
        }
        // Obtener tiempos de las transiciones que no se completaron antes (por no tener informacion de predecesoras)
        while !trans_pending_time.is_empty() {
            let trans = trans_pending_time.pop_back().unwrap();

            // Lista de tiempos de los caminos predecesores
            let mut times_pre: Vec<usize> = Vec::new();
            for trans_pre in self.transition_list[trans.0].pre.iter() {
                if trans_times.contains_key(&trans_pre.0) {
                    // El camino de la transicion tiene tiempo: guardarlo
                    times_pre.push(*trans_times.get(&trans_pre.0).unwrap());
                } else {
                    break;
                }
            }
            if times_pre.len() == self.transition_list[trans.0].pre.len() {
                // Se tiene la informacion de predecesoras, se obtiene su tiempo
                times_pre.sort();   // Ordenar tiempos de menor a mayor
                // trans_times[trans] = times_pre[trans.lef-1] + trans.firing_time
                trans_times.insert(trans.0, times_pre.get(self.transition_list[trans.0].lef_value.0 as usize - 1).unwrap()
                                            + self.transition_list[trans.0].firing_time.0);
            } else {
                // Informacion incompleta, marcar transicion para calcularla despues
                trans_pending_time.push_front(trans);
            }
        }
        return SimulatedClock(*trans_times.get(&transition.ind_local.0).unwrap());
    }
}
return SimulatedClock(0); // No se utiliza
}
```

Figura 26: Implementación en Rust del cálculo dinámico de lookahead en simulación (parte 2).