Universidad Zaragoza
1542

Trabajo Fin de Grado

Desarrollo de un Framework de Simulación de Sistemas de Eventos Discretos Complejos

Development of a Simulation Framework for the Simulation of Complex Discrete Event Systems

Autor

Sergio Herrero Barco

Director

José Ángel Bañares

Codirector

José Manuel Colom

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2020

AGRADECIMIENTOS

Quiero dar mi agradecimiento a mis directores *José Ángel Bañares* y *José Manuel Colom* y a *Unai Arronategui* por su paciencia, docencia y su inestimable ayuda a la hora de la elaboración de este proyecto fin de grado. Gracias a ellos el trabajo ha sido mucho más llevadero y me han permitido adquirir una gran cantidad de conocimientos.

Finalmente, agradezco a mi familia, pareja y compañeros el apoyo moral recibido a lo largo de todo el proyecto sin el cual la elaboración del trabajo hubiera sido mucho más difícil.

RESUMEN

Bajo el término de Sistemas de Eventos Discretos Complejos se agrupan aquellos sistemas dinámicos cuyo estado se ve modificado de una forma puntual y distinguible en su evolución. Este tipo de sistemas son el contrapunto a los Sistemas Continuos en los que el estado se ve modificado de una forma gradual y permanente sin posibilidad de distinguir puntos de la evolución de una forma diferenciada.

Los Sistemas de Eventos Discretos son de una gran importancia hoy en día por la multitud de ellos que tienen relevancia en nuestra sociedad: desde la propia *Internet de las Cosas* (IoT) a la evacuación de un estadio deportivo en caso de incidente grave, pasando por el sistema de salud pública donde un conjunto de recursos de asistencia sanitaria, tratan de atender a una serie de pacientes siguiendo una serie de protocolos médicos. Todos estos sistemas tienen como característica principal su complejidad y tamaño, que puede crecer hasta valores enormemente grandes. Estas propiedades, unidas al hecho de que no se pueden construir prototipos que permitan ensayar soluciones o mejoras en la gestión, hacen que la construcción de modelos que representen su comportamiento (o aspectos de este) de la forma más fidedigna posible sea prácticamente la única manera de tratar con estos sistemas.

Estos modelos son utilizados por el diseñador para simular el comportamiento del sistema bajo diversos escenarios que permitan investigar la evolución bajo distintas condiciones. El análisis de estos resultados para diferentes entradas permitirá comprender mejor el sistema, ayudar a tomar decisiones en situaciones reales o incluso diseñar sistemas de optimización que mejoren su comportamiento.

Las redes de Petri son una familia de modelos matemáticos que se han mostrado especialmente adecuados para modelar (representar) el comportamiento de sistemas de eventos discretos. A su simplicidad conceptual y representación gráfica, añaden la característica de ser modelos que se pueden ejecutar. En ellos se representa el estado mediante unas variables llamadas lugares, y la modificación del estado mediante transiciones que cambian los valores enteros no negativos almacenados en los lugares. La modificación del estado se realiza de una forma local y atómica.

En este proyecto se aborda el diseño e implementación de un entorno para la simulación (ejecución) de modelos de Redes de Petri que representen sistemas de eventos discretos complejos y, además, escalables. Las partes innovadoras del trabajo desarrollado se centran en los siguientes aspectos que se enumeran a continuación:

1) **Descripción de sistemas de eventos discretos complejos y escalables que sea viable y compacta**. Con respecto a otros proyectos en los que se ha considerado la simulación de redes de Petri como tema central, este proyecto se centra en sistemas complejos con una estructura intrincada y de dimensiones escalables que puedan crecer hasta valores enormes. Estas dos características se han contemplado tradicionalmente de una forma limitada (e incluso drásticamente limitada) en las propuestas de simuladores de redes de Petri, dando lugar a entornos que llevan inevitablemente a:

    a) Usar modelos planos (no jerárquicos) de los sistemas, listas de lugares y de transiciones y sus conexiones estructurales de un solo nivel, haciendo que el modelo tenga unas dimensiones enormes y generando una maraña en el grafo que lo representa. Esto hace no útil la descripción de sistemas complejos ya que la modificación de cualquier especificación se convierte en una tarea titánica. Así mismo, los resultados son de difícil, si no imposible,

traducción a términos del sistema especificado ya que no se diferencia entre modelo de descripción y modelo de ejecución bajo escenarios de simulación. Por otra parte, la introducción de redes de Petri de alto nivel no mejora significativamente este problema dado que los conceptos de jerarquía y modularidad no están pensados desde el lado del diseñador, si no que están pensados desde el lado de la coherencia matemática interna de la definición del modelo formal de red de Petri.

b) Usar técnicas de simulación centralizada que hacen que la explotación de la concurrencia en modelos de grandes dimensiones se vea mermada por el entrelazamiento de actividades concurrentes en un entorno centralizado. En otras palabras, la selección de simulación centralizada no permite la simulación de sistemas escalables de grandes dimensiones por lo que resulta imposible estudiar estos sistemas mediante esta aproximación. Aunque existen propuestas de simuladores distribuidos, no poseen soluciones para la gestión de nombres globales de la red de Petri distribuida de una forma eficiente y tampoco proponen técnicas eficientes para la explotación de la concurrencia, ni consideran una representación del modelo que facilite su interpretación distribuida eficiente facilitando el despliegue y/o el balanceo dinámico de carga.

Por las razones citadas anteriormente, la propuesta innovadora que se realiza en este proyecto es **la separación entre dos modelos claves** del sistema real bajo consideración.

El **primer modelo descriptivo** será una red de Petri que servirá **para definir el sistema que se pretende estudiar**. Para ello se enriquece el lenguaje tradicional de descripción de una red de Petri con construcciones de alto nivel que permiten declarar sistemas de grandes dimensiones y complejos. En este proyecto se consideran tres construcciones de alto nivel:

- Componentes funcionales que permiten describir mediante una red de Petri una funcionalidad del sistema. La definición puede ser jerárquica usando componentes previamente declarados y convenientemente interconectados.
- Recursos de sistemas que serán modelados mediante redes de Petri (en su versión más simple pueden ser lugares de redes de Petri cuyo marcado inicial representa el número de copias disponibles al inicio de la ejecución) y representan entes pasivos adquiridos por los componentes funcionales para poder progresar en su actividad, a cuyo término son liberados.
- Canales de comunicación para el intercambio de mensajes, que son redes de Petri a través de las cuales se representan los mecanismos a través de los que se imponen relaciones de causalidad entre componentes funcionales del sistema mediante intercambio de mensajes.

La declaración de un sistema estará basada en la declaración de sus componentes funcionales y las relaciones entre ellos mediante compartición de recursos y canales de comunicación. En su nivel más bajo de declaración de cualquiera de estos tres tipos de constructores será necesario especificar una red de Petri mediante sus lugares, transiciones, arcos de conexión entre ellos y el marcado inicial de sus lugares. Esta descripción estará cercana al diseño estructural que haya realizado el diseñador y por tanto con alto significado en términos de los componentes del sistema modelado.

Desde un punto de vista técnico, los problemas afrontados para la definición de modelos complejos y gran escala son:

- La *definición de primitivas* que permitan la definición de diferentes perspectivas, especialmente la perspectiva del modelo desde el punto de vista de la asignación de recursos, o en la definición de comportamientos y la interacción de componentes en el contexto de competición de recursos.
- La definición de componentes atómicos, componentes compuestos y sus relaciones y que permitan definir *estructuras y jerarquías*. La definición de la estructura requiere precisa de clases e Interfaces que pueden ser implementadas en cualquier lenguaje de programación orientado a objeto y que permitan la definición de entidades y la estructura del sistema.

El segundo modelo implicado es el modelo de ejecución. En la propuesta que aquí se hace es un modelo de redes **Lugar/Transición** con la ventaja de que el diseñador no tiene por qué conocer ni manejar dicho modelo. El modelo de ejecución se obtiene a partir del modelo descriptivo del sistema como se comenta a continuación.

2) **Desarrollo de procedimientos y mecanismos para la generación de un modelo de red de Petri ejecutable** con el que se pueda realizar la simulación del sistema en un escenario concreto, a partir de un modelo de red de Petri que especifique/describa el comportamiento del sistema. El modelo ejecutable es una red de Petri Lugar/Transición plana, sin jerarquía, obtenida de la descripción del sistema realizada en un lenguaje de redes de Petri orientado a componentes, recursos y canales de comunicación. El proceso de obtención del modelo ejecutable consta de dos procesos, uno primero de **elaboración,** que obtiene una red de Petri Lugar/Transición plana, sin jerarquía, y otro proceso de **compilación** que traduce la red plana en una codificación eficiente para su ejecución. La selección de redes de Petri Lugar/Transición como modelo de ejecución está basada en la existencia de técnicas de simulación muy eficientes que están dirigidas por las listas de transiciones sensibilizadas (transiciones listas para ser disparas y producir un cambio de estado en el sistema). La caracterización de la sensibilización de las transiciones se realiza mediante **funciones lineales del marcado,** cuyo valor se actualiza mediante simples constantes emitidas por transiciones vecinas disparadas que producen algún cambio en el grado de sensibilización de la transición propietaria de la función. Estas técnicas están especialmente indicadas en contextos distribuidos ya que los mensajes intercambiados son bajos en número (sólo los cambios inducidos por el disparo de una transición en el grado de sensibilización de las transiciones conectadas mediante un lugar) y muy simples (se trata de constantes enteras evaluadas en el proceso de elaboración).

   Desde un punto de vista técnico, los problemas afrontados para la obtención del modelo ejecutable son:

   - Concepción de un *algoritmo para la elaboración* del modelo descriptivo del sistema que genere el modelo ejecutable basado en redes Lugar/Transición.
   - Definición de un *sistema de información* que permita conectar los diferentes componentes y construcciones del modelo jerárquico con los objetos de la red de Petri ejecutable. En este punto juega un papel importante el sistema de nombres globales de objetos de la red de Petri ejecutable y su gestión.
   - *Definición de las gramáticas* para la descripción de redes planas, y un *compilador* para generar la red de Petri ejecutable basada en el método de simulación de redes de Petri elegido, es decir la obtención y codificación de las funciones lineales de sensibilización a partir de la red de Petri plana.
   - *Definición y gestión de las particiones* de la red ejecutable de cara a una *simulación distribuida*. Para el caso centralizado se aplica el mismo

procedimiento, pero la partición de la red consta de un único componente que engloba toda la red.

El sistema de información generado durante la elaboración es clave de cara al procesamiento de los *logs* obtenidos del proceso de simulación de la red de Petri y para la construcción de los resultados del modelo.

3) **Desarrollo de una máquina virtual de simulación de redes de Petri Lugar/Transición que implemente el método de transiciones sensibilizadas caracterizadas mediante funciones lineales de sensibilización**. En el proyecto, a cada una de estas máquinas virtuales se le ha denominado **Simbot.** Dichos simbots implementan el algoritmo de simulación, pero también incluyen los elementos de administración que permiten al simbot.

   Desde un punto de vista técnico, los problemas afrontados para el desarrollo de los simbots son:

   - *Interaccionar y solicitar recursos* de la plataforma física donde se ejecuta el simbot.
   - Encargarse del *envío y recepción de los mensajes* a intercambiar con otros simbots (en el caso de una simulación distribuida) o con otros servicios del entorno como el servidor de nombres o el sistema de registro de logs.
   - *Gestionar los tiempos de simulación y los tiempos del sistema simulado* para la gestión de estados seguros en el caso de simulaciones distribuidas conservadoras.
   - *Elaboración de figuras de rendimiento de la simulación* que permitan realizar un *balanceo dinámico* de carga de ejecución durante la simulación
   - Lanzamiento y gestión de *"threads" asociados a funciones* que deban ejecutarse durante el disparo de transiciones, y reserva de recursos necesarios
   - *Capacidad para alojar cualquier otra función* que pueda desarrollarse en el futuro para ampliar o mejorar el sistema como arranques en caliente de la simulación o migraciones de carga de simulación entre simbots.

   Para el diseño eficiente de este simbot, desde un punto de vista metodológico, se ha procedido a construir un simulador centralizado que permita determinar la eficiencia de las distintas alternativas que aparecen a la hora de implementar el simulador de redes de Petri. Por ejemplo, alternativas en la gestión e inserción ordenada de eventos en las colas de eventos.

   Esta forma de proceder aísla de otros fenómenos como pueden ser el intercambio de mensajes entre simbots, que son costosos en tiempo y pueden ocultar la simulación propiamente dicha.

4) **Propuesta y demostración de la viabilidad de un entorno de simulación distribuida basada en simbots.** Se han realizado pruebas y ensayos para exponer la viabilidad de la simulación distribuida, así como la demostración de la eficiencia obtenida frente a la simulación centralizada. Se han usado principalmente 2 casos de estudio que han permitido evaluar las soluciones adoptadas.
   1) Construir una *máquina virtual de simulación* de redes de Petri que sea al menos tan eficiente como un simulador centralizado comercial. Para ello se ha realizado un experimento para comparar entre el *simulador centralizado* desarrollado y el simulador de redes de Petri ***Cosmos*** [1] que presenta los tiempos de simulación. En dicho experimento se ha comprobado la

escalabilidad de los sistemas, usando un sistema con una estructura similar al problema de ***la cena de los filósofos*** [2].

2) Construir un *simulador distribuido* basado en un conjunto de simuladores virtuales coordinados cada uno de los cuales contiene un pedazo de una red. El objetivo es demostrar la **viabilidad de un simulador distribuido basado en la máquina virtual diseñada.** La viabilidad de un simulador distribuido es demostrar que lo hace mejor que un simulador centralizado **sacando ventaja de la concurrencia.** En este experimento se ha demostrado que una red particionada y ejecutada en un simulador distribuido se ejecuta **más rápido** (ganancia computacional o speed-up) que en el simulador centralizado desarrollado.

Como conclusiones de este trabajo se puede afirmar que el simulador distribuido desarrollado logra demostrar la viabilidad de la simulación distribuida basada en simbots. Se ha demostrado la eficiencia obtenida en simulación distribuida en cuanto a paralelismo se refiere, así como la distribución de redes grandes en diferentes máquinas.

El framework desarrollado representa un prototipo inicial de lo que puede llegar a ser un framework completo y comercial. Como prototipo inicial, el proyecto ha perseguido el desarrollo de todos los servicios básicos necesarios para la definición de un framework de simulación. Como trabajo futuro quedan aspectos como incorporar sistemas de tolerancia a fallos, balanceo de carga dinámica y la mejora de paso de mensajes, entre otros, que harán del framework un trabajo más robusto y completo.

A nivel personal, el trabajo ha supuesto un aprendizaje del papel e importancia de las redes de Petri como modelo formal ejecutable para la simulación de sistemas de eventos discretos complejos, así como la ampliación de conocimiento adquiridos sobre sistemas distribuidos ya introducidos a lo largo del grado. Además, se ha acercado al estilo de trabajo de los investigadores que han ayudado y colaborado a la hora de la realización del trabajo.

INDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

ACRÓNIMOS

- RdP: Red de Petri
- LEF: *Linear Enabling Function*
- DES: *Discrete-event simulation*
- DEVS: *Discrete Event System Specification*

1. INTRODUCCION

1.1. Motivación

Los sistemas de eventos discretos son sistemas cuya evolución dinámica está dirigida por hechos puntuales en el tiempo, los cuales están aislados y son distinguibles y asíncronos.

Bajo este concepto se agrupan sistemas que estudian el comportamiento de grupos sociales, diagnósticos de problemas en procesos, aplicaciones de gestión hospitalaria o sistemas tecnológicos como internet [3]. Todos ellos comparten dos características importantes: *la complejidad* (relaciones intrincadas entre diferentes partes del sistema) y *la escalabilidad* (dimensiones crecientes manteniendo todas las relaciones estructurales entre componentes del sistema).

El estudio de estos sistemas se hace cada vez más necesario tanto para comprender su naturaleza, como para la ayuda en la toma de decisiones a lo largo de su ciclo de vida, o al diseño de sistemas de control/optimización, etc.

La simulación, cómo método de análisis, y los simuladores, como herramienta soporte de la metodología asociada, se constituyen en la aproximación práctica para tratar con estos sistemas (siendo muchas veces la única factible). En una metodología basada en la simulación, el sistema se representa/describe a través de un modelo de su comportamiento, junto con un escenario o entorno donde el sistema debe evolucionar, operar e interaccionar.

En el caso de los Sistemas de Eventos Discretos, los modelos de la familia de las redes de Petri se han demostrado como uno de los más adecuados para representar/describir el sistema y proceder a su simulación dado que una red de Petri es una especificación ejecutable. Teniendo en cuenta que en este proyecto el enfoque es hacia sistema complejos y escalables, esto conlleva a que se requieran métodos altamente eficientes, desde un punto de vista computacional, para su simulación.

En la actualidad existen muchas propuestas metodológicas y herramientas que las soportan, para la simulación de sistemas de eventos discretos. Un ejemplo significativo de esta abundancia de propuestas puede ser *Omnet*++ [4]. No obstante, la revisión realizada de estas propuestas ha puesto en evidencia que cuando se trata de sistemas complejos y escalables, todas ellas presentan grandes carencias para hacer simulaciones eficientes. Esta es la razón de este proyecto: tratar de cubrir las carencias del dominio con una propuesta integral, metodología y herramientas, para la simulación de sistemas de eventos discretos complejos y escalables.

1.2. Aproximación

El trabajo realizado en este proyecto tiene como antecedente directo la investigación presentada en los artículos [5] [6] [7]. En ellos se propone una aproximación integral para el estudio a lo largo del ciclo de vida de sistemas de eventos discretos complejos y escalables basada en simulación. La figura incluida a continuación es la figura 1 del artículo [5] en la que se describen el proceso,

basado en redes de Petri, desde la especificación hasta la simulación distribuida del modelo elaborado a partir de dicha especificación.

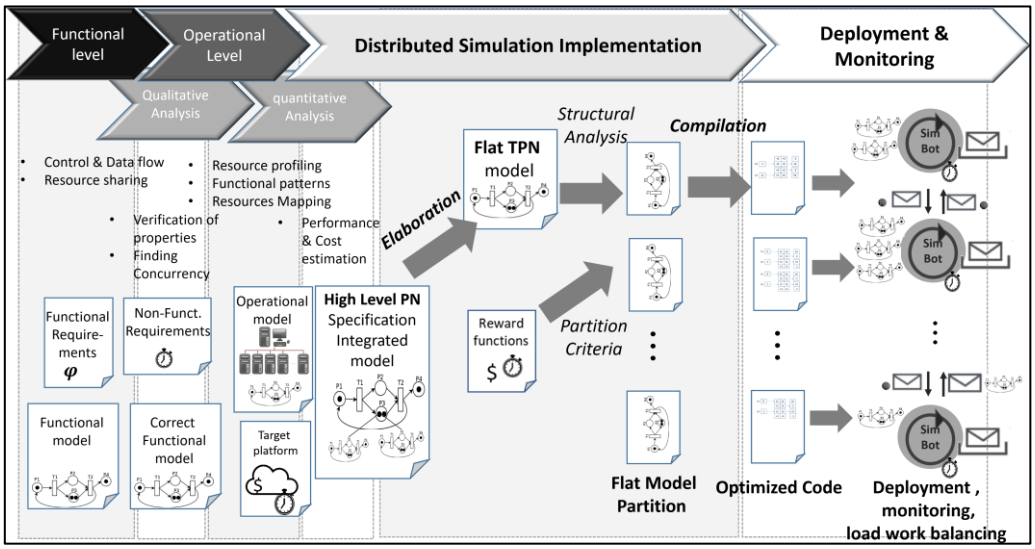


Figura 1: Descripción del proceso de simulación

Puntos claves de la aproximación propuesta, y que son los elementos de partida adoptados en este proyecto son:

- ***La definición de un lenguaje basado en componentes funcionales que se relacionan mediante recursos y canales de comunicación***. Dicho lenguaje es un lenguaje basado en redes de Petri incorporando jerarquía y modularidad. El lenguaje sirve para la especificación del sistema que sirve como especificación/descripción formal del comportamiento del sistema próxima a la concepción del ingeniero de diseño.
- ***La definición de un proceso de elaboración que permite obtener a partir de la especificación del sistema, representada en el lenguaje anterior, una red de Petri Lugar/Transición global, sin jerarquía ni modularidad, orientada a la ejecución mediante su simulación alimentada por datos del entorno del sistema***. Dicho modelo ejecutable está conectado con el modelo de especificación mediante un sistema de información construido en el proceso de elaboración.
- ***La simulación distribuida de la red de Petri ejecutable, obtenida del proceso de elaboración, como técnica de obtención de eficiencia computacional para sistemas complejos y escalables***. En la simulación distribuida juega un papel fundamental la explotación de la información estructural que aporta el modelo de redes de Petri para, por ejemplo, pronosticar de forma segura el avance de los relojes de simulación en ausencia de eventos recibidos en un momento dado por los simuladores, o para realizar un balanceo dinámico de la carga de trabajo de los simuladores locales con el ánimo de aumentar la concurrencia real de cara a mejorar la eficiencia.

Los trabajos más próximos a la aproximación que se ha adoptado de partida se encuentran alrededor de la iniciativa DEVS. No obstante, las diferencias esenciales como se indica en [5] [6] [7], que hacen que la propuesta aquí considerada presente ventajas importantes son: (1) La separación entre el modelo de especificación y el modelo de ejecución que en DEVS no es clara, lo cual hace que el ingeniero se vea

condicionado en su labor de especificación del sistema por la implementación del simulador distribuido, que hace que los modelos deban ser retocados cada vez que el simulador evoluciona tecnológicamente. En la aproximación adoptada, este gap entre especificación y ejecución lo salva el proceso de elaboración; (2) En DEVS la jerarquía, modularidad y conectividad de componentes es limitada y bastante compleja, haciendo que el ingeniero se tenga que adaptar a los posibilidades ofrecidas por el entorno limitando la expresividad de las especificaciones; (3) La ausencia de información estructural del modelo a simular que hace complicado el incremento de la eficiencia de la simulación mediante técnicas de pronóstico del avance de relojes, o balanceo dinámico de carga, debiendo pasar por técnicas genéricas dentro de la simulación distribuida que hacen más pesada computacionalmente la simulación.

Por todo lo anterior, es por lo que en este proyecto se procederá a la construcción de herramientas software que den soporte a la metodología propuesta en [5] [6] [7] valorando en cada caso las tecnologías software más adecuadas para lograr la máxima eficiencia del proceso de simulación distribuido.

1.3. Objetivos

El objeto de este proyecto es el diseño de un entorno para la simulación distribuida que tenga como entrada una descripción del sistema de eventos discretos en un lenguaje orientado a componentes basado de redes de Petri jerarquizadas y modulares.

En el diseño de la arquitectura de dicho entorno, el núcleo está en la construcción de un simulador de una red de Petri que desde el punto de vista funcional sea eficiente y permita coordinarse con otras réplicas del simulador encargadas de simular otras partes de una red global ejecutable y así realizar una simulación distribuida de un sistema complejo y escalable de forma eficiente.

A los aspectos funcionales de la simulación de redes de Petri, el simulador local incorporará funciones administrativas que permitan realizar la simulación de una forma distribuida, como por ejemplo: gestión de los relojes, pronósticos coordinados de los avances del reloj, recogida de información sobre la carga de trabajo de simulación y la utilización de recursos, gestión de nombres globales, etc.

1.4. Alcance

Para alcanzar el objetivo general planteado en este proyecto, los objetivos específicos que se abordarán son los siguientes:

- Formalización del lenguaje de especificación de sistemas de eventos discretos complejos y escalables, orientado a componentes, con jerarquía y basado en redes de Petri a partir de la propuesta realizada en [5] [6] [7]
- Diseño y construcción de un elaborador y compilador que, a partir de la descripción, en el lenguaje definido anteriormente, del sistema de eventos discretos a simular genere un modelo de red de Petri ejecutable, sistema de información para relacionar ambos modelos y partición de la red para repartir entre el conjunto de simbots que realizarán la simulación distribuida.
- Diseño del simulador local, denominado simbot, que incorporará los aspectos funcionales necesarios para una simulación eficiente de redes de Petri y todas aquellas funciones de carácter administrativo que

permitan coordinar simbots para una simulación distribuida de un sistema particionado en varias subredes (cada una simulada en un simbot). El algoritmo de simulación deberá ser el más adecuado para permitir una simulación distribuida por coordinación de simbots.

- Construcción y evaluación del simulador local analizando alternativas de implementación buscando la máxima eficiencia medida en número de transiciones disparadas por unidad de tiempo. En este proceso de construcción y evaluación de alternativas el simulador local o simbot se utilizará como un simulador centralizado de una única red de Petri (sin coordinación con otros simbots) para seleccionar el algoritmo de simulación más eficiente sin la perturbación de la comunicación de eventos entre simbots en una simulación distribuida. Así mismo, esta forma de proceder permitirá comparar la eficiencia de la simulación con simuladores centralizados existentes y determinar la ganancia computacional obtenida por una simulación distribuida.
- Construcción de un prototipo de simulador distribuido a partir de los simbots construidos, los servicios y servidores necesarios que permitan construir un entorno completo de simulación distribuida y obtención de resultados. Dicho prototipo perseguirá demostrar la viabilidad de la simulación distribuida a partir de las herramientas objeto fundamental de este proyecto: el elaborador y el simbot.
- Diseño y construcción de prototipos de servicios y servidores para construir el entorno de simulación distribuida basado en los simbots, por ejemplo, servicios de nombres globales.
- Selección de casos de estudio y planificación de la experimentación a realizar encaminada a demostrar la viabilidad de la construcción de un simulador distribuido de sistemas de eventos discretos modelados con redes de Petri, a partir de un conjunto de simbots cada uno de los cuales se encarga de simular una parte de la red global coordinándose con el resto de simbots del conjunto. Se analizará la ganancia obtenida con respecto a la simulación centralizada en un único simbot.

Dada las limitaciones temporales que impone el desarrollo de un Trabajo Fin de Grado, el alcance del proyecto definido persigue el desarrollo de una prueba de concepto del entorno de simulación descrito. Dado el número de tareas consideradas en el proyecto, el alcance del trabajo se ha orientado más hacia un desarrollo en amplitud de todas las tareas presentadas más que en profundidad en cuanto al detalle de las funcionalidades desarrolladas y sus prestaciones.

1.5. Tecnologías y Entorno de Trabajo

Las herramientas principales usadas durante el desarrollo han sido *IntelliJ IDEA* [8] y *Eclipse* [9]. Para la edición de los ficheros de texto plano se ha usado el editor de texto *Sublime Text* [10].

Para la compilación y traducción de los ficheros planos en las estructuras se han usado las herramientas de compilación *JCup* y *Jlex.*

El simulador ha sido desarrollado en el lenguaje Java. Se han usado otras tecnologías como *Redis* [11] *y Log4J* [12] ambas de código abierto.

Se han utilizado y adaptado paquetes del `DEVS-Suite_5.0.0` que definen la capa de estructura de los modelos DEVS*. En concreto,* se ha utilizado el paquete `Java GenCol` que define estructuras básicas como *`Bag, Relation y`*

ensembleCollection, y se ha adaptado la capa *GenDevs* para la definición de componentes y la definición de modelos jerárquicos.

El control de versiones se ha realizado con la ayuda de GitHub, y la escritura de la memoria se ha llevado a cabo en Microsoft Word.

El estudio experimental se ha realizado sobre los equipos descritos en el apartado Estudio Experimental.

Como herramientas de *Profiling* se han usado *Java VisualVM, FlameGraph* [13] y la opción de profiling que posee el IDE *IntelliJ*.

1.6. Tareas principales del proyecto

A partir de los objetivos definidos anteriormente se han desglosado en las siguientes tareas.

1) Estudio de los artículos [5] [6] [7].
2) Estudio y compresión del código del que se parte el proyecto elaborado por Miguel Ángel Barcelona y Luis Caballero, en el que se desarrolló un prototipo de intérprete de redes de Petri mediante funciones lineales del marcado [13]
3) Diseño y desarrollo del proceso de elaboración, definición de las gramáticas para definir redes de Petri Lugar/Transición, y desarrollo del compilador.
4) Implementación de mecanismos para la composición de RdP grandes a partir de RdP más pequeñas.
5) Implementación de mecanismos para la obtención de localizaciones de transiciones en una red.
6) Incorporación de un registro de las trazas de simulación con soporte a registros en ficheros y consola.
7) Conversión de simulador centralizado en distribuido. Desarrollando mecanismos para trabajar en red.
8) Implementación de API de envío de mensajes de manera asíncrona entre diferentes máquinas.
9) Depuración de errores a lo largo de todo el proyecto.
10) Realización de estudios experimentales que permitan demostrar la viabilidad de la simulación distribuida.
11) Escritura de la memoria donde reflejar el trabajo realizado, así como los resultados obtenidos.

1.7. Planificación del proyecto

Las horas dedicadas al proyecto se encuentran en la siguiente tabla, y un diagrama de Gantt sobre el mismo en el Anexo 11.1.

Tabla 1: Dedicación de horas

Tarea	**Horas**
Estudio del articulo	10
Estudio del código proporcionado	15
Diseño e Implementación de relaciones jerárquicas y mecanismos de composición de RdP usando metodologías DEVS.	10

Definición e implementación del proceso de elaboración	6
Definición e implementación del proceso de compilación	4
Implementación de servidor de nombres distribuido	10
Implementación de gestor de registros de trazas de cada componente o RdP	20
Diseño de arquitectura distribuida	15
Diseño de API con sockets como primera aproximación a sistema distribuido	25
Corrección de errores	100
Estudio experimental	20
Memoria	100
Total	**335**

2. CONCEPTOS BÁSICOS Y NOTACIÓN

A continuación, se van a explicar algunos de los conceptos básicos de los que se hablan a lo largo de la memoria. En la sección 2.1 se habla de lo que son los sistemas de eventos discretos, cómo evolucionan y algunos ejemplos. En la sección 2.2 se introduce el concepto de Red de Petri desde una vista formal y matemática, y finalmente en la sección 2.3 se habla sobre el concepto de simulación de un sistema, así como de las diferentes estrategias a tomar en cuanto a simulación se refiere.

2.1. Sistema de eventos discretos

Un **Sistema de Eventos Discretos** es un sistema dinámico que avanza y evoluciona de acuerdo con la aparición imprevista de eventos a intervalos irregulares y desconocidos.

Por ejemplo, un evento puede corresponder a la llegada de un cliente, la finalización de una tarea, la pérdida de un paquete de datos en la red o incluso el fallo de una máquina en un sistema de producción.

Los sistemas de eventos discretos aparecen en dominios como los sistemas de producción, la robótica, la logística, las comunicaciones en red, etc. Todas estas aplicaciones requieren control y coordinación para el correcto tratamiento de cada posible evento.

Antiguamente, los sistemas de eventos discretos se realizaban como una solución *ad hoc* a los problemas, pero el incremento de la complejidad en cuestión de los modelos y el aumento de la capacidad de cómputo de los ordenadores ha llevado a detallar modelos formales para el análisis y diseño de estos sistemas.

2.2. Red de Petri

Una RdP (Red de Petri) es una representación matemática usada para el modelado que permite representar y analizar procesos concurrentes.

Consiste en un grafo bipartido dirigido en el que hay nodos llamados transiciones (eventos que pueden ocurrir, gráficamente representados por barras o rectángulos) y nodos lugares (condiciones, gráficamente representados por círculos). Los nodos de este grafo bipartido, si son de distinto tipo pueden conectarse mediante un arco dirigido, es decir, pueden existir arcos desde un lugar a una transición o arcos desde una transición a un lugar.

De una manera formal, una RdP es una 4-tupla $(\mathbf{P}, \mathbf{T}, \mathbf{W}, \mathbf{M_0})$ donde:

- $\mathbf{P}$ es un conjunto finito de lugares, |P| = n
- $\mathbf{T}$ es un conjunto finito de transiciones, |T| = m
- $\boldsymbol{S}$ y $\mathbf{T}$ son disjuntos
- $\mathbf{W}$: $(\mathbf{P} \times \mathbf{T}) \cup (\mathbf{T} \times \mathbf{P}) \rightarrow \mathbb{N}$ que asigna a cada arco un número no negativo llamado peso. El conjunto de los nodos de entrada a un nodo x, es el conjunto de nodos y tales que W(y, x)>0, este conjunto se denota

$^{\cdot}x$. El conjunto de los nodos de salida de un nodo x, es el conjunto de nodos y tales que W(x, y)>0, este conjunto se denota $x^{\cdot}$.

- $\mathbf{M_0}$ es el marcado inicial de la red, es decir, los lugares que poseen una marca o token al inicio del sistema.

Las Redes de Petri permiten definir y representar de una manera formal los **sistemas de eventos discretos.**

Se denomina estado de una red de Petri al contenido de marcas de cada uno de los lugares de la red y se denota mediante un vector M $\in \mathbb{N}^{\boldsymbol{n}}$, y para designar el contenido de marcas de un lugar p $\in$ **P** en el estado M, se denotará M[p]

En una red de Petri un cambio de estado se produce como consecuencia del disparo de una transición t $\in$T. Para que una transición t se pueda disparar desde el marcado M, debe estar sensibilizada en M, lo cual significa que cada uno de los lugares p de entrada a la transición tiene un contenido de marcas igual o superior al peso del arco que va del lugar p a la transición t. Es decir, $\forall\, p \in {}^{\cdot}t, M[p] \geq W(p,t)$. La transición t se puede disparar desde el marcado M si está sensibilizada en M y entonces produce un cambio de estado desde el estado M al estado M' definido de la siguiente forma: $\forall\, p \in P, M'[p] = M[p] - W(p,t) + W(t,p)$

2.3. Simulación

La simulación de un sistema es el proceso a través del cual se ejecuta un modelo que describe (una abstracción) el comportamiento del sistema con un entorno de entrada con el que interacciona el modelo. El modelo debe tener la capacidad de representar el estado del sistema y estar dotado de un conjunto de reglas de modificación del estado que se activan en función de los eventos internos producidos y los recibidos desde el entorno.

En una simulación, los procesos van avanzando en el tiempo atendiendo de manera discreta los eventos que llegan. Un evento se puede definir como el suceso que activa una regla para cambiar las variables de estado del sistema.

Durante una simulación van llegando eventos y no se puede atender todos a la vez. Por tanto, en toda simulación existen listas donde se van almacenando los eventos que van llegando para ser procesados de manera ordenada. Dicha lista suele estar ordenada según los tiempos de ocurrencia del evento dado.

Dichos eventos pueden ser internos o externos. Los eventos internos se refieren a los eventos generados por cambios de estado locales del propio sistema, mientras que los eventos externos se refieren a eventos generados por el entorno con el que interacciona el sistema que está siendo simulado.

El aspecto temporal es de mucha importancia en la simulación. Se debe gestionar una evolución (avance) del tiempo de simulación real. En el caso de eventos discretos, el avance de tiempo no es continuo si no que su avance está gobernado por el tiempo para el que están proyectados los eventos almacenados en la cola de eventos pendientes: el tiempo de simulación real no puede avanzar más allá del tiempo para el que está proyectado el evento en cabeza de la cola, hasta que ese evento, al menos, sea procesado.

Hay que diferenciar el tiempo del sistema y el tiempo de simulación. El tiempo del sistema se refiere al tiempo consumido por los recursos computacionales utilizados para el proceso de simulación del sistema, se trata de la medición del

coste computacional del proceso software de simulación. El tiempo de simulación hace referencia a la ubicación temporal del comportamiento del modelo del sistema que se está observando mediante simulación. Esta ubicación temporal del sistema hace que los eventos generados se etiqueten con esta referencia temporal, proyectándolos a un futuro de la evolución del sistema y los eventos consumidos se procesen cuando su etiqueta temporal coincida con la referencia temporal en la que el sistema se encuentra ubicado.

Dentro de las estrategias de implementación de los procesos de simulación, la simulación distribuida persigue ejecutar la simulación del modelo del sistema sobre múltiples computadores autónomos que se encuentran interconectados por medio de una red de comunicación de datos. La forma en la que cada computador de la plataforma distribuida realiza una parte de la simulación, está basada en el particionado del modelo del sistema en submodelos que se alojan cada uno en uno de los computadores para su simulación. Estos submodelos no son independientes ya que proceden de la partición del modelo completo, por tanto su simulación debe ser coordinada entre todos los computadores que poseen una de las partes del modelo. Dicha coordinación se realiza mediante el intercambio de eventos entre los distintos simuladores ubicados en cada uno de los computadores de la plataforma distribuida y la gestión de los tiempos de simulación en cada uno de los simuladores para que el tiempo del sistema simulado avance de una forma coordinada. El intercambio de eventos con sus etiquetas temporales para la coordinación de los tiempos de simulación locales, se realiza a través de una red de comunicación con todos los problemas clásicos que ello conlleva: retrasos en los mensajes que portan los eventos, pérdida de mensajes, llegada desordenada de mensajes, etc. Es por ello que, para paliar este tipo de problemas, dentro de una simulación distribuida se pueden diferenciar dos tácticas para la gestión de eventos externos:

- ***Simulación optimista:*** Una vez procesados todos los eventos para un tiempo dado, se puede suponer que no va a llegar ningún evento más, que sea externo al simulador local, y por tanto se puede realizar el avance del tiempo de simulación local para así tratar los eventos almacenados en la cola de eventos proyectados para el futuro cuando coincide su ubicación temporal de simulación con el valor del tiempo de simulación local actualizado. Obviamente no es posible tener una completa seguridad de que la suposición sea correcta ya que ha podido ocurrir que:

    - No ha llegado ningún evento más porque no había ninguno ya, y por tanto la suposición es correcta.
    - Puede llegar un evento ubicado en un tiempo de simulación anterior al que el simulador local dispone después de haberlo actualizado. El mensaje que contenía este evento se ha retrasado al atravesar la red de comunicación de datos que interconecta el computador donde se generó el evento y el computador para el que el evento estaba destinado. Las razones para este retraso pueden ser múltiples, pero achacables a la plataforma distribuida de ejecución. Por tanto, en este caso la suposición es incorrecta.

  En el primer caso, se mejora la eficiencia ya que no se ha esperado a ningún mensaje. Sin embargo, en el segundo caso, se debe deshacer toda la simulación que se ha realizado y por tanto la eficiencia empeora. Para evitar perder toda la simulación realizada se definen estados seguros que se registran para retroceder hasta ese punto la simulación. Aunque la técnica puede potencialmente explotar mejor el paralelismo, la

implementación de puntos de recuperación ("*rollback*") es compleja, y los requisitos de memoria pueden ser altos.

- **Simulación pesimista**: Se denomina así ya que no se toma ninguna decisión sin tener la seguridad completa de que no van a llegar más eventos. De esta manera no se continúa simulando hasta que no haya llegado la confirmación de todos los submodelos que están siendo simulados de que no van a enviar ningún evento etiquetado con un tiempo de simulación del pasado. Como ventaja es que se evita tener que deshacer trabajo, pero la desventaja es que habrá situaciones donde se pierda tiempo al esperar el mensaje de los simuladores locales.

Estas técnicas de simulación se trasplantan fácilmente al contexto de simulación de redes de Petri. Por ejemplo, los submodelos a simular en cada computador de la plataforma distribuida son subredes de la red de Petri global del sistema y los eventos intercambiados harán referencia a la consumición y generación de marcas en los lugares de la subred de Petri como consecuencia del disparo de transiciones.

En el Anexo 11.6 se presenta una breve introducción a la simulación distribuida de sistemas de eventos discretos y la arquitectura de un simulador que realiza una simulación pesimista. Una introducción más detallada a la simulación distribuida se puede encontrar en [15].

3. MODELADO Y METODOLOGÍA.

Uno de los mayores problemas en ingeniería del software es reducir el salto entre los diseños de alto nivel y el código generado para una plataforma de ejecución específica. Las aproximaciones dirigidas por el modelo (MDE, *Model Driven Engineering*) proponen transformaciones entre modelos a diferente nivel, y la automatización del proceso mediante herramientas. En este trabajo el modelado y simulación de sistemas complejos se ha articulado alrededor del formalismo de las redes de Petri. La metodología seguida dirigida por los modelos basados en el formalismo de las redes de Petri se presenta en la subsección 3.1. En la siguiente subsección 3.2 se presentan los lenguajes que soportan dicha metodología.

En las subsecciones 3.2.1 y 3.2.2 se presentan el lenguaje de especificación de sistemas de eventos discretos complejos y escalables, orientado a componentes, con jerarquía y basado en redes de Petri. En su diseño se ha tenido en cuenta que la tarea más costosa es el desarrollo de los propios modelos, y por lo tanto es tan importante o más que la generación de un código eficiente para la simulación, o la gestión eficiente de los recursos computacionales en una simulación distribuida. La aproximación más difundida para reducir el coste de desarrollo de los modelos es representar el conocimiento del dominio sobre primitivas propias del dominio que faciliten su representación, de forma modular mediante librerías reusables de modelos que pueden ser especializados y compuestos en modelos mayores. En la subsección 3.2.1 se define la gramática para la descripción funcional del modelo mediante una RdP, y en la sección 3.2.2 se detalla la definición de componentes atómicas y compuestas que soportan la definición de modelos jerárquicos, y el proceso de elaboración que se realiza durante la construcción del modelo jerárquico a partir de la descripción textual del comportamiento de las componentes mediante redes de Petri.

En la sección 3.2.3 se presentan las estructuras de datos para representar una RdP. El proceso de compilación genera un código eficiente para la simulación distribuida que explote el potencial paralelismo y permita un balanceo de carga para abordar la impredecibilidad en un entorno de ejecución distribuido. Finalmente la Sección 4.1 explica la interpretación de la RdP mediante la estructura de datos generada, junto con la definición de transiciones en conflicto acoplado que se realiza en tiempo de compilación y que permite determinar los conjuntos máximos de transiciones que se pueden disparar de manera concurrente bajo un marcado, así como la definición de criterios para seleccionar que transición se dispara cuando hay conflicto.

3.1. Metodología de modelado

La metodología se va a ilustrar con un ejemplo sacado del artículo [7], en el que se presenta el desarrollo de un modelo de un sistema de eventos discretos que representa el tratamiento de enfermos siguiendo unos protocolos médicos dentro del sistema de salud público.

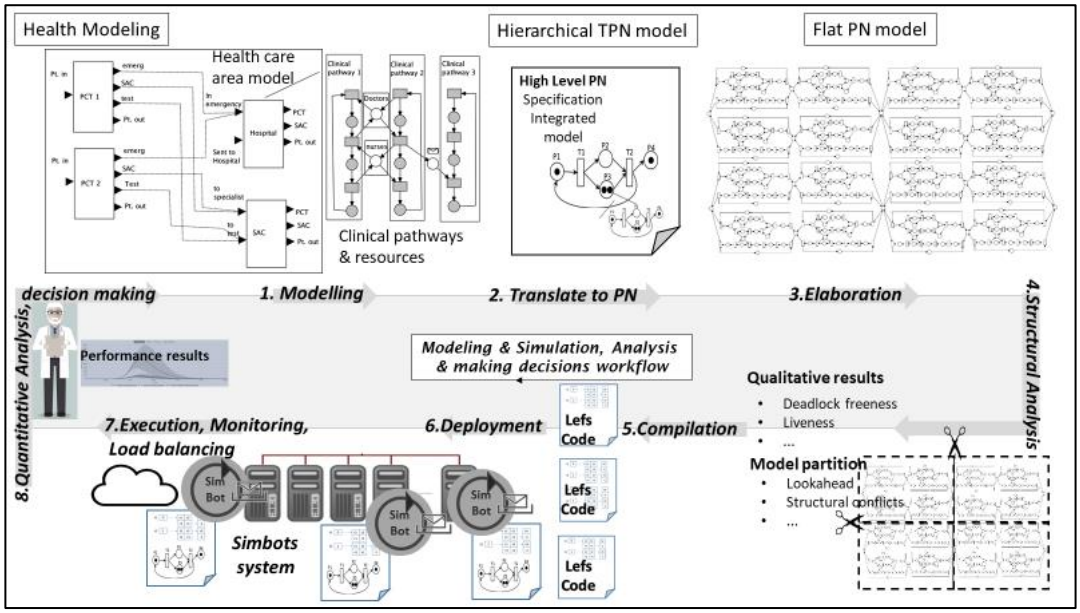


Figura 2: Modelado de sistema de salud, simulación distribuida, análisis y toma de decisiones Fuente: [7]

La figura anterior, extraída del artículo [7] con permiso de los autores, resume los pasos seguidos en la metodología allí propuesta basada en modelos formales.

1) **Modelado. Establecer principios de abstracción** que permitan construir modelos del comportamiento de los flujos de pacientes**,** siguiendo protocolos médicos preestablecidos, cada uno de los cuales está orientado a una patología, y teniendo en cuenta los recursos necesarios para la aplicación de los tratamientos Los principios de abstracción con los que están familiarizados los gestores de los sistemas de salud son los protocolos médicos, los recursos materiales y personal médico necesario en la aplicación de estos protocolos, y el intercambio de información entre los protocolos médicos. Generalizando a otros dominios de aplicación, nos encontramos como primitivas básicas los procesos, los recursos necesarios para llevar a cabo los procesos, y la comunicación asíncrona entre procesos. La composición de estas primitivas para definir componentes más complejos estructurados jerárquicamente permite

la construcción de modelos complejos escalables. La definición de estas primitivas básicas está basado en redes de Petri.

2) **Obtención de red de Petri Jerárquica. Definición de un lenguaje de descripción de modelos basada en las primitivas definidas anteriormente que permite la definición de un modelo de red de Petri jerárquico.** La definición de modelos mediante la composición de las primitivas básicas favorece la reutilización de componentes, las cuales pueden ser conectadas para configurar diferentes sistemas de salud. De cara a construir el modelo completo, se requieren métodos para la estructuración jerárquica y modular de las componentes. El modelo obtenido se completa con datos obtenidos de series de datos históricos. La composición modular y jerárquica da lugar a un modelo **j**erárquico de Red de Petri.
3) **Elaboración. Definición de la semántica operacional del lenguaje de descripción de modelos en términos de redes de Petri Lugar/Transición que permita construir un modelo equivalente plano ejecutable.** El modelo ejecutable así construido del sistema de salud se alimentará con un flujo de pacientes simulado u obtenido de series de datos históricas. El proceso de generación del modelo ejecutable a partir de la descripción se denomina **proceso de elaboración** del modelo y es el paso previo para realizar la simulación/ejecución del sistema de salud a estudiar. El proceso de elaboración transforma la red de Petri jerárquica en un modelo de red plano. El modelo plano se puede partir entre distintos simuladores y compilar para obtener una representación de la red de Petri que puede ser eficientemente ejecutado y particionado en un entorno de ejecución distribuido.
4) **Análisis Estructural. Utilización de un entorno de modelado y simulación con herramientas software para la generación de código ejecutable por la infraestructura de simulación distribuida.** El entorno aportará herramientas para realizar el proceso completo de traducción del modelo a código ejecutable en plataformas distribuidas. Este proceso requiere, además del proceso de elaboración, las siguientes herramientas: 1) Generar un sistema de información que permita traducir resultados y trasladar especificaciones entre el modelo de alto nivel y el modelo ejecutable; 2) Herramientas de análisis de redes de Petri que permitan extraer información estructural del modelo para su partición en subredes para una simulación distribuida segura sacando partido de la concurrencia que puede existir en el modelo, y otros criterios como la optimización del número de mensajes intercambiados entre los sitios del simulador distribuida; 3) Evaluación de prestaciones del modelo ejecutable que permita extraer información estructural sobre las dependencias entre eventos de distintas partes de la red para hacer pronósticos seguros sobre el avance de la simulación cuando hay falta de mensajes; etc.
5) **Compilación**. Generación de estructuras de datos para almacenar la red de Petri de forma que pueda ser ejecutado eficientemente. Las estructuras de datos se basan en la idea de “Linear Enabling Function”(LEF). La idea de función lineal de habilitación permite caracterizar la sensibilización de las transiciones mediante un valor entero. El proceso de compilación traduce la red de Petri en una red de propagación de valores enteros a las transiciones que se ven afectadas por el disparo de una transición [14].
6) **Despliegue. Disponibilidad de un entorno y herramientas software para el despliegue, control y explotación de la simulación distribuida del modelo.** Elemento importante de este entorno de simulación distribuida es la disponibilidad de una máquina virtual (***simbot***) para la simulación de redes de Petri con capacidad para integrarse en un conjunto coordinado de ellas en la simulación distribuida de un modelo particionado y cargado en ellas. Las herramientas de carga y arranque de los simbots; Mecanismos para el balanceo dinámico de carga; Mecanismo para la resolución de conflictos según diferentes

políticas especificadas por el diseñador; Recogida de resultados de simulación y procesamiento integrado de las trazas generadas; son algunas de las herramientas y mecanismos software para hacer operativo dicho entorno de simulación distribuida.

7) **Ejecución, Monitorización y balanceo de carga.** Disponibilidad de un middleware adaptado a la plataforma física de ejecución donde se integren no solo el simulador distribuido si no todos aquellos servicios necesarios para la distribución y localización de los simbots, flete seguros de eventos entre simbots distribuidos, solicitud de recursos físicos de ejecución, labores de contabilización y cualquier otra clase de servicios que permitan disponer de un entorno que permita el uso amigable de la plataforma física de ejecución y optimice la utilización de sus recursos.
8) **Análisis cuantitativo, toma de decisiones.** Finalmente, los resultados de la simulación se recogen para que los responsables puedan **tomar sus decisiones**, que pueden decidir redefinir el modelo buscando una gestión más eficiente de los recursos.

3.2. Lenguajes de modelado para el modelo funcional y estructural. Elaboración y compilación del modelo.

3.2.1. Descripción funcional del modelo. Lenguaje de descripción de RdP

La metodología propuesta parte de la definición de un lenguaje específico de dominio a partir de primitivas básicas especificadas como redes de Petri modulares. En esta sección se presenta la gramática para definir las redes de Petri. Una vez se tiene el modelo jerárquico del sistema, mediante el proceso de elaboración se creará un fichero acorde a la gramática definida en el que se representa la RdP plana equivalente al modelo a simular. En dicho fichero aparecerán las transiciones y lugares de la RdP, los tiempos de disparo de las transiciones, sus marcados iniciales, así como la partición en subredes si lo desea el usuario.

Cabe destacar que el lenguaje no es case-sensitive, es decir, no distingue entre mayúsculas y minúsculas.

La gramática que describe el lenguaje de descripción de las RdP se refleja en el Anexo 11.3. A continuación, se explica cada parte del fichero en el que se declara el modelo de un sistema:

Tabla 2: Explicación de la gramática del lenguaje de RdP

Red <nombreRed>;	Nombre de la RdP global que se va a simular.
Subred <nombreSubRed>;	Comienzo de la descripción de una subred de la red global.
Lugares <Lista_lugares>;	Enumeración de los lugares de la subred.
Transiciones	Inicio de la fase de descripción de las transiciones.
<nombreTransicion> :	Nombre o identificador de la transición. (No se permiten que dos transiciones se llamen igual en la misma subred)

Pre <Lista_lugares>;	Lugares que son entrada en la transición
Post <lugares>;	Lugares que son salida en la transición.
Marcado <Lista_lugares>;	(OPCIONAL) Establece el marcado en los lugares indicados. Por defecto todos los lugares tienen 0 marcas.
Tiempo <Lista_transiciones>;	(OPCIONAL) Establece el tiempo de disparo de las transiciones. Por defecto todas las transiciones son inmediatas, es decir, su tiempo de disparo es 0.
Interfase	(OPCIONAL) Inicio de la descripción de los puertos de las subredes.
Entrada <Lista_lugares>;	Lugares de puertos de entrada
Salida <Lista_lugares>;	Lugares de puertos de salida
Finsubred;	Fin de la descripción de la subred
Sincronizacion	Inicio de la descripción de sincronizaciones entre subredes. Se define un nombre común para aquellos puertos que sean de entrada y salida en subredes conectadas.
<subred, lista_nombreAntes> <= <lista_nombreNuevo>	Cambios de nombre de la subred
Finred;	Fin de la descripción de la red global

La Figura 3 muestra un ejemplo de la descripción de una RdP:

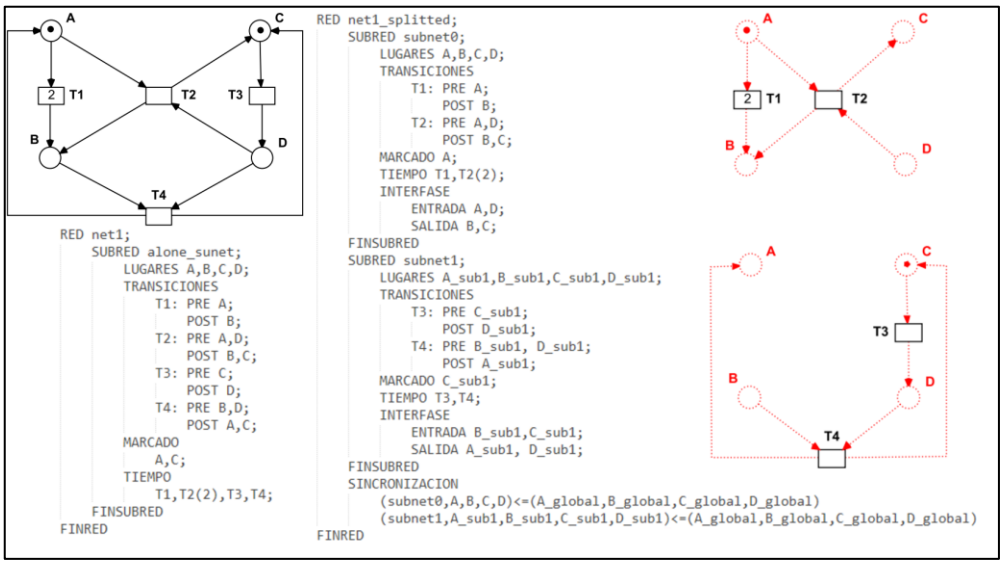


Figura 3: Ejemplo de descripción de una red expresada de manera gráfica, textual y dividida en 2 subredes diferentes Fuente: [5]

Una vez definida la gramática se debe definir las primitivas sobre las que se construirán los modelos. Se debe definir un lenguaje específico del dominio que facilite la construcción de los modelos por parte de los expertos en dicho dominio. Como punto de partida para la definición de dichas primitivas se ha considerado que la mayoría de los sistemas pueden ser modelados a partir de la identificación de tres entidades básicas que configuran un modelo de simulación: **procesos**, **recursos** y **canales**. Los **procesos** son partes del sistema que necesitan **recursos** para poder ejecutar su labor. Los **canales,** a su vez, son los medios por los que se manda la información a los demás procesos. Una definición más concreta de estas primitivas básicas se describen en el artículo [7].

3.2.2. Representación jerárquica del modelo.

El diseño de sistemas complejos y de grandes dimensiones (que pueden escalar a lo largo del ciclo de vida) no se puede abordar de una manera integral y unitaria. Este tipo de sistemas deben ser concebidos con una estructura interna, identificando partes y sus relaciones, que en definitiva consiste en aplicar el principio de “divide y vencerás” de amplia implantación en muchas ramas de la ingeniería. Un sistema completo consta de varios subsistemas denominados Componentes. Cada componente a su vez puede ser un sistema completo compuesto por más componentes o un sistema sencillo compuesto por un único componente

Por ejemplo, el sistema de salud para el tratamiento de flujos de pacientes introducido en el apartado anterior se compone de una serie de protocolos específicos adaptados a las diferentes patologías que presenten. A su vez, estos protocolos están constituidos por más subprotocolos y así hasta llegar a las primitivas más sencillas que compondrán las primitivas básicas que representan los procesos básicos, los recursos que comparten estos procesos, y el intercambio de mensajes entre los procesos.

La idea de estos componentes es similar a la idea de **Componente de Software**, que corresponde a una pequeña unidad modular de un programa que posee interfaces por donde le puede llegar o enviar información.

La gramática presentada permite definir las redes de Petri de forma modular definiendo redes formadas por subredes con una interface definida por lugares de entrada, lugares de salida, y relaciones de sincronización basadas en la fusión de lugares. Sin embargo, el lenguaje presentado no permite más que un nivel de jerarquía. Con el objeto de ofrecer la posibilidad de definir el modelo con mayores niveles de anidamiento, y sobre todo con la posibilidad de incorporar las primitivas definidas a cualquier lenguaje de programación se han definido primitivas de estructuración del sistema. De esta forma, el modelo funcional se representa mediante redes de Petri, y la estructura del sistema se puede representar en cualquier lenguaje de programación orientado a objeto que incorpore las primitivas propuestas. La representación de la jerarquía del modelo ha adoptado una ontología para la estructurar el sistema similar a la SES (*System Entity Structure*) de DEVS, o las entidades y arquitecturas de VHDL. Dicha representación especifica de forma declarativa la estructura de los modelos en términos de componentes y relaciones de acoplamientos. Los modelos se construyen a partir de entidades primitivas que incluyen **puertos de entrada y salida** y que definen su interfaz externa.

Un puerto de entrada hace referencia a una señal o evento recibido, y un puerto de salida hace referencia a eventos o señales para componentes externos. De manera informal, los puertos encapsulan un subsistema en el que se abre una puerta de entrada y salida de datos para que pueda enviar y recibir información del exterior.

Las **entidades atómicas** hacen referencia a redes de Petri especificadas en ficheros de texto. La definición de los puertos de entrada y salida se construye directamente a partir de la especificación del interfaz de la red de Petri. Las relaciones conectan puertos de salida con puertos de entrada. La Figura 4 muestra una entidad atómica que representa un proceso. El componente tiene un interfaz definido por un conjunto de puertos de entrada y salida que se obtienen a partir del interfaz de la RdP a la que se referencia para definir el comportamiento funcional del componente.

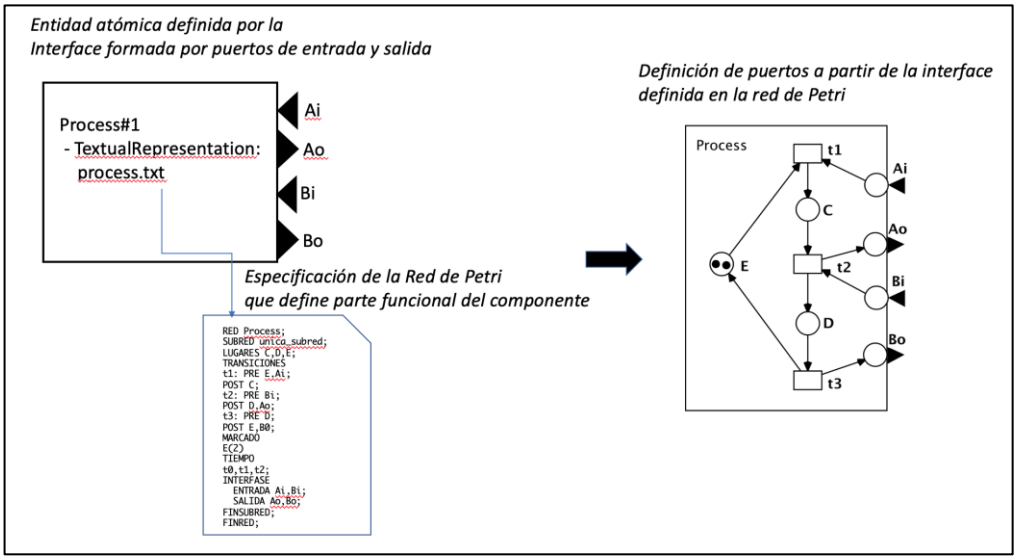


Figura 4: Entidad atómica

A continuación, se va a ilustrar un ejemplo de definición de **entidad compuesta** a partir de entidades atómicas. En la Figura 5 se muestra el modelado de procesos compitiendo por recursos. La parte izquierda de la figura muestra dos procesos secuenciales compitiendo por recursos.

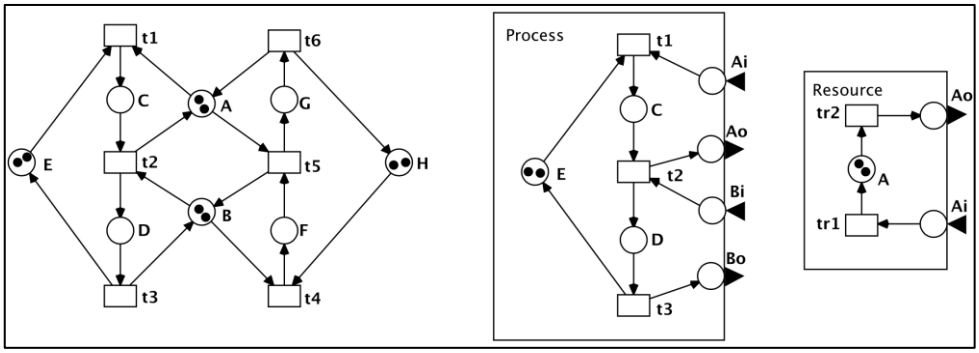


Figura 5: Modelado de procesos compitiendo por recursos.

El primer proceso se representa con los lugares C, D, E y las transiciones t1, t2, t3 que modelan un autómata. El autómata puede representar un proceso de trabajo que describe una secuencia de operaciones que debe realizar un sistema de fabricación para producir un producto, la transformación de datos o movimiento de documentos en un *workflow*, o el protocolo clínico que debe seguir el personal de un hospital en determinadas situaciones. Los lugares F, G, H y las t4, t5, t6 representan el segundo proceso secuencial. Las marcas en el lugar E representan dos procesos concurrentes siguiendo el primer proceso secuencial, por ejemplo, dos piezas que siguen el mismo plan de proceso, o dos pacientes que deben ser atendidos. Lo mismo ocurre con las marcas en el lugar H. Los lugares A and B representan la capacidad de los recursos necesarios para realizar estos procesos. Para un sistema, un recurso puede ser una herramienta necesaria para realizar el plan de proceso de producción de un producto (transporte, operación, almacenaje), un recurso computacional para transformar un dato o documento en un *workflow*, o un doctor o una habitación necesarios para atender o alojar a un paciente. La figura muestra los recursos compartidos por los procesos y representados por dos marcas en A y B. Los recursos pueden ser utilizados en cada etapa de los procesos. El disparo de la transición t1 representa la adquisición de un recurso que es liberado por el proceso cuando se dispara t2. T1 y t5 son transiciones en conflicto, representando que los dos procesos compiten por los recursos.

La parte derecha de la figura muestra las **entidades atómicas** definidas para representar este comportamiento distinguiendo los recursos y los procesos. La representación de las entidades utiliza la representación gráfica de DEVS para las entidades con cajas y las puntas de flecha (triángulos negros) para representar los puertos de entrada y salida. Los puertos de entrada se asocian con lugares sin arcos de entrada y con un arco de salida como máximo. Los puertos de salida se asocian con lugares sin arcos de salida y un arco de entrada como máximo. El proceso representado utiliza recursos en dos etapas, lo que requiere representar cuando el evento de recursos disponible en los puertos A_i y B_i, y liberar los recursos que se representa con los puertos A_o y B_o. El recurso es representado por un solo lugar A con una marca por cada recurso disponible. En la figura, se muestran las dos transiciones para representar la obtención y la liberación del recurso.

En la siguiente figura se muestra la obtención del comportamiento equivalente al proceso planteado en la parte izquierda de la figura anterior mediante la composición de entidades.

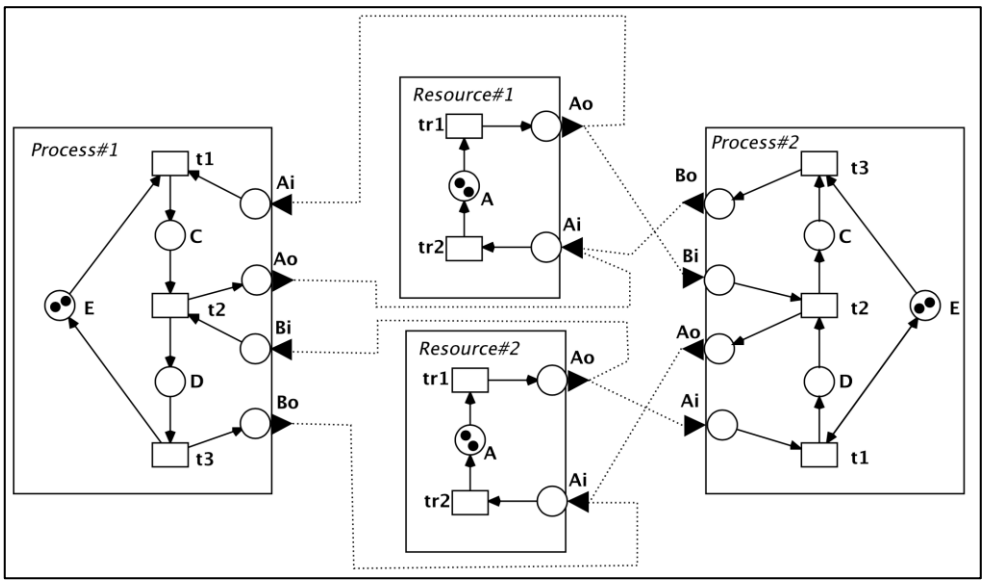


Figura 6: Composición de entidades

Los modelos jerárquicos son modelos que agregan componentes que pueden ser entidades atómicas o agregados y que especifican las conexiones entre los componentes.

La manera de construir una estructura compleja consiste en: (1) Creación de las componentes básicas, definiendo su sistema y estableciendo los puertos de entrada y salida. (2) Conectar los puertos de salida de las componentes con los puertos de entrada de entrada de otras componentes, obteniendo así una nueva estructura más compleja. (3) Definir nuevos puertos de entrada y salida de la nueva estructura, y seguir conectando con otras estructuras.

Las entidades que agregan otras entidades generan una representación textual en el que cada red de un subcomponente es una subred, y las conexiones entre redes se traducen en relaciones de sincronización, generando una representación textual de la red agregada. En el caso de que la red generada sea utilizada como subcomponente de otra componente, se genera la red plana como subred para el siguiente nivel. De esta forma, **el proceso de elaboración** se realiza mientras se genera el modelo.

En la siguiente imagen se puede ver un ejemplo de un sistema compuesto por 3 subcomponentes.

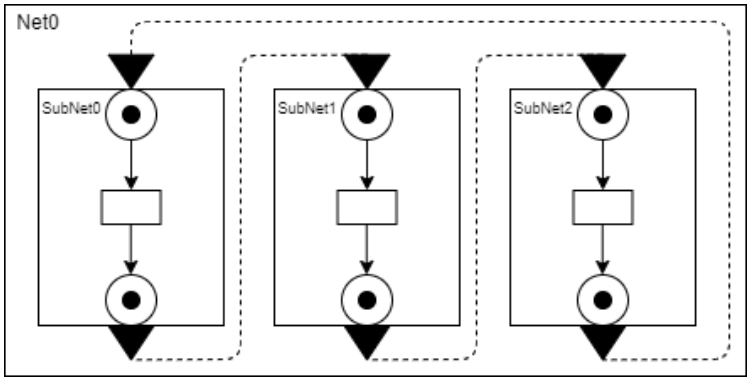


Figura 7: Red compuesta por 3 subcomponentes

A continuación, se muestra otro sistema compuesto por 2 subredes, que a su vez tienen 3 subredes dentro.

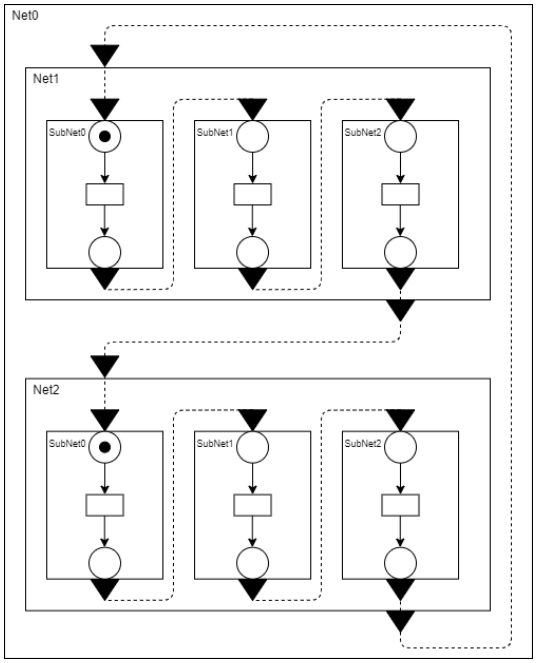


Figura 8: Composición de 2 sistemas con 3 subsistemas cada uno

3.2.2.1. Lenguaje de componentes en Java y proceso de Elaboración

Un modelo jerárquico consiste en un modelo compuesto de manera modular en el que se han ido construyendo, a partir de componentes básicos, una estructura compleja. La manera de construirlo consiste en la reutilización de componentes, conectados entre sí para configurar el sistema. Su uso permite crear estructuras complejas, similares a la de un sistema complejo, de una manera sencilla componiendo a partir de primitivas básicas.

El lenguaje de componentes desarrollado para Java se inspira en DEVSJAVA [15], que implementa clases e interfaces en JAVA para representar la SES de DEVS. Una entidad *DEVS* se especifica a través de entradas, salidas y estados, de manera similar a una máquina de estados [16]. Hay que notar que no se ha seguido

la especificación funcional de DEVS, que ha sido reemplazada por las redes de Petri, y que la semántica de puertos y componentes ha sido definida en función de la especificación textual de la red de Petri. Por lo tanto, el paralelismo con DEVs se reduce a la idea de componente con interfaz especificada con puertos, y a la utilización de las primitivas para la representación de la estructura jerárquica del sistema. Su uso permite crear estructuras complejas, similares a la de un sistema complejo, de una manera sencilla componiendo a partir de primitivas básicas.

Desde el punto de vista de la implementación, el paquete `componentes` es una capa que se inspira en la capa GenDEVs que implementa modelos atómicos y acoplados de DEVS. Se ha utilizado directamente el paquete `GenCol` del DEVS-Suite_5.0.0. La implementación del paquete Java `GenCol` define la capa de estructura de los modelos DEVS definiendo estructuras básicas como `Bag`, *`Relation`*, y *`ensembleCollection`*. Para definir el paquete `componentes` se ha revisado la parte del paquete `GenDevs` correspondiente al modelado de componentes (denominado `model.modeling` en DEVS-Suite_5.0.0) sirviendo de inspiración para definir componentes y la definición de modelos jerárquicos de nuestro framework. El objetivo de la implementación ha sido definir las clases e interfaces que puedan servir de referencia para cualquier implementación en un lenguaje orientado a objeto. La siguiente figura muestra cada interface del paquete components, excepto las interfaces `EntityInterface` y `EnsembleInterface` de la capa `GenCol`. Cada interface define un subgrupo de funcionalidad relacionada. La Interface `CoupleComponentInterface` permite añadir componentes a un componente agregado (*Coupled Component),* y el componente *IOComponentsInterface* definir la interface añadiendo, quitando, y conectar los puertos de un componente. `ComponentsInterface` permite recorrer todos los puertos, y `IoBasicSubNetInterface` recuperar la especificación textual de un componente con comportamiento funcional representado por una RdP, o el fichero que contiene dicha representación.

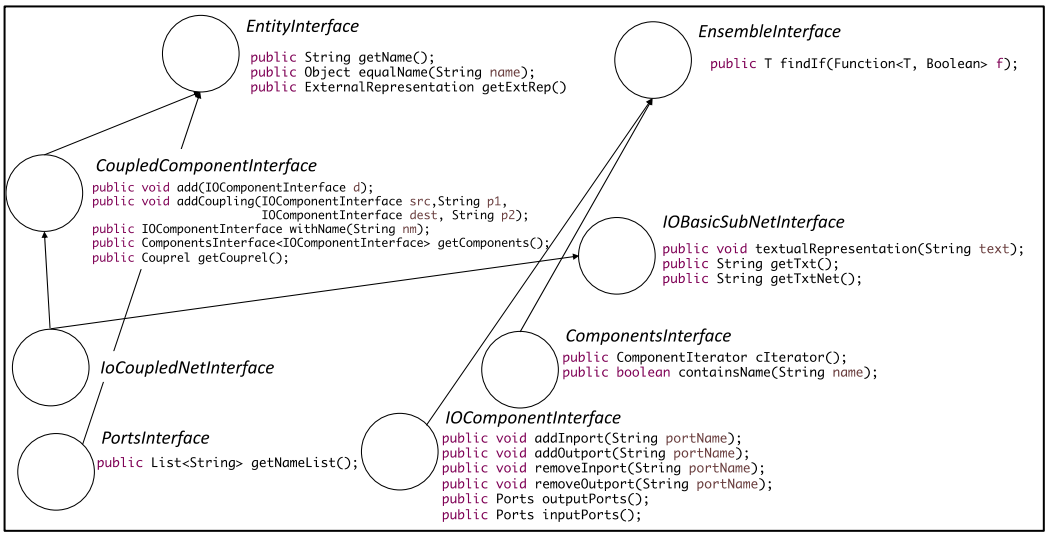


Figura 9: Interfaces JAVA para la que agrupan la funcionalidad para definir componentes

En el paquete componentes se definen las clases que implementan dicha funcionalidad. La clase `IOComponentImpl` implementa la interface `IOComponentInterface` y la clase `IOCoupledNetImpl` hereda de IOComponentImpl e implementa IOCoupledNetInterface, y la clase `IOSubNetImpl` hereda de `IOComponentImpl` e implementa IOBasicSubNetInterface. De esta forma las instancias de `IOSubNetImpl` implementan componentes con la descripción de

subredes que pueden ser agregadas jerárquicamente en instancias de la clase `IOCoupledNetImpl` para definir un modelo complejo. A continuación, se ilustra brevemente las primitivas y se presenta un ejemplo.

```
1  // Atomic entities
   IOSubNetImpl subnet_p1= new IOSubNetImpl("process1");
3  subnet_p1.textualRepresentation("process");
   IOSubNetImpl subnet_p2= new IOSubNetImpl("process2");
5  subnet_p2.textualRepresentation("process");
   IOSubNetImpl subnet_r1= new IOSubNetImpl("resource1");
7  subnet_r1.textualRepresentation("resource");
   IOSubNetImpl subnet_r1= new IOSubNetImpl("resource2");
9  subnet_r1.textualRepresentation("resource");
   // Coupled entity
11 IOCoupledNetImpl GlobalNet =
        new IOCoupledNetImpl("processes_Resources");
13 netGlobal.add(process1);
   netGlobal.add(process2);
15 netGlobal.add(resource1);
   netGlobal.add(resource2);
17 // Coupling relations
   netGlobal.addCoupling(subnet_r1,"Ao",subnet_p1,"Ai");
19 netGlobal.addCoupling(subnet_p1,"Ao",subnet_r1,"Ai");
   netGlobal.addCoupling(subnet_r2,"Ao",subnet_p1,"Bi");
21 netGlobal.addCoupling(subnet_p1,"Bo",subnet_r1,"Bi");
   netGlobal.addCoupling(subnet_r1,"Ao",subnet_p2,"Ai");
23 netGlobal.addCoupling(subnet_p2,"Ao",subnet_r1,"Ai");
   netGlobal.addCoupling(subnet_r2,"Ao",subnet_p2,"Bi");
25 netGlobal.addCoupling(subnet_p2,"Bo",subnet_r1,"Bi");
```

Figura 10: Ejemplo con guía de estilo JavaDevs para redes acopladas Fuente: [7]

La Figura 10 muestra un ejemplo de definición de componentes independientes como instancias de la clase `IOSubNetImpl` definidas como subredes, cuatro procesos y un recurso (Líneas 1-9). La red `GlobalNet` es una instancia de `IOCoupleNetImpl`, a la que se agregan las subredes y se definen las relaciones entre puertos de sus componentes (Líneas 10-25).

La primitiva de conexión `addCoupling`, conecta puertos de salida con puertos de entrada, dando lugar a la fusión de los lugares de salida declarados en la subred de una componente con los lugares de entrada declarados en la subred de otra componente. Una vez se tiene toda la jerarquía y las entidades ya están conectadas se debe realizar la fusión de lugares. Esta etapa consiste en cambiar el nombre de los puertos de entrada y salida que estén conectados. Para ello se recorre cada puerto de entrada y salida de las subredes y se va cambiando en nombre por un nombre global. Esto aparecerá en el fichero plano para que el simulador sepa que red están conectadas y que lugares son los mismos.

Como se puede apreciar, la generación de un modelo jerárquico empieza creando un RdP acoplada llamada *IOCoupledNetImpl.* Dicha red albergará a su vez otra red IOCoupledNetImpl creando así el primer nivel de jerarquía. La idea de ir creando niveles de jerarquía viene dada por la transformación de una red acoplada a una nueva subred vista como una red completa plana. A su vez, esa subred puede ser añadida a otra red acoplada añadiendo otro nuevo nivel de jerarquía. La transformación de red completa acoplada (que contiene un numero de subredes, conectadas entre ellas) a una subred que pueda ser conectada con otras subredes se realiza con el método *convertirRed(IOCoupledNetImpl)*. Para ello, realizará la fusión de nombres globales.

Por construcción, cuando se genera una componente a partir de subcomponentes, se genera la especificación textual como modelo plano a partir

de las especificaciones textuales de los subcomponentes. Un modelo plano consiste en un modelo sin jerarquía. Dicho modelo se puede describir con un lenguaje en un fichero de texto que luego será interpretado por el simulador. La conexión de puertos de las componentes agregadas en un Coupled Component definida mediante la primitiva `addCoupling`, se traduce en la fusión de los lugares correspondientes de las componentes. De esta forma, mediante el proceso de definición de la especificación textual del comportamiento de las componentes agregadas se realiza el proceso de elaboración al obtener una red plana mediante la fusión de los lugares de las componentes agregadas siguiendo la especificación de conexiones de los puertos de entrada y salida.

En el Anexo 11.11, se muestra un diagrama de todas las clases implementadas. En el Anexo 11.12 se presentan las fases del compilador y, por último, en el Anexo 11.13 se detalla el proceso de elaboración. En el paquete `componentes` se definen los constructores de las componentes atómicas y agregadas, y clases auxiliares como la clase `FusionPlaces` que soportan el proceso de elaboración.

3.2.3. Compilación de una RdP – Sistema de información asociada a la declaración del modelo de un sistema de eventos discretos

Una vez se tiene el fichero creado, lo primero es compilar el fichero y representarlo de manera interna en el simulador. El objetivo de esta transformación es obtener una representación de la RdP para simular de formar eficiente el comportamiento definido. En esta sección mostraremos como traducir la estructura y marcado de Place/Transition net, representada con la gramática definida en la sección 3.2 a un conjunto de funciones de sensibilización de transiciones como Código optimizado para motores de simulación centralizado o distribuidos.

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:$\boldsymbol{R}(N,\mathbf{m_0}) \rightarrow Z$ que hace corresponder a cada marcado alcanzable de la red de Petri, $m \in \boldsymbol{R}(N,\mathbf{m_0})$, un entero de manera que la transición t está sensibilizada si y solo si $f_t(\mathbf{m}) \leq 0$. Por ejemplo, para la transición T2 de la red de la figura 3 su Lef es: $f_{T2}(\mathbf{m}) = 2 - (m[A]+m[D]), \forall m \in \boldsymbol{R}(N,\mathbf{m_0})$, donde m_0 es el marcado inicial. Por tanto, $f_{T2}(\mathbf{m_0}) = 2 - (m_0[A] + m_0[D]) = 2-1=1$, 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 $f_{T1}(\mathbf{m_0})=0$, por lo que está sensibilizada en el marcado inicial.

La utilización de LEFS tal como se han presentado para la caracterización de la sensibilidad de una transición, requiere de una representación explícita del marcado de la red y de la LEF como función. Para una simulación distribuida la representación del estado como un conjunto de variables compartidas requiere de mecanismos para mantener la consistencia del marcado, y las funciones requieren de una reevaluación continua. Para evitar estos problemas, la representación de los LEFS ha tenido en cuenta que: (1) Sólo es necesario guardar el valor de la función LEF (inicialmente este valor se corresponde con el valor en el marcado inicial $\mathbf{m_0}$, que puede calcularse y almacenarse en tiempo de compilación; y 2) Cada vez que se dispara una transición en la red, se propaga una constante a las transiciones cuya sensibilidad se ve afectada. Esta constante se utiliza para actualizar el valor de las LEF afectadas.

Esta estrategia de propagación de constantes a las transiciones afectadas es fácilmente implementable mediante la traducción de la RdP a una red de propagación de constantes, en la que no es necesaria la representación explícita del marcado ni reevaluar constantemente el valor de las funciones LEF. De esta forma, el estado de la simulación viene determinado por los valores en curso de las funciones LEF y los cambios de estos valores se basan en constantes enviadas a las transiciones que cambian sus condiciones de disparo.

Dado el disparo de la transición t' en el marcado m, $\mathbf{m} \xrightarrow{t'} \mathbf{m'}$, el valor de la LEF de las transiciones que se ven afectadas por ese disparo, $f_t(\mathbf{m'})$ se puede calcular a partir del valor $f_t(\mathbf{m})$ y de un parámetro estático conocido en tiempo de compilación que representa el cambio de f_t cuando se ocurre t'. Dicho parámetro corresponde a los cambio en el contenido de marcas de los lugares de entrada de t como consecuencia del disparo de t'. La equiación de actualización de la función LEF de t cuando t' ocurre se puede definir como $f_t(\mathbf{m'}) = f_t(\mathbf{m})+UF(t' \rightarrow t)$, donde $UF(t' \rightarrow t)$ es el Factor de actualización (*Updating Factor*) de t' sobre t obtenido de la estructura de la red y del marcado inicial.

De acuerdo con lo dicho, el resultado de la compilación para generar una codificación LEF agrupa la siguiente información para cada transición:

- **Identificador de transición**: Identificadores globales que se asignan a cada transición de manera única.
- **$\tau(t')$**: Tiempo determinista de disparo de la transición, asignado entre paréntesis en cada transición
- **Contador**: Variable que contiene el valor actual del LEF asignado a cada transición que se inicializa a $\mathbf{m_0}$.
- **Lista de actualizaciones inmediata** (Immediate Updating List, **IUL**(t')): Que contiene la lista de transiciones con sus correspondientes UFs cuyos LEFS deben ser actualizados de manera inmediata al disparar una transición t'.
- **Lista de actualizaciones proyectadas** (*Projected Updating list*, **PUL**(t')): Lista de transición cuyos LEFS deben ser actualizados con sus correspondientes UFs **después** de que haya pasado el tiempo de disparo de la transición.

En la Figura 11 se puede ver el resultado de la compilación de la RdP de la Figura 2, y que mostramos de nuevo a la izquierda para facilitar la comprensión de la estructura de datos generada. Además de las estructuras LEFs, el proceso de compilación como se verá en la sección 3.3.2 agrupa las transiciones en relaciones de conflicto acoplado. En el ejemplo mostrado, presentará dos grupos conflicto, el primero con las transiciones 1,2 y 4, y el segundo con la transición 3.

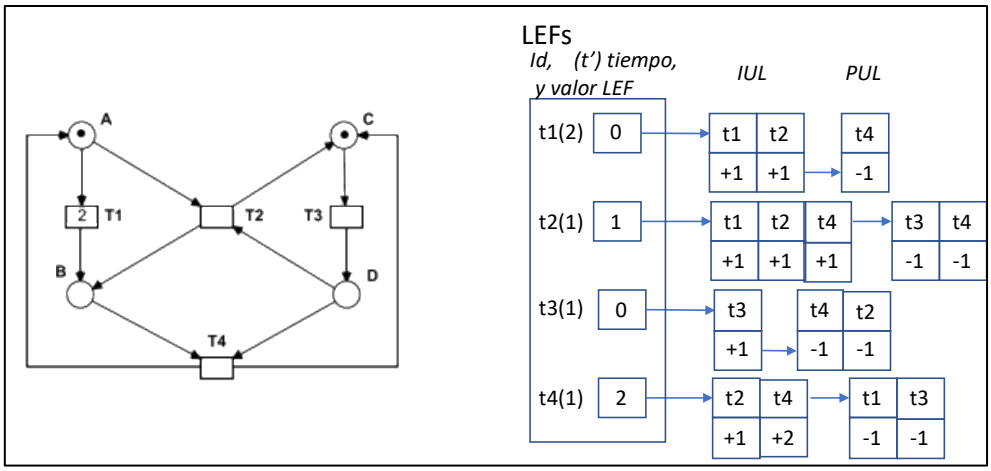


Figura 11: Resultado de la compilación/elaboración de la RdP de la figura 7 Fuente: [5]

El objetivo de esta compilación no es únicamente traducir el lenguaje de alto nivel de las redes al tipo de dato usado en el simulador, si no es que es obtener una representación **muy eficiente para la interpretación y posterior simulación del modelo introducido**. Es importe observar que la información asociada a cada transición es independiente de las descripciones de otras transiciones si no tenemos en cuenta las transiciones en conflicto. Por lo tanto, la partición del modelo para una simulación distribuida solo requerirá definir el conjunto de transiciones que se agruparán en cada motor de simulación. De esta forma, el balanceo de carga se simplifica al poder mover la información de la codificación LEF de las transiciones correspondientes de un motor a otro. Las transiciones en conflicto pertenecerán se agruparán siempre en el mismo motor de simulación para evitar comunicaciones adicionales a la hora de elegir que transición se dispara entre las que se encuentran en conflicto.

El compilador usa la librería JLex y *Jcup* y se ha basado en el trabajo previo de TFG de *Miguel Ángel Barcelona Liédana y Luis Caballero Fernández* [13]. La compilación de una RdP se compone básicamente de 4 fases:

- Recorrido del fichero plano para la obtención de los tamaños de la lista de transiciones, lista de subredes, etc...
- Relleno de las estructuras de datos.
- Conversión de la estructura de datos en una nueva estructura de datos LEF.
- Obtención de las redes que son adyacentes y siguientes.

En la **primera fase** se comprueba si el fichero es léxica, sintáctica y semánticamente correcto. Se hace un recorrido del fichero plano para la obtención de los tamaños de la lista de transiciones, lista de subredes, etc. Además, se guarda el número de subredes y de lugares para la segunda fase. En la **segunda fase** se puede suponer que el fichero es correcto y por tanto se va rellenando la estructura interna. La RdP se almacena en una estructura de datos en la que se guarda la información relativa a la compilación de cada subred. De cada subred se almacena la lista de lugares, de transiciones, los marcados iniciales, etc. En la **tercera fase**, se hace la conversión de la estructura de datos en la nueva estructura de datos LEF presentada en esta sección. Finalmente, en **la cuarta fase**, se obtienen las redes que son adyacentes, que pueden ser utilizadas como partición inicial en una simulación distribuida. Esta partición inicial manualmente especificada, puede realizarse de diferente maneara atendiendo a diferentes criterios.

A partir de las estructuras de datos generados, es posible la traducción de la representación de red de Petri en la gramática especificada a otros formatos de texto, y otras representaciones. En esta sección se ha presentado la representación basada en la idea de LEF, que permitirá una ejecución eficiente del modelo.

Respecto a las estructuras de datos utilizadas, cabe destacar que las listas pueden ser implementadas con dos enfoques diferentes: los vectores estáticos y los vectores dinámicos. Los **vectores** estáticos reservan una cantidad fija de memoria para su uso. Dicha cantidad es invariable en tiempo de ejecución. El inconveniente de los vectores estáticos es que es necesario definir antes de usarlos el tamaño que van a tener. Como ventaja tienen que son más eficientes y sus accesos son muchos más rápidos que los dinámicos. Los **vectores dinámicos** no reservan una cantidad fija invariable, si no que reservan una cantidad determinada y mientras va creciendo se va moviendo a nuevos sitios con más espacio. Cabe destacar que es el sistema operativo el encargado de esa sobredimensión. Como ventaja tienen que pueden ir creciendo en tiempo de ejecución. Como desventaja es que son menos eficientes que los estáticos y que los accesos son más lentos.

Como el propósito de este TFG es la obtención de un simulador eficiente de RdP se ha elegido usar vectores estáticos. Por ello en la primera pasada del compilador se obtienen las dimensiones de cada lista y se crea con dicho tamaño para, en la segunda pasada, rellenar dichas listas con los datos obtenidos.

4. SIMULACIÓN DEL MODELO BASADO EN LEF y MICRO-KERNEL DE SIMULACIÓN.

En esta sección se presenta la interpretación de las RdP basadas en la sensibilización de transiciones y en la idea de LEF. Para dicha interpretación, debemos tener en cuenta las transiciones que se encuentran en conflicto que se presenta en la subsección 4.2.3. Estas transiciones no podrán ejecutarse concurrentemente y deberán tener asociadas una estrategia de disparo conjunta. En la sección 4.1.4 se presenta el Micro-kernel de simulación con las componentes y servicios básicos para interpretar el código obtenido. Las componentes típicas de una arquitectura de simulación distribuida aparecen en el Anexo 11.6. En las subsecciones 4.2, 4.3 y 4.4 se presentan los detalles de la arquitectura concreta desarrollada y los algoritmos de simulación centralizado y distribuido.

4.1. Ejecución de RdP basadas en el seguimiento de la sensibilización de sus transiciones: Linear Enabling Function (LEF)

4.1.1. Ejecución de una RdP y tiempos de disparo

La ejecución de una RdP está basada en el disparo de sus transiciones sensibilizadas en el instante de tiempo de reloj previsto. Por tanto, la ejecución de una RdP se realiza mediante un proceso iterativo dirigido por el tiempo de simulación, donde en cada iteración se distinguen dos fases diferenciadas:

- *Detección de las transiciones sensibilizadas* para el tiempo de simulación actual después de haber tomado efectos los eventos previstos para ese tiempo de simulación. Los eventos proyectados son los cambios en el marcado de los lugares como consecuencia del disparo de transición en tiempos de simulación anteriores al actual o causalmente anteriores a las

transiciones que están siendo analizadas para su sensibilización. Se recuerda que una transición está sensibilizada cuando cada uno de sus lugares de entrada contiene un numero de marcas igual o superior al peso del ardo de entrada de la transición. Además, en esta fase del ciclo de simulación se deberá resolver los conflictos entre transiciones, es decir, se deberá decidir qué transiciones se disparan de un conjunto de transiciones acopladas, por compartir lugares de entrada, y que no todas pueden ser disparadas de manera concurrente porque los lugares compartidos no poseen suficientes marcas para todas.

- Una vez se detecta el conjunto de transiciones sensibilizadas y seleccionado el conjunto de las que se dispararán de manera concurrente se procede a su disparo. *El disparo* de esas transiciones producirá modificaciones en los contenidos de marcas de los lugares de entrada y de salida conectados a la transición que se ha disparado. Cada uno de esos cambios de marcado es lo que se denomina **evento** y estará proyectado *para un tiempo futuro* que se construye a partir del tiempo de simulación actual más el tiempo asociado al disparo de la transición que ha producido el evento. Estos eventos se irán propagando por el sistema de simulación encolándose de manera ordenada según el tiempo futuro para el que fueron proyectados y tomarán efecto, si su tiempo asociado coincide con el tiempo de simulación actual, en la fase 1 del ciclo de simulación en el momento que se actualicen las condiciones de sensibilización de las transiciones.

Los tiempos de disparo están relacionados con el tiempo que se tarda desde que se dispara la transición t hasta que los efectos del disparo llegan a los lugares siguientes a esta. Por ejemplo, la transición T1 de la Figura 11 posee un tiempo de disparo de 2 y por tanto al dispararse sus efectos no se verán hasta pasado 2 unidades de tiempo desde el tiempo actual del ciclo de simulación en el que se ha producido el disparo. De ahí que al dispararse T1 en el tiempo 0, hace que la transición T4 no esté sensibilizada hasta tiempo 2.

4.1.2. Conflictos – Coupled Conflict Sets

De una manera informal, dos transiciones se encuentran en conflicto estructural si y solo si comparten algún lugar de entrada. Por ejemplo, en la Figura 11, las transiciones T1 y T2 se encuentran en conflicto dado que ambas tienen como lugares de entrada el lugar A. La denominación de conflicto procede del hecho de que, bajo algún marcado dado, las dos transiciones se encuentran sensibilizadas (los lugares de entrada de cada una de ellas tienen suficientes marcas como para dispararlas) pero no se pueden disparar las dos, ya que supondría retirar un número de marcas del lugar compartido que no están disponibles. Considérese el marcado sucesor de la inicial en la figura 11 después del disparo de la transición T3. Este marcado contiene una marca en el lugar A y una marca en el lugar D, es decir, la transición T1 está sensibilizada porque su único lugar de entrada, A, contiene una marca y la transición T2 también se encuentra sensibilizada porque sus dos lugares de entrada, A y D, contienen cada uno una marca. No obstante, las dos transiciones T1 y T2 no se pueden disparar juntas ya que ello requeriría retirar dos marcas (una por el disparo de T1 y otra por el disparo de T2) del lugar de entrada compartido, A, que no se encuentran disponibles. Es decir, las transiciones están en conflicto efectivo en el marcado descrito dado que, aunque las dos están sensibilizadas, el disparo de una de ellas inhibe el disparo de la otra por culpa del marcado del lugar de entrada compartido.

La relación de conflicto es una **relación binaria**, pero no transitiva en general. Por ejemplo, en la Figura 11 la transición T1 se encuentra en conflicto con la

transición T2 ya que comparten el lugar de entrada A; y la transición T2 se encuentra en conflicto con la transición T4 porque comparten el lugar de entrada D; sin embargo, la transición T1 no está en conflicto con la transición T4 ya que no comparten ningún lugar de entrada. No obstante, la existencia de estas dos relaciones de conflicto sí que introduce interferencias en el disparo simultaneo de estas tres transiciones y, sobre todo, para la decisión de que transición disparar del conjunto de estas 3 transiciones. Obsérvese que en el análisis de que transición es disparable de entre todas las sensibilizadas, se deberá estudiar conjuntamente todas las transiciones que tienen alguna relación de conflicto entre ellas para la determinación del conjunto máximo de transiciones que se pueden disparar a la vez de entre las transiciones con conflictos acoplados y seleccionar uno de estos conjuntos máximos como candidatos a disparar. Por ejemplo, en la red de la Figura 7, se puede considerar un marcado inicial que contiene una marca en cada uno de los lugares A, B y D, y cero marcas en C. En este marcado, se encuentran sensibilizadas las transiciones T1, T2 y T4, pero solo existen dos alternativas de máximo número de disparos en este marcado: (1) Disparar T2, invalidando el disparo de T1 y T4; (2) Disparar concurrentemente T1 y T4, invalidando el disparo de T2. Este conjunto de transiciones T1, T2 y T4 se denomina ***Conjunto de Transiciones en Conflicto Acoplado*** *(en inglés, Coupled Conflict Set, CCS)*. Desde un punto de vista de cálculo, la relación *Coupled Conflict* es la cerradura transitiva de la relación de Conflicto y los *Coupled Conflict Sets* son las clases de equivalencia (conjunto cociente) de esta relación de equivalencia que es la relación *Coupled Conflict.*

En el proceso de compilación se realiza el análisis de la RdP y se determinan los *Coupled Conflict Sets* de la red de forma que el *Coupled Conflict Set* al que pertenece una transición es una propiedad asociada a cada transición y será utilizada en la primera fase del ciclo de simulación, encargada de determinar los conjuntos máximos de transiciones que se pueden disparar de manera concurrente bajo un marcado. Obsérvese que dado un *Coupled Conflict Set*, en un marcado dado, habrá que determinar en esa fase primera de la simulación los subconjuntos máximos del CCS compuestos por transiciones disparables de manera concurrente bajo ese marcado. En el análisis de sensibilización habrá que seleccionar uno y solo uno de estos subconjuntos máximos como transiciones a disparar en la segunda fase del ciclo de simulación. Los criterios para la selección de uno y otro de estos subconjuntos pueden ser diferentes, pero en esencia será un conjunto de reglas que pueden utilizar información histórica de las secuencias de disparo para primeras uno conjunto sobre otro. Entre los criterios más populares están: (1) Selección aleatoria; (2) Disparo del subconjunto menos frecuentemente disparado; (3) Disparo del subconjunto más frecuentemente disparado; (4) Establecimiento de un turno fijado por el diseñador; (5) etc. Todas estas políticas serán especificadas por el diseñador como parte de las especificaciones de la simulación como “políticas de resolución de conflictos”.

4.1.3. Caracterización de los eventos en una ejecución de la red de Petri basada en LEFs e información extraída de la ejecución de una red

La propagación de los efectos del disparo en una transición dada consiste en la transmisión de las variaciones del contenido de marcas de los lugares de entrada y de salida de la transición disparada. Cada una de estas variaciones del marcado de un lugar tiene efectos sobre el grado de sensibilización de las transiciones de salida de los lugares cuyo contenido de marcas ha sido modificado. Además, cada una de estas variaciones de marcado se ha producido en un instante de tiempo de simulación determinado por el ciclo de simulación en el que se disparó la transición y la duración de la transición disparada. Estas dos informaciones: variación del marcado de un lugar y tiempo en el que se produce dicha variación es lo que

constituye un evento en el proceso de simulación. Un evento es generado por una transición disparada y debe ser enviado a cada una de las transiciones de salida del lugar cuyo contenido de marcas varía según el valor entero (positivo en caso de que aumente el número de marcas y negativo si se disminuye el contenido de marcas) almacenado en el evento. Estos eventos se almacenan en **una cola de eventos ordenada por tiempo** en el que se debe producir la variación de marcado. Cuando un evento toma efecto por que se ha alcanzado el tiempo por el que está proyectado, se toma la constante que representa la variación y se actualiza la LEF de la transición para la que está proyectado el evento (la transición de destino también está almacenada en el propio evento).

El valor de las constantes puede ser negativo o positivo. El valor positivo indica quitar marcas de un lugar y el valor negativo indica añadir marcas a un lugar. Los valores de estas constantes son calculados de manera estática en tiempo de compilación/elaboración y forman parte de la información asociada a una transición, para que al disparar una transición se pueda enviar las constantes a las transiciones que se vean afectadas por el disparo realizado.

En el ejemplo de la Figura 11, al disparar T1 ocurren dos eventos inmediatos y un evento proyectado. Los dos eventos inmediatos envían la constante +1 a T1 y T2 ya que se quita una marca de su lugar adyacente (son inmediatos ya que t1 al lanzarse deja de estar sensibilizada y T2 porque su adyacente es compartido con T1). El evento proyectado implica enviar una constante -1 ya que el disparo de T1 implica añadir una marca en el lugar adyacente a T4. El evento enviado contendría la constante -1, la transición destinataria t4 y el tiempo de aplicación 2.

Los eventos contienen el valor de la constante que se envía, la transición destino y el tiempo en el que debe ser aplicada.

Anotar cada disparo de transición y el tiempo en el que se ha realizado permite extraer información relativa al progreso del sistema a lo largo del tiempo y ver cómo ha evolucionado. Además, obteniendo los disparos es posible la reconstrucción del estado viendo la evolución del marcado y determinar cómo se han ido moviendo las marcas a lo largo de la simulación.

El análisis de dicha información extraída es relevante para desarrollar particiones más optimas y evaluar el número de recursos requeridos para simular el sistema en un tiempo estimado, y el coste de dichas decisiones.

4.1.4. Poniendo todo junto: Arquitectura del micro-kernel de simulación: Simbot

En la Figura 12, se presentan las componentes básicas del micro**-kernel del simulador** al que denominamos **Simbot** y una introducción típica de simulador distribuido se encuentra en el Anexo 11.6.

Un **simbot** es un proceso ligero que posee las directivas de simulación y comunicación entre otros simbots.

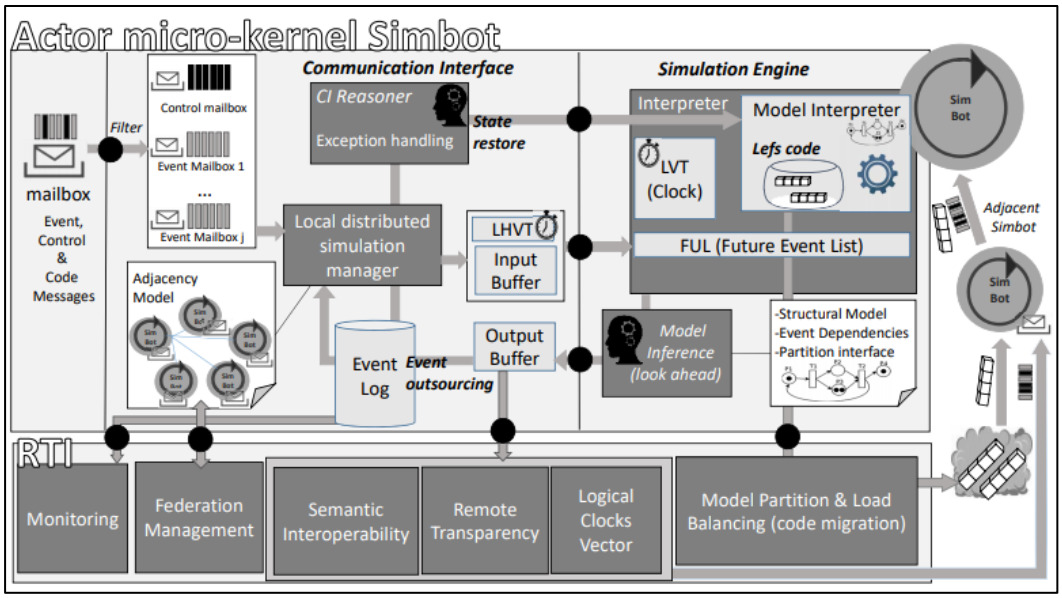


Figura 12: Micro-Kernel o Simbot del simulador Fuente: [6]

Los ***mensajes*** son enviados de manera asíncrona al “buzón” del simbot y estos son obtenidos según el orden de una pila FIFO. Se pueden poner filtros de manera que se puedan insertar o no determinados mensajes. En la figura, se puede apreciar un mailbox por cada simbot adyacente del que recibe mensajes, así como el modelo de adyacencia que le permite conocer y enviar mensajes a sus simbot vecinos. Además, se define un mailbox de control que se utilizará para mensajes de control que pueden consistir en detener la simulación para realizar balanceos de carga, y continuar la simulación una vez finalizada. Los balanceos de carga se pueden realizar eficientemente moviendo la información de los LEFs de un simbot a otro. Los eventos recibidos y emitidos pueden ser almacenados localmente, y se pueden juntar los eventos de todos los simbot para recopilar la información de la simulación global.

La ***interfaz de comunicación*** permite seguir el tiempo de simulación y ordenar los eventos internos y externos según el reloj de simulación que poseen. En el caso de simulaciones optimistas, pueden detectar eventos fuera de tiempo y ejecutar rollbacks.

El **motor de simulación** interpreta el modelo. Dicho interprete puede ser sustituido por diferentes simuladores siempre usando la misma interfaz de comunicación.

En la parte inferior, encontramos servicios del **middleware de simulación**, RTI (Runtime Interface) que permite dar servicios adicionales al simulador como monitorización, vectores lógicos de Lamport, particiones del modelo y balanceo de carga, etc.

El trabajo de este proyecto se ha centrado en el desarrollo del motor de simulación basado en LEFs, pero teniendo en cuenta el marco de todos los componentes que configuran el micro-kernel de simulación.

4.2. Algoritmo de Simulación basado en LEFs

El algoritmo de RdP basados en LEFs compone lo que se denomina el **kernel del simulador**. A continuación, se detallan los servicios incorporados al simulador.

4.2.1. Tiempo de simulación

El tiempo de simulación es el tiempo de computación consumido en la simulación del modelo de un sistema con un escenario de entrada. El recuento de este tiempo de simulación es una medida cuantitativa para evaluar la eficiencia del simulador implementado.

Hay muchas maneras de medir el tiempo de simulación: Se puede medir desde que se ejecuta el programa hasta que finaliza, desde que empieza a realizarse la simulación hasta que acaba, etc.

En este proyecto se ha elegido que el tiempo de simulación se mide desde el punto que empieza a simular el sistema hasta que la maquina principal o receptora cumpla con el tiempo de simulación.

Cabe destacar que no se debe confundir con el **tiempo real.**

4.2.2. Ordenación de eventos

Cada evento se puede caracterizar con tres campos: el tiempo para el que se debe considerar el evento, la transición a la que va dirigida y la constante que se envía.

Cuando se genera un evento este se inserta en la **lista de eventos** en función a un criterio dado. El criterio dado puede ser en función al tiempo del evento, a la transición que va dirigida para darle más importancia, en función de las constantes para priorizar las transiciones que están habilitadas, etc.

Se ha elegido que el criterio de ordenación de los eventos sea en **función del tiempo** ya que es lo más sencillo de implementar y se sigue el orden lógico del reloj de simulación.

4.2.3. Conflictos

Un conflicto en una RdP se puede dar cuando existe un lugar P que posee dos transiciones de salida. Se puede ver un ejemplo en la Figura 13.

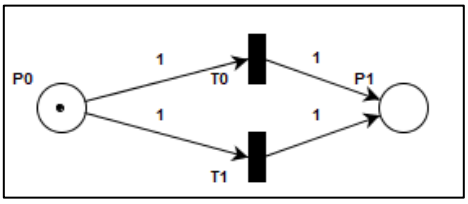


Figura 13: Conflicto en RdP

Como se puede ver el lugar P0 posee dos transiciones de salida T0 y T1. En este caso se puede ver el conflicto ya que al ir a disparar las transiciones que estén sensibilizadas es indeterminado cual va a ser.

Existen diferentes técnicas de **resolución de conflictos**. Se ha decido usar la técnica de disparar las transiciones que menos veces ha sido disparada. Se ha elegido esta porque es realmente sencillo de implementar y porque otras técnicas

pueden llevar a problemas que no se han querido abordar. Por ejemplo, si en un conflicto siempre se dispara la que menor tiempo tenga se puede tener un bucle y tener transiciones que no se disparen nunca.

La implementación es muy sencilla ya que la clase que implementa la transición tiene un campo que almacena las veces que ha sido disparada. De esta manera, cuando se comprueba las transiciones que están sensibilizadas se lanza aquella que menor número de veces se haya disparado.

4.2.4. Nombres globales de las transiciones

Cada subred cuyos lugares son puertos de entrada o salida se traducen en **nombres globales** compartidos entre las diferentes conexiones entre subredes.

Internamente, dichos lugares se codifican con números positivos o negativos. Los números positivos corresponden con las transiciones locales a la subred, y los números negativos corresponden a transiciones de otras subredes. Por tanto, al disparar una transición y generar los eventos se comprueba si el código de la transición es negativo; en caso afirmativo se añade un nuevo evento exterior que será procesado más tarde, en caso negativo se añade un nuevo evento interior.

4.2.5. Generación de logs de ejecución

Ver las trazas de las ejecuciones en la simulación permite saber cómo va progresando el sistema en función del tiempo. En el caso del simulador, una traza que puede ser útil es ver las transiciones que se han disparado y en qué momento ha sido.

Ver las trazas en tiempo real es realmente costoso ya que la entrada/salida en fichero es muy costosa. Por ello se ha decidido ir almacenando las trazas en un vector de manera dinámica y al terminar la simulación se muestran los datos.

En el caso distribuido, existe una maquina llamada **“principal”** que es la encargada de recibir las trazas de cada máquina y ordenarlas según el tiempo de simulación. Finalmente, se muestran todas las trazas de cada máquina.

La implementación consiste en poner una barrera para que al terminar la simulación se sincronicen todas las máquinas. Además, si hay n máquinas y una es principal espera el mensaje de las n – 1 máquinas. Si es una maquina no principal, al terminar envía su traza a la maquina principal.

Para el simulador desarrollado a parte de las transiciones disparas se muestra el tiempo real simulado, el número de transiciones disparadas por segundo y el tiempo de simulación.

A continuación, se muestra una traza devuelta por el simulador:

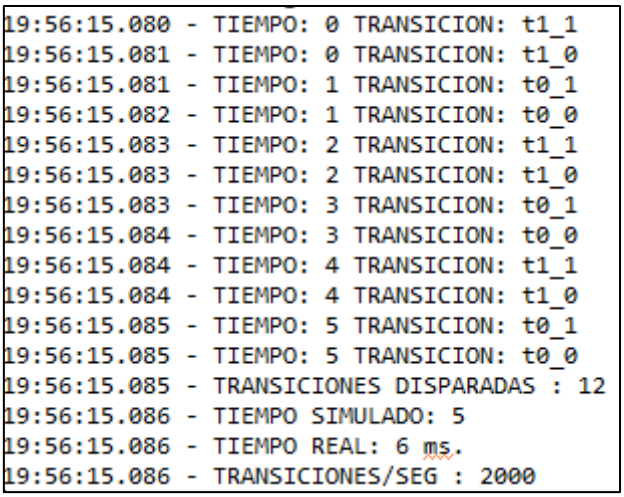


Figura 14: Traza de una simulación

Como se puede ver en la imagen, se detalla en cada tiempo de simulación que transiciones han sido disparadas, así como información sobre cuantas han sido y algunas métricas de eficiencia.

4.2.6. Gestión eventos exteriores

Cuando una subred posee un evento exterior significa que es necesario el envío de constantes a otra subred. La incorporación de eventos exteriores al motor de simulación, tal como se ha explicado anteriormente, se realiza en la fase de disparo de transiciones.

Cuando se genera un evento exterior para un simbot adyacente se envía el evento con su reloj, transición y código a las máquinas **adyacentes,** es decir, aquellas máquinas que están conectadas a través de alguna transición externa. El servidor de nombres, con la información de todas las transiciones, mantiene la información actualizada de la máquina en la que se encuentra el simbot que contiene dicha transición. Dicha información es almacenada en los modelos de adyacencia del simbot para evitar comunicaciones adicionales con el servidor de nombre, y sólo se actualiza con el servidor de nombre en el caso de que el proceso de balanceo de carga supongo una reubicación de la transición a la que se dirige el evento.

La recepción de eventos exteriores se realiza de manera asíncrona asimilando el concepto de un buzón. Por ello se tiene un proceso independiente encargado de recibir eventos y otro para la simulación. El proceso va incorporando los eventos en una lista y al ir a procesar los eventos exteriores se van sacando de la lista como una cola FIFO.

Dichos eventos pueden ser la recepción de una constante para una transición propia en cuyo caso se almacena el evento como si fuera local. En el caso de la simulación conservativa, si una maquina no tiene ningún evento para su reloj local se espera a que le lleguen de todas máquinas para poder avanzar el reloj o que ha terminado.

En el Anexo 11.8 se muestra el protocolo usado para el envío de mensajes y la estructura de los mensajes que se envían entre las diferentes máquinas.

4.2.7. Ejecución de tareas al disparar transiciones

En un sistema real cada transición puede significar ejecutar una determinada tarea. Para simularlo, al disparar cada transición se puede poner cualquier código que simule la ejecución de una tarea. Dicha tarea puede ser la simulación de un acceso a memoria caracterizado por un tiempo dado, mostrar algún mensaje por pantalla, el movimiento de un robot, etc.

La implementación actual es que al disparar una transición se generan los eventos ocurridos por el disparo de una transición, pero se podría añadir la simulación de cualquier tarea en dicho código o incluso dormir al proceso durante un tiempo dado.

4.2.8. Simulación pesimista

Aunque ya se ha comentado anteriormente, la simulación seguida es del tipo pesimista, es decir, no se avanza el reloj hasta que no avisan todas las máquinas que no tienen ningún evento para el tiempo local.

La implementación se basa en el envío de un evento con el tiempo del reloj local de cada máquina a todas las demás. Si hay n máquinas, una maquina puede avanzar el reloj local si recibe n – 1 eventos de que no hay nada para dicho tiempo.

4.2.9. Algoritmo de simulación centralizada

Aunque el propósito del trabajo descrito en esta memoria es la elaboración de un simulador **distribuido**, se toma como referencia la simulación centralizada para comprobar que la simulación distribuida devuelve buenos resultados, y que podemos mejorar las prestaciones de la simulación.

Ya se ha descrito anteriormente en que consiste un simulador de RdP, por lo que únicamente se va a explicar de qué manera se ha realizado un simulador centralizado.

- En el proceso de elaboración, se componen todas las subredes en una única subred de mayor tamaño. El simulador centralizado recibe el resultado del compilador que toma como entrada el fichero de texto plano que compone el sistema a simular mediante RdP. En el caso de la simulación centralizada no es necesario enviar eventos externos, ya que todas las subredes están contenidas en una única sola.
- Una vez se tiene la red en una única sola se convierte a la estructura de los LEFs y se lanza el simulador de igual manera que en distribuido.
- Una vez se finaliza la simulación se muestran los resultados.

Cabe destacar que se ha desarrollado una API que ofrece métodos tanto para el centralizado como para el simulador por lo tanto el código base consiste en el mismo.

En la siguiente figura se puede ver un pseudo-código sobre el simulador centralizado:

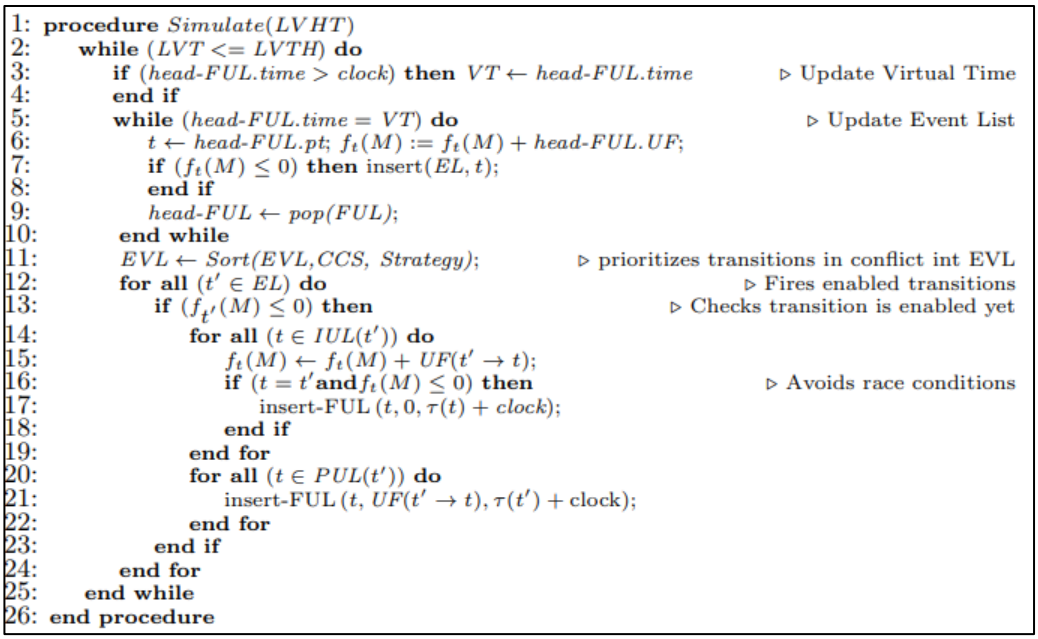


Figura 15: Pseudo-código sobre el simulador centralizado Fuente: [6]

En el código sobre el simulador centralizado se pueden observar

- La línea 2 consiste en la actualización del tiempo de simulación. Si el tiempo del primer evento proyectado es mayor que el tiempo actual, se actualiza.
- En la línea 5 se actualiza la lista de eventos. Se obtienen los eventos cuyo tiempo coincide con el tiempo de simulación actual y se actualizan los valores de las LEFs.
- En la línea 11 se establece el orden de disparo de las transiciones que estén en conflicto según una política dada.
- En la línea 12 se disparan todas las transiciones que están sensibilizadas, es decir, si el valor de su LEF es menor o igual a cero.
- En la línea 14 se actualizan los eventos inmediatos, es decir, aquellos que tienen efecto en el instante de disparo.
- En la línea 20 se actualizan los eventos proyectados, es decir, aquellos que tienen efecto en un tiempo superior al del tiempo de simulación actual del disparo.

En el Anexo 11.4 se entra más en detalle sobre el código del algoritmo centralizado.

Los detalles de implementación del algoritmo realizado se encuentran en el Anexo 11.5.

4.3. Simulación distribuida

La diferencia del algoritmo distribuido frente al centralizado radica principalmente en 3 aspectos: (1) La lista de eventos contiene un nuevo tipo de evento, el evento exterior, que corresponde al envío de un evento a una transición que no está en la red que se está simulando. Cuando se dispara una transición cuyos lugares de salida son lugares de entrada de una transición externa, se inserta en la lista de eventos exteriores para que, al tratarlos, se envíe el evento por la red. En este caso existen 3 tipos de eventos (inmediatos, futuros y exteriores) que se han

implementado con tres listas diferentes; (2) Cuando no se tienen eventos para el tiempo de simulación actual, ni transiciones sensibilizadas ni eventos exteriores se envía un mensaje a las demás máquinas con el valor del **lookahead**. El valor de lookahead es un valor temporal que indica a las otras redes el tiempo mínimo que pueden avanzar sus relojes porque no se va a generar ningún evento exterior desde la máquina que emite el lookahead; (3) El avance del tiempo que se puede realizar es el mínimo valor de tiempo de todos los mensajes de lookahead que han llegado de las demás redes y de los eventos propios de la red.

Una de las ventajas de utilizar el modelo de redes de Petri temporizado es que podemos calcular el lookahead a partir del modelo estructural de la red, y de la definición de las interfaces de la componente que contiene la red. Esta información se puede calcular en tiempo de compilación. Aspectos como el cálculo del lookahead cuando hay balanceo de carga no han sido considerados en este trabajo. Información más detallada sobre el cálculo del lookahead puede encontrarse en la sección 4.3.3.

Al simulador distribuido se le han añadido una serie de características que permiten su correcto funcionamiento. A continuación, se detallan las características más importantes implementadas.

4.3.1. Asignación de máquinas

Cuando el simulador recibe el número de subredes a simular y el número de máquinas virtuales que se encuentran disponibles se debe realizar una asignación de las subredes a cada máquina.

Debido al alcance de este Trabajo Fin de Grado, se ha impuesto una limitación para el entorno de ejecución donde cada máquina física posee un **único Simbot,** así como una repartición de trabajo sencilla. Según el número de cada parámetro se pueden ver tres casos diferentes:

- El número de subredes a simular es igual al número de máquinas virtuales disponibles. Por tanto, el simulador asignará a cada máquina una única subred.
- El número de subredes es mayor al número de máquinas virtuales. En este caso se debe unir algunas subredes hasta tener el mismo número de máquinas. Dicha agrupación puede ser equitativa asignando el mismo número de subredes a cada máquina o puede seguir alguna optimización, por ejemplo, asignando a las máquinas con procesadores más veloces y más memoria RAM, mayor número de subredes en caso de que las maquinas no sean homogéneas. La opción más sencilla y la que se ha seguido por el momento en este trabajo es dividir de manera equitativa las subredes entre todas las máquinas disponibles y si la división no es entera, se asigna el resto a la máquina considerada como principal.
- El número de subredes es menor al número de máquinas. En este caso se irá asignando una subred a cada máquina hasta que no se disponga de ninguna y por tanto las siguientes máquinas no realizarán nada.

En la siguiente imagen se muestra la idea de las 3 asignaciones diferentes que se pueden dar:

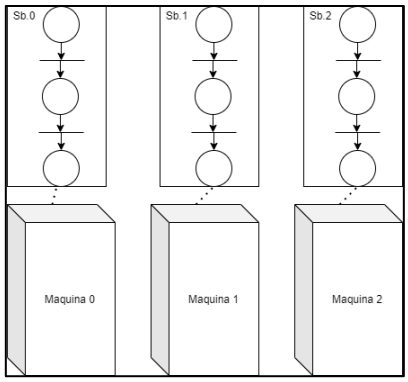


Figura 16: Asignación de 3 subredes a 3 máquinas

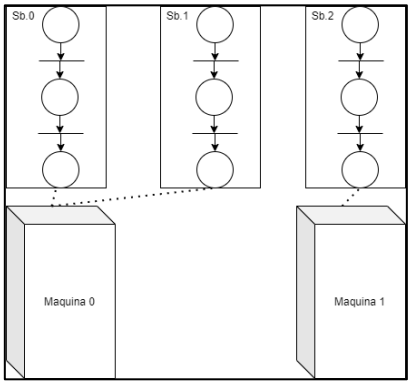


Figura 17: Asignación de 3 subredes a 2 máquinas

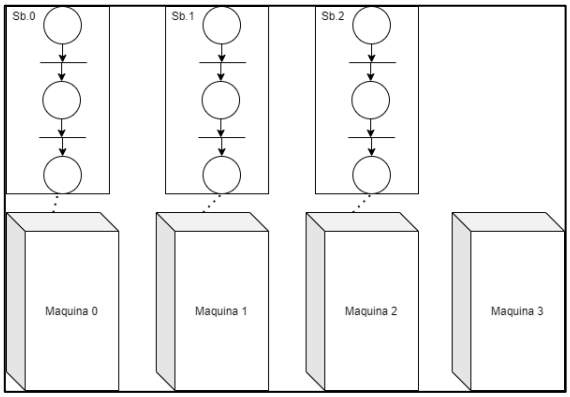


Figura 18: Asignación de 3 subredes a 4 máquinas

Una vez que cada máquina sabe cuántas y cuales subredes debe simular se empieza el proceso de simulación.

4.3.2. Iniciación y finalización sincronizada

El inicio y finalización de un algoritmo distribuido es un problema ya conocido. De todas las aproximaciones que resuelven este problema se ha decidido usar la más sencilla a priori.

Se debe asignar una máquina como **máquina principal.** Dicha máquina es elegida por el usuario estableciendo dicha máquina en primer lugar en el listado de máquinas. Su propósito es la de comprobar el estado, sincronizar y recoger todas

las trazas generadas por todas las máquinas involucradas en el proceso de simulación.

Para la inicialización, la maquina designada como principal espera a que todas las maquinas estén operativas y esperando a recibir el mensaje de barrera, algo parecido al concepto "*¿areYouAlive*?".

Cuando la maquina es conocedora de que todas las demás máquinas están operativas, les manda un mensaje para iniciar el proceso de barrera. Al recibir cada máquina el mensaje, se lo devuelve para que empiecen todos a la vez.

En la siguiente imagen se puede ver un ejemplo del proceso de barrera seguido por todas las máquinas para sincronizarse.

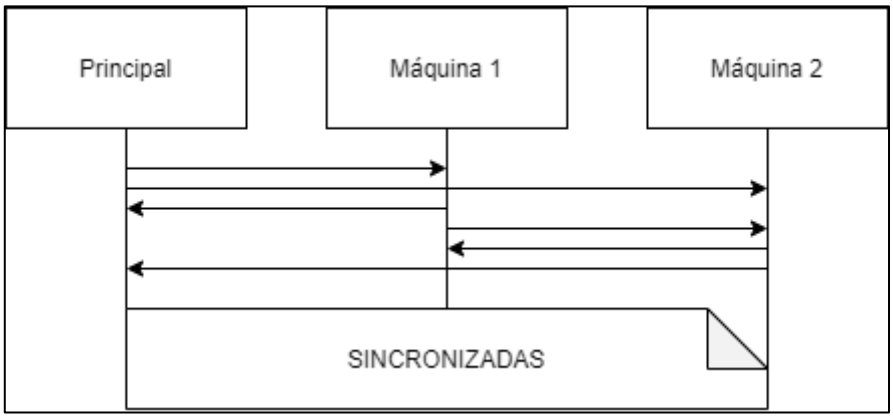


Figura 19: Proceso de barrera de 3 máquinas

Una vez concluida esta fase se puede decir que las máquinas están listas y sincronizadas para empezar a simular.

En la fase de finalización se realiza una operación similar a la explicada. La máquina principal espera que le lleguen los mensajes de finalización de las n – 1 máquinas y cuando le llegan les avisa a todas de que pueden continuar sus operaciones.

4.3.3. Cálculo del lookahead

El lookahead es una variable que se envía entre los diferentes Simbot que permite avisarles hasta qué tiempo pueden avanzar su reloj ya que no van a recibir ningún evento.

Siguiendo la simulación pesimista, cuando una máquina no tenga ningún evento exterior que enviar para un tiempo dado, enviará un mensaje a las demás máquinas para notificarles hasta que tiempo pueden avanzar ya que no les va a enviar ningún evento hasta dicho tiempo. Las máquinas al recibir el mensaje avanzaran el tiempo mínimo entre todos los lookahead recibidos por todas las máquinas adyacentes y el tiempo de su primer evento.

De esta manera se está seguro de que el avance del tiempo no va a ser optimista y no habrá que retroceder.

La implementación para el cálculo del lookahead empieza en la fase de compilación:

- Para todas las transiciones de salida se almacena el tiempo que va a costar llegar una marca desde cada transición.
- Para obtener el valor del lookahead en cada momento de la simulación, se recorren todas las transiciones que están sensibilizadas y devuelve el menor de los tiempos calculados. En caso de que no haya ninguna sensibilizada, se envía el menor de todas las transiciones que componen la red.
- Una vez se tiene el valor del lookahead, se envía a las maquinas siguientes el valor del reloj actual más el valor obtenido.

A continuación, se muestra una traza sencilla para ver el cálculo realizado:

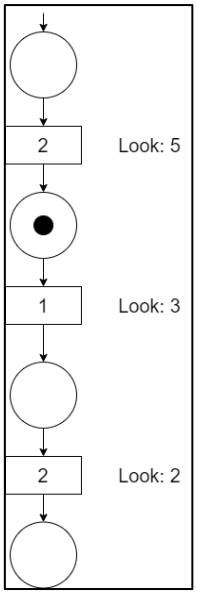


Figura 20: Ejemplo de cálculo de lookahead

Como se puede ver en la Figura 22, cada transición tiene una etiqueta con el valor que le costaría llegar una marca desde que se dispara cada transición hasta la salida. Por eso la primera transición tiene un valor 5 (2 de su transición + 1 de la siguiente transición + 2 de la última transición).

Por tanto, en el caso de que no hubiera ningún evento exterior que mandar, el Simbot encargado de esta subred le enviaría a sus siguientes el mínimo entre los valores de todas las transiciones sensibilizadas, siendo en este ejemplo el valor 3. Eso significa que esta subred está segura de que no va a enviar ningún evento exterior hasta dentro de 3 unidades temporales.

En el caso de que no hubiera ningún evento exterior, ni ninguna transición sensibilizada se toma la metodología conservativa y se envía el mínimo tiempo de todas las transiciones de salida. De esta manera, se establece una seguridad de que no se vaya a avanzar el reloj más allá del próximo evento exterior.

4.3.4. Consideraciones

Existen una serie de recomendaciones que pueden llevar a una mejor optimización de la simulación distribuida

- La simulación distribuida aprovecha en gran medida el paralelismo de las redes que componen el sistema. Esto implica que la mejor manera de **partir la red** es agrupar las partes secuenciales en la misma máquina (evidentemente, sin sobrecargar en gran medida una única máquina).
- El cuello de botella de la simulación distribuida son las comunicaciones. Por tanto, lo mejor es agrupar las subredes que más comunicaciones tengan en una misma máquina para así evitar la saturación de mensajes circulando por la red de interconexión circunscribiéndoles a circulación de eventos entre colas internas del simbot.
- Es evidente que, en cuanto a rendimiento computacional de las máquinas, es mejor asignar las subredes con mayor número de transiciones y lugares a las máquinas más potentes (que posean procesadores con más núcleos, más memoria RAM, mayor frecuencia de reloj…etc.) Todas estas consideraciones se pueden tener en cuenta a nivel de diseño del sistema, agrupando las diferentes subredes en redes más grandes que luego se asignen a una sola máquina. Por otra parte, la asignación **no equitativa** sería una idea de trabajo futuro sobre el framework, comprobando en la fase de compilación que redes son más densas y así poderlas asignar de manera óptima [17].

4.3.5. Ejemplo de ejecución distribuida

En la Figura 21 se muestra una ejecución de una simulación distribuida.

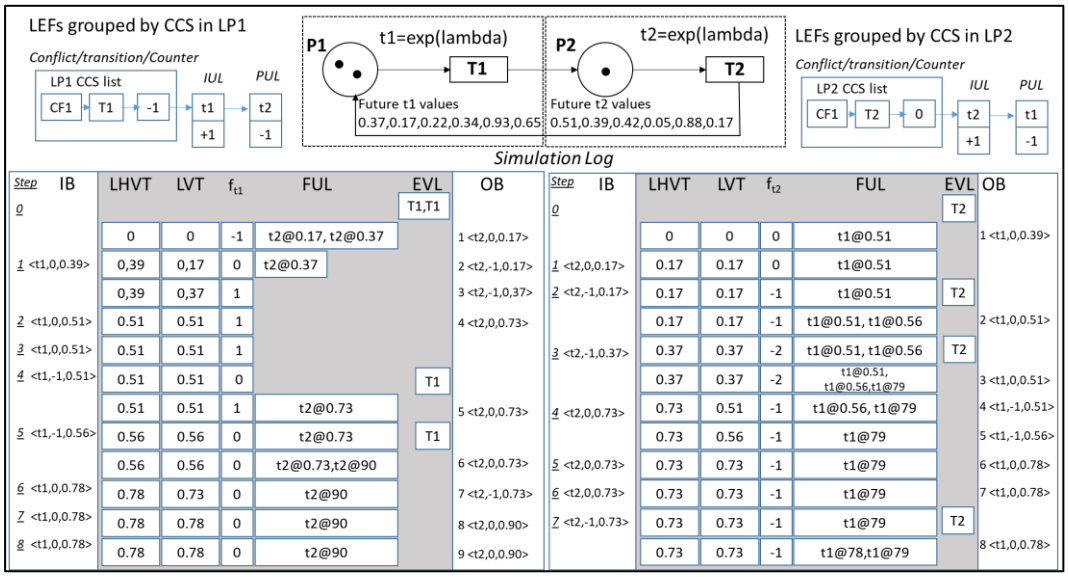


Figura 21: Ejemplo de ejecución distribuida. Fig. 3 de [6]

Donde LVT (*Local Virtual Time*) es el tiempo local del simulador, LHVT (Local Horizon Virtual Time) es el mínimo tiempo de los recibidos por las demás máquinas adyacentes, EVL (*Event List*) es la lista de eventos, IB (*Input Buffer*) es el buffer de entrada donde llegan los eventos externos, OB (*Output Buffer*) es el buffer de salida donde se inserta los eventos externos.

En la parte superior de la Figura 21 se presenta una sencilla red de Petri que se ha partido en dos partes. Se presentan los valores en los que se dispararán las transiciones `T1` y `T2` que siguen una distribución exponencial exp(λ) con λ=0.5. A la izquierda se muestra la codificación LEF de la transición `T1`, y a la derecha la de la transición `T2`.

En el primer paso, podemos observar en la partición izquierda que al tener dos marcas el lugar P1, el valor de la función de la LEF de T1 es -1. Al ser un marcado menor o igual a cero, La T1 aparece dos veces en la EVL y se inserta el disparo de T1 en la FUL en los tiempo 0.17 y 0,37 (designado en la figura como t2@0.17 y t2@0.37 en la FUL).

El cálculo del *lookahead* que una partición envía a sus simbot adyacentes se realiza teniendo en cuenta que, dado el marcado inicial, la T1 y T2 se podría disparar como máximo tres veces de forma concurrente. En este caso, se calcula el mínimo tiempo entre los tres valores de tiempo que podemos obtener del *time stamp* (marcas de tiempo) en la FUL y los valores previamente generados de la distribución exp(0.5) que siguen la transición T1 y T2 y a los que les sumaremos el LVT. En este paso inicial, el mínimo de los valores (0.17, 0.22+0 y 0.37). Como el mínimo corresponde a un evento externo en la FUL que corresponde a la transición T2 que no pertenece a la partición, se coloca en el buffer de salida el evento <t2,0,0.17> que se enviará a la partición de la derecha y que aparece su buffer de entrada (*IB*) en el paso uno de su ciclo de simulación. Se debe observar, que en el mensaje enviado el segundo valor de la tupla corresponde con el valor del UF(*updating factor*) para la transición T2. En este caso es cero, porque corresponde a un mensaje null, que de acuerdo a la técnica de simulación conservativa permite comunicar que no ha habido ningún evento en la partición de la izquierda, pero que con el *lookahead* calculado la partición de la derecha puede avanzar su *LVT* hasta 0.17. En la partición izquierda no podemos avanzar el *LVT* hasta que no recibamos algún mensaje de la partición derecha.

De la misma manera, en la partición derecha podemos observar que al tener una marca inicial en el lugar P2, el valor LEF de la transición T2 es cero, por lo que aparece una vez en la *EVL* y da lugar a que aparezca en la *FUL* un evento t1@0.51. En el cálculo del lookahead obtenemos que el próximo evento a enviar a la partición izquierda es el mínimo de (0.51+0, 0.39+0, 0,22+0). Por lo tanto, el lookahead es 0.39, y como es anterior al evento que se encuentra en la FUL t1@0.51, de acuerdo a la técnica conservativa enviamos un evento null con el evento <t1, 0, 0.39>, indicando esta partición que puede avanzar hasta el tiempo 0.39.

En el primer paso de la partición izquierda, se recibe en el buffer de entrada el mensaje <t1,0,0.39>, por lo que es posible avanzar el reloj hasta el tiempo hasta el *LHT* =0.39. Se saca del *FUL* el evento t2@0.17, el *LVT* pasa a ser 0.17 y se envía a la partición derecha el mensaje <t2,-1,0.17> que ahora contiene un valor de UF=-1 para la transición T2. A continuación se envía en el mismo paso el evento <t2,-1,0.37> avanzándose el reloj al *LVT* =0.37. El resto de pasos se puede seguir con la traza presentada.

4.3.6. Despliegue manual

Durante todo el desarrollo del trabajo, el despliegue del simulador distribuido se ha realizado de manera manual a través de *ssh*. Se ha elegido de manera manual por su sencillez y como toma de contacto para comprobar la viabilidad del sistema.

Para desplegar y realizar una simulación distribuida entre diferentes máquinas se debe poseer en todas las máquinas: (1) El propio simulador; (2) El mismo fichero que contiene la lista de máquinas disponibles; (3) El mismo fichero de texto que modela el sistema a simular.

El despliegue se ha realizado con la herramienta *Terminator* que permite lanzar múltiples comandos ssh en varias terminales a la vez. De esta manera, con todos

los requisitos anteriormente descritos en todas las máquinas disponibles, se puede lanzar el simulador a la vez gracias a dicha herramienta.

4.4. Optimizaciones

4.4.1. Paso de mensajes de eventos exteriores

Inicialmente el envío de los eventos exteriores se hacía a todas las máquinas que se encontraban en el proceso de simulación. Esto conllevaba a una gran sobrecarga de la red ya que no todas las subredes están conectadas con todas.

Se optimizó dicho proceso obteniendo el número de redes a la que el disparo de cualquier transición propia conllevaba el envío de una constante a una subred externa. Por tanto, inicialmente se calculaba el número de redes “siguientes”, es decir, las redes que se verían afectadas por el disparo de una transición.

Al generar un evento exterior, se enviaba únicamente a todas las subredes siguientes de la subred cuya transición ha sido disparada.

Con esto se mejora el rendimiento ya que el número de mensajes se reduce. Obviamente si la red está fuertemente acoplada y todas las subredes están conectadas entre sí unas con otras el rendimiento será igual.

Como una posible mejora de cara al trabajo futuro, se podría enviar el evento únicamente a la subred siguiente cuya transición afectada fuera de su propiedad.

4.4.2. Desacoplo de escritura por pantalla

La entrada y salida por pantalla es un cuello de botella que conlleva que el simulador no sea tan eficiente como se desea.

Al ir disparando y generando trazas se pueden ir mostrando por pantalla para que el usuario pueda ver cómo va evolucionando el sistema.

En caso de que no interese ver el estado real si no una vez terminado, se puede establecer un flag para que no muestre nada. Este caso es por tanto el más eficiente de todos.

En caso de querer verlo, se ha realizado una optimización con un **patrón observer** de manera que el simulador se dedica únicamente a simular y a sus observadores les notifica cuando ha habido algún cambio.

Más detalles se encuentran en el apartado de Patrones 5.2.

5. ARQUITECTURA DEL ENTORNO DE SIMULACIÓN

A continuación, se va a comentar la arquitectura del entorno de simulación, así como los patrones de programación realizados. En la sección 5.1, se comenta la arquitectura que compone el simulador, comprendiendo desde la arquitectura de red utilizada hasta un servidor de nombres. Finalmente, en la sección 5.2 se encuentra los patrones *Software* usados en el simulador.

5.1. Arquitectura del entorno

5.1.1. Modelo de actores con comunicación asíncrona

El modelo de actores que se ha usado para el desarrollo está basado en el paso de mensajes de manera asíncrona.

Consiste en un modelo en base a eventos que permite escalar y elimina la complejidad de los sistemas de mensajes por bloqueo.

Consiste en un buzón que va almacenando los mensajes que van llegando y se van obteniendo en función a una política dada, como por ejemplo FIFO (First Input First Output)

Las ventajas que ofrecen son la sencillez para la realización de los modelos, así como la eficiencia que se gana al no tener un proceso bloqueante a la espera de mensajes.

5.1.2. Servidor de nombres

Cuando se posee un sistema muy complejo, con una gran cantidad de componentes jerárquicos, obtener los nombres globales de las transiciones y lugares es una tarea con gran relevancia.

Para ello se ha realizado un servidor de nombres que proporcione servicios para atender esta demanda, así como la flexibilidad para poder añadir otros servicios.

Frente a todas las posibles tareas que puede realizar un servidor de nombres se han implementado dos:

- Obtención en el proceso de compilación de la ***localización de la máquina*** donde se encuentran **las transiciones** de aquellas subredes adyacentes a las que se les debe enviar algún mensaje con el disparo de alguna transición.
- **Traducción de los nombres de las transiciones** que se han disparado a lo largo de la simulación en los logs de ejecución, y así obtener los nombres globales que son realmente los que proporcionan la utilidad al usuario. Los nombres globales se obtienen a partir de las construcciones jerárquicas de las componentes, de forma que a partir del nombre es posible identificar la transición de cada subcomponente.

Para más detalles sobre la implementación del servidor de nombres acudir al Anexo 11.9.

5.1.3. Diagramas

En el Anexo 11.10 se puede ver el diagrama de paquetes con la explicación de lo que representa cada uno, y en el Anexo 11.11 el diagrama de clases de cada paquete.

5.2. Patrones

5.2.1. Patrón observer

A la vez que se va simulando se van generando trazas para saber que transiciones han sido disparadas y en qué tiempo. Si se desea mostrar esas trazas a la vez que se va simulando la eficiencia de esta decrece ya que la entrada/salida es un cuello de botella.

Por eso se ha decidido desacoplar la simulación de la muestra de las trazas usando un patrón observer.

La idea es que la entidad encargada de mostrar las trazas sea independiente y se defina como observadora del creador, en este caso el simulador.

Cuando el simulador dispare una transición, notificará a su observador para que actualice su estado. El observador por su parte al ser notificado mostrará el evento por pantalla para mostrárselo al usuario.

Para que la eficiencia no decrezca se ha implementado el patrón observador usando hilos. De manera que el sujeto que produce los eventos esté localizado en un hilo y el observador en otro hilo diferente.

Sin embargo, si no se ve necesario el tener que ver los disparos en tiempo real, se puede establecer un flag para que muestre los resultados al final y ser lo más óptimo posible.

6. ESTUDIO EXPERIMENTAL

A continuación, se detallan las metodologías usadas con relación a la experimentación realizada, los equipos usados y las pruebas realizadas. En la sección 6.1 se detalla la metodología usada y la descripción de los equipos usados. Finalmente, en la sección 6.2 se detalla los experimentos realizados y se comentan los resultados obtenidos.

6.1. Metodología y entorno experimental

Para el estudio experimental se han realizado pruebas con los equipos del laboratorio 1.02 de la **Escuela de Ingeniería y Arquitectura de Zaragoza** con las siguientes características:

- Sistema Operativo: CentOS Linux x64
- Procesador: Intel(R) Core (TM) i5-4570 CPU @ 3.20GHz
- Memoria RAM: 12 GB

Las pruebas con el simulador centralizado se han realizado sobre un solo equipo de los anteriormente expuestos, mientras que con las pruebas del simulador distribuido se han usado varios ordenadores de los mismos ordenadores.

Para los tipos de pruebas se han realizado 2 diferentes versiones para comprobar el correcto funcionamiento de ambos simuladores y para corroborar la eficiencia de estos.

Cabe destacar que se han ido realizando pruebas a lo largo de todo el desarrollo del código de ambos simuladores, no obstante, solo se incluyen en esta memoria las 2 pruebas principales que se han realizado para demostrar la viabilidad y optimización conseguida en cuanto a la comparación entre los simuladores se refiere.

6.2. Experimentos y Resultados

Experimento 1: Comparación con otro simulador centralizado

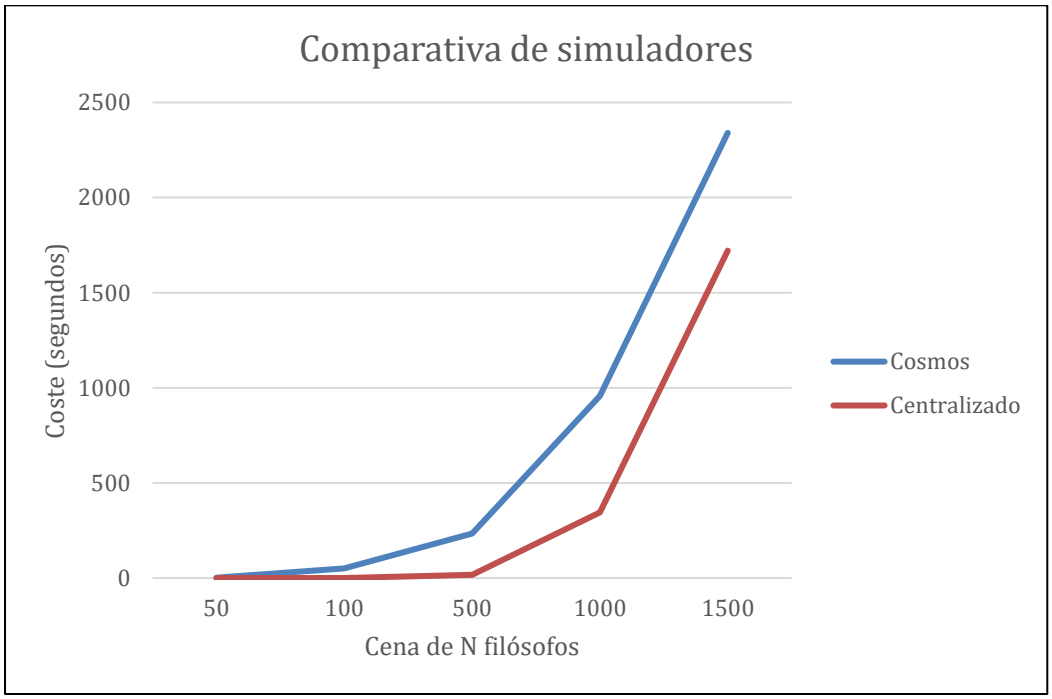
En este experimento se ha tratado de ver que el simulador centralizado basado en LEFS posee unas características igual o mejor en cuanto a eficiencia que algún simulador comercial o disponible en el ámbito de académico.

Para el experimento, se ha elegido el simulador ***Cosmos***, que permite la simulación de RdP Lugar/Transición a partir de modelos definidos en un lenguaje con *XML.* Debido a que el lenguaje de descripción de *Cosmos* y el desarrollado para el simulador no coinciden, se ha tenido que realizar una conversión entre el lenguaje definido y el lenguaje admitido por el simulador foráneo.

El sistema a simular se basa en ***el problema de la cena de los Filósofos*** [2] por la facilidad de escalar el modelo aumentando el número de filósofos***.*** En el Anexo 11.14 se explica en que consiste dicho problema.

Para el experimento, se han realizado simulaciones variando el número de filósofos empezando por 50 filósofos hasta llegar a N filósofos.

Los resultados obtenidos se pueden ver en la siguiente gráfica.



Como se puede apreciar en la gráfica, inicialmente con 50 filósofos los dos simuladores devuelven el mismo coste. Esto es debido a que el número de eventos generados no es de gran tamaño y por tanto la similitud entre ambos simuladores es grande.

A medida que va aumentando el número de filósofos la diferencia se va haciendo más notoria llegando a su máximo en la cena de 1000 filósofos. Esta diferencia viene marcada por la eficiencia de los LEFS en cuanto a simulación se refiere. Esto

es debido a que la comprobación de la sensibilización de una transición se realiza comprobando el signo de un entero.

A medida que se sigue aumentando el número de filósofos, la diferencia se estabiliza.

Por tanto, podemos concluir que la prueba de concepto del simulador centralizado devuelve unos resultados similares o incluso mejores que los comerciales y por tanto que tiene unas prestaciones similares.

Además, para concluir con este experimento, se ha usado una herramienta de ***profiling*** que permite ver que funciones son las que más tiempo de CPU ocupan y así saber que funciones se pueden optimizar y así obtener unos mejores resultados.

En el Anexo 11.15 se detallan las funciones más costosas de manera gráfica.

Experimento 2: Incorporación de máquinas

En el siguiente experimento se ha tratado de ver como la explotación del paralelismo frente a la opción secuencial arroja mejores resultados. Para ello, cada transición disparada conlleva un tiempo de cálculo de CPU.

Para simular ese tiempo de cálculo de CPU se ha procedido a dormir el proceso durante 50 milisegundos al disparar la transición, y así emular que el disparo de una transición conlleva una tarea que dura 50 ms.

La red de Petri usada se puede ver a continuación:

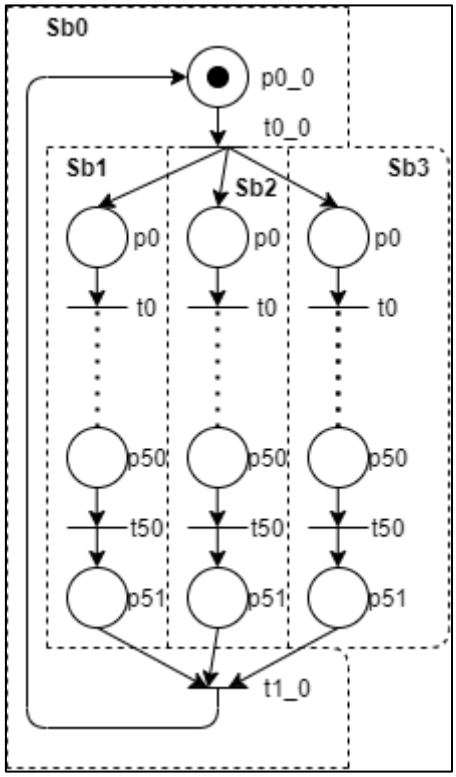


Figura 22: Red de Petri del experimento 2

Para el experimento, se han realizado 3 diferentes configuraciones:

- Simulador centralizado
- Simulador distribuido con 2 máquinas
- Simulador distribuido con 3 máquinas
- Simulador distribuido con 4 máquinas

Los resultados obtenidos se pueden ver en la siguiente gráfica:

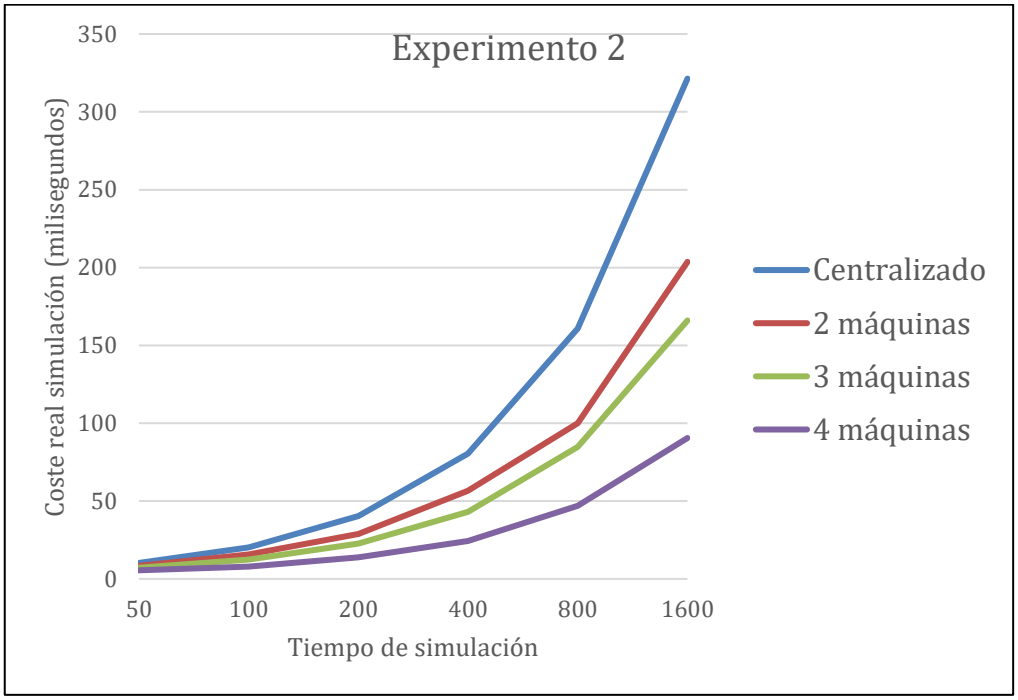


Figura 23: Resultados obtenidos en el experimento 2

Como se puede observar en la gráfica, el simulador centralizado es el que peores resultados arroja. Esto es debido a que no se explota el paralelismo y por tanto cada transición horizontal se procesa de manera secuencial, y los tiempos de cálculo se suman.

La solución con el simulador distribuido y 2 máquinas arroja mejores resultados que el centralizado, pero se encuentra aún lejos de las otras soluciones. Esto es debido a que se explota el paralelismo 2 a 2, es decir, cada máquina simula a la vez 2 subredes y por tanto se suman los tiempos de cálculo de las 2 subredes.

El coste se va viendo cómo va bajando a medida que se va aumentando el número de máquinas obteniendo el mínimo coste con 4 máquinas. Esto se debe a que cada máquina, está simulando una única red y por tanto es más eficiente.

Viendo los resultados se puede afirmar que en redes que sean potencialmente paralelas, distribuirlas sobre más de una máquina ofrece una mejora sustancial; incluso añadiendo únicamente una sola máquina más, el resultado que se obtiene es altamente beneficioso.

7. CONCLUSIONES

Los sistemas de eventos discretos permiten modelar muchos de los sistemas de hoy en día.

El simulador desarrollado va a permitir realizar la simulación de los sistemas de eventos discretos complejos de gran tamaño en tiempos relativamente cortos y permitiendo escalar el tamaño.

No obstante, el principal factor que ha llevado a la realización de este simulador ha sido ver la inexistencia de simuladores distribuidos que escalen de una manera correcta.

Viendo los resultados obtenidos, se puede concluir con que el simulador centralizado es una buena solución para modelos sencillos que no tienen una gran cantidad de transiciones. Por otro lado, el simulador distribuido es una muy buena solución para aquellos modelos **complejos** que tengan una gran cantidad de transiciones.

Obviamente, las prestaciones de un simulador y otro son muy dependientes del modelo y ya no solo hablando del número de transiciones, si no de cuan acopladas están las subredes entre ellas. No es lo mismo una red con 3 subredes que solo tienen una salida y entrada donde la comunicación es mínima, que el mismo caso pero con 100 salidas y entradas por ejemplo, donde la comunicación es exhaustiva y muy dependiente de la congestión de la red.

Así mismo, se han usado herramientas de *Profiling* para comprobar las funciones que más tiempo de CPU requieren y así poder buscar, de cara al trabajo futuro, estructuras más optimas en cuanto a tiempo se refiere.

8. TRABAJO FUTURO

El Framework desarrollado es una primera versión, usada para comprobar la viabilidad del sistema y para ver si la escalabilidad que se promete es cierta. Partiendo de las premisas de los artículos [5] [6] [7], se ha desarrollado el framework con las características más sencillas para corroborar y comprobar los resultados prometidos.

Partiendo del framework, hay una gran cantidad de trabajo que se puede ir incorporando para hacer del simulador una solución completa y muy robusta.

Mirando el futuro del framework, aquí se muestran algunas de las incorporaciones que se pueden hacer al simulador:

- Incorporar una estructura de datos más óptima en la cola de eventos que mejore el rendimiento del proceso de incorporación y obtención de eventos.
- Incorporar un servicio de tolerancia a fallos, para proteger al sistema de los problemas que pueda haber en la red.
- Incorporar otros modelos de red que sean más óptimos que los sockets TCP, como el modelo de actor. Para ello ya existen tecnologías que se podrían incorporar de manera relativamente sencilla como Aka de Java.

- Mejorar el reparto de subredes entre las diferentes máquinas de manera que sea lo óptimo posible.
- Realizar un balanceador de carga que reparta el trabajo cuando una máquina este más saturada que otra.

Aquí aparece una mínima parte de las incorporaciones que se pueden añadir, pero es un trabajo a largo plazo donde se puede llegar a tener un simulador increíblemente potente y con la premisa de que de momento no existe ningún simulador que sea tan eficiente ni escale de esta manera.

9. VALORACIÓN PERSONAL

Además de usar y ampliar los conocimientos adquiridos durante el grado sobre Redes de Petri y Sistemas distribuidos, el Trabajo Final de Grado me ha permitido acercarme al estilo de trabajo de los investigadores que han ayudado y colaborado conmigo.

El simulador ha sido toda una superación personal, pero ha sido realmente gratificante ver como los resultados prometidos a lo largo de los artículos se iban consiguiendo.

Como valoración personal, puedo afirmar que ha sido un proyecto entretenido, aunque duro en algunos momentos. Aun así, no me arrepiento de haber elegido dicho proyecto ya que veo que tiene mucho futuro y puede ser de gran utilidad.

10. BIBLIOGRAFÍA

[1] "http://cosmos.lacl.fr/index.html," [Online].

[2] "https://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_fil%C3%B3sofos," [Online].

[3] Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Discrete-event_simulation#Common_uses.

[4] "Omnet++," [Online]. Available: https://omnetpp.org/.

[5] J. Á. Bañares and J. M. Colom, "Model and Simulation Engines for Distributed Simulation of Discrete Event Systems," *In GECON 2018 - International Conference on the Economics of Grids, Clouds, Systems, and Services,* vol. 11113 of Lecture Notes in Computer Science, pp. 77-91, 2018.

[6] U. Arronategui, J. Á. Bañares and J. M. Colom, "Towards an Architecture Proposal for Federation of Distributed DES Simulators," *In: Djemame K., Altmann J., Bañares J., Agmon Ben-Yehuda O., Naldi M. (eds) Economics of Grids, Clouds, Systems, and Services. GECON 2019 Lecture Notes in Computer Science,* vol. 11819, 2019.

[7] U. Arronategui, J. Á. Bañares and J. M. Colom, A MDE approach for Modelling and Distributed Simulation of Health Systems, 2020.

[8] "JetBrains," [Online]. Available: https://www.jetbrains.com/es-es/idea/.

[9] "Eclipse," [Online]. Available: https://www.eclipse.org/.

[10] "Sublime," [Online]. Available: https://www.sublimetext.com/.

[11] "Redis," [Online]. Available: https://redis.io/.

[12] "Log4j," [Online]. Available: https://logging.apache.org/log4j/2.x/.

[13] B. Gregg, "http://www.brendangregg.com/flamegraphs.html," [Online]. Available: http://www.brendangregg.com/flamegraphs.html.

[14] M. Á. Barcelona Liedana and L. Caballero Fernandez, "Simulación distribuida de sistemas de eventos discretos basada en agentes móviles," Proyecto Fin de Carrera de Ingeniería Informática, Zaragoza.

[15] R. M. Fujimoto, Parallel and Distribution Simulation Systems, USA: John Wiley & Sons, Inc., 1999.

[16] J. M. Colom and J. L. Briz, "Implementation of weighted place/transition nets based on Linear Enabling Functions," *Valette R. (eds) Application and Theory of Petri Nets 1994. ICATPN 1994. Lecture Notes in Computer Science,* vol. 815.

[17] X. Hu and B. P. Zeigler, "The Architecture of GenDevs: Distributed Simulation in DEVSJAVA.".

[18] B. P. Zeigler and A. Muzy, "From Discrete Event Simulation to Discrete Event Specified Systems (DEVS)," *20th IFAC World Congress,* vol. 50, pp. 3039-3044, 2017.

[19] T. C. K. Chou and J. A. Abraham, "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering,* Vols. SE-8, no. 4, pp. 401-412, 1982.

[20] J. L. Peterson, "http://jklp.org/profession/books/pn/3.html," [Online].

[21] "https://www.upwork.com/resources/java-vs-c-which-language-is-right-for-your-software-project," [Online].

11. ANEXOS

11.1. Planificación

ACTIVIDAD	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5	Semana 6	Semana 7	Semana 8	Semana 9	Semana 10	Semana 11	Semana 12	Semana 13	Semana 14	Semana 15	Semana 16	Semana 17	Semana 18	Semana 19	Semana 20
TFG (Completo)																				
Estudio del artículo																				
Estudio del código inicial																				
Implementación de jerarquía y composición																				
Implementación fase elaboración/compilación																				
Implementación del servidor de nombres																				
Implementación del generador de trazas																				
Diseño de arquitectura distribuida																				
Diseño de API con sockets																				
Corrección de errores																				
Estudio experimental																				
Memoria																				

11.2. Problemas durante el desarrollo

Durante el desarrollo del trabajo de fin de grado realizado, ha habido una serie de problemas acontecidos:

- **Tamaño de máquina virtual *JAVA*:** Por defecto, la máquina virtual de Java reserva una cantidad de memoria muy limitada para el simulador. En algunas simulaciones con una gran carga de redes se ha tenido que cambiar el tamaño en las ejecuciones.
- **Disponibilidad de puertos**: Inicialmente se había propuesto que los sockets se abrieran y cerraran cada vez que se quisiera enviar un mensaje a una máquina. El problema está en que el SO de los ordenadores no cierra inmediatamente el socket cuando recibe la directa de cerrar, si no que espera un tiempo determinado por si acaso llega un mensaje perdido. Esto conlleva un problema y es que, si hay una gran cantidad de mensajes que enviar, la lista de puertos disponibles se va agotando y por tanto no se pueden enviar los mensajes. La solución afrontada fue usar las directivas del SO para que cierre los puertos inmediatamente al mandar la directiva de cerrar.
- **Congestión de la red**: En las pruebas con el simulador distribuido, si el paso de mensajes es muy exhaustivo la red se puede ir congestionando. Además, al ser ordenadores compartidos por toda la universidad, la red puede estar bastante colapsada y por tanto puede haber pérdida de paquetes o latencias grandes que degradan el funcionamiento del simulador.
- **Problemas con "Are you Alive?":** Comprobar si las máquinas están vivas y preparadas para simular requiere enviar un mensaje y esperar respuesta. La idea que he ha llevado a cabo ha sido realizar n mensajes hasta esperar la respuesta. Si se envían los n mensajes y no se recibe respuesta se asume que la maquina no está viva y por tanto no se puede realizar la simulación. Pero esto no tiene porqué ser así, ya que puede ser que la máquina haya tardado más en llegar a ese punto. Este problema se ha solucionado incrementando el número de mensajes y añadiendo un tiempo de espera entre envío y envío.
- **Compartición de los ordenadores**: Como son ordenadores compartidos con todos los alumnos de Informática de la Escuela, los recursos son compartidos y por tanto no se dispone de toda la infraestructura dada por los computadores. Esto ha conllevado en memoria de RAM insuficiente ante las simulaciones o incluso pérdida de computación de los procesadores. Si había dos alumnos en el ordenador y se lanzaba el simulador, el resultado se degradaba de manera que los resultados no eran concluyentes.

11.3. Gramática del lenguaje

$G = (V, \Sigma, Q_0\ P)\ donde$:

- V es el conjunto finito de símbolos no terminales compuesto por:

RdP, nombrered, lsitasubredes, subred, nombresubred, cuerpo, lugares, listalugares, lugar, transiciones, listatransiciones, transición, pre, post, marcados, tiempos, lista_tiempo, lugar_tiempo, interfases, entrada, salida, finsubred, finred, sincronizaciones, listasincro, sincro, lugarviejo, lugarnuevo.

- Σ es el conjunto finito de símbolos terminales compuesto por:

RED, SUBRED, LUGARES, TRANSICIONES, PRE, POST, MARCADO, TIEMPO, INTERFASE, ENTRADA, SALIDA, FINSUBRED, FINRED.
CONST_ENTERA ::= (0 – 9)+
IDENTIFICADOR ::= [a - Z] (a – Z | 0 – 9 | "_")*

- Q0 representa el símbolo inicial, RdP
- P representa el conjunto de producciones:

RdP ::= nombrered listasubredes sincronizaciones finred ;
nombrered ::= RED identificador ";"
listasubredes ::= subred | listasubredes subred
subred ::= nombresubred cuerpo finsubred ";"
nombresubred ::= SUBRED identificador ";"
cuerpo ::= lugares transiciones marcados tiempos interfases
lugares ::= LUGARES listalugares ";"
listalugares ::= lugar | lugar "," listalugares
lugar ::= identificador
transiciones ::= TRANSICIONES listatransiciones
listatransiciones ::= transición | listatransiciones transición
transición ::= identificador ":" pre post
pre ::= PRE listalugares ";"
post ::= POST listalugares ";"
marcados ::= MARCADO listalugares ; | E
tiempos ::= TIEMPO lista_tiempo ";" | E
lista_tiempo ::= lugar_tiempo | lugar_tiempo "," lista_tiempo;
lugar_tiempo ::= IDENTIFICADOR | IDENTIFICADOR "(" CONST_ENTERA ")"
interfases ::= INTERFASE entrada salida ";" | E
entrada ::= ENTRADA listalugares ";" | E
salida ::= SALIDA listalugares ";" | E
finsubred ::= FINSUBRED ";"
finred ::= FINRED ";"
sincronizaciones ::= SINCRONIZACION listasincro ";"
listasincro ::= sincro | sincro listasincro
sincro ::= lugarviejo "<=" lugarnuevo
lugarviejo ::= "(" identificador "," listalugares ")"
lugarnuevo ::= "(" listalugares ")"

11.4. Algoritmo centralizado

```
1: procedure Simulate(LVHT)
2:     while (LVT <= LVTH) do
3:         if (head-FUL.time > clock) then VT ← head-FUL.time          ▷ Update Virtual Time
4:         end if
5:         while (head-FUL.time = VT) do                               ▷ Update Event List
6:             t ← head-FUL.pt; f_t(M) := f_t(M) + head-FUL.UF;
7:             if (f_t(M) ≤ 0) then insert(EL, t);
8:             end if
9:             head-FUL ← pop(FUL);
10:        end while
11:        EVL ← Sort(EVL,CCS, Strategy);          ▷ prioritizes transitions in conflict int EVL
12:        for all (t' ∈ EL) do                                        ▷ Fires enabled transitions
13:            if (f_t'(M) ≤ 0) then                                   ▷ Checks transition is enabled yet
14:                for all (t ∈ IUL(t')) do
15:                    f_t(M) ← f_t(M) + UF(t' → t);
16:                    if (t = t' and f_t(M) ≤ 0) then                 ▷ Avoids race conditions
17:                        insert-FUL (t, 0, τ(t) + clock);
18:                    end if
19:                end for
20:                for all (t ∈ PUL(t')) do
21:                    insert-FUL (t, UF(t' → t), τ(t') + clock);
22:                end for
23:            end if
24:        end for
25:    end while
26: end procedure
```

A continuación, se explica el funcionamiento del algoritmo centralizado:

El simulador ejecuta la simulación hasta el tiempo ***LVHT*** (Horizonte lógico de tiempo virtual o en inglés, *Logical Horizontal Virtual Time*).

Inicialmente, se avanza el reloj lógico hasta el tiempo del primer evento de la lista de eventos futuros (*Future Event List o **FUL**)* (líneas 3 – 4). Entonces, el algoritmo actualiza los valores de los LEFs con el valor de la constante (*Updating Factor o **UF***) de cada evento cuyo tiempo coincide con el tiempo de simulación actual (*Logical Virtual Time o **LVT**)*, y actualiza la lista de transiciones sensibilizadas.

Head-ful es un puntero a la lista de eventos futuros ***FUL***, y *pop(**FUL**)* devuelve la cabeza de la lista.

Continuando, el algoritmo resuelve los conflictos ordenando los conflictos acoplados (*Conflict Coupled Set o **CCS***) (línea 11) acorde a alguna estrategia definida.

Por cada transición sensibilizada, el algoritmo aplica los valores de actualización inmediatos (*Inmediate Updating Factor o **IUF***), que representa el recogido de marcas de los sitios de entrada de las transiciones que han sido disparadas (líneas 14 – 19), e inserta los eventos en la lista de eventos futuros (*Proyected Updating List o **PUL**)* que representa las marcas que aparecerán en los lugares posteriores de la transición disparada en un tiempo futuro (líneas 20-22).

11.5. Implementación del algoritmo centralizado

A continuación, se va a remarcar las ideas principales sobre la implementación realizada del algoritmo centralizado.

- El algoritmo recibe como parámetros la red a simular, que puede ir particionada o en una única subred. En caso de que sea particionada, se procederá a la fusión de lugares.
- Se compila y elabora la red a simular almacenando en una lista el conjunto de transiciones que se encuentran sensibilizadas con el marcado inicial.
- Se inicia el algoritmo de manera iterativa hasta que el tiempo de simulación sobrepasa la frontera de tiempo, es decir, el tiempo máximo a simular.
    - Se comprueba si existen transiciones sensibilizadas para el tiempo actual. Para ello se recorre la pila donde se encuentran las transiciones sensibilizadas. En caso afirmativo, se disparan dichas transiciones creando los eventos generados por el disparo de cada transición. En caso negativo se sigue la ejecución. El disparo de una transición consiste en insertar los eventos futuros en la lista de eventos y actualizar los valores de la función LEF debido al disparo (Evento inmediato). Además, se almacena en un vector el identificador de la transición disparada, así como el tiempo en el que se ha realizado.
    - Se comprueba si existen eventos a tratar para el tiempo actual. En caso afirmativo se tratan por separado, es decir, se actualizan los LEFS de la red. En caso negativo, se continua la ejecución. La cola de eventos se ha implementado como una *ArrayList* de *Java*, en la que se van insertando los eventos calculando primero la posición relativa del evento teniendo en cuenta el tiempo para el que está proyectado.
    - Debido a los eventos tratados en la fase anterior, se actualiza la pila de transiciones sensibilizadas.
    - Si no existe ningún evento para el tiempo de simulación actual ni ninguna transición sensibilizada, se incrementa el reloj local de la simulación hasta el tiempo mínimo de entre todos los eventos futuros.
- Una vez se finaliza la simulación, se recorre la lista de todos los disparos registrados y se muestran por pantalla en la consola y se genera un fichero log donde se vuelcan dichos disparos y estadísticas sobre las transiciones disparadas por segundo, el tiempo de simulación y el tiempo real.

A continuación, se muestra el código del algoritmo centralizado:

```
private void simularCent(int inicio, int numCiclos) throws IOException {
    Date ld_ini = new Date();
    // Inicializamos el reloj local
    int relojlocal = ii_cicloinicial;
    // Inicializamos las transiciones sensibilizadas, es decir, ver si con el
    // marcado inicial tenemos transiciones sensibilizadas
    il_lefs.inicializa_tiempos(relojlocal);
    il_lefs.actualiza_sensibilizadas(relojlocal);
    while (relojlocal <= numCiclos) {
        // Si existen transiciones sensibilizadas para el reloj local las disparamos
        if (il_lefs.hay_sensibilizadas())
            disparar_transiciones_sensibilizadas(relojlocal);
        // Si existen eventos para el reloj local los tratamos
        if (il_lefs.hay_eventos(relojlocal))
            tratar_eventos(relojlocal);
        // Los nuevos eventos han podido sensibilizar nuevas transiciones
        il_lefs.actualiza_sensibilizadas(relojlocal);
        // Tras tratar todos los eventos, si no nos quedan transiciones sensibilizadas no podemos simular
        // nada mas procedemos a avanzar el reloj local
        if (!il_lefs.hay_sensibilizadas()) {
            if (!il_lefs.hay_eventos(relojlocal)) {
                relojlocal = avanzar_tiempo();
                if (relojlocal == -1)
                    relojlocal = numCiclos + 1;
            }
        }
    }
    Date ld_fin = new Date();
    tiempoReal = (int) (ld_fin.getTime() - ld_ini.getTime());    // Milisegundos
    tiempoSimulado = (numCiclos - getTiempoInicial());
    System.out.println("FIN DE LA SIMULACION");
}
```

Figura 24: Código del simulador centralizado implementado

11.6. Introducción al Diseño de un Simulador Distribuido

Una simulación es una representación de un sistema físico evolucionando en el tiempo. Un sistema físico o procesos se modela mediante un proceso lógico (LP, Logical Process). Un LP consta de un número de entidades virtuales que realizan tareas e interactúan mediante intercambio de mensajes que ser presentan mediante eventos al nivel de los LP. El estado de las entidades cambia en el tiempo, y por lo tanto provocan la evolución del sistema. Los cabios de estado en el tiempo son dirigidos por el avance el tiempo virtual (VT, virtual time) en la simulación. En una simulación de eventos discretos el tiempo de ejecución avanza o salta discretamente hasta el tiempo de ejecución del siguiente evento que ocurre en la simulación. Un evento tiene un tiempo de disparo, indicando el tiempo de simulación en que el evento ocurre. La ejecución de un evento puede crear nuevos eventos, y la simulación completa acaba cuando todos los eventos son procesados.

Para mejorar las prestaciones y escalabilidad del modelo se introduce la simulación distribuida entre varias máquinas. Existen varias opciones para llevar a cabo la simulación de eventos discreta como se ha presentado en la sección 3. Un núcleo de un simulador ejecuta los procesos lógicos mediante un bucle que ejecuta continuamente los eventos en el tiempo de disparo correspondiente. Los principales componentes de son el reloj que mantiene el tiempo virtual de simulación y la cola de eventos o event list (EVL).

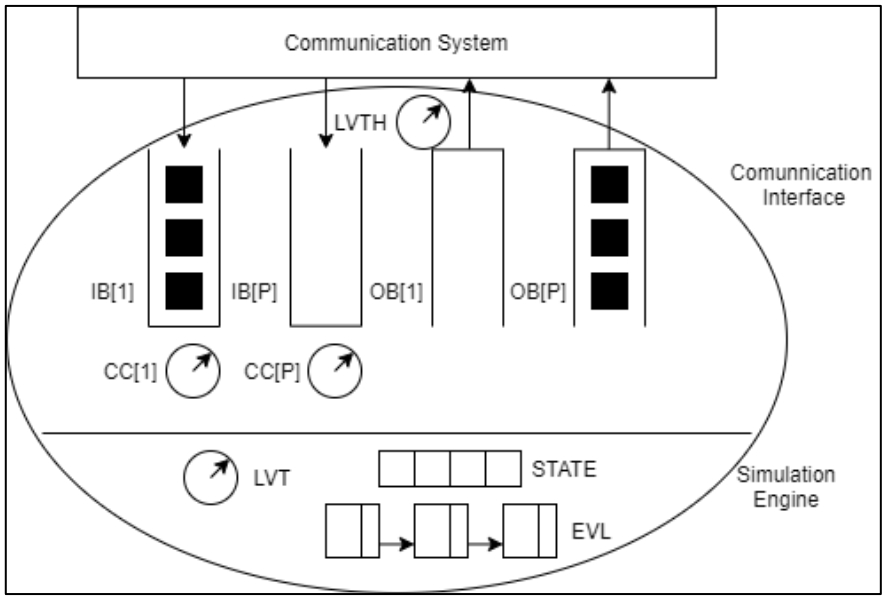


Figura 25: Arquitectura de un simulador distribuido conservativo

La figura presenta la implementación básica de un simulador conservativo. Dado una topología de simuladores que pueden intercambiar mensajes a través de un sistema de comunicación, el simulador consta de una interface de comunicación que consta de un buffer de entrada IB[i] (IB, input buffer) donde se almacenan los eventos en orden FIFO por cada uno de los simuladores (LP) adyacentes. CC[i] (CC, Canales de Comunicación) mantiene una copia de los time stamp de los mensajes que se encuentran en la cabeza de los IB[i]. Inicialmente CC[i] es cero. El tiempo de horizonte de simulación (LVTH, Local Virtual Time Horizon) es el tiempo que puede avanzar el simulador sus eventos sin tenerse que preocupar de que llegue un evento externo anterior. Durante este tiempo, el simulador puede producir eventos internos que se almacenan en la EVL, o eventos externos que son depositados en los buffers de salida OB[i], uno por cada simulador adyacente. Cuando dentro del horizonte temporal marcado por LVTH no hay

eventos internos ni externos que consumir, el simulador se bloquea hasta que no tiene un mensaje de cada uno de los simuladores adyacentes. Para evitar el bloqueo, los simuladores calculan los lookahead, mensajes sin contenido (null), pero que llevan información de hasta qué punto pueden avanzar el LVT sin que puedan recibir posteriormente mensajes con time stamp anterior. El simulador elegirá el tiempo menor de todos los recibidos en los canales de comunicación CC[i] para establecer el nuevo LVTH y reanudar la simulación. Los detalles de los algoritmos de simulación se han presentado en la sección 4.2 y 4.3.

La Figura 25 tiene por objeto poner en contexto los elementos de un simulador distribuido y como breve introducción al resto de esta memoria.

11.7. Implementación del algoritmo distribuido

A continuación, se va a explicar el funcionamiento del algoritmo distribuido, así como los detalles de la implementación.

Teniendo en consideración que actualmente solo se permite tener una máquina virtual por cada máquina física, y que cada máquina virtual admite un único simbot el funcionamiento del algoritmo para una máquina dada es el siguiente:

- El algoritmo recibe como parámetros el fichero de texto con la descripción del sistema a simular, el cual puede estar particionado en varias subredes y por tanto se procederá al despliegue en las máquinas disponibles, o puede estar en una subred y por tanto se realice la simulación centralizada. Además, recibe el número de ciclos máximos de simulación, la lista con las direcciones ip's de las máquinas disponibles siendo la primera la máquina considerada como principal, y un índice para seleccionar que máquina corresponde de la lista máquinas.
- Lo siguiente es realizar la asignación de las subredes a las diferentes máquinas. En función del número de subredes a simular y del número de máquinas disponibles, se hace un reparto equitativo y en caso de que la división de subredes no sea entera, el resto de las redes la simulará la máquina principal.
- Una vez se compila y elabora, se inicia el algoritmo distribuido que se divide en 4 fases.
    - 1. Sincronización y areYouAlive: La máquina principal espera hasta que las máquinas de la lista estén operativas, es decir, hayan lanzado el simulador y estén esperando algún mensaje. Seguidamente, se realiza un proceso de barrera para sincronizar y que empiecen todas las maquinas a la vez.
    - 2. Algoritmo distribuido, similar al centralizado. Los cambios añadidos son que las transiciones de salida generan eventos exteriores, dichos eventos se caracterizan por que afectan a subredes que no están en el propio simbot. Se ha decidido etiquetar las transiciones negativas cuando son para eventos exteriores. Si al tratar los eventos se comprueba que hay eventos exteriores, se mandan los mensajes con el evento y el tiempo actual. Si no hay eventos, se envía un mensaje avisando a las maquinas adyacentes hasta el tiempo que pueden avanzar ya que no van a recibir ningún mensaje de este simbot. El avance del reloj se hace con el mínimo de los tiempos recibidos y el del próximo evento local.
    - 3. Sincronización: Se sincronizan todas las máquinas para terminar en el mismo momento.
    - 4. Envío de logs: La máquina principal espera a que le lleguen los mensajes con los logs generados por cada máquina que ha intervenido en la simulación.
- Finalmente se muestran los logs distribuidos.

A continuación, se muestra el código del algoritmo distribuido.

```java
public void simularDist(int inicio, int numCiclos) {
    int relojLocal = inicio;
    int lookahead = 0;
    Date ld_ini = new Date();
    try {
        System.out.println("INICIO FASE 1: SINCRONIZACION");
        barreraMaquinas();    // Sincronizacion
        System.out.println("INICIO FASE 2: SIMULACION\n\n");
        // Inicializamos transiciones sensibilizadas de manera inicial
        il_lefs.inicializa_tiempos(relojLocal);
        il_lefs.actualiza_sensibilizadas(relojLocal);
        while (relojLocal <= numCiclos) {
            lookahead = relojLocal + Math.max(il_lefs.dame_lookahead(), 0);
            // Miramos si hay sensibilizadas para el tiempo local
            if (il_lefs.hay_sensibilizadas())
                // Si hay sensibilizadas se disparan generando nuevos eventos
                disparar_transiciones_sensibilizadas(relojLocal);
            // Si hay eventos para el reloj local lo tratamos
            if (il_lefs.hay_eventos(relojLocal))
                tratar_eventos(relojLocal);
            // Los nuevos eventos han podido sensibilizar nuevas transiciones
            il_lefs.actualiza_sensibilizadas(relojLocal);
            // Si no hay sensibilizadas
            if (!il_lefs.hay_sensibilizadas()) {
                if (lookahead <= numCiclos) {    // No va a mandar ningun evento que esté dentro del numero de ciclos
                    // Si existen eventos exteriores
                    if (il_lefs.hay_eventos_exteriores()) {
                        // Mandamos mensaje del evento
                        while (il_lefs.hay_eventos_exteriores()) {
                            Evento ex = il_lefs.pop_primer_evento_exterior();
                            if(il_lefs.tengo_la_transicion(ex.ii_transicion)) {
                                il_lefs.agnade_evento(ex);
                            }else {
                                // Enviamos el evento a todas las maquinas y ellas se encargaran de saber
                                // si es para ella
```

Figura 26: Código del algoritmo distribuido

```
                        enviarEventoMaquinas(ex, EVENTO);
                    }
                }
            } else {    // No existen eventos exteriores
                // No tengo nada para tiempo reloj_local
                // Envio a mis siguientes el tiempo hasta donde yo no les voy a enviar nada
                // Es decir mi reloj local mas el lookahead
                enviarEventoMaquinas(new Evento(lookahead, ai_transicion: 0, ai_cte: 0), RELOJ);
            }
        } else {
            // Les envio de que yo ya no les enviaré nada mas
            enviarEventoMaquinas(new Evento(lookahead, ai_transicion: 0, ai_cte: 0), TERMINANDO);
        }


        // Momento de recibir los demas mensajes
        int avanzarTiempo = recibirEventosMaquinas();
        // Ahora deberiamos mirar si no hay eventos para el reloj local
        if (!il_lefs.hay_eventos(relojLocal) &&
                !il_lefs.hay_sensibilizadas()) {
            if ((avanzarTiempo != ERROR) && (avanzarTiempo > relojLocal)) {
                if (MODE_SIMULATION == DEBUG)
                    System.out.println("Avanzo el tiempo de " + relojLocal
                            + " a " + avanzarTiempo);
                relojLocal = avanzarTiempo;

            }
        }

    }


}// fin del while

// Envio por sia caso evento de que he terminado
enviarEventoMaquinas(new Evento(relojLocal, ai_transicion: 0, ai_cte: 0), TERMINANDO);
```

```
        /*
         * FASE FINAL: Finalizado de todas las maquinas
         */
        if (esPrincipal) {
            if (MODE_SIMULATION == DEBUG)
                System.out.println("(PRINCIPAL): Ya he terminado, espero los mensajes de terminacion");
            // Si soy la maquina principal espero hasta que me hayan llegado n - 1 mensajes de terminacion
            int i = listaMaquinas.length - 1;
            while (i > 0) {
                ArrayList<Paquete> mensajes = mochila.getEvento();
                for(Paquete paq : mensajes) {
                    if (paq.direccionIp.equals(listaMaquinas[i])) {
                        if (paq.motivo == SINCRONIZACION) {
                            if (MODE_SIMULATION == DEBUG)
                                System.out.println("Me llega mensaje de " + paq.direccionIp + " de que ha terminado");
                            i--;
                        } else if (paq.motivo == RELOJ || paq.motivo == TERMINANDO) {
                            if (MODE_SIMULATION == DEBUG)
                                System.out.println("Envio mensaje de finalizacion porque yo estoy esperando");
                            mochila.enviaEvento(listaMaquinas[i], new Evento(relojLocal, ai_transicion: 0, RELOJ), RELOJ, miIp);
                        }
                    }
                }
            }
            // Una vez me han llegado los n - 1 mensajes de terminacion, les envio de que yo tambien he terminado
            for (i = 1; i < listaMaquinas.length; i++) {
                if (MODE_SIMULATION == DEBUG) {
                    System.out.println("Envio mensaje de sincro a " + listaMaquinas[i]);
                }
                mochila.enviaEvento(listaMaquinas[i], new Evento(relojLocal, ai_transicion: 0, SINCRONIZACION), SINCRONIZACION, miIp);
            }


        } else {
            // No es la maquina principal, envia mensajes de que ha terminado hasta que le llegue el de central
```

```
                if (MODE_SIMULATION == DEBUG)
                    System.out.println("Mando mensajes de que he terminado");
                boolean stopEscucha = false;
                while (!stopEscucha) {
                    if (MODE_SIMULATION == DEBUG)
                        System.out.println("Envio mensaje");
                    // Le envio el mensaje de que yo he terminado y estoy a la espera
                    mochila.enviaEvento(central, new Evento(relojLocal, ai_transicion: 0, SINCRONIZACION), SINCRONIZACION, miIp);
                    ArrayList<Paquete> mensajes = mochila.getEvento();
                    for(Paquete paq : mensajes)
                        if (paq != null && paq.motivo == SINCRONIZACION && paq.direccionIp.equals(central)) {
                            stopEscucha = true;
                            break;
                        }

                }
                // Ya me han contestado de que puedo continuar
            }


        } catch (Exception ignored) {

        }

        Date ld_fin = new Date();
        tiempoReal = (int) (ld_fin.getTime() - ld_ini.getTime());    // Milisegundos
        tiempoSimulado = (numCiclos - getTiempoInicial());
        System.out.println("FIN DE LA SIMULACION");
    }
```

11.8. Protocolo de mensajería y estructura de mensajes

- El protocolo usado para el envío de mensajes entre las diferentes máquinas ha sido **TCP/IP** sobre la implementación de *sockets* de *JAVA*. Dicho protocolo está muy estudiado, existe mucha documentación sobre ello y además existen directivas en *JAVA* de alto nivel que abstraen de la problemática de conexión de red. Como es una comunicación asíncrona, existe un proceso que mantiene un servidor abierto a nuevas conexiones que encola en una lista los mensajes que llegan. Al llegar una nueva conexión, se crea un nuevo hilo de ejecución para atender dicha conexión sin dejar de estar abierto a nuevas conexiones. Las clases usadas han sido *java.net.Socket* y *java.net.ServerSocket*.
- La estructura del mensaje que se envía entre las máquinas llamado en el código *Paquete.java* se puede ver en la siguiente figura. Un paquete se compone del evento que se pretende enviar (constante, transición a la que se le envía la constante y tiempo en la que hace efecto), el motivo por el que se envía dicho mensaje (Nuevo evento a enviar, no hay evento, sincronización de máquinas, finalización de simulación) y la dirección ip donde se origina dicho mensaje. Dicho mensaje es enviado como la clase *Paquete* gracias a *ObjectOutputStream* que permite enviar clases completas a través de sockets.

```
public class Paquete implements Serializable {

    private static final long serialVersionUID = 1L;

    public Evento evento;
    public int motivo;
    String direccionIp;

    public Paquete(Evento e, int motivo, String direccionIp) {
        evento = e;
        this.motivo = motivo;
        this.direccionIp = direccionIp;
    }

    @Override
    public String toString() {
        return "Paquete [evento=" + evento + ", motivo=" + motivo + ", direccionIp=" + direccionIp + "]";
    }


}
```

Figura 27: Estructura de los mensajes

11.9. Servidor de nombres

El servidor de nombres implementado permite obtener datos relacionados con las transiciones involucradas en la simulación del sistema. Los datos que se pueden obtener son: (1) Localización (dirección ip) de la subred que posee dicha transición; (2) Nombre global de la transición (El nombre original del fichero, ya que al elaborar se cambian los nombres de las transiciones).

Se ha definido una API que realiza las conexiones con el servidor de *Redis*. Se han implementado las siguientes funciones:

- Comprobar si el servidor se encuentra activo y operativo. En caso afirmativo, se establece un flag para que el simulador sepa que el servidor se encuentra operativo y se puede utilizar. En caso negativo, se establece un flag para que el simulador no use el servidor y las direcciones se establecen de manera manual por el usuario a través del fichero máquinas.
- Obtener los datos de una transición dado su nombre local.
- Obtener todas las transiciones de las que se tiene información en el servidor.
- Insertar los datos de una transición usando como clave su nombre local.

Para la conexión con el servidor de *Redis* se ha usado la librería *redis.clients.jedis* de *JAVA.*

El servidor de nombres usa la tecnología *Redis* para la persistencia de los datos. *Redis* implementa una tabla hash que permite indexar los datos con una clave única y el valor puede ser una lista de cadenas de texto. En el caso de la implementación realizada, la clave es el nombre local de la transición (tras la elaboración) y el valor contiene los datos relacionados que se han comentado antes.

Inicialmente al compilar y elaborar la red se almacena en el servidor de nombres toda la información relativa a las transiciones que intervienen en la simulación. Después se comprueba si existen transiciones de salida. En caso afirmativo, se pregunta a Redis la localización (dirección ip) de las transiciones que se ven alteradas por el disparo de la transición de salida y se almacena dicha dirección como red adyacente.

En el proceso de recuperación de todos los logs, se deben traducir los nombres locales a los nombres globales. Para ello, se usa el servidor de nombres para traducir los nombres y así mostrar los nombres conocidos (globales).

11.10. Diagrama de Paquetes

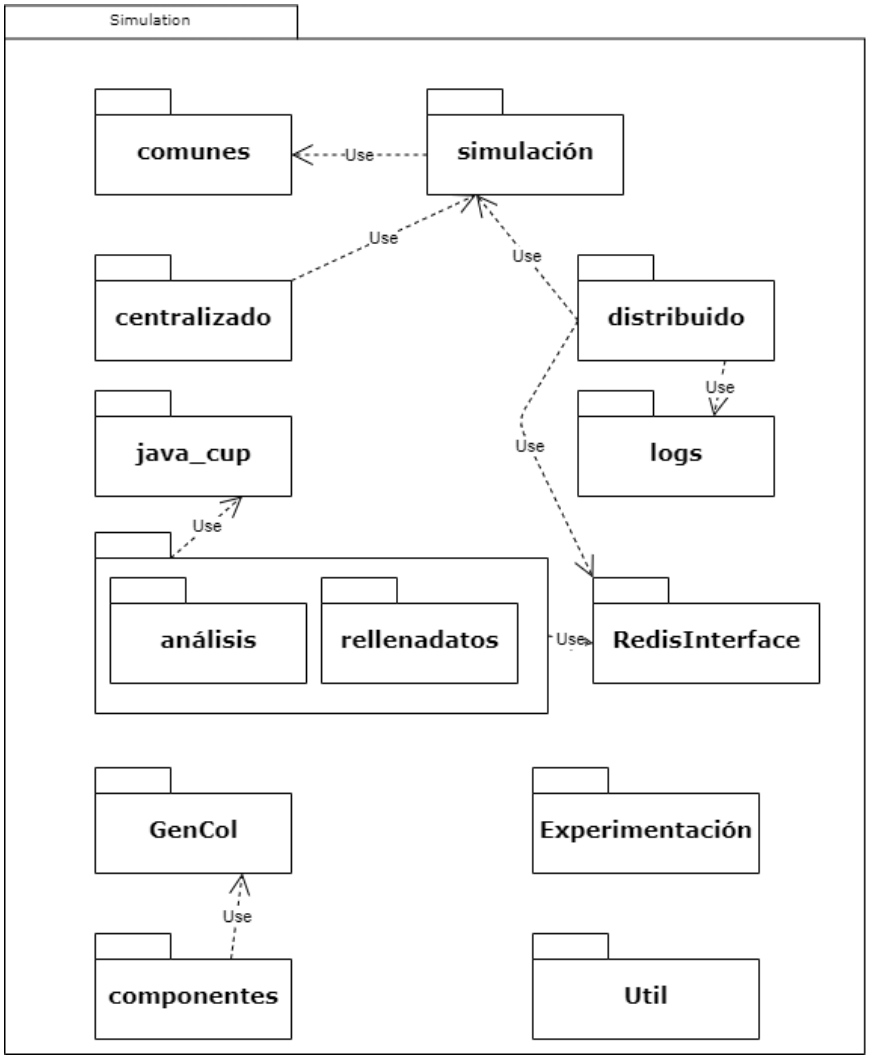


Figura 28: Diagrama de paquetes

A continuación, se remarca la funcionalidad de cada paquete:

- Comunes: Contiene variables globales compartidas entre las clases de la simulación.
- Simulación: Contiene las clases del kernel del simulador.
- Centralizado y Distribuido: Contiene las clases correspondientes al simulador centralizado y distribuido respectivamente.
- Logs: Contiene la clase que gestiona los logs.
- Java_cup: Contiene las clases para la generación de un parser.
- Análisis y rellena datos: Clases para parsear y rellenar con los datos de los ficheros planos.
- Redis_interface: Clase que ofrece API para conexión con redis.
- GenCol: Clases auxiliares para la creación de componentes.
- Componentes: Clases para la creación de componentes en el sistema.
- Experimentación: Clases para generar ficheros planos usados en las experimentaciones.
- Útil: Clases auxiliares.

11.11. Diagrama de clases

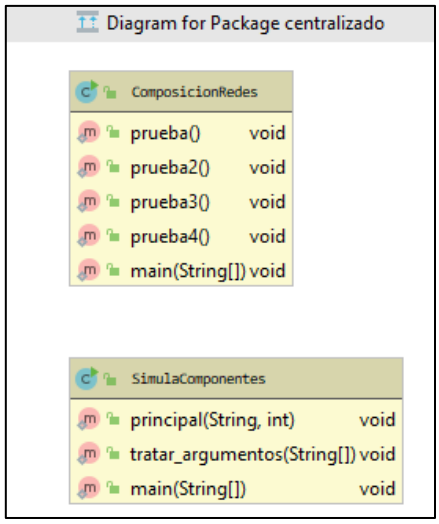


Figura 29: Diagrama de clases del paquete centralizado

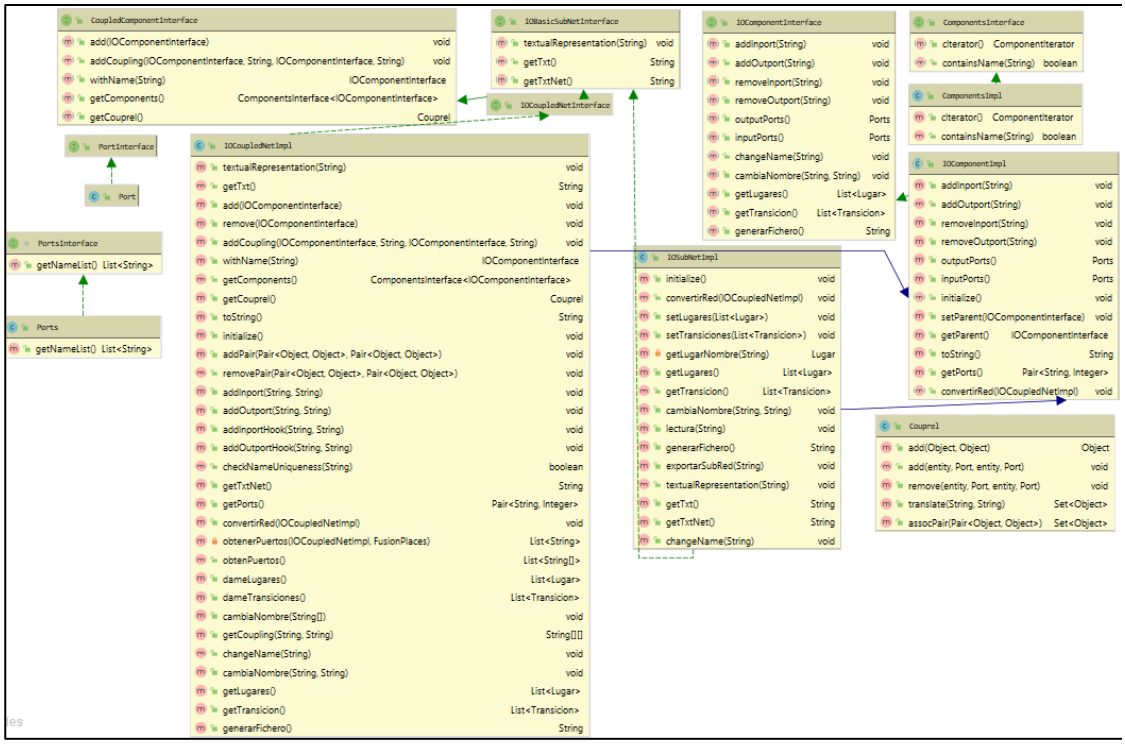


Figura 30: Diagrama de clase del paquete componentes

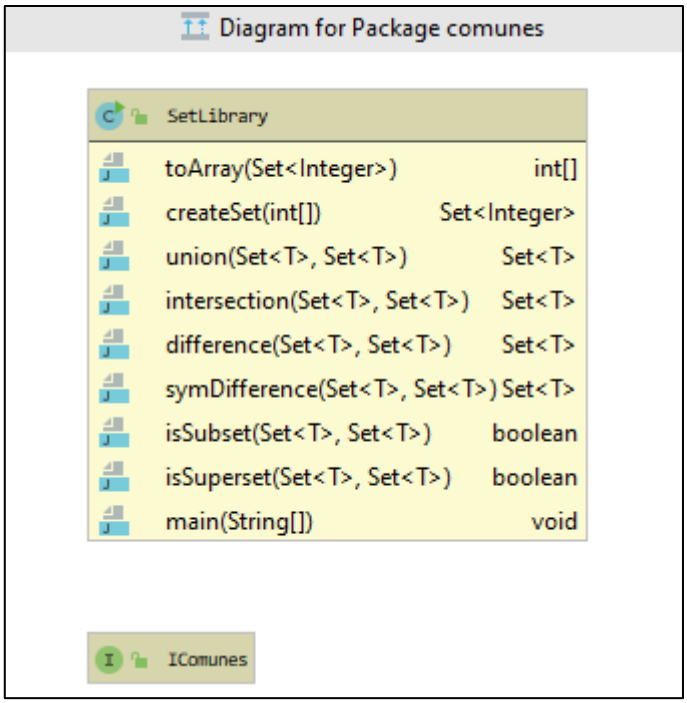


Figura 31: Diagrama de clases del paquete comunes

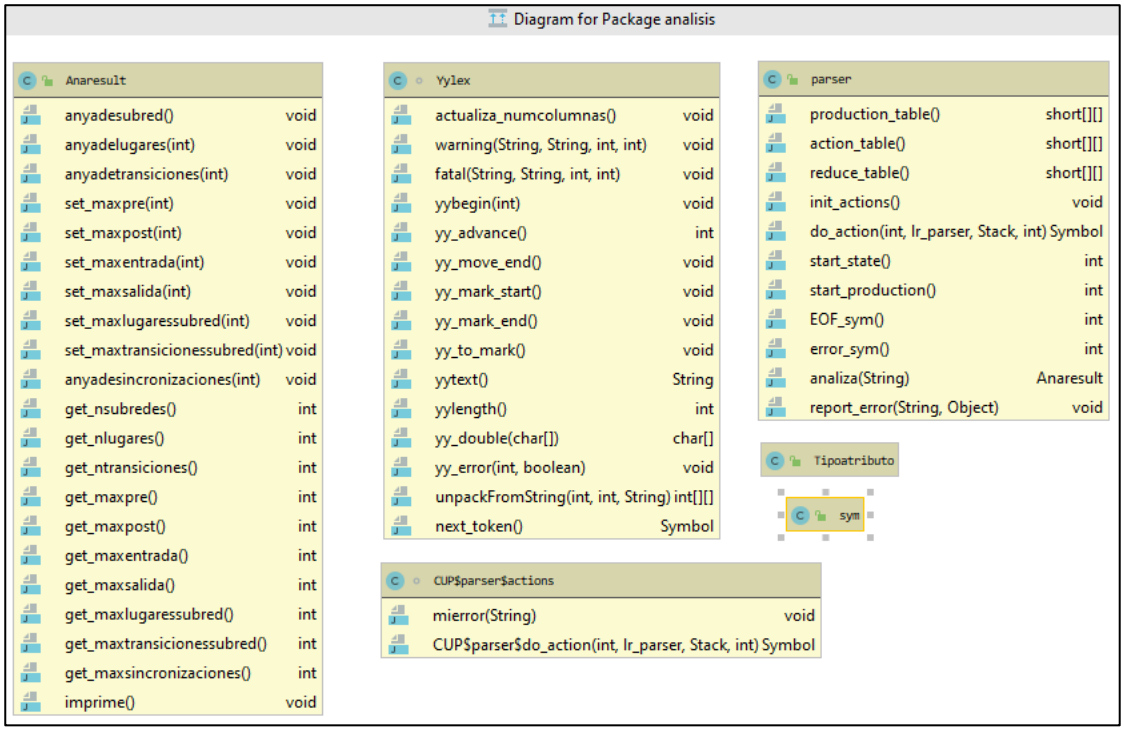


Figura 32: Diagrama de clases del paquete análisis

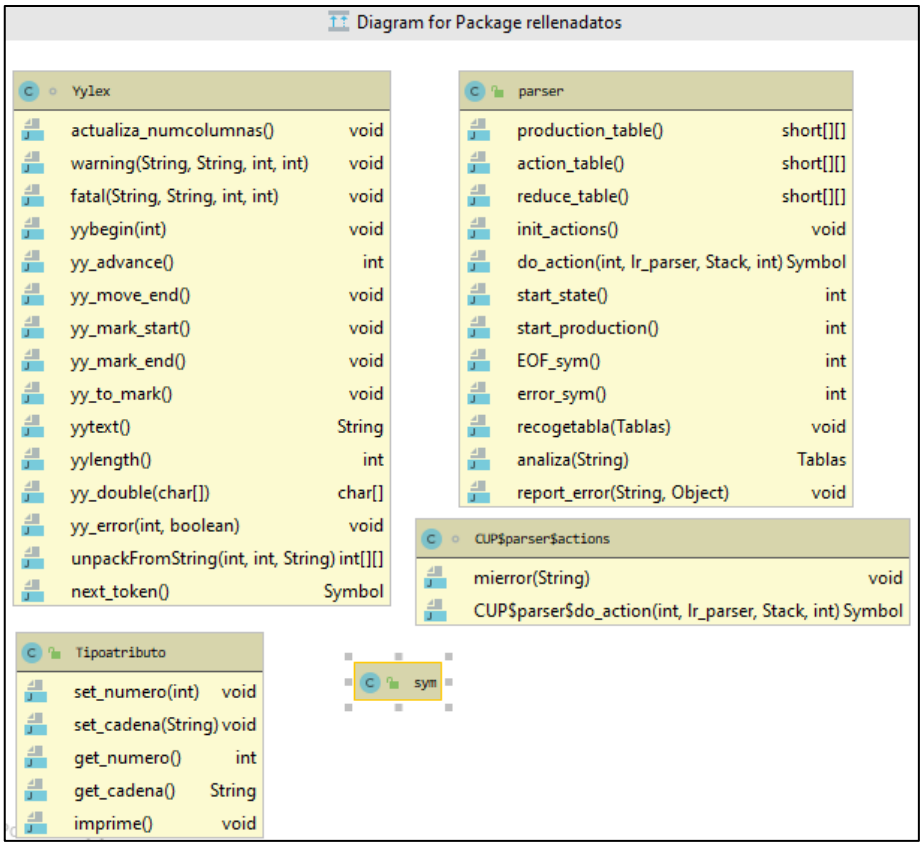


Figura 33: Diagrama de clases del paquete rellenadatos

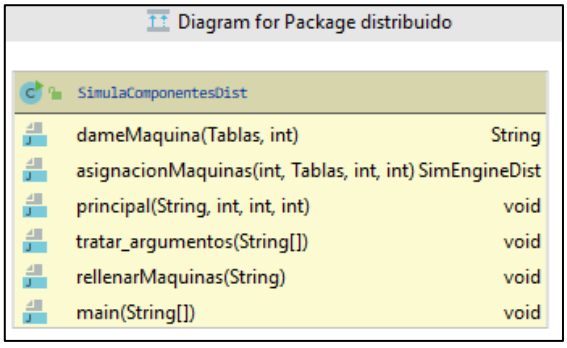


Figura 34: Diagrama de clases del paquete distribuido

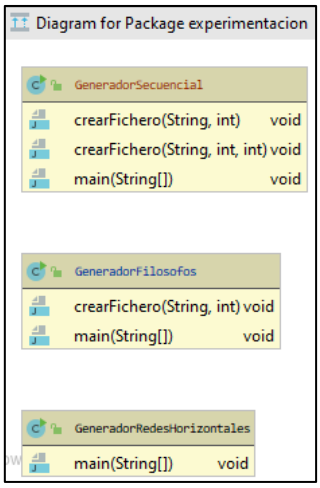


Figura 35: Diagrama de clases del paquete experimentación

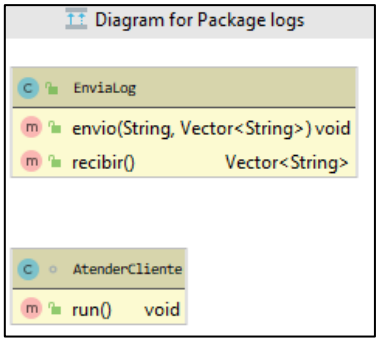


Figura 36: Diagrama de clases del paquete logs

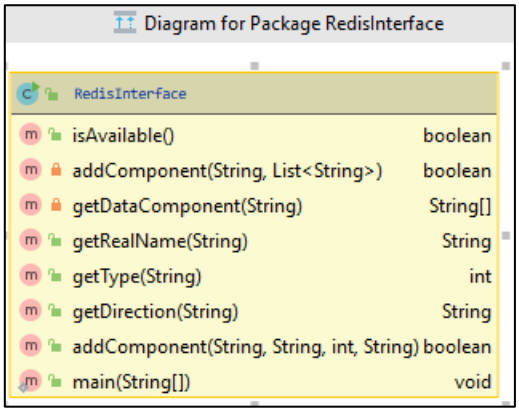


Figura 37: Diagrama de clases del paquete redisInterface

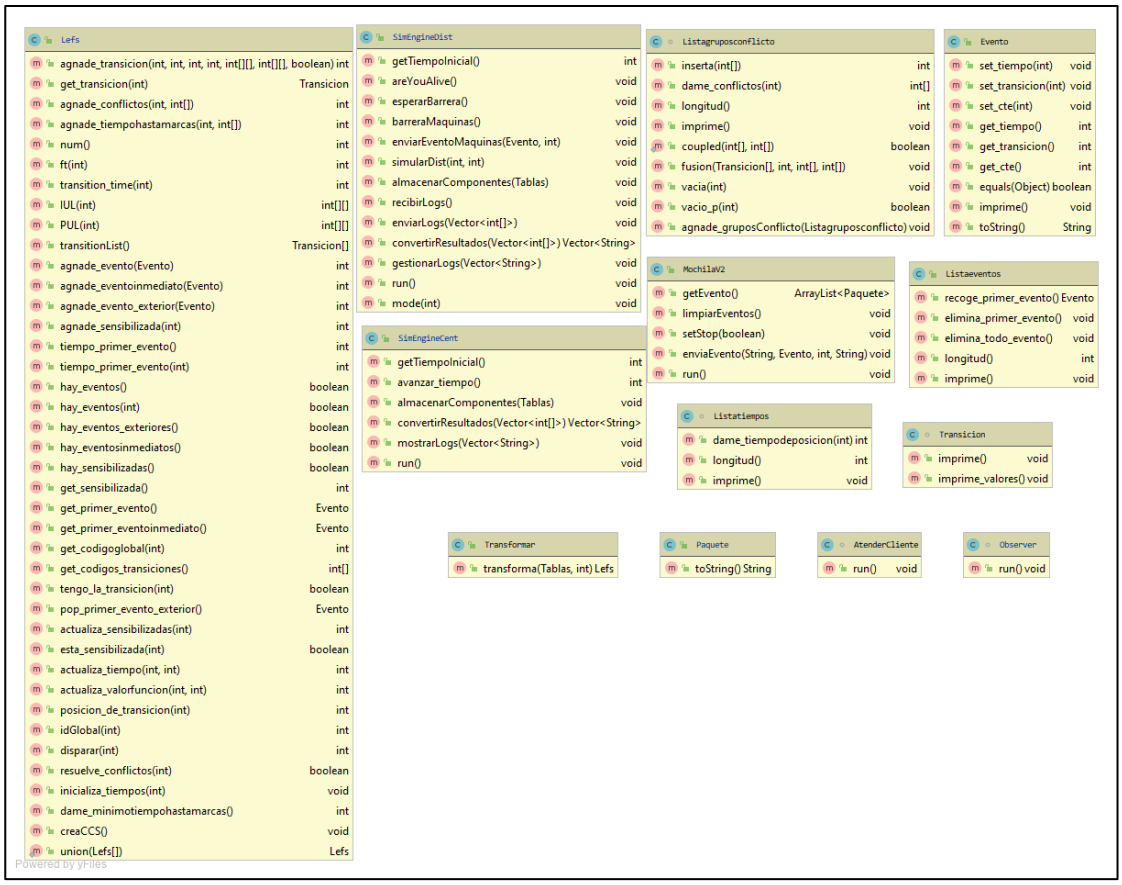


Figura 38: Diagrama de clases del paquete simulación

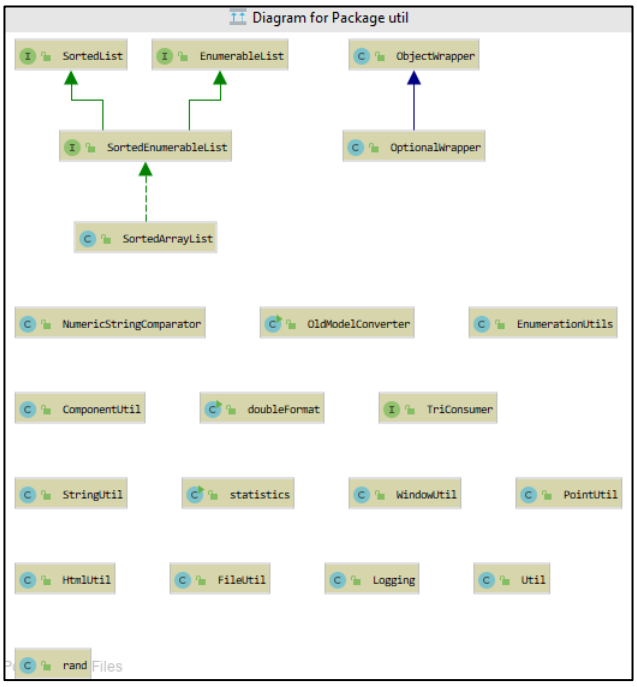


Figura 39: Diagrama de clases del paquete util

Cabe destacar que no se han mostrado las clases de algunos paquetes ya que son de tal envergadura que no se podría ver de manera correcta en esta memoria.

11.12. Algoritmo de compilación

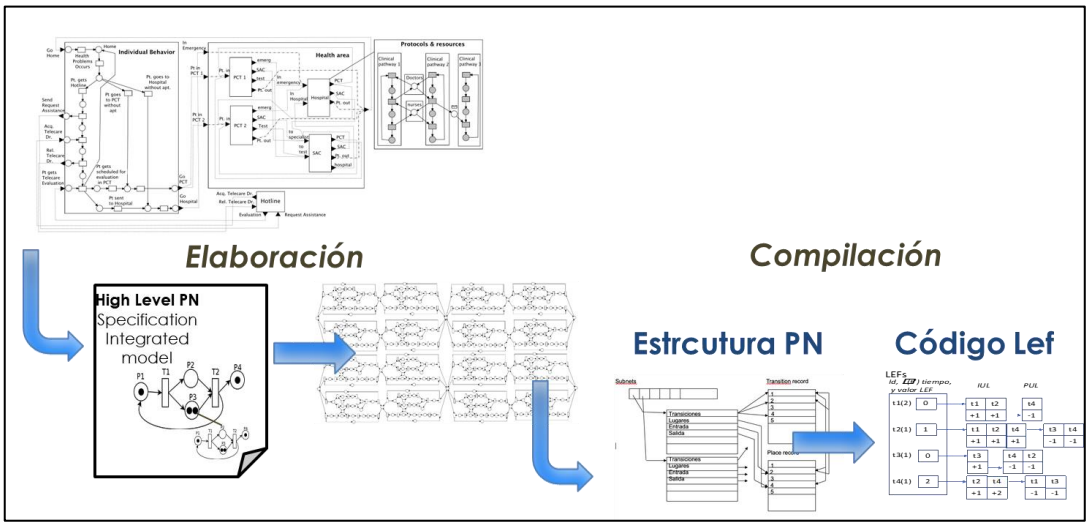


Figura 40: Transformación de Tabla a Lef

En la Figura 40 se presenta el proceso de elaboración y compilación en dos fases a partir de un modelo jerárquico.

El proceso de compilación se realiza sobre un modelo plano. La gramática especificada sólo permite definir un único nivel de jerarquía de redes conectadas. Tras el proceso de elaboración que se realiza al definir las componentes agregadas, el modelo generado se puede compilar. La compilación en detalle, en realidad consta de las siguientes cuatro fases de:

1. Precompilación: Se analiza con el parser desarrollado, *descripción.analisis,* y se obtiene, si el fichero es gramáticamente correcto, el número de transiciones, el número de subredes, el número de lugares, etc. Con todos los tamaños ya obtenidos, se crea una nueva estructura de datos para albergar dichos datos llamada *descripción.Tabla*.
2. Compilación: Se analiza con el parser *descripción.rellenadatos* y se va rellenando la estructura *Tabla* a medida que se va analizando. Como el tamaño ya está establecido no dará ningún problema de stackoverflow.
3. Transformación: Se transforma la estructura *Tablas* en una lista de nuevas estructuras llamada *Lefs*. Se va recorriendo cada subred obtenida y se transforma su información en un Lef. Por cada subred:
    a. Se recorre toda la lista de transiciones que tiene dicha subred.
    b. Se calcula la función de sensibilización de cada transición. Para ello se suma los pesos de todas las transiciones que son PRE de la transición a calcular.
    c. Se obtiene la lista de transiciones IUL y las constantes que serán enviadas, es decir aquellas transiciones que se ven afectadas de manera inmediata al actualizar la transición. Para ello se obtiene las transiciones que tienen como lugar de entrada aquellos que tiene la transición a calcular.
    d. Se obtiene la lista de transiciones PUL y las constantes que serán enviadas, es decir aquellas transiciones que se ven afectadas de manera futura al actualizar la transición. Para ello se obtiene las transiciones que tienen como lugar de entrada aquellos lugares que la transición tiene de

salida. El tiempo será el tiempo actual más el tiempo de disparo de la transición.

e. Se obtiene la lista de conflictos acoplados, buscando entre todas las transiciones que comparten lugares de entrada y agrupándolos en una misma clase CCS.

4. Adyacentes: Se obtiene la lista de subredes a las que se les debe enviar alguna constante. Para ello se recorre la lista de transiciones de salida de una subred y se añade a una lista las transiciones de otras subredes que se ven afectadas.

A continuación, se presenta el código de transformación de Tabla a LEF:

```
public class Transformar {

    // Va haciendo el traspaso de los datos de la subred dada, de la clase Tablas al formato que tenemos en la clase Lefs
    public Lefs transforma(Tablas aT_red, int ai_subred) {
        int li_ntransiciones, li_i;
        // int li_ntransiciones_total; //NEVER USED
        int li_duracion, li_fs, li_tiempo, li_dimension_IUL, li_dimension_PUL, li_cod_trans;
        int[][] la_ctes_IUL;  int[][] la_ctes_PUL;int[] la_conflicto; int[] la_tiempohastamarcas;
        try {
            // sacamos el numero de transiciones que tiene la subred
            li_ntransiciones = aT_red.dame_ntransiciones(ai_subred);
            // creamos la instancia de la clase que va a contener la informacion de las Lefs
            Lefs iL_subred = new Lefs(li_ntransiciones);
            // bucle para ir transicion por transicion sacando la informacion e introduciendola en la instancia de Lefs
            for (li_i = 0; li_i < li_ntransiciones; li_i++) {
                // cogemos la duracion del disparo de la transicion
                li_duracion = aT_red.dame_duracion(ai_subred, li_i);
                // calculamos la funcion de sensibilizacion de la transicion
                li_fs = aT_red.dame_fsensibilizacion(ai_subred, li_i);
                // el tiempo en el que es valida este valor de la funcion de sensibilizacion es t=cero
                li_tiempo = 0;
                // cogemos el codigo global de la transicion para usarlo amo identificador
                li_cod_trans = aT_red.dame_cod_trans_global(ai_subred, li_i);
                // pasamos a ver las constantes que tendra que generar la transicion y a que transiciones tendra que generarselas

                // recoger la informacion de la transicion li_i dentro de la subred ai_subred llevarla a las Lefs
                // agnadir las funciones necesarias a la clase Tablas
                li_dimension_IUL = aT_red.dame_numconstantes_IUL(ai_subred, li_i);
                //la_ctes_IUL=new int[li_dimension_IUL][2];
                la_ctes_IUL = aT_red.dame_constantes_IUL(ai_subred, li_i, li_dimension_IUL);
                li_dimension_PUL = aT_red.dame_numconstantes_PUL(ai_subred, li_i);
```

```
            //la_ctes_IUL=new int[li_dimension_IUL][2];
            la_ctes_PUL = aT_red.dame_constantes_PUL(ai_subred, li_i, li_dimension_PUL);

            // para agnadir la transicion a las Lefs, tenemos que comprobar si es transicion de salida de la subred,
            // es decir si alguno de sus lugares sumideros son interfaz de salida.
            if (aT_red.es_transicion_salida(li_cod_trans, ai_subred)) {
                // si la transicion es de salida, lo decimos con el ultimo parametro
                iL_subred.agnade_transicion(li_cod_trans, li_fs, li_tiempo, li_duracion, la_ctes_IUL, la_ctes_PUL,  ab_desalida: true);
                // Lookahead
                la_tiempohastamarcas = new int[li_ntransiciones];
                iL_subred.agnade_tiempohastamarcas(li_cod_trans, la_tiempohastamarcas);
            } else
                iL_subred.agnade_transicion(li_cod_trans, li_fs, li_tiempo, li_duracion, la_ctes_IUL, la_ctes_PUL,  ab_desalida: false);

            la_conflicto = aT_red.dame_conflicto(li_cod_trans, ai_subred);
            iL_subred.agnade_conflictos(li_cod_trans, la_conflicto);
        }
        iL_subred.creaCCS(); // Agnadido 30/4/2019
        return (iL_subred);

    } catch (Exception e) {
        System.out.println("Excepcion en transforma");
        System.out.println(e);
        System.exit( status: 1);
        return (new Lefs( ai_ntrans: 3));    // Aunque nunca se alcanza, necesario para compilar
    }
}
```

Figura 40: Transformación de Tabla a Lef

11.13. Proceso de elaboración. Clase FusionPlaces

El proceso de elaboración para obtener una especificación del comportamiento definida por una red Lugar/Transición sin jerarquía se realiza en la definición del comportamiento de una componente acoplada a partir de la especificación textual del comportamiento de las componentes agregadas. Este proceso utiliza la especificación textual de sincronizaciones que permite definir un nivel de jerarquía con subredes y fusiones de lugares de entrada y salida de las subredes especificadas en sus `INTERFASE`. La fusión de lugares de ENTRADA con lugares de SALIDA se especifica en `SINCRONIZATION`. La clase *`FusionPlaces`* dentro del paquete *`componentes`* se encarga de recoger todos aquellos puertos de entrada y de salida de los diferentes sistemas que están conectados y cambiarles el nombre a un nuevo nombre global para que el simulador sepa que están conectados y que representan el mismo lugar.

Las subcomponentes se definen con el constructor `IOSubNetImp(String nombreRed)` que reciben como argumento el nombre de se dará a la red, y el método `textualRepresentation(String nombreFichero)` el nombre del fichero de texto donde se especifica la red Lugar/Transición de define el comportamiento de la subred. De esta forma, podemos instanciar distintas componentes con diferente nombre, pero que utilizan la misma especificación textual del comportamiento. A partir del `INTERFASE` definido en este fichero de texto, se definen los puertos de entrada y salida, correspondientes a los lugares de entrada y salida de la subred. Los nombres de los puertos son los mismos que los nombres de los lugares que definen el `INTERFASE`.

La componente agregada se define mediante el constructor `IOCoupledSubNetImp(String nombreRed)`. El método addCoupling `(IOcomponetInteface src, String p1, IOcomponetInteface dest, String p2)` conecta puertos de salida con puertos de entrada de las subcomponentes. Estas relaciones se traducen en las relaciones de sincronización que se han de especificar para fusionar los lugares de entrada con los de salida de las subredes. Además, de agrupar aquellos lugares que son entrada y/o salida y están conectados entre ellos, hay que cambiarles el nombre a uno global para que el simulador sepa que representan el mismo lugar. El método `textualRepresentation(String nombreFichero)` obtiene genera el fichero con la representación textual con un modelo jerárquico a un nivel que se puede compilar.

En la siguiente figura se puede ver la creación de un sistema a partir de dos primitivas básicas:

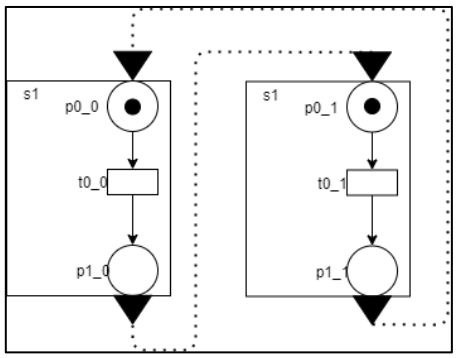


Figura 41: Unión de dos subredes

Tras la unión y posterior exportación, la clase FusionPlace se encarga de recorrer todos los puertos de entrada y salida que estén conectados (p0_0, p1_0, p0_1 y p1_1), e ir

cambiándoles el nombre a un nombre global (p0_0 y p1_1 son iguales, y p1_0 y p0_1 también son iguales) para identificarlos como iguales y así generar una red equivalente como la siguiente:

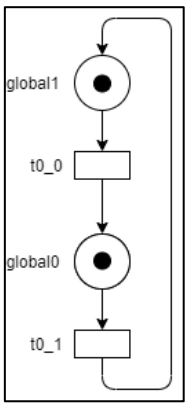


Figura 42: FusionPlace ha cambiado los nombres y agrupado

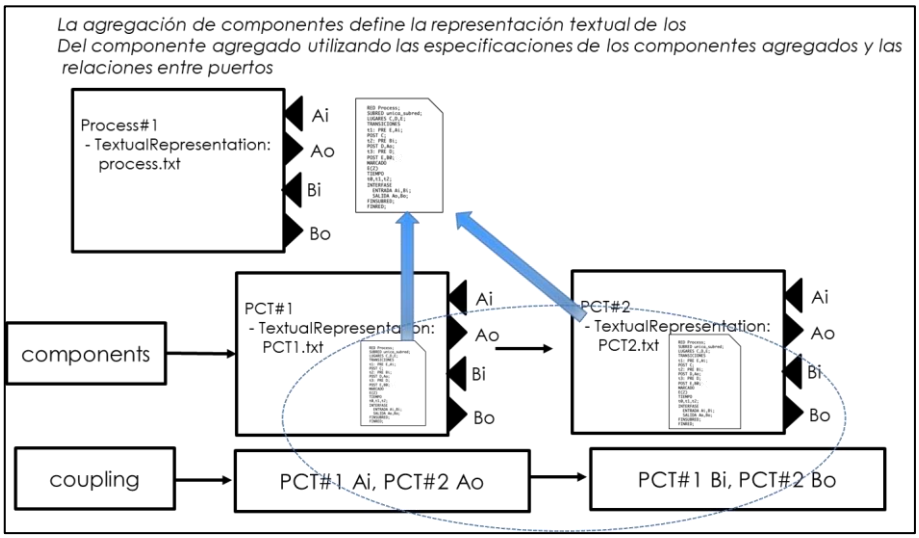


Figura 43: Obtención de la especificación textual de una entidad compuesta a partir de los agregados y las relaciones entre puertos

Para ilustrar el proceso de elaboración en más detalle, suponiendo que se parte de la especificación textual de las tres siguientes subredes y que se visualiza en la Figura 43:

SubRed0.txt
```
SUBRED sub0;
LUGARES p0,p1,p2,p3,p4;
TRANSICIONES
   t0: PRE p0;
       POST p1,p2;
   t1: PRE p3,p4;
       POST p0;
MARCADO p0;
TIEMPO
    t0,t1;
INTERFASE
  ENTRADA p3,p4;
  SALIDA p1,p2;
FINSUBRED;
SUBRED sub1;
```

SubRed1.txt
```
  LUGARES p0,p1,p2;
  TRANSICIONES
     t0: PRE p0;
         POST p1;
     t1: PRE p1;
         POST p2;
  TIEMPO t0,t1;
  INTERFASE
     ENTRADA p0;
     SALIDA p2;
  FINSUBRED;
```

SubRed2.txt
```
  SUBRED sub2;
  LUGARES p0,p1,p2;
  TRANSICIONES
     t0: PRE p0;
         POST p1;
     t1: PRE p1;
         POST p2;
  TIEMPO t0,t1;
  INTERFASE
     ENTRADA p0;
     SALIDA p2;
  FINSUBRED
```

La definición de la componente agregada `netGlobal` de la siguiente manera:

```
//Ejemplo Definición componente agregado
//Instancia componente agregada netGlobal
IOCoupledNetImpl netGlobal = new IOCoupledNetImpl("netGlobal");
//Instancia subredes
IOSubNetImp subnet0= new IOSubNetImpl("subred0");
Subnet0.textualRepresentation("Subred0.txt");
IOSubNetImpl subnet1= new IOSubNetImpl("subred1");
subnet1.textualRepresentation("Subred1.txt");
IOSubNetImpl subnet2= new IOSubNetImpl("subred2");
subnet2.textualRepresentation("Subred2.txt");
//Añadimos componentes al componente agregado
netGlobal.add(subnet0);
netGlobal.add(subnet1);
netGlobal.add(subnet2);
//Conectamos componentes
netGlobal.addCoupling(subnet0, "p1", subnet1, "p0");
netGlobal.addCoupling(subnet1, "p2", subnet0, "p3");
netGlobal.addCoupling(subnet0, "p2", subnet2, "p0");
netGlobal.addCoupling(subnet2, "p2", subnet0, "p4");
//Se genera fichero con especificación de red para el compilador
netGlobal.textualRepresentation("redGlobalGenerado.txt");
```

Da lugar a la siguiente representación textual de la red global en el fichero `redGlobalGenerado.txt`:

```
RED netGlobal;
SUBRED subred1;
                LUGARES p0,p1,p2;
                TRANSICIONES
                        t0: PRE p0;
                         POST p1;
                        t1: PRE p1;
                         POST p2;
                TIEMPO t0,t1;
                INTERFASE
                        ENTRADA p0;
                        SALIDA p2;
FINSUBRED;
SUBRED subred0;
                LUGARES p0,p1,p2,p3,p4;
                TRANSICIONES
                        t0: PRE p0;
                         POST p1,p2;
                        t1: PRE p3,p4;
                         POST p0;
                MARCADO
                        p0;
                TIEMPO
                        t0,t1;
                INTERFASE
                        ENTRADA p3,p4;
                        SALIDA p1,p2;
FINSUBRED;
SUBRED subred2;
                LUGARES p0,p1,p2;
                TRANSICIONES
                        t0: PRE p0;
                         POST p1;
                        t1: PRE p1;
                         POST p2;
                TIEMPO t0,t1;
                INTERFASE
                        ENTRADA p0;
                        SALIDA p2;
FINSUBRED;
SINCRONIZACION
```

```
(subred1,p2,p0) <= (p1global,p3global)
(subred0,p3,p2,p1,p4) <= (p1global,p2global,p3global,p4global)
(subred2,p0,p2) <= (p2global,p4global)
FINRED;
```

A continuación, se muestra sólo los métodos principales del código para generar la especificación textual de una componente agregada:

```
/**
 * Obtiene la representación textual de la red a partir de las representaciones
* textuales de las subredes componente y las conexiones
* @return   Cadena que representa la red.
 */
public String getTxtNet(){
        StringBuilder sb = new StringBuilder();
        boolean sinSincronizacion = false;//Variable para saber si hay distribucion
        try{
            PrintWriter salida = new PrintWriter(new FileWriter(SubnetTxt));
            sb.append("RED ").append(this.getName()).append(";\n");

     // Si la red no tiene puertos, no se pone la sincronización y se escribe tal cual
     // Esto sirve para cuando se convierte una red distribuida en una centralizada
            for(IOComponentInterface iod : this.getComponents()){
                // Por cada componente de la red
                if(iod.outputPorts().isEmpty()) // Si no tiene puertos de salida
                    sinSincronizacion = true;
                sb.append(iod.generarFichero());//Se añade el contenido de componente
            }
            // Si no hay sincronización, se finaliza y termina
            if(sinSincronizacion){
                // Se termina de definir la red
                sb.append("FINRED;");
                salida.write(sb.toString());
                salida.close();
                return sb.toString();
            }

            // Queda poner las sincronizaciones
            sb.append("\nSINCRONIZACION\n");
            ComponentIterator componentIterator=getComponents().cIterator();
            FusionPlaces fusion = new FusionPlaces(getComponents().cIterator());

            int idConexion=1;
            while(componentIterator.hasNext()){
                IOComponentInterface component= componentIterator.nextComponent();
                Iterator outPortsIterator =component.outputPorts().iterator();
                while (outPortsIterator.hasNext()) {
                    String puerto = outPortsIterator.next().toString();
                    String nombreFusionPlace="global"+idConexion;
                    String[][] conexiones = getCoupling(component.getName(), puerto);
                    for (int i=0; i<conexiones.length;i++){
                        fusion.add(component.getName(), puerto, nombreFusionPlace);
                        fusion.add(conexiones[i][0], conexiones[i][1],
                                nombreFusionPlace);
                        idConexion++;
                    }
                }
            }
            sb.append(fusion.toString(null));
            // Se termina de definir la red
            sb.append("FINRED;");
            salida.write(sb.toString());
            salida.close();
        }catch(Exception e){
          e.printStackTrace();
        }
        return sb.toString();
    }
```

```
    /**
     * getCoupling: Devuelve vector con pares de cadenas componente, puerto a los que
     * está conectado
     *              el puerto de salida
     * @param subnet Nombre de la subred
     * @param port   Nombre del puerto de salida
     * @return       Matriz de cadenas, tantas filas como conexiones, primera columna
     * nombre subred, segunda columna nombre puerto entrada
     */
    public String [][] getCoupling(String subnet, String port){


       Set<Object> s = cp.translate(subnet, port);
       String [][] result = new String [s.size()][2];
       s.forEach((Object o) -> {
           try
           {   int i=0;

               @SuppressWarnings("unchecked")
               Pair<Object, Object> p = (Pair<Object, Object>) o;
               result[i][0]=p.getKey().toString();
               result[i][1]=p.getValue().toString();
              // System.out.println("Clave:"+p.getKey()+ " valor:"+p.getValue());
               i++;
           }
           catch (ClassCastException e)
           {
               e.printStackTrace();
           }
       });

       return result;
     }
```

```
package componentes;

import java.util.ArrayList;

public class FusionPlaces {
         public ArrayList<String[]> fusionList = new ArrayList<String[]>();
         private ComponentIterator componentIterator;

         public FusionPlaces(ComponentIterator componentIterator){
                   this.componentIterator=componentIterator;
         }

         public void add(String Component, String localPlace, String globalPlace){
                   String[] data = new String[4];
                   data[0]=Component;
                   data[1]=localPlace;
                   data[2]=globalPlace;
                   data[3] = "False"; // Indica si ha cambiado de nombre
                   fusionList.add(data);
         }

         /**
          * Cambia el nombre de la red en la lista de nombreviejo a nombrenuevo
          * @param nombreViejo
          * @param nombreNuevo
          */
         public void cambiarNombre(String nombreViejo, String nombreNuevo){
                   if(!nombreNuevo.contentEquals(nombreViejo)){
                            for(String[] redes : fusionList){
                                     // Si coincide nombre y no ha sido cambiado
                                     if(redes[0].contentEquals(nombreViejo) &&
                                        redes[3].contentEquals("False")){
                                               redes[0] = nombreNuevo;
                                               redes[3] = "True";
                                     }
                            }
                   }
         }
```

```
        public String toString(ComponentIterator prueba){
                String local ="";
                String global="";
                String lineas="";

                //System.out.println("imprimiendo fusion");

                 while(componentIterator.hasNext()){
                          // Por cada componente una línea de fusiones
                          String component=
                                  componentIterator.nextComponent().getName();
                          //cuenta veces en lista / cuantas fusiones de lugares
                          int cont=0;
                          for (int i=0; i<fusionList.size(); i++ ){
                                   if(component.equals(fusionList.get(i)[0]))
                                       cont++;
                          }
                          //construye cadena
                         // línea si componente tiene conexiones
                          if (cont>0){
                                   int cont2=0;
                                   local="("+component+",";
                                   global = " <= (";
                                   for (int i=0; i<fusionList.size(); i++ ){

                                      if(component.equals(fusionList.get(i)[0])) {
                                          cont2++;
                                          if(cont2<cont) {
                                             local+=fusionList.get(i)[1]+",";
                                              global+=fusionList.get(i)[2]+",";
                                          }
                                          else {
                                             local+=fusionList.get(i)[1]+")";
                                             global+=fusionList.get(i)[2]+")\n";
                                          }
                                       }
                                   }
                                   lineas+=local+global;
                          }
                 }
                return lineas;
        }

}
```

11.14. Cena de los filósofos

El **problema de la cena de los filósofos** es un problema típico de las ciencias de la computación propuesto por ***Edsger Dijkstra*** para la representación del problema de la sincronización de procesos.

Problema

N filósofos se sientan en una mesa redonda con cuencos de espaguetis. Se colocan tenedores entre cada par de filósofos adyacentes.

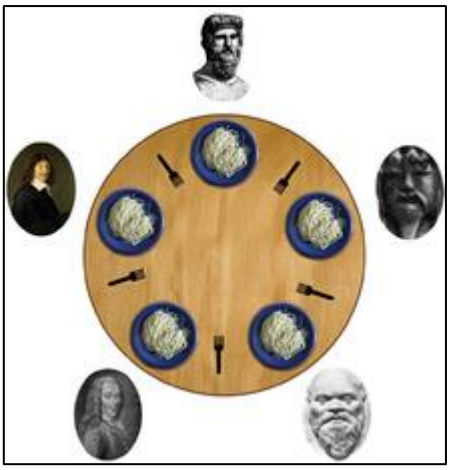


Figura 44: Ejemplo gráfico de la cena de los filósofos Fuente: [2]

Cada filosofo debe pensar y comer de manera alternativa. Sin embargo, un filósofo solo puede comer espaguetis cuando tiene los tenedores izquierdo y derecho. Cada tenedor puede ser sostenido por un solo filósofo, por lo que un filósofo puede usar un tenedor si y solo si no lo está usando otro filosofo.

Una vez que el filósofo termina de comer, debe dejar ambos tenedores para que los demás tenedores estén disponibles para el resto de los filósofos.

La comida no está limitada, supone una oferta infinita.

El problema es como diseñar una disciplina de comportamiento (algoritmo concurrente) de manera que ningún filosofo se muera de hambre.

Dicho problema se puede modelar a través de una red de Petri como se puede ver en el siguiente ejemplo:

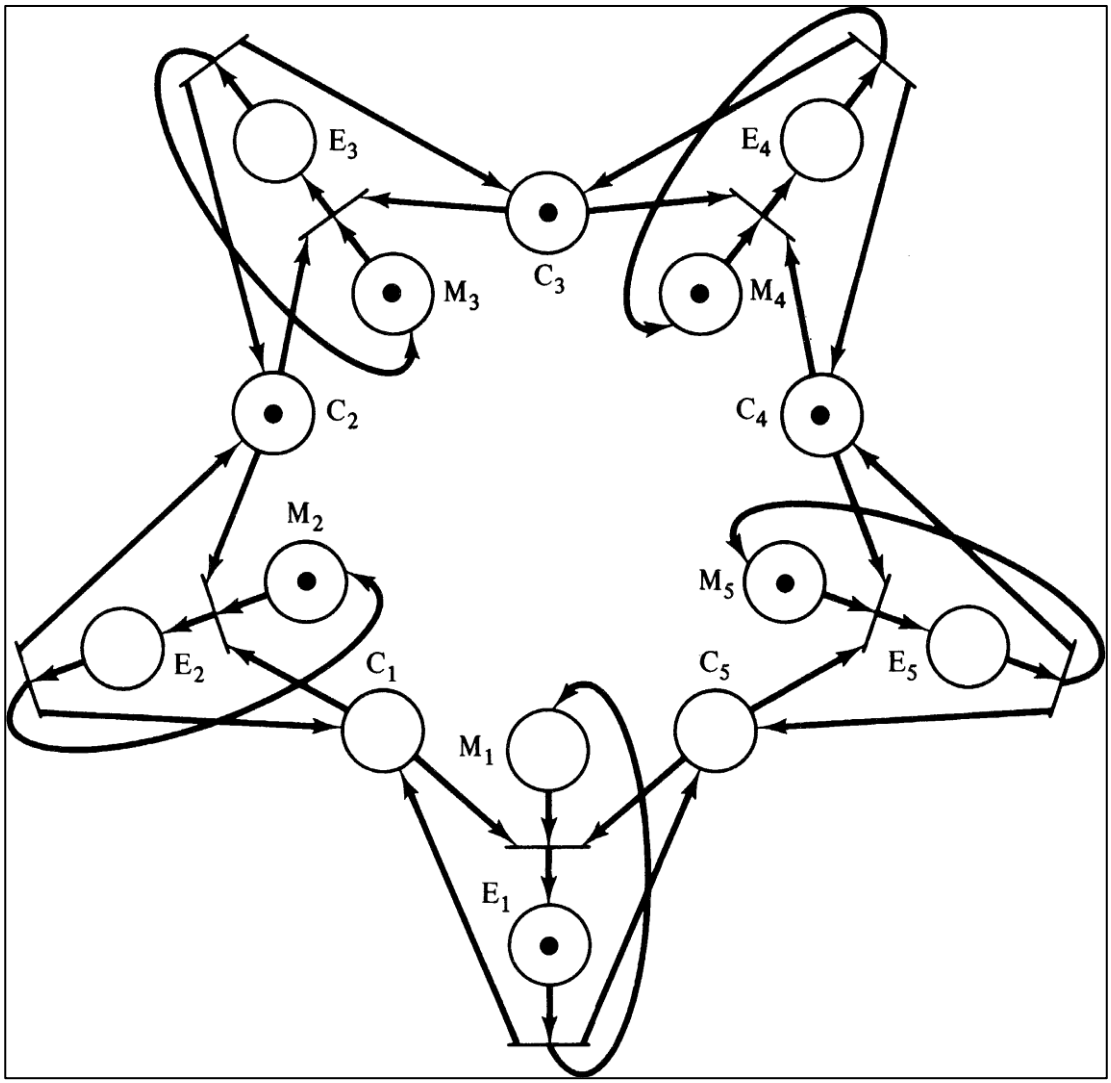


Figura 45: Problema de la cena de 5 filósofos en una RdP Fuente: [19]

Los lugares de C1 a C5 representan los tenedores e inicialmente están libres y por tanto contienen una marca. Cada filosofo se ha representado mediante dos sitios, M_i y E_i representando los estados de pensar y de comer respectivamente. Para que un filósofo deje de pensar y se ponga a comer debe poder obtener los dos tenedores, uno en la izquierda y otro en la derecha.

11.15. Profiling

Se ha usado la herramienta de *Java Profiling* integrada en el IDE *IntelliJ*. Para ello, se ha lanzado la simulación con el profiling activo obteniendo los siguientes resultados:

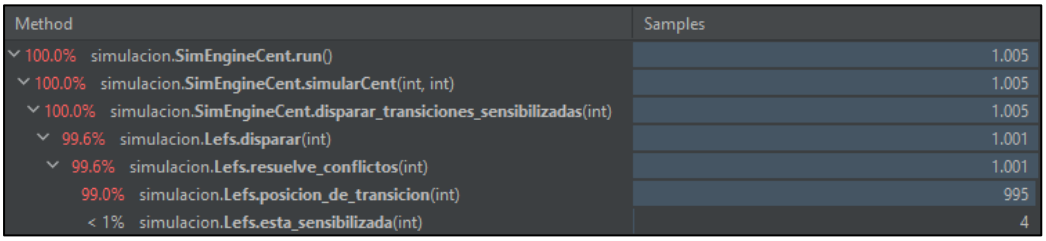


Figura 44: Resultado de profiling

En esta imagen se observa las funciones más costosas de manera compuesta, de manera que las funciones superiores están compuestas o integradas por las funciones inferiores. Con esto se puede ver que la función simple que más veces es llamada, y más tiempo de CPU conlleva es la función **posición_de_transicion(int).** Esta función devuelve el índice de la transición que se le pasa por parámetro. En dicha función se debe recorrer todo el vector de transiciones que hay disponibles y por tanto es objeto de ser mejorado a una estructura de datos óptima.