Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

▪ Proyecto Fin de Grado ▪

Ingeniería de Computadores

Modelado de sistemas concurrentes con Omnet++

Autor: Ignacio Garbizu

Director: F. Xabier Albizuri

Febrero 2015

informatika fakultatea
facultad de informática

Resumen

El objetivo del siguiente proyecto es proporcionar una herramienta para estudiar el rendimiento de sistemas multiprocesador. Para ello estos sistemas serán previamente modelados como redes de Petri y simulados mediante el entorno de simulación de sistemas de eventos discretos OMNeT++.

Las redes de Petri son grafos dirigidos formados por dos tipos de elementos: Lugares (Places) y Transiciones (Transitions). Sirven para la representación de eventos discretos y serán muy útiles a la hora de representar la sincronización de procesos en sistemas concurrentes.

OMNeT++ es un entorno para la simulación de sistemas de eventos discretos. Puede ser usado para desarrollar simuladores de redes de comunicación, redes de colas, modelos de protocolos, modelos de multiprocesador o arquitecturas hardware. Está orientado a objetos y hace uso del lenguaje NED, para la descripción de la estructura de los módulos que componen el modelo, y C++ para la programación de la lógica de éstos.

En este proyecto se han implementado en OMNeT++ los módulos Place y Transition (Inmediatas y Temporales) mediante los cuales se podrá construir cualquier topología de red de Petri. Además se podrán obtener datos estadísticos de éstos. Esto será especialmente útil en el estudio del rendimiento de sistemas informáticos, y en el estudio del rendimiento del sistema multiprocesador de este proyecto.

Índice

Lista de Figuras

Ubicación del proyecto

El código del proyecto se encuentra disponible en la siguiente dirección web:

http://www.sc.ehu.es/ccwalirx/usr/15IgnacioGarbizu.zip

1 Antecedentes.

El interés inicial de este proyecto era trabajar con el entorno de simulación OMNeT++ y aportar a éste módulos para la simulación de sistemas concurrentes. Al ser un entorno para la simulación de eventos discretos ofrecía bastantes posibilidades de desarrollo. Como la mayoría de los proyectos y ejemplos de muestra desarrollados eran redes de comunicaciones, se planteó el desarrollo de librerías para poder simular redes de Petri.
Este desarrollo estaba motivado por tres aspectos:

- Las posibilidades que este tipo de redes ofrecen en el modelado de sistemas concurrentes.
- No se han encontrado librerías ya desarrolladas para este tipo de redes en OMNeT++. Para ver los modelos desarrollados hasta la fecha consultar el catálogo de modelos de: http://www.omnetpp.org/models/catalog
- La mayoría de las implementaciones de redes de Petri en otros entornos o lenguajes estaban basadas en redes de Petri clásicas y no en redes de Petri temporales

Finalmente también había interés en modelar el funcionamiento de un multiprocesador. Éste no ha sido modelado directamente, sino que se han usado librerías desarrolladas de las redes de Petri en OMNeT++ para modelarlo ya que previamente había sido modelado como red de Petri.

2 Planificación.

2.1 Objetivos.

Si bien el objetivo final de este proyecto será desarrollar una extensión en OMNeT++ con el fin de modelar, simular y evaluar el rendimiento de sistemas concurrentes, se han definido los siguientes objetivos a completar durante el desarrollo del mismo:

- Aprender a usar el entorno de simulación OMNeT++ y descubrir sus posibilidades.
- Repasar y aplicar conceptos de la evaluación de sistemas informáticos.
- Estudiar las aplicaciones de las redes de Petri temporales para el modelado de sistemas concurrentes representados por modelos formales.
- Desarrollar módulos en OMNeT++ que simulen los componentes de este tipo de redes.

2.2 Fases en el desarrollo.

El desarrollo del proyecto se dividirá en distintas fases o etapas a completar:

- **Planificación:** En esta fase se determinarán los objetivos a cumplir, se identificarán las fases del proyecto, se ideará un plan de comunicación con el director y se elaborará un plan de riesgos. No se ha realizado una estimación en horas de las tareas a realizar.
- **Aprendizaje y estudio:** En esta fase se repasarán los conceptos aprendidos sobre las redes de Petri y la evalución del rendimiento de sistemas informásticos. También se estudiará el manual de OMNeT++ y sus ejemplos de muestra.
- **Desarrollo:** En esta fase se desarrollarán los módulos que simularán el comportamiento de los componentes de las redes de Petri.
- **Pruebas:** En esta fase se realizarán las pruebas. Para ello se simularán distintas topologías de redes de Petri que usen los módulos que hemos desarrollado. Además, también se realizará la evaluación del rendimiento del multiprocesador que se ha modelado.
- **Tareas de seguimiento y control:** Se ha llevado un seguimiento del desarrollo del proyecto en un cuaderno dónde se han ido anotando las explicaciónes de algunos conceptos de la teoría, el progeso en el desarrollo y lo que se ha ido haciendo cada día. Estas anotaciones han sido usadas en la redacción de esta memoria.

En el siguiente gráfico aparecerán organizadas las fases y subfases de las que se compone el desarrollo del proyecto:

Estructura de Desglose de trabajo (EDT)

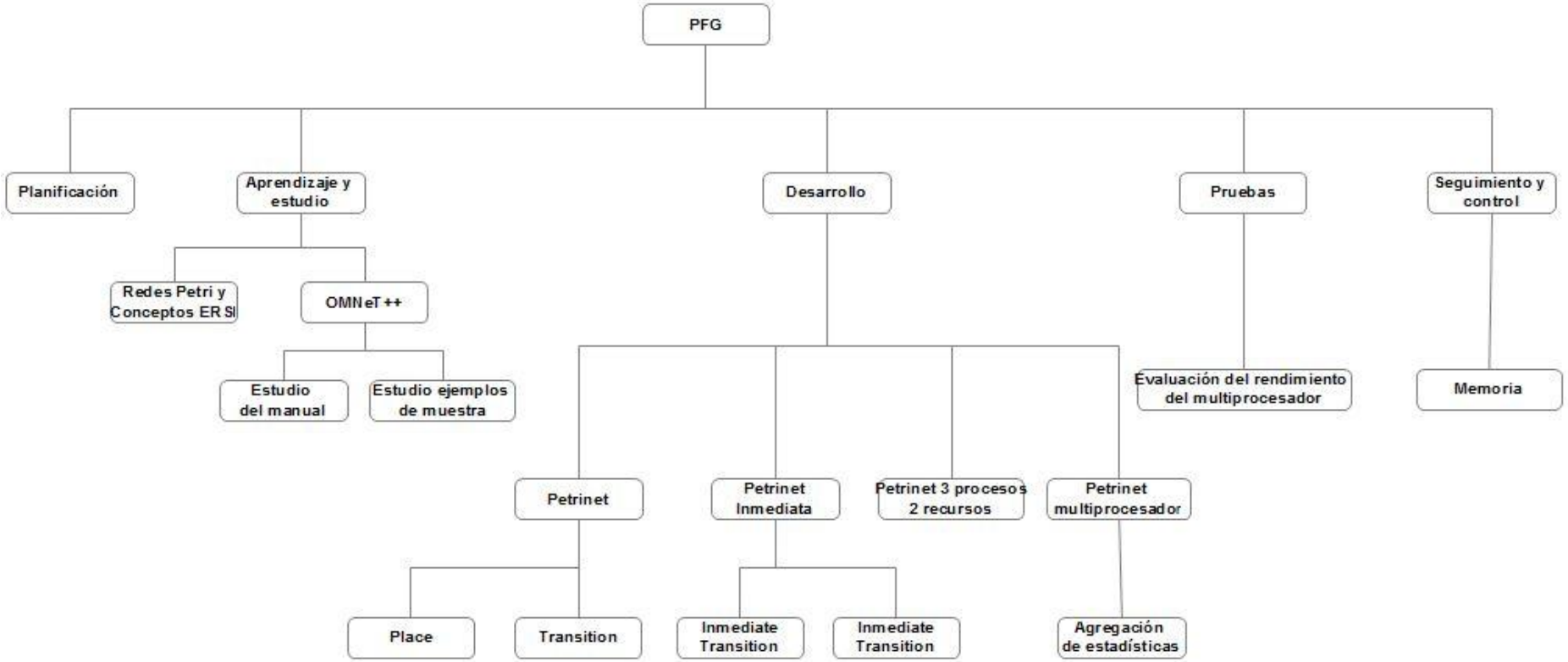

Ilustración 1 - EDT

2.3 Plan de comunicación.

La comunicación con el director de proyecto se ha planificado de la siguiente manera:

- Se realizarán reuniones cada dos semanas en las fases iniciales del proyecto, mientras que en las fases avanzadas del desarrollo y durante la redacción de la memoria serán al menos semanales. En estas reuniones se tratarán los avances en el proyecto y las dudas que han podido surgir sobre el proyecto y sus conceptos teóricos. Además, también se establecerán nuevos hitos en el proyecto y el día de la próxima reunión.

- Las dudas puntuales se podrán comunicar por correo electrónico o tratar en las reuniones del proyecto.

- Los entregables acordados se enviaran por correo electrónico con al menos 24 horas de antelación a la reunión.

- En caso de no poder asistir a una reunión de proyecto, el alumno avisará al director con 24 horas de antelación a la misma

2.4 Plan de riesgos.

Se han detectado los siguientes riesgos:

- **Riesgo nº1:** Posible rotura del equipo donde se realiza el proyecto. Aunque su probabilidad sea baja, el impacto en el avance del proyecto puede ser alto.

- **Riesgo nº2:** Pérdida del código del proyecto o de partes del mismo. Su probabilidad es baja y su impacto en el desarrollo puede ser alto.

- **Riesgo nº3:** Errores en la lógica debidos a modificaciones incorrectas que han podido pasar desapercibidas. La probabilidad de que ocurra es media – baja, mientras que su impacto en el desarrollo del proyecto será medio.

- **Riesgo nº4:** Pérdida de apuntes y documentos ligados al seguimiento del desarrollo del proyecto. La probabilidad de que ocurra es baja y su impacto será medio.

Para cada riesgo se han preparado los siguientes planes de contingencia:

- **Riesgo nº1:** Guardar los avances del proyecto en la nube usando un servicio de alojamiento de archivos para ello. Tener instalado OMNeT++ en otro equipo y comprobar su buen funcionamiento en éste, para que en caso de tener que usarlo funcione adecuadamente.

- **Riesgo nº2:** Después de cada sesión de trabajo, los avances se guardarán en la nube.

- **Riesgo nº3:** Se conservarán al menos las quince últimas versiones del proyecto guardadas en la nube.

- **Riesgo nº4:**Se extraerá lo más significativo de los apuntes del cuaderno usado para el seguimiento, se redactará y se almacenará en la nube junto a las copias del proyecto.

2.5 Diagrama de Gantt.

En este apartado se puede ver como se ha distribuído el tiempo de trabajo en la realización de las distintas tareas del proyecto. Para ello se ha usado un diagrama Gantt.

A diferencia del gráfico de la Estructura de Desglose de Trabajo, en este diagrama de Gantt las tareas estarán organizadas por entregables u objetivos a cumplir.

Nótese que ciertas tareas como el aprendizaje y en estudio de OMNeT++ se han prolongado durante todo el desarrollo del proyecto. Esto es debido a que ha habido que consultar permanentemente información en el manual de la herramienta.

Diagrama de Gantt.

(Tareas organizadas por entregables)

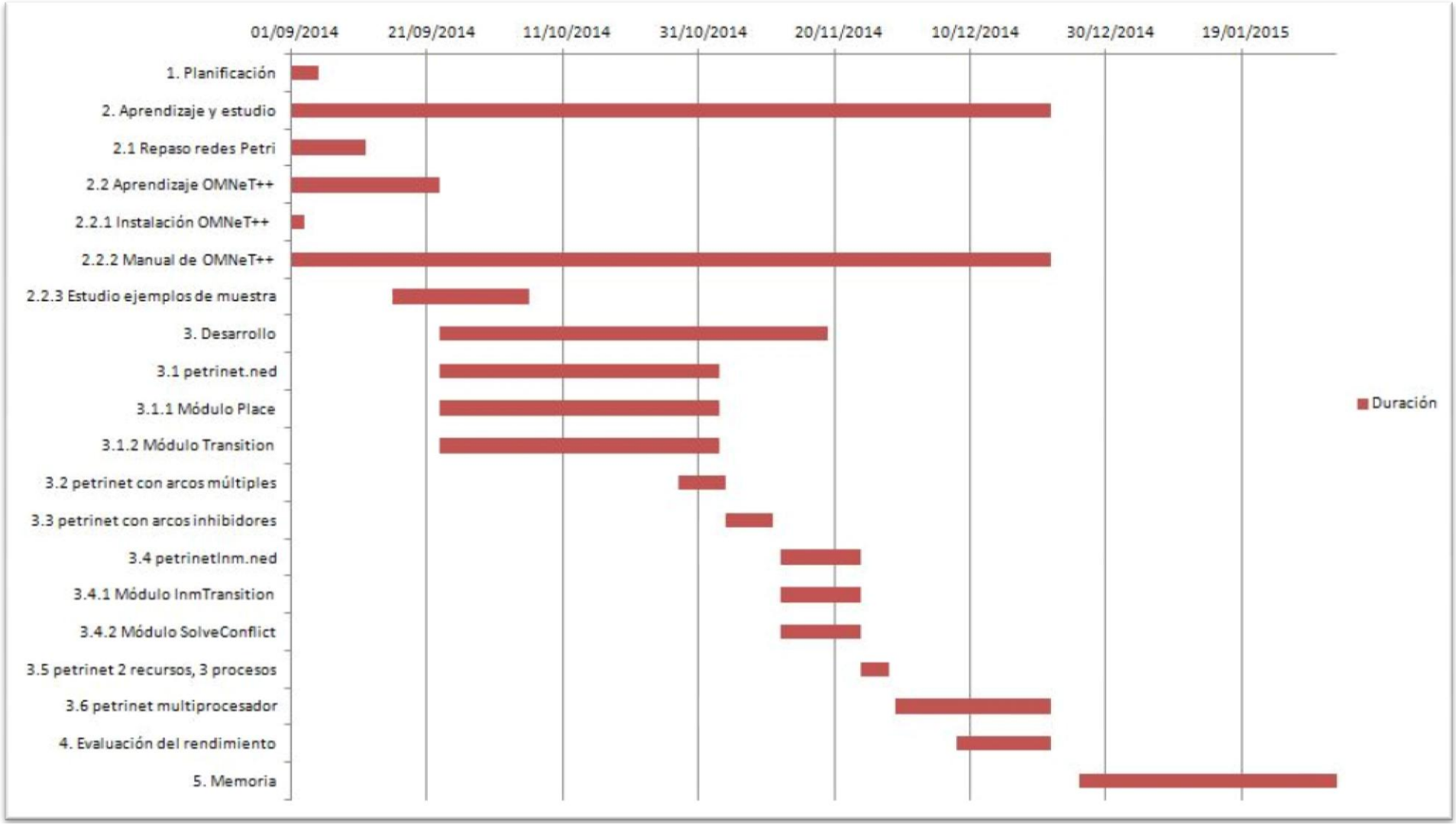

Ilustración 2 - Diagrama de Gantt

3 Introducción a la evaluación del rendimiento en sistemas informáticos.

3.1 Tipos de medidas de evaluación del rendimiento.

En los sistemas informáticos se evalúa el rendimiento con el objetivo de saber si son fiables, si son compatibles con una determinada aplicación, si son eficientes ejecutándola, etc.

En general, las medidas de evaluación de rendimiento intentarán medir tres aspectos:

- Rapidez del sistema a la hora de terminar de ejecutar una tarea.
- Efectividad del sistema a la hora de usar los recursos disponibles.
- Cómo responde el sistema ante fallos.

Las medidas de evaluación del rendimiento se pueden clasificar en los siguientes tipos:

- **Rapidez de respuesta:** Se trata de la rapidez con la que el sistema completa la ejecución de una determinada tarea dada. En esta categoría encontraremos medidas como el tiempo de espera, tiempo de procesado y longitud de una cola.

- **Nivel de uso:** Las medidas de este tipo evaluarán si están siendo bien usados los componentes del sistema. En esta categoría de medidas entrarían la productividad (throughtput) y la utilización. Esta categoría puede entrar en conflicto con la anterior.

- **Compleción de la misión:** Esta categoría mide si el sistema será capaz de completar la ejecución de una tarea. Son medidas como la duración de la ejecución de la tarea en el sistema o el tiempo de disponibilidad del sistema para la ejecución de la tarea. Serán de utilidad para detectar comportamientos imprevistos en la ejecución de la tarea.

- **Fiabilidad:** Medirán la fiabilidad del sistema ante periodos largos de tiempo. Se trata de fallos previsibles. En esta categoría entrarían medidas como el número de fallos en un periodo de tiempo o el coste de los fallos.

- **Productividad del usuario:** Medirán la efectividad con la que un usuario podrá completar su trabajo. Se trata de medidas como la amigabilidad del sistema o productividad del trabajo de usuario.

En este proyecto se usarán las redes de Petri para representar situaciones en las que hacer uso de las medidas de tiempo de respuesta y de nivel de uso. De todas formas las redes de Petri también podrán ser útiles para representar situaciones en las que interesa medir la fiabilidad del sistema o la capacidad de completar una misión o tarea.

3.2 Tipos de medidas y sistemas informáticos.

La importancia de los tipos de medidas dependerá del sistema informático y de la aplicación que se vaya a usar. A continuación veremos los tipos de medidas que se tendrán en cuenta dependiendo del sistema informático:

- **Propósito general:** Están diseñados para ofrecer una funcionalidad general. En este tipo de sistemas serán importantes las medidas de tiempo de respuesta, nivel de uso y de productividad. Las medidas de fiabilidad no serán tan importantes.

- **Alta disponibilidad:** Están diseñados para dar soporte a un gran número de transacciones. Se utilizan en bancos, compañías aéreas, etc. En este tipo de sistemas importarán las medidas de tiempo de respuesta y de fiabilidad. La productividad del usuario también será importante.

- **Tiempo real:** Están diseñados para dar respuesta dentro de una limitación de tiempo determinada. En este tipo de sistemas serán de vital importancia las medidas de tiempo de respuesta y la fiabilidad

- **Orientados a cumplir una tarea:** Requieren niveles muy elevados de fiabilidad de cara a cumplir bien con su objetivo. Serán importantes las medidas de capacidad de completar la tarea o misión. Las medidas de tiempo de respuesta no serán muy importantes en este tipo de sistemas.

- **"Vida larga":** Se trata de sistemas como los usados por los satélites o sondas. No podrán ser accedidos para realizar reparaciones durante largos periodos de tiempo. Las medidas de fiabilidad serán importantes en este tipo de sistemas.

3. 3 Técnicas de evaluación del rendimiento.

Existen tres técnicas básicas para la evaluación del rendimiento de los sistemas informáticos:

- Medición.
- Simulación.
- Modelado analítico.

3.3.1 Medición.

Es la técnica básica de la evaluación del rendimiento. Servirá también para calibrar los sistemas reales diseñados a partir de la simulación y del modelado analítico. Se puede llevar a cabo por software o por hardware. Como al hacer una medición hay factores que no

pueden ser del todo controlados, se usarán técnicas estadísticas para analizar la información obtenida y sacar conclusiones.

3.3.2 Simulación.

Se construirá un modelo que represente el funcionamiento de un sistema y se simulará. Ofrece una gran flexibilidad en la simulación de distintos comportamientos del sistema. Sin embargo habrá que tener en cuenta el nivel de detalle del sistema que se quiere modelar, los datos de salida a analizar estadísticamente y el tipo de experimento a realizar para que sea factible. A la hora de sacar conclusiones, puede ser necesario usar técnicas estadísticas para analizar la información obtenida.

De entre todos los tipos de simulación que existen nos centraremos en la simulación de eventos discretos.

3.3.2.1 Simulación de eventos discretos.

Este tipo de simulaciones modelará los eventos un sistema como una secuencia discreta de eventos ordenados por tiempo. Cada evento ocurrirá en un determinado instante y producirá un cambio de estado en el sistema, y se asumirá que entre eventos consecutivos no ocurrirán cambios de estado. De esta forma la simulación podrá avanzar o dar saltos en el en tiempo directamente de un evento al siguiente.

Un ejemplo de evento discreto es la adquisición de un recurso o la finalización de un servicio. El simulador manejará una lista de eventos futuros que se irá actualizando al procesar secuencialmente los eventos.

En el caso de OMNeT++ veremos que los eventos de la simulación serán representados como mensajes entre el mismo o distintos módulos. Estos se manejarán y ordenarán en una estructura de datos llamada FES (Future Event Set).

3.3.2.2 Tipos de eventos discretos.

Existen dos tipos: los eventos discretos deterministas y no deterministas. Para hacer la simulación de un sistema necesitaremos generar eventos no deterministas y para ello habrá que generar números aleatorios. Esto se puede hacer siguiendo una distribución de probabilidad y generando números a partir de esta mediante un algoritmo de generación de números aleatorios, o también a partir de los datos de una medición del sistema real.

3.3.2 Modelado analítico.

Se construirá un modelo matemático que refleje el comportamiento del sistema con cierto nivel de detalle. Este modelo estará formado por ecuaciones basadas en fórmulas probabilísticas. Normalmente se usará para representaciones del sistema que no exijan gran

cantidad de detalle, ya que en ese caso se usará la simulación como herramienta de modelado.
A diferencia de la simulación y de la medición no será necesario analizar estadísticamente los datos de salida ya que no habrá incertidumbre en ese aspecto, pero habrá que fijarse en el nivel de detalle con el que se ha modelado el sistema.

En el caso de este proyecto, habrá una parte del sistema descrita por una fórmula probabilística. Esta parte se trata del cálculo de tiempo aleatorio siguiendo una determinada distribución en las transiciones temporales o con retardo temporal.

3.4 Aplicación de las técnicas de evaluación del rendimiento en sistemas informáticos.

A continuación citaremos las aplicaciones más destacadas de las técnicas de la evaluación del rendimiento en los sistemas informáticos:

- **Diseño de sistemas:** En las fases iniciales del diseño de un nuevo sistema además de realizarse un diseño básico de la arquitectura se establecen los objetivos de rendimiento y fiabilidad. Para alcanzar estos objetivos habrá que construir un modelo que describa el funcionamiento del sistema y evaluarlo usando éstas técnicas. En fases más avanzadas del diseño la simulación se convertirá en una herramienta esencial a la hora de tomar decisiones y de evitar fallos. En una fase inicial se puede usar el modelado analítico para examinar diversas opciones de diseño con modelos simples del sistema.

- **Elección de sistema:** Las técnicas de evaluación del rendimiento serán útiles a la hora de elegir el sistema que más se adapte a nuestros objetivos de compatibilidad, disponibilidad, tiempo de respuesta, etc. La mejor técnica de evaluación del rendimiento para llevar a cabo dicha elección es la medición, aunque ésta no siempre será posible, por lo que si no es así, nos tendremos que basar en los datos que poseamos del sistema o en un modelo simple de este.

- **Actualización de sistemas:** La actualización puede implicar el cambiar partes del sistema con el objetivo de que sea compatible con una nueva parte. Se buscará alcanzar nuevos objetivos de rendimiento, compatibilidad, etc. Aunque el modelado analítico puede ser de utilidad, en sistemas que tengan interacciones complejas entre subsistemas será necesario el uso de la simulación como técnica de evaluación del rendimiento.

- **Puesta a punto de sistemas:** En la puesta a punto se buscará optimizar el rendimiento del sistema cambiando ciertas políticas de gestión de recursos, el mecanismo de la planificación de procesos, tamaño del almacenamiento en buffer, etc. La experimentación y medición de los resultados obtenidos es el método más simple para llevar a cabo la puesta a punto, aunque no siempre será posible. Se recomienda el uso de la simulación como técnica de evaluación del rendimiento ya que el uso del modelado analítico puede ser más dificultoso.

- **Análisis de sistemas:** En el análisis de sistemas se buscaran las razones de un funcionamiento lento o inadecuado del sistema. Esto puede ser debido a que el sistema tenga hardware inadecuado o a una mala gestión de los recursos del sistema. Este proceso será necesario para determinar si se quiere hacer una actualización o puesta a punto del sistema.

El método más sencillo para evaluar el sistema será la experimentación y medición de los resultados. Sin embargo en casos en los que haya interacciones complejas en el sistema la simulación será necesaria.

3.5 Redes de colas.

Se tratan de sistemas en los que los existen una o varias estaciones o colas en las que se van acumulando objetos, a los que se irá dando servicio. Estos objetos pueden ser clientes, peticiones, trabajos, etc.

Existen dos tipos de redes de colas: las abiertas y las cerradas. En las redes de colas abiertas los objetos o clientes llegarán y se marcharán, mientras que en las cerradas el número de clientes siempre será constante.

Hay conceptos de las redes de colas aplicables a ciertas partes de las redes de Petri. Estas partes tienen un comportamiento similar a las estaciones de las redes de colas y en ellas podrán medirse los parámetros de rendimiento y aplicarse las Leyes de la utilización y de Little.

3.5.1 Parámetros de rendimiento de una red de colas.

Las redes de colas estarán compuestas por estaciones, que en este caso funcionarán como colas con servidores que van dando servicio a los clientes que se van acumulando en ellas. Para cada estación i en una red de colas tendremos los siguientes parámetros:

- **λi**: Es la tasa de llegadas a la estación y de salidas. Se le denomina throughput o productividad de la estación.

- **Si**: Es la esperanza del tiempo que un cliente permanecerá en una estación tomando servicio.

- **Qi:** Es el promedio de la longitud de la cola de clientes en la estación.

- **Wi**: Es la esperanza del tiempo que estará un cliente en la cola de una estación o tiempo de respuesta, incluyendo el servicio.

- **Ui**: Es el promedio de la fracción de tiempo en que la estación ofrece servicio o la utilización de la estación.

3.5.2 Ley de Little.

Esta Ley es aplicable al sistema entero o a los subsistemas (serían las estaciones) del sistema.

Tendremos un sistema al que llegan aleatoriamente clientes de fuera, permanecerán un tiempo aleatorio dentro y después cada cliente saldrá del mismo.

La fórmula de la Ley de Little es la siguiente:

$$L = \lambda W$$

- **W:** Es la esperanza del tiempo que los clientes permanecen dentro del sistema.

- **λ**: Es la productividad del sistema, es decir, la tasa de clientes que entran al sistema igual a la tasa de clientes que salen, para un sistema estable.

- **L o Q**: Es el promedio temporal del número de clientes en un sistema. Si la Ley la hemos aplicado a un subsistema y queremos saber el promedio temporal del número de clientes fuera de este, si N es el número total de clientes totales del sistema habrá que calcular N – L.

3.5.3 Ley de utilización.

A diferencia de la Ley de Little es aplicable a una única estación con un único servidor.

La fórmula de la Ley de la utilización es la siguiente:

$$U = \lambda S$$

- **U o utilización:** Es trata de la fracción de tiempo que la estación está dando servicio a los clientes.

- **λ:** Es la productividad de la estación. Se calculará haciendo el promedio temporal del número de los servicios dados a los clientes.

- **S:** Es la esperanza del tiempo de servicio de los clientes en la estación.

3.6 Redes de Petri.

Se trata de un lenguaje de modelado matemático para la descripción de sistemas. Usa grafos dirigidos y compuestos por dos tipos de elementos: lugares (Places) y transiciones (Transitions). Se definen formalmente como una cuádrupla (P,T,I,O), donde P es el conjunto de lugares o puestos, T es el conjunto de transiciones, I c P x T son todas las entradas de cada transición, y O c T x P todas las salidas de cada transición.

En las redes de colas se puede representar la disputa de recursos, pero no será posible representar la sincronización de varias actividades en sistemas concurrentes. Las redes de Petri se usarán para realizar esta representación y además permitirán un mayor nivel de abstracción a la hora de representar un modelo.

3.6.1 Componentes de las redes de Petri.

Como ya hemos dicho, las redes de Petri estarán compuestas por lugares y transiciones unidos por arcos. Funcionarán de la siguiente manera:

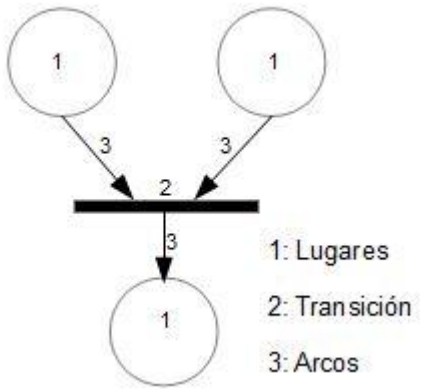

Ilustración 3 - Componentes red de Petri

3.6.1.1 Lugares.

Albergarán un número de marcas (tokens). Podrán comenzar con un número de marcas mayor o igual que 0. Las transiciones sabrán el número de marcas de sus lugares predecesores y en el caso de que se produzca un disparo, los lugares predecesores de cada transición perderán y los sucesores ganarán tokens.

3.6.1.2 Transiciones.

Si se produce un disparo consumirán marcas de los lugares predecesores y añadirán marcas a los lugares sucesores. Para que se dé la condición de disparo de una transición debe ocurrir que en todos sus lugares predecesores su cumpla que el número de marcas iguale al menos

la multiplicidad del arco correspondiente, o en el caso de que se trate de un arco inhibidor que no haya ninguna marca.

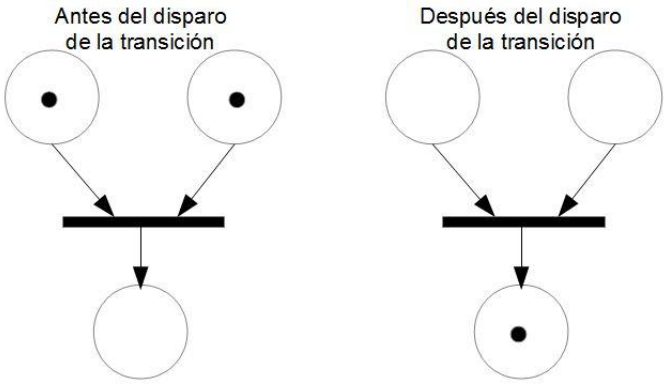

Ilustración 4 - Disparo de una transición

Además habrá dos tipos de transiciones, las temporales y las inmediatas. Cuando en las temporales se cumpla la condición de disparo se pondrá en marcha un tiempo de retardo. Si tras haberse cumplido ese tiempo aún se cumple la condición de disparo, la transición se disparará. En el caso de las transiciones inmediatas no habrá tiempo de retardo y se dispararán al cumplirse la condición de disparo.

3.6.1.3 Arcos.

En las redes de Petri existen dos tipos de arcos: los de entrada a la transición, que unen los lugares con las entradas de las transiciones y los de salida de la transición, que unen las salidas de las transiciones con los lugares.

En el caso de los arcos de entrada a la transición podrán tener multiplicidad mayor que uno (arcos múltiples) o ser inhibidores:

- **Multiplicidad:** Es un parámetro del arco significa que si hay n arcos de entrada en una transición procedentes de un mismo lugar se necesitarán k >= n marcas o tokens para que esa transición tenga el permiso de ese lugar para ser disparada.

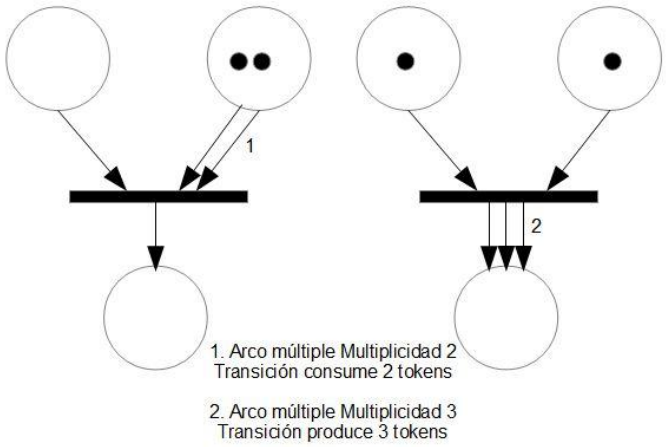

Ilustración 5 - Arcos múltiples

- **Arcos Inhibidores:** El objetivo de los arcos inhibidores consiste en impedir que una transición pueda dispararse mientras que haya marcas en los lugares precedentes. En el proyecto serán representados como arco con multiplicidad 0.

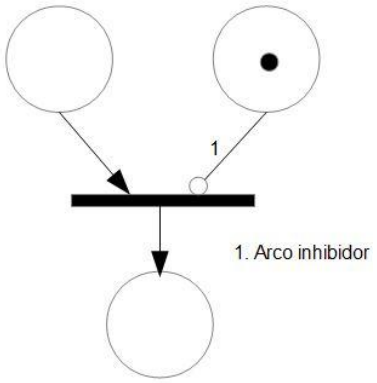

Ilustración 6 - Arco Inhibidor

3.6.2 Dinámica de las redes de Petri.

Cuando en una transición se dé la condición de disparo podrá dispararse. Al poderse habilitar varias transiciones en el mismo instante la dinámica de las redes de Petri será no determinista, es decir, habrá distintos casos de evoluciones posibles en el estado de estas redes.

Si varias transiciones se habilitan en el mismo instante surgirá un conflicto entre ellas a la hora de disparar que se resolverá de distinta manera si se trata de transiciones inmediatas o de transiciones temporales. Un vez resuelto el conflicto se disparará una de ellas, y en caso de que lugares precedentes no tengan marcas suficientes como para cumplir la condición de disparo, se deshabilitarán.

El criterio a seguir para la resolución de conflicto entre varias transiciones dependerá del tipo de transición:

- Si se trata de un conflicto entre transiciones temporales, la que tenga menor tiempo de retardo se disparará antes.

- Si el conflicto es entre transiciones inmediatas se elegirá de entre ellas la transición a disparar de manera aleatoria o por un criterio de prioridad.

- En caso de que el conflicto sea entre transiciones inmediatas y temporales, se dispararán antes las temporales ya que no tienen tiempo de retardo.

3.6.3 Redes de Petri clásicas y temporales.

Las redes de Petri temporales son una extensión que añade nuevas propiedades a las redes de Petri clásicas o "normales". Además de permitir modelar la estructura del modelo como las redes clásicas, permitirán modelar los tiempos asociados a los eventos discretos. Estos tiempos estarán asociados a los disparos de las transiciones y podrán ser constantes, indeterminados o probabilísticos.

Las distribuciones probabilísticas pueden ser usadas, por ejemplo, para generar tiempos aleatorios para las transiciones de las redes temporales.

3.6.4 Semánticas de disparo en la redes de Petri.

En las transiciones de las redes de Petri hay dos semánticas de disparo posibles: disparo atómico (AF) y disparo no atómico (NF). Ambas semánticas tienen la siguiente diferencia:

- En la semántica de disparo atómico si se cumple la condición de disparo de la transición y si transcurrido un tiempo t >= 0 se produce el disparo, se restarán las marcas de los lugares precedentes y se añadirán marcas a los lugares sucesores en el mismo instante.

- En la semántica de disparo atómico cuando en una transición se cumpla la condición de disparo se restarán las marcas de sus lugares precedentes, y tras transcurrir un tiempo t >= 0 se añadirán las marcas a los lugares sucesores

.
Aunque las redes con semántica NF sean más simples que las que tienen semántica AF habrá situaciones como la condición de carrera que no podrán representar.

El proyecto ha sido desarrollado usando la semántica de disparo atómico AF.

3.6.5 Semánticas de servidor en redes de Petri.

En una transición de una red de Petri puede ocurrir una habilitación múltiple, es decir, se podría habilitar dándose la condición de disparo simultáneamente varias veces si los lugares entrantes a la transición tienen un número de marcas k > = 2*multiplicidad del arco (en caso de que no sea un arco inhibidor).

Por ejemplo, si tuviéramos un lugar entrante a una transición con 5 marcas y un arco entre ellos con multiplicidad 0, esa transición podría ser habilitada 5 veces simultáneamente.

Dependiendo del tratamiento que se le dé en la red a la habilitación múltiple de una transición habrá dos tipos de semánticas:

- **Servidor único o SS (Single Server):** Un lugar entrante sólo podrá provocar la habilitación de la misma transición una vez. O visto de otra manera, las peticiones de habilitación de la transición serán atendidas por un único servidor.

- **Servidor de retraso o DS (Delay Server):** Un lugar entrante podrá provocar la habilitación simultánea de la misma transición tantas veces como k marcas tenga, donde k > = multiplicidad del arco. O visto de otra manera, las peticiones de habilitación de la transición serán atendidas por n servidores, donde n será el número de veces que se podrá habilitar.

El proyecto ha sido desarrollado usando la semántica de servidor único o SS.

4. OMNeT++.

4.1 Introducción.

Es un entorno para la simulación de eventos discretos. Es modular, es decir, los elementos de un sistema se dividirán en módulos, y además está orientado a objetos.

Gracias a su arquitectura genérica permitirá el modelado y la simulación de cualquier sistema, siempre y cuando éste se pueda modelar mediante eventos discretos y pueda ser representado mediante una red de módulos que se comunican entre sí intercambiando mensajes.

Será muy útil para el modelado de redes de comunicación, protocolos, redes de colas, multiprocesadores y sistemas distribuídos, etc.

Para la descripción y programación de los modelos usará dos lenguajes: el lenguaje NED y C++.

4.2 Instalación de OMNeT++.

OMNeT++ se encuentra disponible para las siguientes Sistemas Operativos: Windows (7, 8 y XP), Mac OS X (10.7, 10.8 y 10.9) y las siguientes distribuciones de Linux: Ubuntu, Fedora, Red Hat, y openSUSE.

4.2.1 Instalación de OMNeT++ en Windows 7.

Iremos a la web de OMNeT++, http://omnetpp.org y en el apartado de Downloads (descargas) buscaremos la versión más reciente de OMNeT++ para Windows. Para la versión 4.6 el archivo se llama omnetpp-4.6-src-windows.zip y en la web lo encontraremos en "OMNeT++ 4.6 win32 (source + IDE + MinGW, zip)". Lo descargamos.

Extraemos el contenido del .zip en el directorio donde queremos instalar OMNeT++. Es importante que el path o la ruta completa a este directorio no contengan ningún carácter de espacio (por ejemplo Archivos de programa).

Entramos en el directorio llamado omnetpp-4.6 e iniciaremos mingwenv.cmd con un doble click. Aparecerá una consola de MSYS bash shell con el contenido de la carpeta omnetpp-4.6/bin ya incluido en el path.

Miraremos el contenido del fichero configure.user del directorio omnetpp-4.6. Esto puede hacerse introduciendo el comando notepad configure.user en la consola que hemos abierto antes. En la mayoría de los casos no hará falta cambiar nada.

Introduciremos los siguientes comandos en la consola:

```
$  ./configure
$ make
```

Para verificar que la instalación se ha realizado correctamente podemos simular un ejemplo de omnetpp-4.6/samples. Por ejemplo ejecutaremos el ejemplo de fifo:

```
$ cd samples/fifo
$ ./fifo
```

Se podrá iniciar OMNeT++ bien escribiendo su comando en la consola: $ omnetpp o bien haciendo doble click en el ejecutable omnetpp.exe del directorio omnetpp-4.6/ide.

Nota: Si al hacer el build de un proyecto desde el IDE nos sale el mensaje: "Toolchain is not supported on this platform or installlation…" iremos a Project > Properties > C/C++ Build > Tool Chain Editor.

4.2.2 OMNeT++ en Linux.

Además de Ubuntu, Fedora, Red Hat, y openSUSE OMNeT++ también es compatible con las siguientes distribuciones:

- Derivadas de Ubuntu como Kubuntu, Mint, etc.
- Algunas derivadas de Debian como Knoppix o Mepis (Al estar Ubuntu basado en Debian se puede usar la misma forma de instalación que en Ubuntu).
- Derivadas de Fedora: Simplis y Eeedora.

Aunque existen formas de instalación específicas para Ubuntu, Fedora, Red Hat y openSUSE, en esta guía solo nos centraremos en la instalación específica en Ubuntu y en la instalación general en Linux.

4.2.3 Instalación en Ubuntu (Probado en Ubuntu 14.04.LTS).

1- En el botón Dash de la barra lateral buscamos "terminal" y lo abriremos.

2- Una vez en el terminal, actualizaremos el catálogo de paquetes disponibles:

```
$ sudo apt-get update
```

3- Para instalar los paquetes necesarios introducimos:

```
$ sudo apt-get install build-essential gcc g++ bison flex perl tcl-dev tk-dev libxml2-dev \ zlib1g-dev default-jre doxygen graphviz libwebkitgtk-1.0-0 openmpi-bin libopenmpi-dev \ libpcap-dev
```

Ante las preguntas que nos haga el terminal introduciremos S o Y (si nuestro sistema está en inglés).
Otra opción es instalar todos estos paquetes de manera gráfica con Synaptic. Para ello tendremos que instalarlo ya que en Ubuntu 14.04 LTS no viene instalado por defecto.

4- Para seguir con la instalación iremos al siguiente apartado: "Instalación en Linux (general)".

4.2.4 Instalación en Linux (general).

A continuación se describirá el proceso de instalación general de OMNeT++ para una distribución Linux con bash como shell:

1- Instalaremos los paquetes del compilador de C++ gcc y el de Java runtime. Dependiendo de la distribución Linux que se trate es posible que haya que instalar más paquetes adicionales. Para más información consultar: omnetpp.org/doc/omnetpp/InstallGuide.pdf.

2- Iremos al apartado "Downloads" de la web http://omnetpp.org y haremos click en "OMNeT++ 4.6 (source + IDE, tgz)". Descargaremos el archivo omnetpp-4.6-src.tgz.

3- Copiamos el archivo al directorio donde lo queremos instalar (se aconseja el directorio /home/<nuestro usuario>) y los extraemos en el con el siguiente comando:

```
$ tar xvfz omnetpp-4.6-src.tgz
```

4- Ajustaremos las variables de entorno. Desde el terminal nos dirigiremos al directorio omnetpp-4.6. Si lo hemos extraído en nuestro directorio HOME, $ cd omnetpp-4.6. Una vez estemos en el directorio, ejecutaremos el script setenv que añadirá a PATH el contenido del directorio /bin y a LD_LIBRARY_PATH el contenido del directorio /lib:

```
$ . setenv
```

Si queremos fijar las variables de entorno de forma permanente podemos editar el fichero .bashrc o .bashcon un editor de texto, por ejemplo gedit:

```
$ gedit ~/.bashrc
```

Se añadirán al final del fichero las siguientes líneas y se salvarán cambios:

```
export PATH=$PATH:$HOME/omnetpp-4.6/bin
export LD_LIBRARY_PATH=$omnetpp_root/lib:$LD_LIBRARY_PATH
```

5- Ejecutaremos el script configure del directorio omnetpp-4.6:

```
$ ./configure
```

Si nos encontramos en una sesión ssh cambiaremos el comando a:

```
$ NO_TCL=1 ./configure
```

En caso de que se produjese algún error en la operación anterior se inspeccionará el fichero config.log.

6- Introduciremos el comando make:

```
$ make
```

Este proceso podrá durar varios minutos.

7- Para verificar que la instalación se ha realizado correctamente podemos simular un ejemplo de omnetpp-4.6/samples. Por ejemplo ejecutaremos el ejemplo de fifo:

```
$ cd samples/fifo
$ ./fifo
```

8- Para iniciar el IDE de OMNeT++ simplemente introduciremos el siguiente comando en el terminal:

```
$omnetpp
```

Para acceder al IDE de OMNeT++ desde el lanzador de aplicaciones introduciremos el siguiente comando:

```
$ make install-menu-item
```

Y para crear un acceso directo en el escritorio:

```
$ make install-desktop-icon
```

4.3 El lenguaje NED.

Se trata de un lenguaje de descripción que servirá para describir la estructura del modelo de simulación entero y la estructura de cada módulo. Permite la declaración de los módulos simples y la agrupación de éstos en módulos compuestos. El lenguaje NED tendrá ciertas características que facilitarán su escalabilidad a grandes proyectos
:

- **Jerarquización:** Un módulo complejo podrá ser dividido en módulos simples y ser usado como un módulo compuesto.

- **Basado en componentes:** Tanto los módulos simples como los compuestos serán reusables, lo que permite la existencia de librerías.

- **Interfaces:** Las interfaces de los módulos y de los canales podrán ser usadas como placeholders en vez de usar un tipo de módulo o canal. El tipo del módulo o del canal será determinado al construir el modelo de simulación y fijar los parámetros.

- **Herencia:** Los módulos y canales podrán ser subclases. Los módulos y clases derivados podrán tener nuevos parámetros, puertas, submódulos y conexiones. También se podrán usar para modificar los valores de los parámetros y el tamaño de los vectores de puertas de las superclases.

- **Otras características:** La organización de clases mediante el uso de paquetes como Java. Los tipos internos como los tipos de canales o tipos de módulos usados en la definición de un módulo compuesto. La anotación de metadatos de una simulación.

4.4 Modelado en OMNeT++.

4.4.1 Tipos de Módulos en OMNeT++.

Los modelos en OMNeT++ están compuestos por módulos que se comunican entre sí mediante el intercambio de mensajes. Existen dos tipos de módulos, los módulos simples y los compuestos.

- **Módulos simples:** Son los componentes más básicos del modelo y están compuestos por dos partes: descripción y la parte lógica. Su descripción se realiza en lenguaje NED, mientras que su parte lógica está escrita en C++ usando la librería de clases de la simulación de OMNeT++.

- **Módulos compuestos:** Los módulos simples podrán ser agrupados en módulos compuestos, y además los módulos compuestos también podrán agruparse en módulos compuestos, por lo que los niveles de jerarquía serán ilimitados. Por ejemplo, un modelo de red compuesto por módulos simples o compuestos, podrá ser considerado un módulo compuesto.

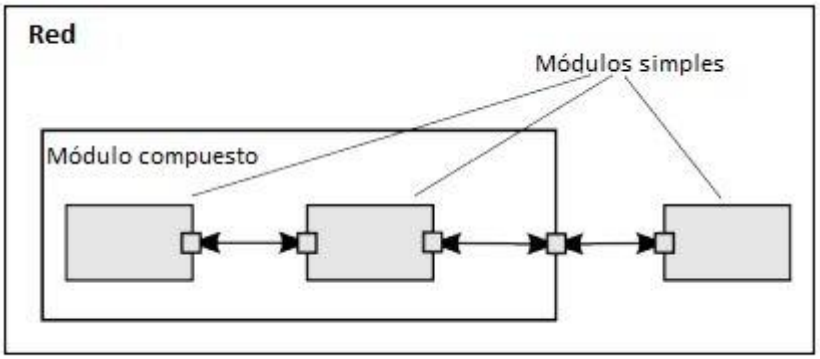

Ilustración 7 - Módulos simples y compuestos

Un modelo en OMNeT++ será representado como una agrupación de módulos anidados jerárquicamente. Las instancias de los módulos más simples servirán como componentes para módulos más complejos, como por ejemplo, el módulo del sistema.

4.4.2 Construcción de un módulo simple.

Para ver la forma en la que se construye un módulo simple, nos basaremos en el módulo Queue incluido en el directorio queueinglib de los proyectos de muestra de OMNeT++ (ver carpeta samples en el directorio de instalación de OMNeT++).

Un módulo simple constará de dos ficheros, uno en NED con la descripción del módulo, y otro en C++ con la programación de la parte lógica.

La descripción de un módulo tendrá la siguiente estructura:

```
simple Queue
{
    parameters:
        @display("i=block/queue;q=queue");
        int capacity = default(-1);
        bool fifo = default(true);
        volatile double serviceTime @unit(s);
    gates:
        input in[];
        output out;
}
```

Con estas líneas hemos descrito un módulo llamado Queue definiendo sus parámetros y sus puertas:

- **Parámetros:** Hemos añadido un parámetro de display que asocia al módulo una imagen de la librería de imágenes de OMNeT++, en este caso la imagen de una cola (queue). También hemos añadido dos parámetros (capacity y queue) con valores por defecto y un parámetro de tiempo de servicio volátil (volatile double servicetime). El valor de este último parámetro cada vez que se vuelva a recalcular el tiempo de servicio (definido en omnetpp.ini), cambiará.

.

- **Puertas:** Hemos añadido al módulo un array de puertas de entrada (in) y una única puerta de salida (out).

Nota: Esta parte se podrá hacer en el fichero .ned del modelo del sistema o en un fichero .ned aparte dónde se describirá únicamente el módulo.

Ejemplo de la programación de la parte de la implementación:

```
#include "Job.h"

namespace queueing {// declaración de la clase
class QUEUEING_API Queue : public cSimpleModule
{
    private:
       //...
    public:
        Queue();
        virtual ~Queue();
       //...
    protected:
        virtual void initialize();
        virtual void handleMessage(cMessage *msg);
        virtual void finish();
       //...
};

Define_Module(Queue); // asociación del módulo

Queue::Queue()// constructor
{
       //...
}
Queue::~Queue()// destructor del módulo
{
       //...
}
void Queue::initialize()
{
    //...
    endServiceMsg = new cMessage("end-service");
    fifo = par("fifo");
    capacity = par("capacity");
    //...
}
void Queue::handleMessage(cMessage *msg)
{
    if (msg==endServiceMsg)
    {
      //...
    }
    else
    {
      //...
    }
}
       //...
```

```
void Queue::finish()
{
      //...
}
};// end namespace
```

Con estas líneas hemos definido la estructura y los métodos a destacar para explicar el funcionamiento de la parte lógica de un módulo. A continuación se explicarán en detalle:

- **Declaración de la clase:** Se declaran los parámetros y cabeceras de los métodos que usará el módulo. Se podrá restringir su accesibilidad definiéndolos como protected, public o private.

- **Define Module:** En esta línea se realiza el registro del módulo en OMNeT y se asocia la parte lógica del módulo en C++ a su descripción en NED.

- **Constructor y destructor:** El constructor será llamado durante la construcción del módulo al principio de la simulación, mientras que el destructor será llamado en la destrucción del módulo o al final de la simulación.

- **Método initialize:** Este método será llamado por el módulo al inicio de la simulación. Es el método de arranque en el que se realiza lo necesario para poner la simulación en marcha. En este caso, se han tomado los valores del archivo de configuración omnetpp.ini para los parámetros fifo y capacity mediante el método par y se ha creado el mensaje endServiceMsg. Otra operación muy típica en el método initialize será el envío de un mensaje a otro módulo o el envío de un automensaje.

- **Método handlemessage:** Este método será llamado por el módulo cada vez que éste reciba un mensaje. Puede activarse al recibir un mensaje o automensaje. Este método tendrá el mensaje recibido como parámetro de entrada, por lo que podrá usar los datos que contiene en sus operaciones, como por ejemplo, saber a qué módulo enviar el próximo mensaje.

- **Método finish:** Este método será llamado por el módulo al final de la simulación. No se debe confundir con el método destructor, que se llamará después durante la destrucción del módulo.

Nota: Queue.cc no se ha representado tal y como aparece en queuinglib, con la finalidad de explicar mejor las partes más importantes de la parte lógica de un módulo simple.

Nota 2: Esta parte se podrá hacer en un fichero .cc o dividirse en dos ficheros: un fichero .h con las cabeceras de los métodos y otro .cc con la lógica. En caso de hacer esto último, la parte de “class Queue” y los includes irán en el fichero .h y el resto en el fichero .cc. Para más información consultar los ejemplos de muestra de OMNeT++.

4.4.3 Comunicación entre módulos.

Como ya hemos dicho, los módulos se comunicarán entre sí mediante el intercambio de mensajes. Estos mensajes podrán ser vistos como los paquetes en una red de computadores o como las tareas o los clientes en una red de colas. Además podrán contener cualquier tipo de datos.

Los mensajes podrán llegar de otro módulo o del mismo módulo (auto-mensaje). El tiempo de simulación local de un módulo se actualizará cuando éste reciba un mensaje. Los auto-mensajes, por ejemplo, serán usados para realizar temporizadores en los módulos.

El envío y recepción de mensajes entre módulos se realizará a través de puertas (gates) que funcionarán como interfaces de entrada y salida de los módulos. Los mensajes serán enviados desde un módulo por una puerta de salida y entrarán a otro por una puerta de entrada.

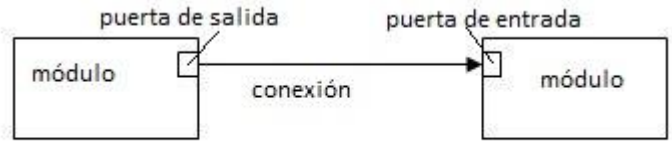

Ilustración 8 - Puertas y conexiones

Una puerta de salida y otra de entrada podrán comunicarse mediante una conexión (connection). En el caso de los módulos simples el envío de mensajes podrá ser directo o sin hacer uso de las conexiones, no obstante, esta posibilidad no se ha contemplado en el proyecto.

Para realizar las conexiones se usarán objetos de la clase de OMNeT++ cChannel (canales). Inicialmente esta clase tendrá unos parámetros por defecto, pero podrá ser extendida y añadírsele nuevos parámetros de forma que pueda ser personalizada.

Para ver qué componentes entran en juego en la conexión de módulos, nos basaremos en el modelo de la red RingQueue.ned incluido en el directorio queuenet de los proyectos de muestra de OMNeT++ (ver carpeta samples en el directorio de instalación de OMNeT++).

4.4.4 Definición de las puertas del módulo.

En esta red usaremos los módulos Queue y Source. Las puertas de estos dos módulos estarán descritas en sus respectivas descripciones NED:

```
simple Queue
{
    parameters:
    //...
    gates:
        input in[];
        output out;
}
```

En el caso del módulo Queue se han descrito un array de puertas de entrada al módulo y una única puerta de salida. El array de puertas permitirá la creación de múltiples conexiones de entrada.

```
simple Source
{
    parameters:
    //...
    gates:
        output out;
}
```

En el caso del modulo Source solamente se ha definido una única puerta de salida llamada out.

Nota: Un módulo podrá tener puertas del tipo input (entrada), output (salida) o inout (bidireccionales).

4.4.5 Definición de canales.

Para hacer las conexiones entre las puertas de los módulos se usarán canales. Si no se define ningún canal en el fichero .ned de la red se usará el canal IdealChannel, que no tendrá parámetros asociados. Los valores de los parámetros de un canal podrán ser leídos por la parte lógica de los módulos (ficheros .cc). El objetivo de la definición de canales es poder añadir nuevos parámetros al canal o darle valores a los parámetros ya existentes.

```
channel C extends ned.DatarateChannel {
      datarate = 100Mbps;
      int newparameter;
}
```

En este ejemplo se ha definido un canal llamado C que es una extensión de DatarateChannel. El canal DatarateChannel tendrá asociados los parámetros datarate, disabled, ber y per. El canal C será una extensión que conservará todas las propiedades de cDatarateChannel. Su parámetro datarate será de 100Mbps y además tendrá un nuevo parámetro llamado newparameter cuyos valores podrán ser cambiados en la definición de las conexiones.

Nota: Aunque en la red RingQueue.ned original se usa un canal IdealChannel (el tipo de canal que se usa por defecto), se ha optado por cambiar esto para poder explicar la definición de canales.

4.4.6 Definición de las conexiones.

A continuación se realizará la conexión de los módulos.

```
network RingQueue
{
    parameters:
        @display("i=device/lan-ring");
    types:
channel C extends ned.DatarateChannel {
             datarate = 100Mbps;
             int newparameter;
}
    submodules:
        source1: Source {
            @display("p=370.0,36.0");
        }
        source: Source {
            @display("p=54.0,45.0");
        }
        queue: Queue {
            @display("p=110.0,124.0");
        }
        queue1: Queue {
            @display("p=139.0,241.0");
        }
        queue2: Queue {
            @display("p=282.0,251.0");
        }
        queue3: Queue {
            @display("p=327.0,134.0");
        }
        queue4: Queue {
            @display("p=232.0,64.0");
        }
    connections:
        queue4.out --> C --> queue.in++;
        queue.out --> C --> queue1.in++;
        queue1.out --> C --> queue2.in++;
        queue2.out --> C --> queue3.in++;
        queue3.out --> C --> queue4.in++;
        source.out --> C --> queue.in++;
        source1.out --> C --> queue3.in++;
}
```

En este ejemplo se ha definido una red llamada Ring Queue.

En el apartado submodules se han definido todos los módulos que aparecerán en la red (2 módulos source y 5 módulos Queue) y su posición en la red.

En el apartado connections se han definido las conexiones entre los distintos módulos de la red, que usarán un canal de tipo C definido en el apartado types. Esta será la topología de red resultante:

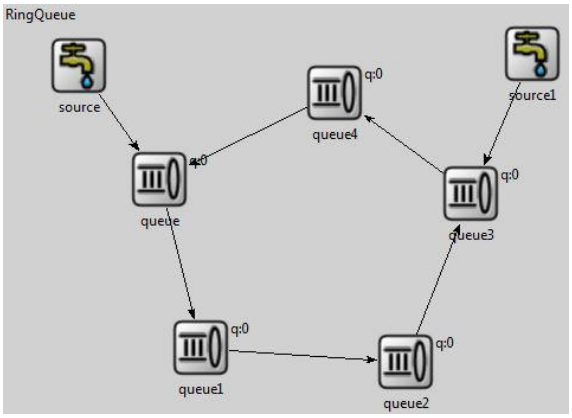

Ilustración 9 - Topología ringqueue

4.4.7 Parametrización de los módulos.

Los módulos pueden tener parámetros, que pueden tomar valores bien del archivo NED del correspondiente módulo o bien del archivo de configuración del modelo omnetpp.ini.

Estos parámetros permitirán la personalización del comportamiento de los módulos simples. Se podrán parametrizar datos del tipo string, numéricos, boolean e incluso árboles de datos XML. Los datos de los parámetros serán usados por la parte lógica del módulo escrita en C++.

En el siguiente ejemplo veremos la parametrización de los módulos y configuración de la red en un archivo omnetpp.ini.

```
[General]
#...
[Config SmallCQN]
network = SmallCQN
#...
[Config Ring]
description = "a server ring"
network = RingQueue
**.source.numJobs = 15
**.source.interArrivalTime = 0
**.source1.interArrivalTime = exponential(2s)
**.source1.startTime = 20s
**.source1.numJobs = 25
#...
```

Se han definido tres tipos de configuraciones: la General, SmallCQN y Ring. La configuración general tendrá la parametrización general de todas las configuraciones, la configuración SmallCQN tendrá la parametrización de la red SmallCQN (parámetro network), y la configuración Ring tendrá la parametrización de la red RedQueue. En esta configuración se darán valores a los parámetros numJobs e interArrivalTime del módulo source y a los parámetros numJobs, interArrivalTime y startTime del módulo source1.

4.4.8 Uso de señales para la grabación de estadísticas.

Los módulos y canales podrán emitir señales con la finalidad de:

- Mostrar las propiedades estadísticas de un modelo.
- Mostrar los cambios que se han producido durante la simulación del modelo y notificarlos.
- Permitir entre los módulos una comunicación tipo publicador-subscriptor.
- Emitir información con otros propósitos.

Las señales se identifican por sus nombres de señal y por sus identificadores numéricos de señal del tipo simsignal_t, que tendrán valores asignados dinámicamente durante la ejecución de la simulación. Para registrar un nombre de señal se usará el método registerSignal que devolverá el identificador de señal, con la siguiente sintaxis:

```
Simsignal_t identificador = registerSignal(“señal”)
```

Para emitir una señal se usará la función emit, que permitirá asociar a la señal emitida el valor que se quiere emitir. Su sintaxis es la siguiente:

```
emit(identificador, valor_a_emitir);
```

Para poder grabar los resultados basados en señales de la simulación habrá que agregar las propiedades @signal y @statistics a la definición NED de los módulos o canales. La propiedad @signal declara que una señal con determinado nombre se emite desde la parte lógica del módulo, mientras que la propiedad @statistics define qué señales serán usadas como entrada (en source), qué propiedades les serán aplicadas en el procesado (por ejemplo filtrado), qué propiedades serán grabadas (máximo, mínimo, media), y de qué forma (vectores, escales, histogramas). A continuación veremos un ejemplo:

```
simple Queue
{
    parameters:
        //...
        @signal[señal](type="int");
        //...
        @statistic[estadistica](source=señal; record=max,timeavg,vector?);
        //...
}
```

En estas líneas se ha asumido que la parte en C++ del módulo o parte lógica emite una señal de nombre “señal”, nombre que le habrá sido dado por la función registerSignal y que además esta señal será de tipo int. La propiedad @statistics que hemos agregado a este módulo grabará el máximo (max), el promedio temporal (timeavg) y el vector de forma opcional (“?” significa opcional) de la señal “señal”.
También se podrán aplicar propiedades a la señal de entrada como en el siguiente ejemplo:

```
@statistic[estadistica1](source="count(señal1)/count(señal2)";record=last,
mean,max?,vector?);
```

En este ejemplo se le ha aplicado la propiedad count a las señales señal1 y señal2. La estadística "estadistica1" tendrá como entrada el resultado de la división de estas dos. Para ver una lista más extensa de las propiedades, consultar el manual de OMNeT++.

Por último, hay que mencionar que habrá que ajustar el modo de grabación de las estadísticas en el fichero de configuración omnetpp.ini, en el apartado correspondiente al modelo que estemos usando. Esto se podrá ajustar mediante la opción de configuración result-recording-modes. Por ejemplo, para la estadística "estadística1" del ejemplo de arriba tendremos los siguientes modos de configuración (en los comentarios se ha escrito el resultado de cada modo de configuración):

```
**.result-recording-modes = default
# --> graba last, mean
**.result-recording-modes = all
# --> graba last, mean, max
**.result-recording-modes = -
# --> no graba ninguna
**.result-recording-modes = mean
# --> solo graba mean (deshabilita 'default')
**.result-recording-modes = default,-vector,+histogram
# --> graba count,mean,histogram
**.result-recording-modes = -vector,+histogram
# --> igual que arriba
**.result-recording-modes = all,-vector,+histogram
# --> graba count,mean,max,histogram
```

Tras la simulación, las estadísticas serán guardadas como datos de salida. Se guardarán en ficheros de tipo vector o de tipo escalar en la carpeta output del proyecto. OMNeT++ ofrecerá una herramienta para el análisis de estos tipos de ficheros.

4.5 Compilación de modelos en OMNeT++.

Un modelo a los ojos de OMNeT++ estará compuesto por las siguientes partes:

- **Descripciones en NED:** Tanto la estructura de los módulos como la descripción de la topología del sistema se harán en lenguaje NED. OMNeT++ permite la creación de estos ficheros escribiéndolos directamente o usando un editor gráfico. Este tipo de archivos tendrán asociada la extensión .ned.

- **Definiciones de los mensajes:** Se podrán definir varios tipos de mensajes y añadirles campos de datos. OMNeT++ se encargará de traducir estas definiciones a clases en C++. Se almacenarán en ficheros con la extensión .msg.

- **Ficheros con la lógica de los módulos simples:** Contienen la programación de los módulos simples. Son ficheros de C++ con la extensión .cc o .h.

- **Fichero "omnetpp.ini":** Se trata del archivo de configuración del modelo. Contendrá los ajustes de simulación del modelo, la parametrización de los módulos y del modelo, la forma de grabación de datos estadísticos, etc.

La compilación de las partes se llevará a cabo en el siguiente orden:

1. Traducción de los mensajes a clases en C++ usando el programa opp_msgc.
2. Compilación y enlace con el kernel de simulación de todos los ficheros en C++.
3. Cuando el programa de simulación arranque, se cargarán los datos de los ficheros NED.
4. Se realizará la lectura del archivo de configuración "omnetpp.ini".

Para compilar el proyecto en el IDE haremos click derecho en el explorador de proyectos y daremos a Build Project. En caso de trabajar con una consola de Linux o MinGW para Windows usaremos el comando $opp_makemake para crear el archivo Makefile y el comando $make para compilar el proyecto.

4.6 Simulación de modelos en OMNeT++.

La parte para la simulación de OMNeT++ ofrecerá:

Un kernel para la simulación: Contiene el código que gestionará la simulación y la librería con las clases de la simulación.

Interfaces de usuario: Permitirán debugear, ejecutar, y controlar simulación del modelo. También permitirán visualizarlo y cambiar el valor de las variables de la simulación. Al ser un entorno para la simulación de eventos discretos, las simulaciones de OMNeT++ serán una sucesión de este tipo de eventos. Cada evento discreto estará asociado a un tiempo en el que ocurrirá dicho evento.

OMNeT++ usa los mensajes para representar los eventos discretos, por lo que en la simulación cada vez que un módulo envíe o reciba un mensaje, habrá sucedido un evento. Cada evento será representado por una instancia a la clase cMessage o a las subclases de esta.

La organización de los eventos discretos futuros en OMNeT+ se realiza mediante la estructura de datos FES (Future Event Set). El funcionamiento de dicha estructura se explica en el siguiente trozo de pseudocódigo:

```
Inicialización – construcción del modelo e inserción de los eventos iniciales en FES
while (FES no vacía y la simulación no se haya completado)
{
Recuperar primer evento en la lista de FES
t:= tiempo de suceso del evento
Procesar evento
```

(El procesado de eventos puede añadir nuevos eventos o borrar los eventos existentes de la lista de FES)
}
Terminar simulación (escribir resultados estadísticos, etc)

Los eventos discretos futuros se ordenarán en la lista de la estructura FES de acuerdo a los siguientes criterios de ordenación:

1- El criterio más prioritario será el tiempo de llegada del mensaje al módulo. Cuanto antes haya llegado, antes estará situado en la lista del FES. Si dos mensajes tienen el mismo tiempo de llegada se tomará en cuenta el siguiente criterio de ordenación.
2- El evento con menor prioridad de planificación se ejecutará antes. Por defecto todos los eventos de los mensajes tendrán prioridad 0, pero está podrá ser cambiada usando el método setSchedulingPriority. Si dos mensajes tienen la misma prioridad, se tomará en cuenta el siguiente criterio de ordenación.
3- El evento que antes se haya planificado e introducido en el FES se ejecutará antes.

Si la simulación genera datos de salida, podrán generarse ficheros de tipo vector o de tipo escalar en la carpeta output del proyecto. OMNeT++ ofrecerá una herramienta para el análisis de estos tipos de ficheros.

Para arrancar la simulación del proyecto en el IDE haremos click derecho sobre el proyecto en el explorador de proyectos y elegiremos Run As > OMNeT++ Simulation. En caso de trabajar con una consola de Linux o MinGW para Windows usaremos el comando $./nombre_del_proyecto.

4.7 Construcción de un modelo TicToc con OMNeT++.

A continuación detallaremos la construcción de un modelo muy simple llamado TicToc. El modelo será una red que constará únicamente de dos módulos: Tic y Toc.

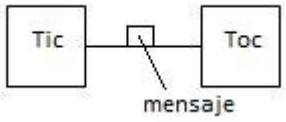

Ilustración 10 - Modelo TicToc

La comunicación entre estos dos módulos será muy simple, Tic generará un mensaje que recibirá Toc, y este último se lo devolverá. Dicho de otra forma, solo se usará un único mensaje que irá "rebotando" de Tic a Toc y de Toc a Tic.

El proyecto se podrá crear bien desde el IDE de OMNeT++, o bien desde una consola en Linux o MinGW en Windows. Estos son los pasos a seguir para la implementación del modelo:

Paso 1: Crearemos un nuevo proyecto llamado tictoc. Para crearlo desde el IDE haremos click en File > New > OMNeT++ Project. Escribiremos su nombre (tictoc) y le daremos a "finish". Se crearán dentro del proyecto una librería con ficheros de Include, un fichero llamado package.ned y otro llamado Makefile. Si estamos usando la consola crearemos un directorio con el nombre del proyecto (por ejemplo, usando mkdir) y nos posicionaremos dentro de este (por ejemplo, usando cd nombre_del_directorio).

Paso 2: A continuación crearemos un fichero NED para describir la topología de la red que vamos a crear. Si lo creamos desde el IDE haremos click derecho sobre el nombre del proyecto en el explorador de proyecto eligiendo New > File. Escribiremos tictoc1.ned y haremos click en "finish". Si estamos usando la consola, podemos crear el fichero con un editor de texto. Dentro del fichero escribiremos las siguientes líneas de código:

```
//descripción del módulo simple Txc1, será instanciados por Tic y Toc
simple Txc1{
    //declaracion puerta de entrada (in), de su puerta de salida (out)
    gates:
        input in;
        output out;
}
//Definición de la red Tictoc1,
network Tictoc1
{
    //compuesta de Tic & Toc de tipo Txc1
    submodules:
        tic: Txc1;
        toc: Txc1;
    //definición de las conexiones entre ambos
    connections:
        tic.out --> {  delay = 100ms; } --> toc.in;
        tic.in <-- {  delay = 100ms; } <-- toc.out;
}
```

Con estas líneas hemos descrito:

- Un módulo simple con una puerta de entrada (input) y una puerta de salida (output). Su parte lógica se programará en C++.
- La topología de la red que constará de dos submódulos tic y toc conectados de tic a toc y de toc a tic usando un canal con un retardo de 100ms.

Estas dos partes podrían llevarse a ficheros diferentes. La descripción del módulo simple se podrá hacer en otro fichero y ser llamada desde el fichero con la descripción de la topología de la red.

Paso 3: Crearemos el fichero txc1.cc que contendrá la programación en C++ del módulo simple. Para crearlo con el IDE haremos click derecho sobre el proyecto en el explorador de proyectos y elegiremos New > File. Escribiremos txc1.cc y le daremos a "finish". En el fichero txc1.cc escribiremos las siguientes líneas de código:

```
#include <string.h>
#include <omnetpp.h>

class Txc1 : public cSimpleModule
{
  protected:
    //definición de las cabeceras de los métodos
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

//Asociar la clase al módulo Txc1
Define_Module(Txc1);

//método initialize, será llamado al principio de la simulación
void Txc1::initialize()
{
    //si se trata del módulo tic, crear y enviar mensaje
    if (strcmp("tic", getName()) == 0)
    {
        cMessage *msg = new cMessage("tictocMsg");
        send(msg, "out");
    }
}

//método handlemessage, será llamado cuando el módulo al reciba un mensaje
void Txc1::handleMessage(cMessage *msg)
{
    //reenviar el mensaje recibido
    send(msg, "out");
}
```

Con estas líneas hemos:

- Declarado la clase y definido las cabeceras de los métodos.
- Asociado la clase al módulo Txc1 (Define_module).
- Programado la lógica del método initialize(), que será llamado al principio de la simulación. Hemos hecho que cree un mensaje y lo envíe por la puerta de salida "out" del módulo.
- Programado la lógica del método hadlemessage(), que será llamado cada vez que un mensaje llegue al módulo. Hemos hecho que vuelva a enviar el mensaje recibido por la puerta de salida "out" del módulo.

Paso 4: Crearemos el fichero de configuración omnetpp.ini. Para crearlo con el IDE, haremos click derecho sobre el proyecto en el explorador de proyectos y elegiremos New > File. Escribiremos omnetpp.ini y le daremos a finish. En este fichero escribiremos:

```
[General]
network = Tictoc1
```

Con estas líneas hemos elegido Tictoc1 como red de la configuración general.

Paso 5: Para compilar el proyecto en el IDE haremos click derecho en el explorador de proyectos y daremos a Build Project. Si la compilación ha ido bien, se podrá ejecutar la simulación haciendo click derecho sobre el proyecto en el explorador en el explorador de proyecto y eligiendo Run As > OMNeT++ Simulation.

Si estamos usando la consola, antes de compilar habrá que crear el fichero Makefile usando el comando $opp_makemake. Si el fichero se ha creado, para compilar el proyecto se usaremos el comando $make. Si la compilación ha sido exitosa, podremos iniciar la simulación con el comando ./tictoc.

5. Módulos desarrollados en OMNeT++.

5.1 Introducción.

Para poder simular redes de Petri en OMNeT++ habrá que desarrollar los módulos de los que éstas se componen: los módulos Place (lugar) y módulo Transition (transición). Ambos serán de tipo simple y a partir de ellos se podrá modelar cualquier topología de red de Petri con transiciones no inmediatas, ya que los disparos en el módulo Transition serán retardados. El buen funcionamiento de estos dos módulos se comprobará simulando distintas topologías de redes de Petri y la red llamada "petrinet.ned".

Una vez probado el modelo "petrinet.ned", se introducirán los módulos InmediateTr (transición inmediata) y SolveConflict (resolución de conflictos), ambos también de tipo simple. El módulo InmediaeTr funcionará como el módulo Transition, pero a diferencia de éste, su disparo no será retardado y se producirá en el mismo instante en el que se haya producido la condición de disparo, mientras que el módulo SolveConflict será el gestor de los disparos de todos los módulos InmediateTr y el encargado de evitar los conflictos entre estos. Para probar el correcto funcionamiento de un modelo de red de Petri que contenga todos estos módulos se simulará la red de Petri llamada "Tres Procesos que Compiten por Dos Recursos".

5.2 Módulo Place.

Se trata de un módulo simple. Está compuesto por los ficheros Place.cc y Place.h que contienen la implementación del módulo en C++, y por el fichero Place.ned que contiene la descripción del módulo escrita en lenguaje NED.

Cada módulo Place de la red de Petri comenzará con un número determinado de marcas que a lo largo de la simulación irá variando. El número de marcas de cada Place disminuirá cuando éstas sean consumidas por las transiciones y aumentará cuando reciban nuevas marcas producidas por las transiciones. Estas marcas también serán conocidas con el nombre de "tokens".

5.2.1 Descripción NED del módulo Place.

```
simple Place
{
    parameters:
    int tokens = default(0);
    gates:
    input inputport[];
    output outputport[];
    input outputportin[];
}
```

En la descripción NED de Place se ha establecido que éste será un módulo simple. Su parámetro será el número de marcas o tokens. Además, tendrá tres arrays de puertas: el array inputport que permitirá la interacción con los módulos Transition predecesores, y los arrays outputport y outputportin que permitirán la interacción con los módulos Transition sucesores.

En la siguiente imagen se mostrarán los tipos de puertas del módulo Place y la conectividad con la puertas del módulo Transition.

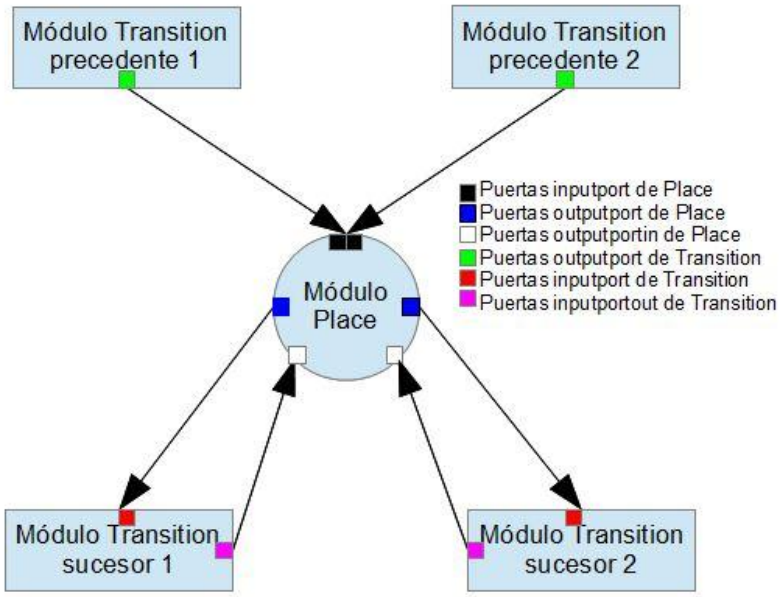

Ilustración 11 - Conectividad módulo Place

Esta definición de puertas se debe a que las comunicaciones entre los módulos Transition predecesores y el módulo Place son unidireccionales, es decir, necesitarán una única conexión de Transition a Place, mientras que las comunicaciones entre el módulo Place y los módulos Transition sucesores son bidireccionales y necesitarán dos conexiones: una de Place a Transition y otra de Transition a Place.

5.2.2 Declaración de la clase Place.

```
class Place : public cSimpleModule
{
  public:
    //Constructor y destructor
  private:
    //...
    int s;//número de puertas outputport o de salida
    int multout;//multiplicidad del canal de una puerta de salida outputport
    int tokens;//cantidad de tokens
    //...
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
```

Los atributos a destacar de esta clase son tres variables de enteros. La variable s albergará el número de puertas de salida (outputport) del módulo, la variable multout albergará la multiplicidad del canal de una conexión realizada entre el módulo Place y el módulo Transition sucesor, y la variable tokens albergará el número de marcas o tokens del módulo.

Los métodos a destacar de esta clase son el método initialize y el método handleMessage. El método initialize será llamado por el módulo al inicio de la simulación y actuará como método de arranque en el que se realizará lo necesario para poner la simulación en marcha, mientras que el método handleMessage será llamado por el módulo cada vez que éste reciba un mensaje. Este mensaje puede ser un mensaje externo procedente de otro módulo o un auto mensaje procedente del mismo módulo Place que lo recibe.

5.2.3 Método initialize.

```
void Place::initialize()
{
    //SECCIÓN 1
    tokens = par("tokens");
    s=gateSize("outputport");

    initializeevent = new cMessage("initializeevent",0);
    cMessage *inhibitorarcmsg = NULL;

    //SECCIÓN 2
    //buscar arco inhibidor
    if (tokens==0){//si 0 tokens
        for (int i=0; i<s; i++){//mirar todas las salidas
            multout=gate("outputport", i)->getChannel()->par("multiplicity");
            if(multout==0){//si tiene un arco inhibidor
                inhibitorarcmsg = new cMessage(getName(),1); //puede disparar
                send(inhibitorarcmsg, "outputport", i);
            }
        }
    }
    else if (tokens>0){//si tokens > 0,
        scheduleAt(simTime(), initializeevent); //automensaje
    }
    //...
}
```

A continuación comentamos las secciones de código a destacar (mirar comentarios) del método initialize del módulo Place:

Sección 1: Las variables tokens y s serán inicializadas. La variable tokens toma el valor que se le haya dado en el archivo de configuración omnetpp.ini a través del método par(), mientras que la variable s toma un valor igual al número de puertas outputport de salida a través del método gateSize().

Sección 2: Dependiendo de la situación, si hay más de 0 tokens en el módulo o si se trata de un módulo con arco inhibidor hacia la transición, el método mandará uno de los dos siguientes tipos de mensaje:

- **Initializeevent:** Es un auto mensaje para la inicialización del módulo Place que se mandará en caso de que la variable tokens sea mayor que 0. Su finalidad es arrancar el módulo mandándole un auto mensaje que será tratado en por el método handleMessage(). Tendrá como atributo Name el texto "initializeevent", que servirá para distinguirlo en el método handleMessage() de otros mensajes, y como atributo Kind el valor 0 para no alterar el valor de la variable tokens.

- **Inhibitorarcmsg:** Es un mensaje que Place mandará a un módulo Transition sucesor en caso de que estén unidos por un arco inhibidor. Para ello, en la inicialización el módulo Place comprobará si tiene arcos inhibidores. Esto lo hará recorriendo todas sus puertas de salida outputport y viendo si el canal que usa cada una tiene multiplicidad 0. En caso de que una puerta de salida use un canal con multiplicidad 0 (variable multout), el módulo Place enviará al módulo Transition correspondiente un mensaje para avisarle que puede disparar.

5.2.4 Método handleMessage.

```
void Place::handleMessage(cMessage *msg)//Gestionar mensajes
{
    //Declaración variables locales método:
    cMessage *message = NULL;
    std::string msgName;
    int ntokens;
    //...
    //SECCIÓN 1
    msgName=msg->getName();//nombre del mensaje recibido
    ntokens=msg->getKind();//cantidad de tokens a sumar o restar
    tokens+=ntokens;//sumar o restar
    //...
    if(msgName.compare("initializeevent")!=0)// Si no es automensaje
        delete msg; //borrarlo

    //SECCIÓN 2
    for (int i=0; i<s; i++){//mandar a todas las transiciones aviso
        multout=gate("outputport", i)->getChannel()->par("multiplicity");
        //multiplicidad de la puerta
        if (multout!=0 && tokens>=multout){//Caso1
            message = new cMessage(getName(),1); //Mensaje disparar
            send(message, "outputport", i);
        }
        else if (multout!=0 && tokens<multout){//Caso2
            message = new cMessage(getName(),0); //Mensaje no disparar
            send(message, "outputport", i);
        }

        else if (multout==0){//Caso3
            if(tokens>0){  //no puede disparar
                message = new cMessage(getName(),0);//Mensaje no disparar
```

```
                send(message, "outputport", i);
            }
            else{ //puede disparar
                message = new cMessage(getName(),1);//Mensaje disparar
                send(message, "outputport", i);
            }
        }
    }
}
```

A continuación comentamos las secciones de código a destacar (mirar comentarios) del método handleMessage del módulo Place:

Sección 1: Las variables locales msgName y ntokens tomarán los valores de los atributos del mensaje recibidos con los métodos getName() y getKind(). El valor de ntokens que puede ser positivo o negativo, se sumará a la variable tokens del módulo. El valor de msgName servirá para distinguir el tipo de mensaje. Si el valor de msgName no es "initializeevent", no se habrá recibido un auto mensaje del módulo y se podrá borrar de manera normal.

Nota: El automensaje "initializaevent" será borrado por el método destructor del módulo, que se ejecutará al finalizar la simulación.

Sección 2: A continuación el módulo Place procederá al envío de mensajes a sus transiciones sucesoras, para ello recorrerá todas sus puertas de salida outputport comprobando la multiplicidad de los canales que éstas usan. Para cada puerta se podrán dar las siguientes situaciones:

- **Caso 1:** Si la multiplicidad de su canal no es 0 y el valor de su variable tokens es igual o mayor que el de la multiplicidad del canal, el módulo Place enviará un mensaje al módulo Transition sucesor indicándole que puede disparar.

- **Caso 2:** Si la multiplicidad de su canal no es 0 y el valor de su variable tokens es menor que el de la multiplicidad del canal, el módulo Place enviará un mensaje al módulo Transition sucesor indicándole que no puede disparar.

- **Caso 3:** Si la multiplicidad de su canal es 0, Place estará conectado a una transición sucesora mediante un arco inhibidor. Si la variable tokens vale 0 enviará un mensaje dando permiso para disparar al módulo Transition sucesor, mientras que si el valor de su variable tokens es mayor que 0, le enviará un mensaje indicándole que no puede disparar.

5.2.5 Señales y obtención de estadísticas del módulo Place.

5.2.4.1 Descripción del módulo Place.

```
simple Place
```

```
{
    parameters:
    //...
    @signal[tokennumber](type="int");
    @signal[isnotempty](type="int");
    @statistic[TokenTimeAvg](source=tokennumber; record=timeavg?,vector?);
    @statistic[NotEmptyTimeAvg](source=isnotempty; record=timeavg?,vector?);
    @statistic[EmptyTimeAvg](source=(1-isnotempty); record=timeavg?);
    //...
}
```

Se han agregado al módulo las señales tokennumber y isnotempty, y las estadísticas TokenTimeAvg, NotEmptyTimeAvg y EmptyTimeAvg. Las dos señales serán emitidas usando el método emit desde la implementación del módulo, como veremos abajo. En cuanto a las estadísticas, las señales serán usadas como fuente de datos (source) en la grabación de las éstas, y se ha optado por grabar el promedio temporal y el vector de manera opcional.

5.2.4.2Declaración de la clase Place.

```
class Place : public cSimpleModule
{
  private:
    //...
    simsignal_t tokensignal, noemptysignal;
    cLongHistogram tokenHistogram;
};
```

En la parte de la implementación del módulo se han declarado dos identificadores de señal (tokensignal y noemptysignal) y un objeto cLongHistogram (tokenHistogram).

5.2.4.3Método initialize.

```
void Place::initialize()
{
    tokensignal=registerSignal("tokennumber");
    noemptysignal=registerSignal("isnotempty");
}
```

Se ha usado el método registerSignal para registrar los nombres de señal tokennumber y isnotempty (estos nombres serán los mismos que se han usado en la descripción NED del módulo). Este método devolverá dos identificadores de señal que se guardará en las variables tokensignal y noemptysignal, respectivamente.

5.2.4.4 Método handleMessage.

```
void Place::handleMessage(cMessage *msg)//Gestionar mensajes
{
    //...
    tokens+=ntokens;//sumar o restar
    emit(tokensignal, tokens);
    tokenHistogram.collect(tokens);
```

```
    if (tokens==0) emit(noemptysignal, 0);
    else if (tokens>0) emit(noemptysignal, 1);
    //...
}
```

Cada vez que el modulo Place reciba un mensaje, emitirá la señal tokensignal con su número actual de tokens, añadirá su número actual de tokens a un objeto histograma y dependiendo si quedan o no quedan tokens en el módulo, se emitirá la señal noemptysignal con valor 1 (en caso de que queden) o con valor 0.

5.3 Módulo Transition.

Se trata de un módulo simple. Está compuesto por los ficheros Transition.cc y Transition.h que contienen la implementación del módulo en C++, y por el fichero Place.ned que contiene la descripción del módulo escrita en lenguaje NED.

El módulo Transition se podrá disparar consumiendo (o restando) tokens de los módulos Place precedentes o produciendo (o sumando) tokens en sus módulos Place sucesores.

Durante la simulación irá recibiendo autorizaciones o prohibiciones de disparo por parte de los módulos Place precedentes. Una vez que todos éstos le hayan dado la autorización de disparo, este módulo planificará un disparo retardado, disparo que ocurrirá una vez que haya pasado el tiempo de retardo, y siempre que el módulo conserve todas las autorizaciones de disparo.

El retardo de las transiciones servirá para representar tiempos reales del sistema simulado como retardos en la lectura de caché o de memoria por parte de un proceso, etc.

5.3.1 Descripción NED del módulo Transition.

```
simple Transition
{
    parameters:
    volatile double firingTime @unit(s);
    gates:
    input inputport[];
    output inputportout[];
    output outputport[];
}
```

En la descripción NED de Transition se ha establecido que éste será un módulo simple. Su parámetro será firingTime o el tiempo de disparo, que será volatile, es decir, cada vez que se recalcule el tiempo de disparo el valor de este parámetro cambiará según la configuración que se haya especificado en el fichero omnetpp.ini. Además, tendrá tres arrays de puertas: los arrays inputport y inputportout que permitirán la interacción con los módulos Place predecesores, y el array outputport que permitirá la interacción con los módulos Transition sucesores.

En la siguiente imagen se mostrarán los tipos de puertas del módulo Transition. y la conectividad con la puertas del módulo Place.

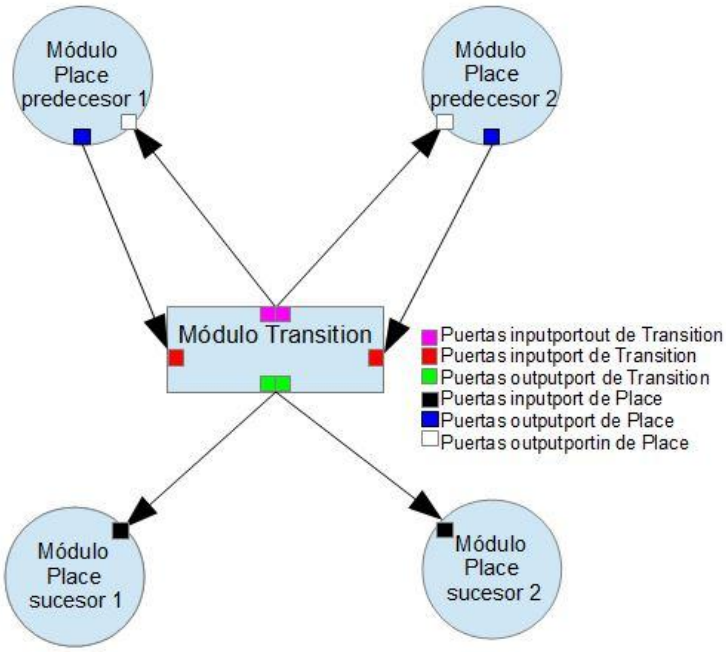

Ilustración 12 - Conectividad módulo Transition

Esta definición de puertas se debe a que las comunicaciones entre los módulos Place predecesores y el módulo Transition son bidireccionales y necesitarán dos conexiones: una de Transition a Place y otra de Place a Transition, mientras que las comunicaciones entre el módulo Transition y los módulos Place sucesores son unidireccionales y necesitarán una única conexión de Place a Transition.

5.3.2 Declaración de la clase Transition.

```
class Transition : public cSimpleModule
{
  public:
    //Constructor y destructor
  private:
    //...
    cMessage *disparo; // declaración mensaje disparo
    int e, s; //e número de puertas de entrada, s número de puertas de salida
    int multout, multinout;
    //multout: multiplicidad del canal de una puerta de salida outputport
    //multinout: multiplicidad del canal de una puerta de salida inputportout
    std::vector<std::string> notemptyplaces;
    //vector que guardará los nombres de los lugares predecesores que le
    //hayan dado permiso de disparo
    //...
  protected:
    virtual void initialize();
```

```
    virtual void handleMessage(cMessage *msg);
    //...
};
```

Los atributos a destacar de esta clase son un objeto de tipo cMessage, cuatro variables de enteros y un vector de strings. El mensaje "disparo" será usado por el módulo como auto mensaje que producirá el disparo del módulo una vez haya pasado un tiempo de retardo. Las variables e y s servirán para almacenar el número de puertas de entrada y el número de puertas de salida, respectivamente. La variable multout albergará la multiplicidad del canal de una puerta de salida outputport, mientras que la variable multinout albergará la multiplicidad del canal de una puerta de salida inputportout. El vector de strings servirá para guardar los nombres de los lugares predecesores que le hayan dado autorización para disparar.

Los métodos a destacar de esta clase son initialize y handleMessage. El método initialize será llamado por el módulo al inicio de la simulación e inicializará las variables necesarias para dejar el módulo preparado para la simulación, mientras que el método handleMessage será llamado por el módulo cada vez que éste reciba un mensaje. Este mensaje puede ser un mensaje externo procedente de otro módulo o un auto mensaje procedente del mismo módulo Transition que lo recibe.

5.3.3 Método initialize.

```
void Transition::initialize()
{
    e=gateSize("inputport");
    s=gateSize("outputport");
    //...
    disparo = new cMessage("disparo",2);
}
```

En este método las variables e y s serán inicializadas. Ambas usan el método gateSize(). La variable e tomará un valor igual al número de puertas inputport de entrada, mientras que la variable s toma un valor igual al número de puertas outputport de salida. También se realizará la inicialización del auto mensaje disparo, que será usado en los disparos retardados.

5.3.4 Método handleMessage.

```
void Transition::handleMessage(cMessage *msg)
{
    //Declaración variables locales método:
    cMessage *message;
    int msgKind;
    std::string msgName;
    int isincluded; //existe el lugar en el vector noemptyplaces?
    //SECCIÓN 1
    msgKind=msg->getKind();//nombre del mensaje recibido
```

```
    msgName=msg->getName();//kind del mensaje recibido
    if(msgName.compare("disparo")!=0) // si no es automensaje
        delete msg; // borrarlo
    //SECCIÓN 2
    if(msgKind==0){// llega mensaje "no disparar" de un Place
        for (unsigned i=0; i<notemptyplaces.size() ; i++){//recorrer vector
            if(notemptyplaces.at(i).compare(msgName)==0){//si existe el place
                notemptyplaces.erase(notemptyplaces.begin()+i);//borrarlo
                if(disparo->isScheduled()){//si disparo estaba planificado
                    cancelAndDelete(disparo); //cancelarlo
                }
            }
        }
    }
    //SECCIÓN 3
    else if(msgKind==1){// llega mensaje "disparar" de un Place
        isincluded=0; // llega mensaje "disparar"
        for (unsigned i=0; i<notemptyplaces.size() ; i++){//recorrer vector
            if(notemptyplaces.at(i).compare(msgName)==0)//place incluido?
                isincluded=1;
        }
        if(isincluded==0){//si place no está incluido, meterlo
            notemptyplaces.push_back(msgName); //meterlo al vector
        }
        if((int)notemptyplaces.size()==e && disparo->isScheduled()==false){
         //si número places precedentes = n entradas inputport &
         //disparo no planificado
            scheduleAt((simTime()+par("firingTime").doubleValue()), disparo);
           //planificar disparo cuando pase x tiempo
        }
    }//... SECCIÓN 4
```

A continuación comentamos las secciones de código a destacar (mirar comentarios) del método handleMessage del módulo Transition:

Sección 1: Las variables locales msgName y msgKind tomarán los valores de los atributos del mensaje recibidos con los métodos getName() y getKind(). El valor de msgName servirá para distinguir el tipo de mensaje. Si el valor de msgName no es "disparo", no se habrá recibido un automensaje del módulo y se podrá borrar de manera normal.

Nota: El automensaje "disparo" será borrado por el método destructor del módulo, que se ejecutará al finalizar la simulación.

Sección 2: Esta sección se ejecutará si el módulo Transition recibe un mensaje y el valor de su atributo Kind es 0. Si esto sucede, el módulo Place precedente remitente del mensaje estará quitando el permiso de disparo al Módulo Transition.

Implementación: Se recorrerá el vector notemptyplaces, que contiene los nombres de módulos Place que han autorizado el disparo, comprobando si el nombre del módulo Place que ha enviado el mensaje está entre ellos. En caso de que este nombre exista en el vector, se borrará, y además se cancelará el disparo de la transición en caso de que se hubiera planificado.

Sección 3: Esta sección se ejecutará si el módulo Transition recibe un mensaje y el valor de su atributo Kind es 1. Si esto sucede, el módulo Place precedente remitente del mensaje estará permitiendo el disparo del Módulo Transition. Cuando Transition tenga el permiso de disparo de todos los módulos Place precedentes se planificará el disparo.

Implementación: En primer lugar, se recorrerá el vector notemptyplaces comprobando si contiene el nombre del módulo Place que ha enviado el mensaje. Si este nombre no está incluido en el vector se meterá, mientras que si está incluido no se podrá volver a meter.

Finalmente se comprobará si el número de nombres de módulos Place que contiene el vector es igual al número de entradas inputport y si el disparo no ha sido planificado. En caso de que se cumplan ambas condiciones, se planificará el disparo, que será un auto mensaje que el módulo Transition recibirá una vez haya pasado un tiempo determinado por el parámetro firingTime.

```
    //SECCIÓN 4
    else if(msgKind==2){//Automensaje de disparo
        notemptyplaces.clear();//vaciar el vector
        for (int j=0; j<s; j++){// lugares sucesores aviso para que sumen
            multout=gate("outputport", j)->getChannel()->par("multiplicity");
            message = new cMessage(getName(), multout);//Mensaje suma
            send(message, "outputport", j);
        }
        for (int i=0; i<e; i++){// lugares predecesores aviso para que resten
                                //y paren a las otras transiciones
            multinout=gate("inputportout", i)->getChannel()
                     ->par("multiplicity");
            if(multinout==0){//Si es arco inhibidor
                message = new cMessage(getName(), 0); //Sumar 0
                send(message, "inputportout", i);
            }
            else{
                message = new cMessage(getName(), -multinout);//Mensaje resta
                send(message, "inputportout", i);
            }
        }
    }
}//End handleMessage
```

SECCIÓN 4: : Esta sección se ejecutará si el módulo Transition recibe un mensaje y el valor de su atributo Kind es 2. El mensaje recibido será un auto mensaje anteriormente planificado. Cuando se reciba este tipo de mensaje, el módulo comprobará si sigue teniendo todos los permisos de disparo de sus módulos Place precedentes, y si está condición se sigue dando, se producirá el disparo.

Los disparos restarán un número de tokens de cada módulo Place precedente igual a la multiplicidad del canal que se esté usando para la conexión, y sumarán un número de tokens en cada módulo Place sucesor igual a la multiplicidad del canal que se esté usando para la conexión. Dicho de otra manera, consumirán tokens de los lugares precedentes y producirán tokens para los lugares sucesores.

En caso de que haya dos o más módulos Transition con disparos planificados, al dispararse uno inmediatamente enviará aviso a sus módulos Place precedentes. Estos últimos una vez hayan actualizado su número de tokens, enviarán mensajes de autorización o de prohibición de disparo a las otros módulos Transition con disparos planificados.

Implementación: En primer lugar, se vaciará el vector notemptyplaces. Para que se pueda volver a producir el disparo el módulo Transition, este vector deberá volver a contener todos los nombres de los módulos Place precedentes.

Tras esto, se realizará el envío de mensajes a todos los módulos Place sucesores para que sumen una cantidad de tokens igual a la multiplicidad de canal que se esté usando para la conexión.

Finalmente, se realizará el envío de mensajes a todos los módulos Place predecesores. Estos mensajes tendrán dos objetivos:

- Restar una cantidad de tokens igual a la multiplicidad de canal que se esté usando para la conexión. Si en la conexión se usa un arco inhibidor (multiplicidad 0) no se modificará la cantidad de tokens.

- Hacer que los módulos Place precedentes autoricen o prohíban el disparo a sus módulos Transition sucesores, para poder cancelar un disparo ya planificado si se diera el caso.

5.3.5 Señales y obtención de estadísticas del módulo Transition.

5.3.5.1 Descripción del módulo Transition.

```
simple Transition
{
    parameters:
      //...
    @signal[shotnumber](type="int");
    @signal[productivity](type="float");
    @statistic[ShotCount](source=shotnumber; record=last?,vector?);
    @statistic[Productivity](source=productivity; record=last?,vector?);
    //...
}
```

Se han agregado al módulo las señales shotnumber y productivity, y las estadísticas ShotCount, y Productivity. Las dos señales serán emitidas usando el método emit desde la implementación del módulo, como veremos abajo. En cuanto a las estadísticas, las señales serán usadas como fuente de datos (source) en la grabación de las éstas, y se ha optado por grabar un escalar (last) y el vector de manera opcional

5.3.5.2 Declaración de la clase Transition.

```
class Transition : public cSimpleModule
{
    //...
```

```
    simsignal_t shotcountsignal, productivitysignal;
    int shotCount;// número de veces que se dispara una transicion
  protected:
    //...
    virtual void finish();
};
```

En la parte de la implementación del módulo se han declarado dos identificadores de señal (shotcountsignal y productivitysignal) y una variable entera llamada shotCount que servirá para contar las veces que el módulo se ha disparado. También conviene resaltar el método finish de este módulo, puesto que las dos señales se emitirán cuando éste se ejecute.

5.3.5.3 Método initialize.

```
void Transition::initialize()
{
    //...
    shotcountsignal=registerSignal("shotnumber");
    productivitysignal=registerSignal("productivity");
    shotCount=0;
    //...
}
```

En este método se ha usado el método registerSignal para registrar los nombres de señal shotnumber y productivity (estos nombres serán los mismos que se han usado en la descripción NED del módulo), que devolverá identificadores de señal que se guardarán en las variables shotcountsignal y productivitysignal, respectivamente. También se inicializará la variable shotChount a 0.

5.3.5.4 Método handleMessage.

```
void Transition::handleMessage(cMessage *msg)
{
    //...
    else if(msgKind==2){//llega automensaje de disparo
        //...
        shotCount++;
        //...
    }
}
```

Cada vez que llegue un auto mensaje de disparo al modulo Transition, además de producirse el disparo, se aumentará en uno el contador de disparos shotCount.

5.3.5.5 Método finish.

```
void Transition::finish()
{
    emit(shotcountsignal, shotCount);
    emit(productivitysignal, shotCount/simTime());
}
```

El método finish se ejecutará al final de la simulación y es entonces cuando se emitirán la señal shotcountsignal con la cuenta de disparos, y la señal productivitysignal con la productividad de disparos del módulo

5.4 Modelado de una red Petri simple.

Este modelo se ha ideado para probar el funcionamiento de los módulos Place y Transition. Para las conexiones se ha definido el tipo de canal Arc (en referencia a los arcos de las redes de Petri), que es una extensión de IdealChannel y que además incluye un parámetro llamado multiplicity, que servirá para especificar la multiplicidad del arco. A continuación se muestra la descripción Ned de su topología.

```
network Petrinet
{
    @display("bgb=417,288");
    types:
        channel Arc extends ned.IdealChannel
             { int multiplicity = default(1); }

    submodules:
        place1: Place { @display("p=87,55;t=tokens: $tokens"); }
        place2: Place { @display("p=205,55;t=tokens: $tokens"); }
        place3: Place { @display("p=320,55;t=tokens: $tokens"); }
        place4: Place { @display("p=137,170;t=tokens: $tokens"); }
        place5: Place { @display("p=281,170;t=tokens: $tokens"); }
        transition1: Transition { @display("p=137,106"); }
        transition2: Transition { @display("p=281,106"); }
        transition3: Transition { display("p=137,232"); }
        transition4: Transition { @display("p=281,232"); }
    connections:
        place1.outputport++ --> Arc --> transition1.inputport++;
        transition1.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place1.outputportin++;
        place2.outputport++ --> Arc --> transition1.inputport++;
        transition1.inputportout++ --> Arc{ @display("ls=,0"); } -->
                    place2.outputportin++;
        place2.outputport++ --> Arc --> transition2.inputport++;
        transition2.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place2.outputportin++;
        place3.outputport++ --> Arc --> transition2.inputport++;
        transition2.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place3.outputportin++;
        transition1.outputport++ --> Arc --> place4.inputport++;
        transition2.outputport++ --> Arc --> place5.inputport++;
        place4.outputport++ --> Arc --> transition3.inputport++;
        transition3.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place4.outputportin++;
        place5.outputport++ --> Arc --> transition4.inputport++;
        transition4.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place5.outputportin++;
        transition3.outputport++ --> Arc --> place1.inputport++;
        transition3.outputport++ --> Arc --> place2.inputport++;
        transition4.outputport++ --> Arc --> place2.inputport++;
        transition4.outputport++ --> Arc --> place3.inputport++;
}
```

Nota 1: En la definición de las conexiones habrá que tener en cuenta que entre los módulos Place y Transition en algunos casos se usarán conexiones unidireccionales y en otros conexiones bidireccionales. Por ejemplo, para añadir un arco que una un módulo Place con un módulo Transition, será necesario definir una conexión de Place a Transition y otra de Transition a Place. Para ver como se tienen que realizar las conexiones, ver los dibujos de la definición de las puertas de los módulos Place y Transition.

Nota 2: Las conexiones que usen el parámetro *@display("ls=, 0")* estarán siendo ocultadas por OMNeT++ en la representación de la topología de la simulación. Se ha decidido ocultarlas para reflejar con mayor claridad la dirección real de los arcos del modelo.

La topología resultante será:

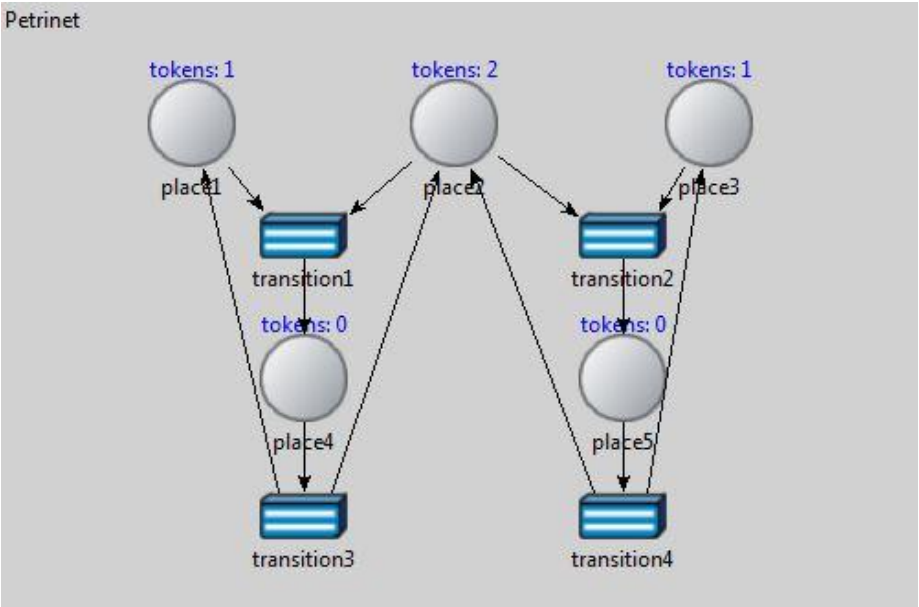

Ilustración 13 - Topología Petri 1

El apartado de configuración de este modelo en el archivo omnetpp.ini tendrá el siguiente aspecto:

```
[Config petrinet]
network = Petrinet
*.place1.tokens = 1
*.place2.tokens = 2
*.place3.tokens = 1
*.place4.tokens = 0
*.place5.tokens = 0

*.transition1.firingTime = exponential(3s)
*.transition2.firingTime = exponential(3s)
*.transition3.firingTime = exponential(3s)
*.transition4.firingTime = exponential(3s)

**.result-recording-modes = all
```

En esta configuración el módulo Place1 comenzará con un token, el módulo Place2 lo hará con dos y el módulo Place3 lo hará con uno. Los módulos Place4 y 5 comenzarán con cero tokens. Los cuatro módulos Transition del modelo han sido configurados para que calculen el tiempo de retardo en el disparo a partir de una distribución exponencial con media de 3 segundos.

La siguiente imagen muestra el funcionamiento del modelo durante la simulación:

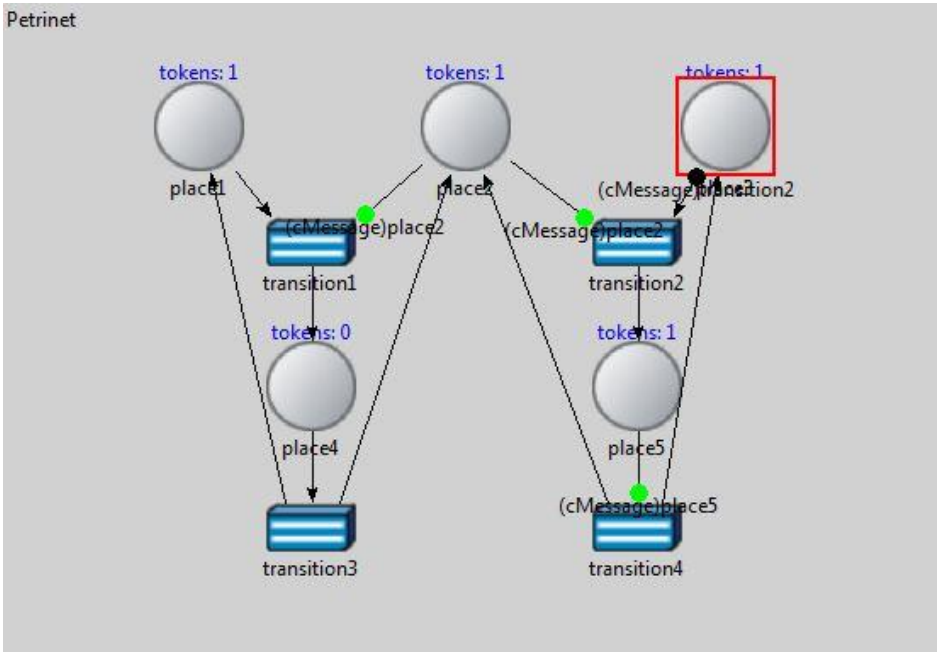

Ilustración 14 – Simulación Petri 1

La simulación de este modelo generará archivos con con la grabación de las estadísticas de los módulos. Estos datos generados dependerán de la configuración de grabación hayamos escrito en el archivo omnetpp-ini para el modelo. Los archivos generados estarán ubicados en la carpeta results del proyecto.

La siguiente variante de petrinet incluirá un arco de multiplicidad 2 (arco múltiple) y un arco de multiplicidad 0 (arco inhibidor) para comprobar el correcto funcionamiento del modelo en caso de que se usen. Los submódulos serán los mismos y las conexiones serán las siguientes:

```
    connections:
        place1.outputport++ --> Arc --> transition1.inputport++;
        transition1.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place1.outputportin++;
        place2.outputport++ --> Arc{ @display("t=2,t;ls=,1,s");
                    multiplicity = 2; } --> transition1.inputport++;
        transition1.inputportout++ --> Arc{ @display("ls=,0");
                    multiplicity = 2; }--> place2.outputportin++;
        place2.outputport++ --> Arc{ @display("t=0,t;ls=,1,s");
                     multiplicity = 0; } --> transition2.inputport++;
```

```
        transition2.inputportout++ --> Arc{ @display("ls=,0");
                  multiplicity = 0; }--> place2.outputportin++;
        place3.outputport++ --> Arc --> transition2.inputport++;
        transition2.inputportout++ --> Arc { @display("ls=,0"); } -->
                  place3.outputportin++;
        transition1.outputport++ --> Arc --> place4.inputport++;
        transition2.outputport++ --> Arc --> place5.inputport++;
        place4.outputport++ --> Arc --> transition3.inputport++;
        transition3.inputportout++ --> Arc { @display("ls=,0"); } -->
                  place4.outputportin++;
        place5.outputport++ --> Arc --> transition4.inputport++;
        transition4.inputportout++ --> Arc { @display("ls=,0"); } -->
                  place5.outputportin++;
        transition3.outputport++ --> Arc --> place1.inputport++;
        transition3.outputport++ --> Arc --> place2.inputport++;
        transition4.outputport++ --> Arc --> place3.inputport++;
```

La topología resultante será:

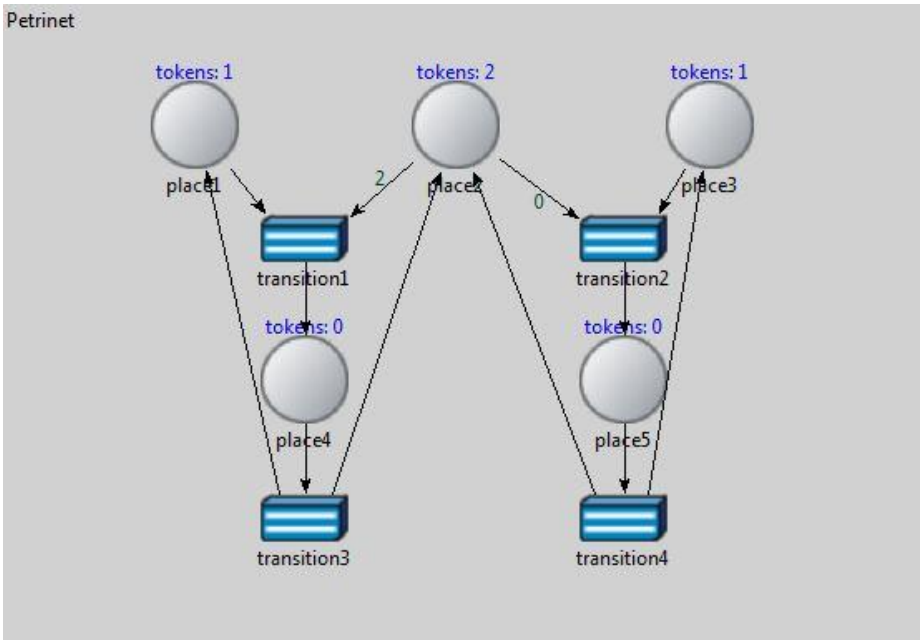

Ilustración 15 - Topología Petri 2

El apartado en el archivo de configuración de la anterior topología será el mismo para ésta.

5.5 Módulo InmTransition.

Se trata de un módulo simple. Está compuesto por los ficheros InmediateTr.cc y InmediateTr.h que contienen la implementación del módulo en C++, y por el fichero InmediateTr.ned que contiene la descripción del módulo escrita en lenguaje NED.

El módulo InmTransition se podrá disparar consumiendo (o restando) tokens de los módulos Place precedentes o produciendo (o sumando) tokens en sus módulos Place sucesores. Durante la simulación irá recibiendo autorizaciones o prohibiciones de disparo por parte de los módulos Place precedentes.

A diferencia del módulo Transition, el disparo del módulo InmediateTr no será retardado, es decir, se producirá en el mismo instante que consiga la autorización de disparo de todos sus módulos Place precedentes.

Para evitar conflictos de disparo entre dos o más módulos, todos los módulos InmediateTr tendrán una prioridad de disparo asignada, y además, se conectarán al módulo SolveConflict. Este último módulo será el encargado de decidir el orden de disparo de los módulos InmediateTr en conflicto, y como criterio de ordenación usará la prioridad de éstos en primer lugar, y en caso de que se dé un empate en las prioridades, usará la aleatoriedad.

5.5.1 Descripción NED del módulo InmediateTr.

```
simple InmediateTr
{
    parameters:
        //...
        int priority;
    gates:
        input inputport[];
        output outputport[];
        output inputportout[];
        inout solveconflict;
}
```

En la descripción NED de InmediateTr se ha establecido que éste será un módulo simple. Su parámetro será priority o la prioridad de disparo. Este parámetro será consultado por el módulo SolveConflict cuando se haya producido la activación de varios módulos InmediateTr en el mismo instante y haya que decidir cuál de ellos se disparará primero. El valor del parámetro de prioridad se configurará en el fichero omnetpp.ini. También tendrá tres arrays de puertas y una puerta bidireccional (inout) que permitirá la realizar la conexión con el módulo SolveConflict. Al igual que en el módulo Transition, los arrays de puertas serán inputport y inputportout que permitirán la interacción con los módulos Place predecesores, y el array outputport que permitirá la interacción con los módulos Transition sucesores.

En la siguiente imagen se mostrarán los tipos de puertas del módulo InmediateTr. y la conectividad con la puertas de los módulos Place y SolveConflict.

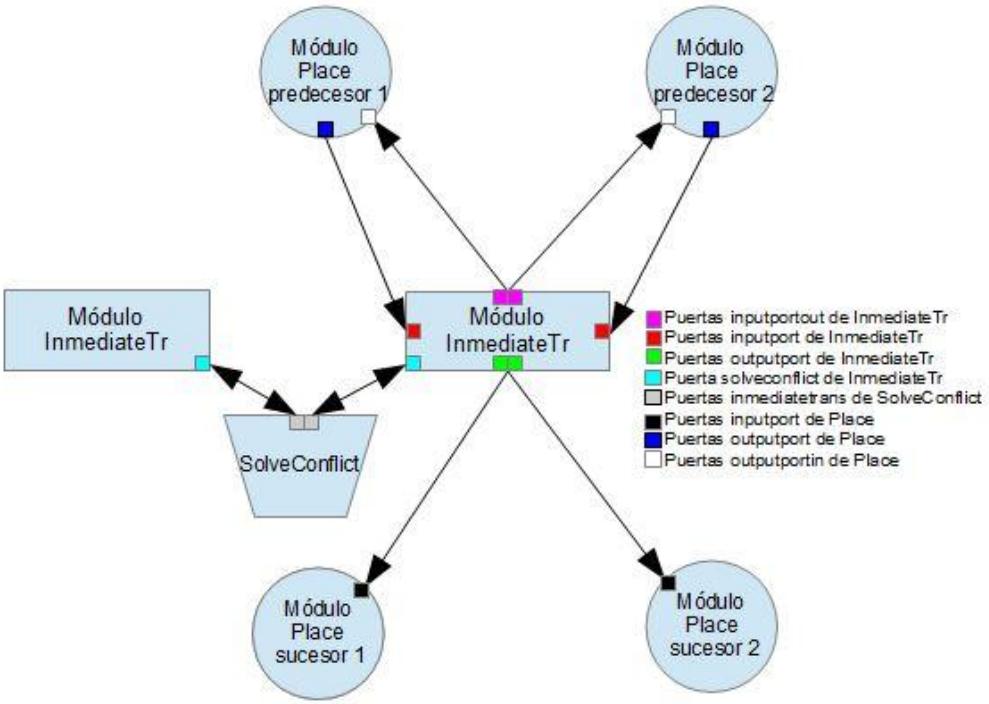

Ilustración 16 - Conectividad módulo InmediateTr

Al igual que en el módulo Transition, las comunicaciones entre los módulos Place predecesores y el módulo InmediateTr son bidireccionales y además necesitarán dos conexiones: de InmediateTr a Place y de Place a InmediateTr, mientras que las comunicaciones entre el módulo Transition y los módulos Place sucesores son unidireccionales y necesitarán una conexión de Place a Transition.

Como novedad, este módulo incluirá la puerta solveconflict cuyo objetivo será conectar los módulos InmediateTr con el módulo SolveConflict. Aunque la conexión con InmediateTr sea bidireccional, al haber definido la puerta solveconflict y las puertas inmediatetrans del módulo SolveConflict como inout, no será necesario hacer la conexión en los dos sentidos y bastará con usar una única conexión.

5.5.2 Declaración de la clase InmediateTr.

```
class InmediateTr : public cSimpleModule
{
  public:
    //Constructor y destructor
  private:
    //...
    cMessage *disparo; // declaración del mensaje de disparo
    int priority; //inmediate transition priority
    int e, s; //e número de puertas de entrada, s número de puertas de
salida
    int multout, multinout;
    //multout: multiplicidad del canal de una puerta de salida outputport
```

```
    //multinout: multiplicidad del canal de una puerta de salida
inputportout
    int solveack;
    // si InmTransition tiene permiso para enviar mensajes a SolveConflict
    //valdrá 1, sino su valor será 0
    std::vector<std::string> notemptyplaces;
    //vector que guardará los nombres de los lugares predecesores que le
    //hayan dado permiso de disparo
    //...
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    //...
};
```

Los atributos a destacar de esta clase son un objeto de tipo cMessage, seis variables de enteros y un vector de strings. El mensaje "disparo" será usado para enviar una solicitud de disparo al módulo SolveConflict. La variable priority albergará la prioridad de disparo asignada al módulo. Las variables e y s servirán para almacenar el número de puertas de entrada y el de salida, respectivamente. La variable multout albergará la multiplicidad del canal de una puerta de salida outputport, mientras que la variable multinout albergará la multiplicidad del canal de una puerta de salida inputportout. La variable solveack servirá para permitir enviar mensajes al módulo SolveConflict. El vector de strings servirá para guardar los nombres de los lugares predecesores que le hayan dado autorización para disparar.

Los métodos a destacar de esta clase son initialize y handleMessage. El método initialize será llamado por el módulo al inicio de la simulación e inicializará las variables necesarias para dejar el módulo preparado para la simulación, mientras que el método handleMessage será llamado por el módulo cada vez que éste reciba un mensaje. Este mensaje será un mensaje externo procedente de otro módulo (Place o SolveConflict) ya que en este módulo no se usarán auto mensajes.

5.5.3 Método initialize.

```
void InmediateTr::initialize()
{
    priority=par("priority");
    solveack=1;
    e=gateSize("inputport");
    s=gateSize("outputport");
    //...
}
```

Las variables priority, solveack, e y s serán inicializadas. La variable priority toma el valor que se le haya dado en el archivo de configuración omnetpp.ini a través del método para(). La variable solveack será inicializada a 1 para permitir el primer envío de mensajes a los módulos SolveConflict. Las variables e y s usan el método gateSize() para tomar un valor igual al número de puertas inputport de entrada y un valor igual al número de puertas outputport de salida, respectivamente.

5.5.4 Método handleMessage.

```
void InmediateTr::handleMessage(cMessage *msg)
{
    //Declaración variables locales método:
    cMessage *message;
    std::string msgName;
    int msgKind;
    int isincluded; //existe el lugar en el vector noemptyplaces?
    //...

    //SECCIÓN 1
    msgKind=msg->getKind();//nombre del mensaje recibido
    msgName=msg->getName();//kind del mensaje recibido
    delete msg;//borra mensaje entrante

    //SECCIÓN 2
    if(msgKind==0){ // llega mensaje "no disparar" de un Place
        for (unsigned i=0; i<notemptyplaces.size() ; i++){//recorrer vector
            if(notemptyplaces.at(i).compare(msgName)==0){ //si existe Place
                notemptyplaces.erase(notemptyplaces.begin()+i); //borrarlo
            }
        }
    }//... SECCIÓN 3
```

Sección 1: Está sección será idéntica a la sección 1 del código de handleMessage del módulo Transition, con la excepción de que no será necesario hacer distinción entre mensajes y auto mensajes a la hora del borrado, ya que estos últimos no se usarán en el módulo.

Sección 2: Está sección se parecerá a la sección 2 del código de handleMessage del módulo Transition, pero a diferencia de ésta, únicamente se realizará el borrado del nombre del lugar precedente en el vector y no se hará la cancelación del auto mensaje, ya que el módulo InmTransition no hace uso de los auto mensajes.

Implementación: Se recorrerá el vector notemptyplaces, que contiene los nombres de módulos Place que han autorizado el disparo, comprobando si el nombre del módulo Place que ha enviado el mensaje está entre ellos. En caso de que este nombre exista en el vector, se borrará.

```
    //SECCIÓN 3
    else if(msgKind==1){ // llega mensaje "disparar" de un Place
        isincluded=0; //inicializar
        for (unsigned i=0; i<notemptyplaces.size() ; i++){//recorrer vector
            if(notemptyplaces.at(i).compare(msgName)==0)//Place incluído?
                isincluded=1;
        }
        if(isincluded==0){//si no esta incluido, meterlo
            notemptyplaces.push_back(msgName); //meterlo al vector
        }
        if((int)notemptyplaces.size()==e){ //n places precedentes = n entr
            if(solveack==1){//si puede mandar el mensaje a solve conflict
                solveack=0;//no permitir envíos hasta recibir respuesta
```

```
                disparo = new cMessage(getName(),priority);
                send(disparo, "solveconflict$o");//enviar mensaje a
                                                 //SolveConflict
            }
        }
    }//... SECCIÓN 4
```

Sección 3: Esta sección tiene similitudes con la sección 3 del código de handleMessage del módulo Transition, pero a diferencia de ésta, cuando el módulo tenga el permiso de disparo de todos los módulos Place precedentes se enviará inmediatamente un mensaje al módulo SolveConflict. Este mensaje contendrá el nombre del módulo InmediateTr remitente y su prioridad de disparo. Ambas variables serán usadas por SolveConflict para establecer un orden de disparo si se activan varios módulos InmediateTr simultáneamente.

Implementación: Se recorrerá el vector notemptyplaces comprobando si éste contiene el nombre del módulo Place remitente del mensaje. Si este nombre no está incluido en el vector se meterá, mientras que si está incluido no se podrá volver a meter.

Finalmente se comprobará si el número de nombres de módulos Place que contiene el vector es igual al número de entradas inputport y si la variable solveack vale 1. En caso de que se cumplan condiciones, el valor de la variable solveack cambiará a 1 para evitar volver a enviar mensajes mientras InmediateTr esté a la espera de una respuesta de SolveConflict, y se enviará un mensaje al módulo SolveConflict con el nombre del módulo InmediateTr remitente y su prioridad de disparo.

```
    //SECCIÓN 4
    else if(msgKind==3){//SolveConflict permite el disparo
        solveack=1;//mensaje recibido, permitir mensajes a solveconflict
        if((int)notemptyplaces.size()==e){//si lugares = n entradas
            notemptyplaces.clear();//vaciar el vector
            for (int j=0; j<s; j++){// lugares sucesores aviso para que sumen
                multout=gate("outputport", j)->getChannel()->par("multiplicity");
                message = new cMessage(getName(), multout);//Mensaje suma
                send(message, "outputport", j);
            }
            for (int i=0; i<e; i++){//mandar a lugares predecesores aviso
                                    //para que resten y paren a las otras transiciones
                multinout=gate("inputportout", i)->getChannel()->par("multiplicity");
                if(multinout==0){//Si es arco inhibidor
                    message = new cMessage(getName(), 0); //Sumar 0
                    send(message, "inputportout", i);
                }
                else{
                    message = new cMessage(getName(), -multinout);
                                                    //Mensaje resta
                    send(message, "inputportout", i);
                }
            }
        }
    }
}//... end handleMessage
```

SECCIÓN 4: El funcionamiento de esta sección también será similar al de la sección 4 del código de handleMessage del módulo Transition, pero a diferencia de ésta, se ejecutará cuando el módulo InmediateTr reciba un mensaje de permiso de disparo por parte del módulo SolveConflict. Cuando se reciba este tipo de mensaje, el módulo volverá a permitir el envío de mensajes a SolveConflict y comprobará si sigue teniendo todos los permisos de disparo de sus módulos Place precedentes. Si está condición se sigue dando, se producirá el disparo.

Implementación: En primer lugar, se cambiará el valor de la variable solveack a 1 para permitir el envío de mensajes al módulo SolveConflict. Tras esto, se vaciará el vector notemptyplaces. Para que se pueda volver a producir el disparo el módulo InmediateTr, este vector deberá volver a contener todos los nombres de los módulos Place precedentes.

Después, se realizará el envío de mensajes a todos los módulos Place sucesores para que sumen una cantidad de tokens igual a la multiplicidad de canal que se esté usando para la conexión.

Finalmente, se realizará el envío de mensajes a todos los módulos Place predecesores. Estos mensajes tendrán dos objetivos:

- Restar una cantidad de tokens igual a la multiplicidad de canal que se esté usando para la conexión. Si en la conexión se usa un arco inhibidor (multiplicidad 0) no se modificará la cantidad de tokens.
- Hacer que los módulos Place precedentes autoricen o prohíban el disparo a sus módulos InmediateTr sucesores, para poder cancelar un disparo antes de que llegue el permiso de disparo de SolveConflict, si se diera el caso.

5.5.5 Señales y obtención de estadísticas del módulo InmediateTr.

5.5.5.1 Descripción del módulo InmediateTr.

```
simple InmediateTr
{
    parameters:
      //...
      @signal[shotnumber](type="int");
      @signal[productivity](type="float");
      @statistic[ShotCount](source=shotnumber; record=last?,vector?);
      @statistic[Productivity](source=productivity; record=last?,vector?);
      //...
}
```

Al igual que en Transition, se han agregado al módulo las señales shotnumber y productivity, y las estadísticas ShotCount, y Productivity. Las dos señales serán emitidas usando el método emit desde la implementación del módulo, como veremos abajo. En cuanto a las estadísticas, las señales serán usadas como fuente de datos (source) en la

grabación de las éstas, y se ha optado por grabar un escalar (last) y el vector de manera opcional.

5.5.5.2 Declaración de la clase InmediateTr.

```
class InmediateTr : public cSimpleModule
{
  private:
    //...
    simsignal_t shotcountsignal, productivitysignal;
    int shotCount;// how many times the inmtransition has been shot
  protected:
    //...
    virtual void finish();
};
```

En la parte de la implementación del módulo se han declarado dos identificadores de señal (shotcountsignal y productivitysignal) y una variable entera llamada shotCount que servirá para contar las veces que el módulo se ha disparado. También conviene resaltar el método finish de este módulo, puesto que las dos señales se emitirán cuando éste se ejecute.

5.5.5.3 Método initialize.

```
void InmediateTr::initialize()
{
    //...
    shotcountsignal=registerSignal("shotnumber");
    productivitysignal=registerSignal("productivity");
    shotCount=0;
}
```

En este método se ha usado el método registerSignal para registrar los nombres de señal shotnumber y productivity (estos nombres serán los mismos que se han usado en la descripción NED del módulo), que devolverá identificadores de señal que se guardarán en las variables shotcountsignal y productivitysignal, respectivamente. También se inicializará la variable shotChount a 0.

5.5.5.4 Método handleMessage.

```
void InmediateTr::handleMessage(cMessage *msg)
{
    //...
    else if(msgKind==3){//SolveConflict manda mensaje de disparo
    //...
        if((int)notemptyplaces.size()==e){//si lugares = n entradas, dispara
            //...
            shotCount++;
            //...
        }
    }
}
```

Cada vez que llegue una autorización de disparo por parte del módulo SolveConflct al modulo Transition y además se cumpla la condición de disparo, se producirá un disparo y se aumentará en uno el contador de disparos shotCount.

5.5.5.5 Método finish.

```
void InmediateTr::finish()
{
    emit(shotcountsignal, shotCount);
    emit(productivitysignal, shotCount/simTime());
}
```

Este método se ejecutará al final de la simulación y es entonces cuando se emitirán la señal shotcountsignal con la cuenta de disparos, y la señal productivitysignal con la productividad de disparos del módulo

5.6 Módulo SolveConflict.

Se trata de un módulo simple. Está compuesto por los ficheros SolveConflict.cc y SolveConflict.h que contienen la implementación del módulo en C++, y por el fichero SolveConflict.ned que contiene la descripción del módulo escrita en lenguaje NED.

Este módulo será el encargado de recibir mensajes con peticiones de disparo por parte de los módulos InmediateTr y de enviar permisos de disparo a éstos.

Los mensajes que recibe contendrán el nombre de cada módulo InmediateTr y su prioridad. Las prioridad, el número de puerta de entrada del mensaje a SolveConflict, y el nombre de cada módulo se irán metiendo en una lista ordenada en función de su prioridad. En caso de que haya empate en este criterio, se usará la aleatoriedad como criterio de ordenación.

Una vez se hayan metido en la lista ordenada todas las peticiones de disparo de los módulos InmediateTr que se hayan producido en un mismo instante, se borrará un nodo de la lista y se enviará un mensaje dando permiso de disparo al módulo InmediateTr del nodo borrado. En caso de que queden nodos en la lista ordenada, el módulo SolveConflict se enviará a sí mismo un auto mensaje para poder volver a borrar otro nodo de la lista y enviar un mensaje dando permiso de disparo a su correspondiente módulo InmediateTr.

5.6.1 Descripción NED del módulo SolveConflict.

```
simple SolveConflict
{
    parameters:
    //...
```

```
    gates:
    inout inmediatetrans[];
}
```

En la descripción NED de SolveConflict se ha establecido que éste será un módulo simple. No tendrá parámetros relevantes en la implementación. En cuanto a las puertas, tendrá un array de puertas bidireccionales (inout) llamado inmediatetrans que permitirá realizar conexiones con varios módulos InmediateTr.

En la siguiente imagen se mostrarán los tipos de puertas del módulo SolveConflict y la conectividad de éstas con las puertas de los módulos InmediateTr.

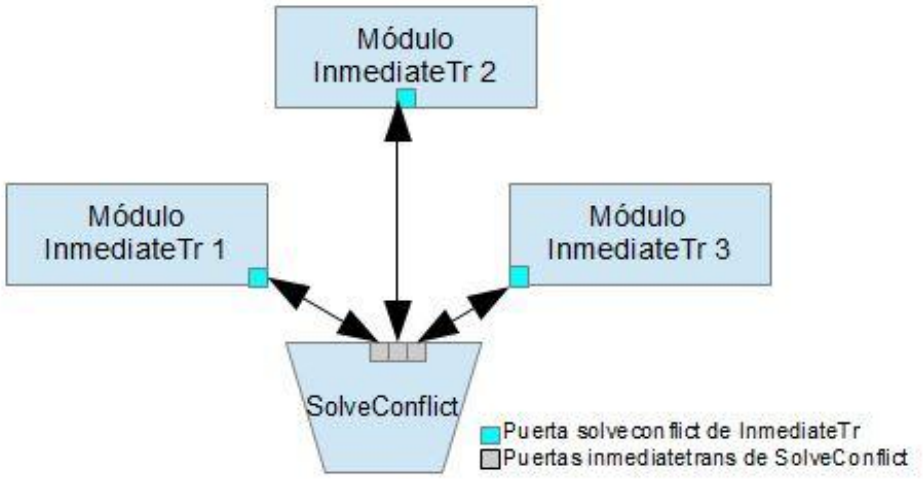

Ilustración 17 - Conectividad módulo SolveConflict

Este módulo incluirá el array de puertas solveconflict cuyo objetivo será conectarlo a todos los módulos InmediateTr de la topología de la red. Aunque la conexión con InmediateTr sea bidireccional, al haber definido las puertas inmediatetrans y cada puerta solveconflict del módulo InmediateTr como puertas inout, no será necesario hacer la conexión en los dos sentidos y bastará con usar una única conexión que conecte las puertas.

5.6.2 Declaración de la claseSolveConflict.

```
class SolveConflict : public cSimpleModule
{
  public:
    //Constructor y Destructor
  private:
    cMessage *solveconflict;
    std::list<InmTrans> transitionpriority;
    unsigned int solve;
    struct InmTrans {
        unsigned int priority, gateid;
        std::string Trname;
```

```
        InmTrans(int priority_, int gateid_, const std::string &Trname_)
                : priority(priority_), gateid(gateid_), Trname(Trname_)
        {};
    };
    static bool trans_priority_asc(const InmTrans& a, const InmTrans& b)
    // método de comparación usado por el sort de la lista
    {
        if(a.priority < b.priority)
            return true;
        else
            return false;
    };
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
```

Los atributos a destacar de esta clase son un objeto de tipo cMessage, una variable entera y una lista ordenada. Como se verá en el apartado de implementación, el auto mensaje "solveconflict" será planificado por el módulo al llegar la primera petición de disparo en un instante concreto, y también después de atender a un nodo de la lista transitionpriority mientras queden nodos en ésta. La variable entera solve actuará a modo de "barrera". Su valor cambiará a 1 cada vez que llegue la primera petición de disparo de un instante y servirá para poder planificar un único auto mensaje solveconflict para todas las peticiones en un mismo instante. Cuando la lista transitionpriority se quede vacía su valor cambiará a 0.

La lista ordenada transitionpriority, estará formada por nodos de la estructura InmTrans que se ha definido en la declaración de la clase. Esta estructura estará formada por dos variables enteras positivas y una variable string, y además tendrá un constructor asociado. Las dos variables enteras son priority que guardará la prioridad del módulo InmTransition remitente del mensaje y gateid que guardará la puerta por la que el módulo ha recibido el mensaje de InmTransition, mientras que la variable string será TrName que contendrá el nombre del módulo InmTransition remitente del mensaje.

Cada vez que se realice una ordenación de la lista, se hará uso del método trans_priority_asc incluido en la declaración de la clase. Este método servirá para establecer el criterio de ordenación en los nodos de la lista, que será su atributo priority, y será usado por el método de ordenación sort predefinido de la lista. Cada ordenación de la lista hará que los nodos se ordenen en función de su prioridad y de manera ascendente, es decir de menor a mayor.

Los métodos a destacar de esta clase son initialize y handleMessage. El método initialize será llamado por el módulo al inicio de la simulación e inicializará las variables necesarias para dejar el módulo preparado para la simulación, mientras que el método handleMessage será llamado por el módulo cada vez que éste reciba un mensaje. Este mensaje puede ser un mensaje externo procedente de un módulo InmediateTr o un automensaje procedente del mismo módulo SolveConflict que lo recibe.

5.6.3 Método initialize.

```
void SolveConflict::initialize()
{
    solve=0;
    solveconflict = new cMessage("solveconflict",0);
}
```

En este método únicamente se inicializarán la variable solve y el mensaje solveconflict. La variable solve se inicializará a 0 para permitir la planificación del auto mensaje "solveconflict" cuando llegue la primera petición de disparo. En cuanto al mensaje solveconflict, se inicializará con un valor 0 en su atributo Kind, que lo identificará como auto mensaje.

5.6.4 Método handleMessage.

```
void SolveConflict::handleMessage(cMessage *msg)
{
    //Declaración variables locales método:
    cMessage *aviso;
    int msgKind, msgArrivalGateId;
    std::string msgName;
    unsigned int nmaxprio, maxpriority, position;
    //

    //SECCIÓN 1
    msgKind=(unsigned int)msg->getKind();//prioridad disparo modulo remitente
    msgName =msg->getName();//nombre del módulo remitente
    if(msgName.compare("solveconflict")!=0) {//si no es el automensaje
        msgArrivalGateId=(unsigned int)msg->getArrivalGate()->getIndex();
        //obtener puerta de entrado al módulo del mensaje
    }

    //SECCIÓN 2
    if(msgKind>0){//mensaje externo petición de disparo si kind>0
                  //(si kind>0 es un prioridad de disparo de InmediateTr)
        if(solve==0){//solve desactivado? Primera petición de ese instante
            solve=1;//Si primera petición de disparo en ese instante
                    //activar solve
            transitionpriority.push_back(InmTrans(msgKind,msgArrivalGateId,msgName));
             //meter en la lista: prioridad, el número de puerta de entrada
             //del mensaje, y nombre de la transición
            scheduleAt(simTime(), solveconflict);//planificar envío
                                                 //del automensaje
        }
        else if (solve==1){ //solve activado?
                            // no es la primera petición de ese instante
            transitionpriority.push_back(InmTrans(msgKind,msgArrivalGateId,msgName));
             //meter en la lista: prioridad, el número de puerta de entrada
             //del mensaje, y nombre de la transición
        }
    delete msg;  //borrar mensaje externo -> su Kind>0
    }
```

Sección 1: Las variables locales msgName y msgKind tomarán los valores de los atributos del mensaje recibidos con los métodos getName() y getKind(). La variable msgName albergará el nombre del módulo InmediateTr remitente, la variable msgKind albergará la prioridad de disparo del módulo InmediateTr remitente (si vale 0 el mensaje recibido será tratado como automensaje). En caso de que el mensaje recibido no se trate de un auto mensaje, la variable msgArrivalGateId tomará el valor del número de puerta de entrada usando los métodos getArrivalGate() y getIndex().

Sección 2: Esta sección se ejecutará si el módulo SolveConflict recibe un mensaje y el valor de su atributo Kind es mayor que 0. Si esto sucede, estará recibiendo una prioridad de disparo de un módulo InmediateTr externo, y por lo tanto, no se tratará de un auto mensaje. Una vez cumplida esta condición se distinguirán dos casos: que al módulo le llegue la primera petición de disparo de un instante concreto (por ejemplo, en el instante 0,2 segundos), o que no se trate de la primera petición de disparo de ese instante. En el primer caso se planificará el envío

Implementación: Para distinguir los dos casos citados se usará la variable solve. Si se trata de la primera petición de disparo que llega en un instante concreto la variable solve valdrá 0, mientras que si no lo es la variable valdrá 1. En el primer caso se añadirá un nodo con la prioridad (msgKind), el número de puerta de entrada (msgArrvialGateId), y el nombre de la transición a la lista transitionpriority, y además se planificará un mensaje solveconflict. En el segundo caso únicamente se añadirá el nodo a la lista.

```
//SECCIÓN 3
    else if(msgKind==0){//se recibe automensaje
        transitionpriority.sort(trans_priority_asc);
        //ordenar la lista por prioridad ascendente
        maxpriority=transitionpriority.begin()->priority; //tomar prioridad maxima
        nmaxprio=0; //variable cuantos con prioridad máxima?
        for (std::list<InmTrans>::iterator it=transitionpriority.begin();
                  it != transitionpriority.end(); ++it){ //recorrer lista
            if(it->priority==maxpriority) //contar veces max prioridad
                nmaxprio++;
        }
        //cálculo de la posición a borrar de la lista
        position=rand()%nmaxprio; //entre todos los nodos con prioridad max
                                  //elegir uno aleatorio, rango 0 nmaxprio-1
        //borrado del nodo
        std::list<InmTrans>::iterator delTrans=transitionpriority.begin();
        //inicializar iterador de la lista
        advance(delTrans,position); //avanzar posición del nodo a borrar
        const char *Trname=delTrans->Trname.c_str();
        //convertir string del nodo a cons char
        int gateid=delTrans->gateid; //tomar puerta del nodo
        transitionpriority.erase(delTrans); //borrar nodo
        //dar permiso disparo a InmediateTr correspondiente
        aviso = new cMessage(Trname, 3);// mensaje permiso disparo
        send(aviso, "inmediatetrans$o", gateid);
        // enviar mensaje usando la puerta tomada del nodo
        if(transitionpriority.empty()==false){ //si lista no vacía
            solveconflict->setSchedulingPriority(1);//prioridad por defecto 0
           //MIRAR NOTA
```

```
            scheduleAt(simTime(), solveconflict); //enviar automensaje
        }
        else{//si la lista está vacía
            solve=0; //solve = 0, a la espera de que vengan más peticiones en
                     //otro instante
        }
    }
}
```

Sección 3: Esta sección se ejecutará si el módulo SolveConflict recibe un mensaje y el valor de su atributo Kind es 0. Si esto sucede, estará recibiendo un auto mensaje. En esta sección la lista transitionpriority será ordenada por prioridad de manera ascendente, se hará la elección del nodo a borrar de la lista, se borrará dicho nodo, y se enviará un mensaje permitiendo el disparo del módulo InmediateTr cuyo nombre estuviese en el nodo borrado.
Si tras esto, aún quedan nodos en la lista ordenada, el módulo se enviará un auto mensaje para poder volver a borrar otro nodo de la lista y enviar otro mensaje de permiso de disparo.

Implementación: Para ordenar la lista transitionpriority se usará el método sort de la misma y a éste se le pasará como parámetro el método trans_priority_asc definido en la declaración de la clase SolveConflict. Una vez se haya ordenado esta lista de manera ascendente, se creará un iterador y se tomará la prioridad del primer nodo, que será la máxima prioridad de disparo de la lista. Se contará el número de nodos con máxima prioridad de la lista y usando el método para el cálculo de aleatoriedad rand(), se elegirá uno de ellos, calculando su posición en la lista. El iterador avanzará hasta la posición calculada y de ese nodo se obtendrá el nombre del módulo InmediateTr y el número de puerta de entrada del mensaje de petición de disparo a SolveConflict. El nodo sobre el que se ha posicionado el iterador se borrará y se creará un mensaje de permiso de disparo que será enviado al módulo InmediateTr correspondiente del nodo borrado a través de la puerta por la que el mensaje ha entrado al módulo.

Finalmente, si quedan nodos en la lista transitionpriority, el módulo SolveConflict se enviará auto mensaje para poder volver a borrar otro nodo de la lista y enviar otro mensaje dando permiso de disparo a su correspondiente módulo InmediateTr. En caso de que la lista se quede vacía, el valor de la variable solve cambiará a 0.

5.7 Modelado de la red “Tres Procesos que Compiten por Dos Recursos”.

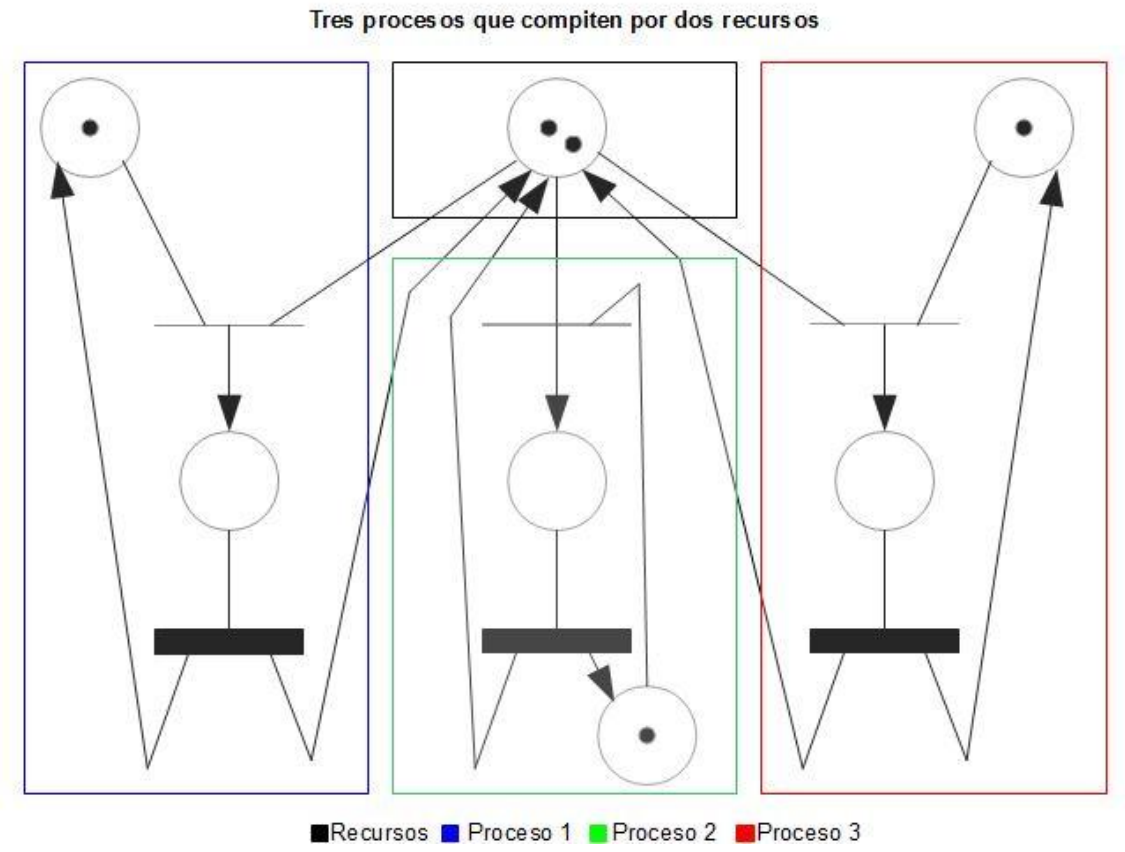

Ilustración 18 - Red Petri 3 procesos 2 recursos

Este modelo se ha ideado para probar el funcionamiento de los módulos Place, Transition, InmediateTr y SolveConflict. Al igual que en la red anterior, para las conexiones se ha definido el tipo de canal Arc (en referencia a los arcos de las redes de Petri), que es una extensión de IdealChannel y que además incluye un parámetro llamado multiplicity, cuyo valor por defecto será 1, que servirá para especificar la multiplicidad del arco. A continuación se muestra la descripción Ned de su topología:

```
network PetrinetInm3
{
    @display("bgb=672,419");
    types:
       channel Arc extends ned.IdealChannel
       {
           int multiplicity = default(1); //por defecto 1
       }
    submodules:
       place1: Place {@display("p=142,55;t=tokens: $tokens");}
       place2: Place {@display("p=324,55;t=tokens: $tokens");}
       place3: Place {@display("p=502,55;t=tokens: $tokens");}
       place4: Place {@display("p=204,216;t=tokens: $tokens");}
       place5: Place {@display("p=324,231;t=tokens: $tokens");}
       place6: Place {@display("p=450,216;t=tokens: $tokens");}
       place7: Place {@display("p=243,351;t=tokens: $tokens");}
```

```
        Inmtransition1: InmediateTr {display("p=204,140;t=p: $priority");}
        Inmtransition2: InmediateTr {@display("p=324,161;t=p: $priority");}
        Inmtransition3: InmediateTr {@display("p=450,140;t=p: $priority");}
        Transition4: Transition {@display("p=204,285");}
        Transition5: Transition {@display("p=417,351");}
        Transition6: Transition {@display("p=450,285");}
        solveConflict: SolveConflict {@display("p=62,231;b=49,33");}
    connections:
        place1.outputport++ --> Arc --> Inmtransition1.inputport++;
        Inmtransition1.inputportout++ --> Arc { @display("ls=,0"); } -->
                place1.outputportin++;
        place2.outputport++ --> Arc --> Inmtransition1.inputport++;
        Inmtransition1.inputportout++ --> Arc { @display("ls=,0"); } -->
                place2.outputportin++;
        place2.outputport++ --> Arc --> Inmtransition2.inputport++;
        Inmtransition2.inputportout++ --> Arc { @display("ls=,0"); } -->
                place2.outputportin++;
        place2.outputport++ --> Arc --> Inmtransition3.inputport++;
        Inmtransition3.inputportout++ --> Arc { @display("ls=,0"); } -->
                place2.outputportin++;
        place3.outputport++ --> Arc --> Inmtransition3.inputport++;
        Inmtransition3.inputportout++ --> Arc { @display("ls=,0"); } -->
                place3.outputportin++;
        Inmtransition1.outputport++ --> Arc --> place4.inputport++;
        Inmtransition2.outputport++ --> Arc --> place5.inputport++;
        Inmtransition3.outputport++ --> Arc --> place6.inputport++;
        place4.outputport++ --> Arc --> Transition4.inputport++;
        Transition4.inputportout++ --> Arc { @display("ls=,0"); } --> place4.outputportin++;
        place5.outputport++ --> Arc --> Transition5.inputport++;
        Transition5.inputportout++ --> Arc { @display("ls=,0"); } --> place5.outputportin++;
        Transition5.outputport++ --> Arc --> place7.inputport++;
        place7.outputport++ --> Arc --> Inmtransition2.inputport++;
        Inmtransition2.inputportout++ --> Arc { @display("ls=,0"); } -->
                place7.outputportin++;
        place6.outputport++ --> Arc --> Transition6.inputport++;
        Transition6.inputportout++ --> Arc { @display("ls=,0"); } --> place6.outputportin++;
        Transition4.outputport++ --> Arc --> place1.inputport++;
        Transition4.outputport++ --> Arc --> place2.inputport++;
        Transition5.outputport++ --> Arc --> place2.inputport++;
        Transition6.outputport++ --> Arc --> place2.inputport++;
        Transition6.outputport++ --> Arc --> place3.inputport++;
        Inmtransition1.solveconflict <--> { @display("ls=,0"); } <-->
                solveConflict.inmediatetrans++;
        Inmtransition2.solveconflict <--> { @display("ls=,0"); } <-->
                solveConflict.inmediatetrans++;
        Inmtransition3.solveconflict <--> { @display("ls=,0"); } <-->
                solveConflict.inmediatetrans++;
}
```

Nota 1: En la definición de las conexiones habrá que tener en cuenta que para añadir los arcos de unión entre un módulo Place precedente y un módulo Transition (o InmediateTr) se usarán conexiones unidireccionales, mientras que para los arcos de unión entre un módulo Transition y un módulo Place sucesor a éste, se usarán conexiones bidireccionales y la conexión además tendrá que realizarse en ambos sentidos (de Transition a Place y de Place a Transition). Para más información ver la conectividad de las puertas de los módulos Transition, InmediateTr y Place.

Nota 2: Para realizar una conexión entre un módulo InmTransition y el módulo SolveConflict bastará con hacer una única conexión debido a que las puertas que intervendrán en ambos módulos son bidireccionales (inout). Para más información ver la conectividad de los módulos InmediateTr y SolveConflict.

Nota 3: Las conexiones que usen el parámetro *@display("ls=, 0")* estarán siendo ocultadas por OMNeT++ en la representación de la topología de la simulación. Se ha decidido ocultarlas para reflejar con mayor claridad la dirección real de los arcos del modelo.

La topología resultante de la red en OMNeT++ será:

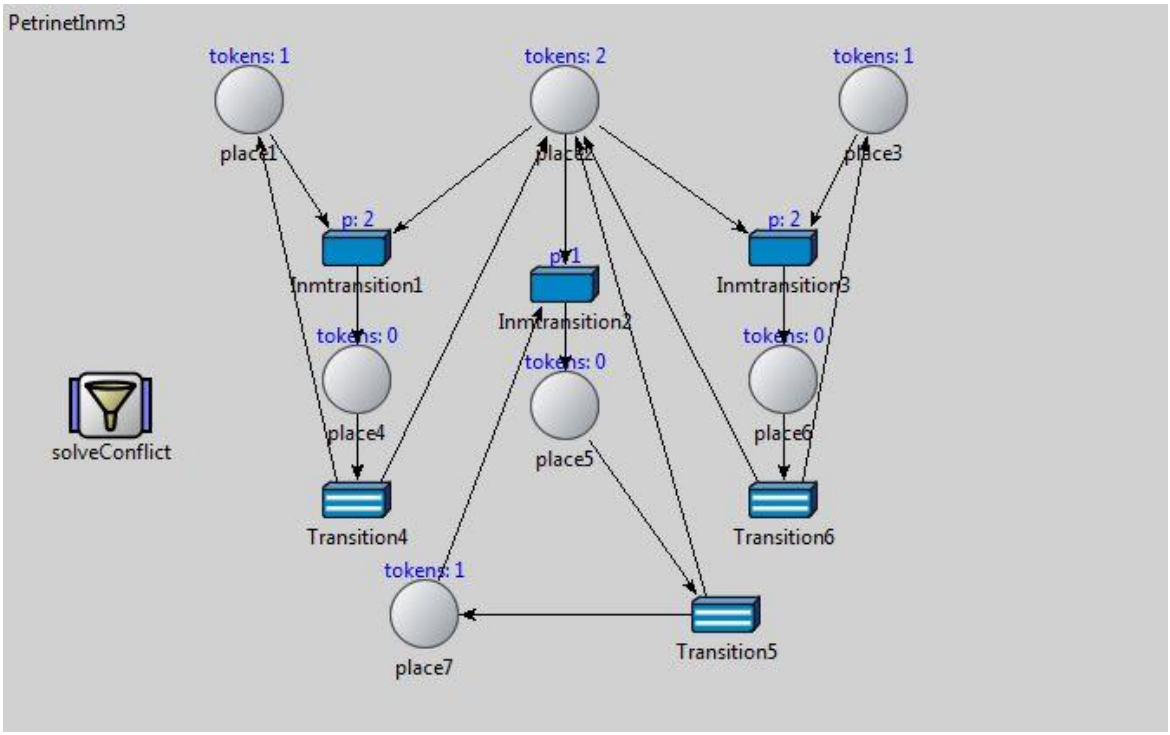

Ilustración 19 - Topología de la red "3 procesos, 2 recursos"

El apartado de configuración de este modelo en el archivo omnetpp.ini tendrá el siguiente aspecto:

```
[Config petrinetInm3]
network = PetrinetInm3
*.place1.tokens = 1
*.place2.tokens = 2
*.place3.tokens = 1
*.place4.tokens = 0
*.place5.tokens = 0
*.place6.tokens = 0
*.place7.tokens = 1

*.Inmtransition1.priority=2
*.Inmtransition2.priority=1
*.Inmtransition3.priority=2
*.Transition4.firingTime = exponential(3s)
*.Transition5.firingTime = exponential(3s)
*.Transition6.firingTime = exponential(3s)
**.result-recording-modes = +vector
```

En esta configuración los módulos Place1, Place3 y Place7 comenzarán con un token y el módulo Place2 lo hará con dos. Los demás módulos Place comenzarán con cero tokens. Los módulos InmTransition1, InmTransition2 y InmTransition3 comenzarán con prioridad dos, uno y dos respectivamente, siendo InmTransition2 el más prioritario. Los tres módulos Transition del modelo han sido configurados para que calculen el tiempo de retardo en el disparo a partir de una distribución exponencial con media de 3 segundos.

La siguiente imagen muestra el funcionamiento del modelo durante la simulación:

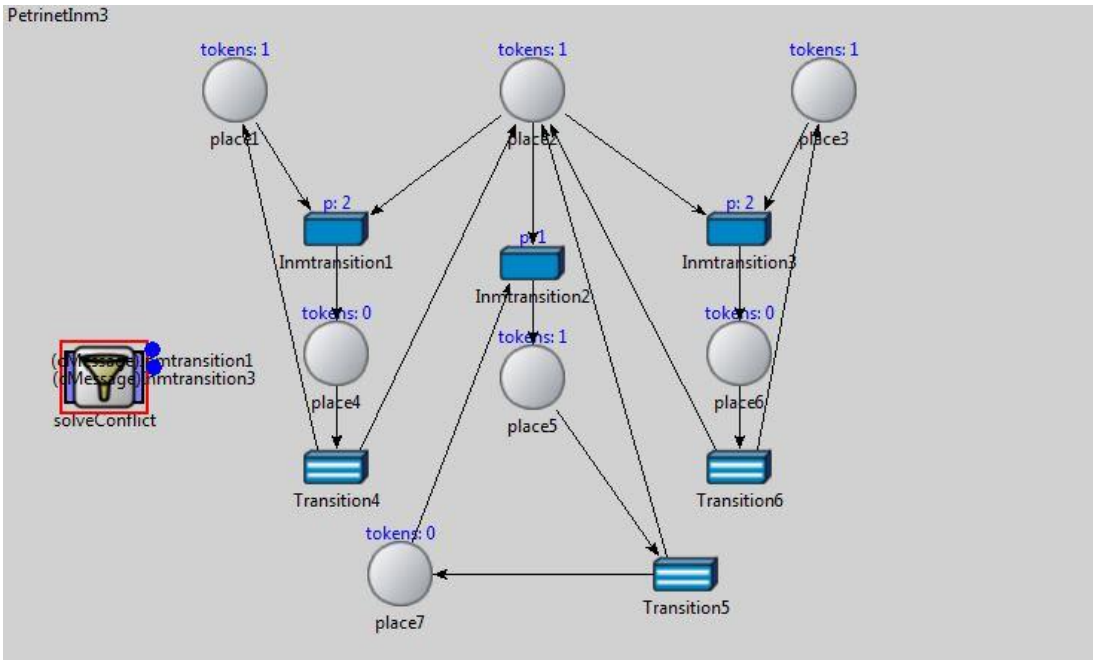

Ilustración 20 - Simulación de la red "3 procesos, 2 recursos"

La simulación de este modelo generará archivos con con la grabación de las estadísticas de los módulos. Estos datos generados dependerán de la configuración de grabación hayamos escrito en el archivo omnetpp-ini para el modelo. Los archivos generados estarán ubicados en la carpeta results del proyecto.

6. Simulación del multiprocesador.

6.1 Introducción.

Finalmente, con todos los módulos ya desarrollados se podrá realizar el modelado de un multiprocesador que previamente haya sido representado como una red de Petri.

Por multiprocesador se comprenderá un sistema que hará uso de varios procesadores, cada uno de ellos con su propia memoria caché, y que tendrán que hacer uso de buses disponibles para poder acceder a la memoria compartida.

A continuación estudiaremos un modelo de multiprocesador que ha sido representado mediante una red de Petri. En este modelo, las marcas o tokens adquirirán un significado diferente dependiendo de la ubicación de sus lugares en la red, y el retardo temporal de las transiciones retardadas servirá para simular el tiempo medio que tarda un procesador en cometer un fallo de lectura en una memoria caché o en la memoria principal.

6.2 Multiprocesador a modelar.

Se considerará un multiprocesador con p procesadores, m módulos de memoria compartida, y b buses. Cada procesador tendrá asociada una memoria caché asociada que será representada mediante la transición T1. Los procesadores accederán a esta memoria local hasta que se produzca un fallo de lectura y cuando esto ocurra, el procesador recuperará un trozo grande de información de la memoria compartida con el fin de minimizar el número de accesos a ésta. Para que un procesador pueda acceder a un módulo de memoria compartida tendrá que adquirir un bus libre. Este bus se liberará una vez haya concluido el acceso del procesador (en el modelo simplificado) o de los procesadores (en el modelo normal) a ese bloque de memoria compartida.

El tiempo de uso de cada caché estará dado por una distribución exponencial con media $1/\lambda$, mientras que el tiempo de acceso a la memoria compartida estará dado por una distribución exponencial con media de $1/\mu$. Se asumirá que el tiempo de adquisición y liberación de un bus será insignificante.

El siguiente modelo tendrá 5 procesadores (marcas en P1), 3 módulos de memoria (3 cajas), 2 buses (2 marcas en P2).

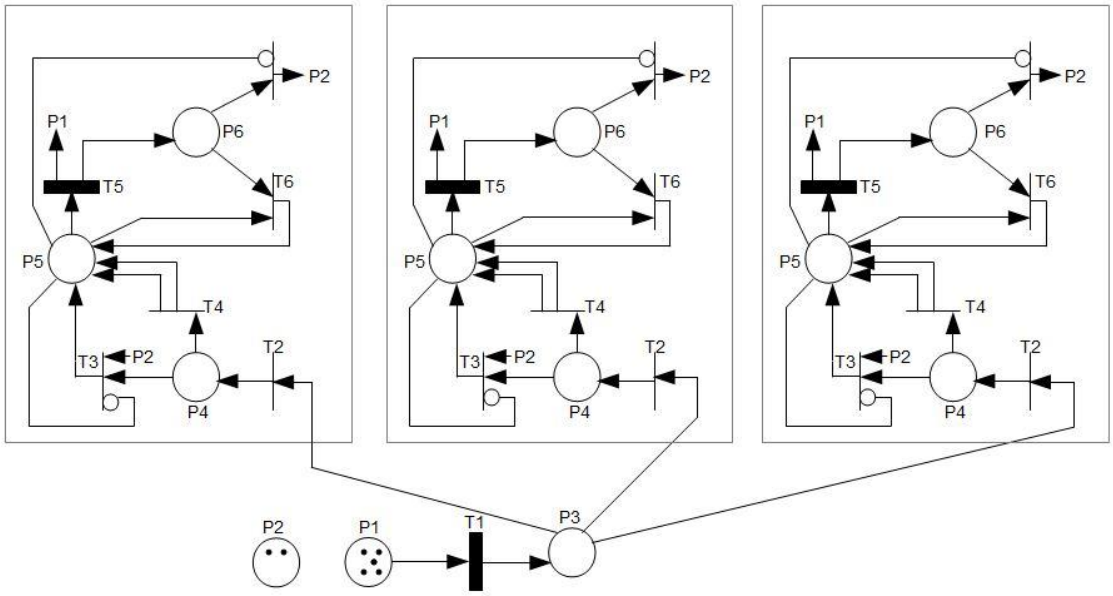

Ilustración 21 - Modelo del multiprocesador.

Como vemos, este modelo estará compuesto de 12 lugares, 4 transiciones retardadas, y 15 transiciones inmediatas. Los tokens del lugar P1 representarán el número de procesadores que están leyendo de sus memorias caché locales. Los tokens del lugar P2 representarán el número de buses disponibles para ser usados. Cada procesador realizará peticiones de acceso a memoria compartida con una tasa λ, de forma que la transición T1 se disparará con una tasa $m\lambda$, donde m será el número de marcas de P1. Esta transición usará la semántica de servidor DS (delay server o servidor de retraso).

Cada token en el lugar P3 representará un procesador que necesita seleccionar un módulo de memoria compartida.

Los módulos de memoria compartida y las acciones de acceso a éstos, como la adquisición o liberación de un bus estarán representados por secciones de la red de Petri idénticas agrupadas en "cajas". En este modelo, hay tres cajas y cada una contendrá una sección de la red de Petri.

Cada procesador (tokens en P3) seleccionará el módulo al que quiere acceder de la memoria compartida disparando una de las tres transiciones inmediatas T2. La probabilidad de selección del módulo será determinada de manera aleatoria. Una vez se haya disparado una de las transiciones T2 el procesador quedará a la espera (como token en el correspondiente lugar P4) de que se le conceda un bus para acceder al módulo de memoria compartida elegido

6.2.1 Política de acceso al bus y a la memoria compartida.

El acceso de los procesadores a memoria compartida a través de un bus se llevará a cabo en los siguientes pasos:

1- Un procesador (token en P3) elegirá uno de los tres módulos de memoria compartida. Una vez haya elegido módulo, quedará esperando adquirir un bus en el lugar P4.

2- Después realizará una petición de acceso al bus. El disparo de la transición T3 significará que la petición de bus ha sido concedida. El bus concedido también se usará para que el resto de procesadores hagan un intento de acceso al módulo de memoria elegido mientras dure el periodo de trabajo del procesador.

3- Los procesadores a la espera de acceder al módulo de memoria i estarán en el lugar P5. Pero sólo uno de ellos podrá usar a la vez el bus concedido para acceder al módulo de memoria debido a que la transición T5 usará la semántica de servidor SS (single server o servidor único).

4- Cuando la transición T5 se dispare el procesador que estaba accediendo a la memoria terminará, quedará libre y se añadirá como token al lugar P1.

5- Si el lugar P5 se ha quedado con algún procesador, éste comenzará inmediatamente a acceder al módulo de memoria usando el mismo bus que el procesador anterior. En caso de que no queden procesadores a la espera de usar el módulo de memoria en P5, la transición T7 se disparará y el bus será liberado.

El propósito de la transición T6 será eliminar los token extra generados por la transición T5 mientras queden marcas en P5. Esto es necesario para evitar la acumulación innecesaria de marcas en el lugar P6.

La política de acceso al bus adoptada minimizará el arbitraje del bus y puede ser recomendable para situaciones donde los niveles de contienda por el bus sean bajos. Sin embargo, para situaciones donde haya niveles altos de contienda por el bus podría ocasionar la inanición de los procesos o tiempos de acceso a memoria excesivamente largos.

Hay que mencionar que la complejidad de la subredes de Petri que contienen las cajas se debe a la política de arbitraje de bus elegida para el acceso a los módulos de memoria compartida, si ésta se simplificase como en el siguiente ejemplo las subredes contenidas en las cajas serían menos complejas.

6.3 Modelo Simplificado.

Si la política de arbitraje del bus es modificada de forma de que una vez el procesador haya terminado su acceso al módulo de memoria, éste se libere, cada caja se simplificará de la siguiente manera:

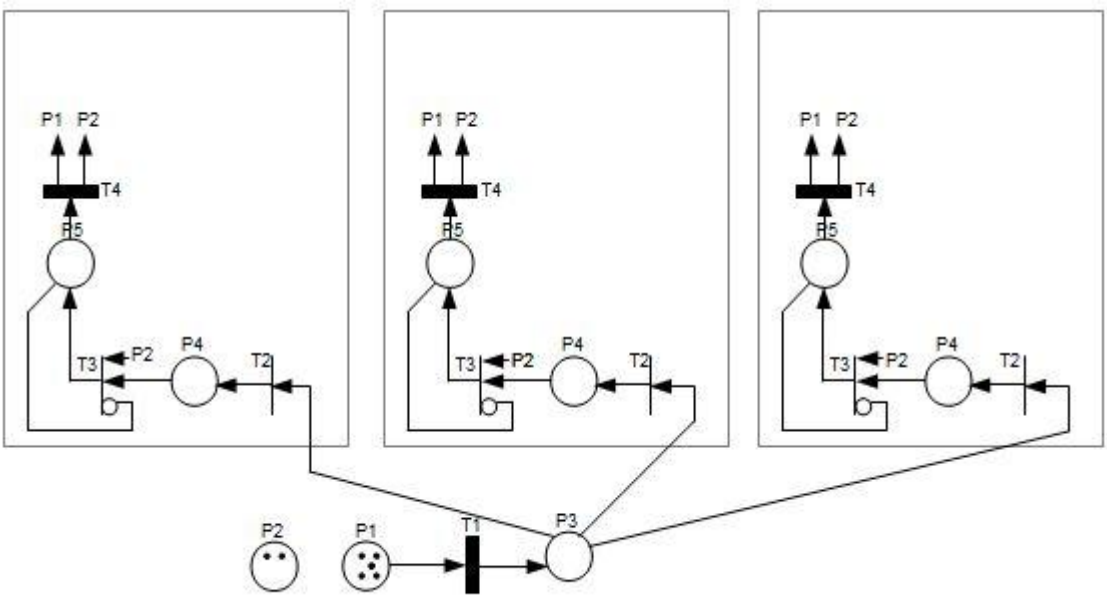

Ilustración 22 - Modelo del multiprocesador simplificado

La diferencia principal será que como mucho un único procesador (un único token) podrá avanzar al lugar P5 y usar la memoria. Cuando éste termine su acceso al módulo de memoria, liberará el bus y otro procesador (token) a la espera en cualquiera de los tres lugares P4 podrá avanzar a P5 y realizar el acceso al módulo de memoria si hay algún bus disponible (si hay tokens en P2).

6.4 Restricciones del modelo.

A partir de aquí nos centraremos únicamente en el modelo normal del multiprocesador y no en el simplificado. Este modelo presentará las siguientes restricciones:

- Este modelo se representarán los procesadores y buses como tokens mientras que los módulos de memoria estarán representados como subredes. Incrementar el número de buses o de procesadores será trivial, pero no el número de módulos de memoria. No será posible dar con un modelo cuya estructura sea independiente de su número de procesadores, memorias y buses.

- En este modelo se ha asumido que cada procesador funcionará a nivel estadístico de la misma manera. Si este no fuera el caso, se tendría que idear un modelo mucho más elaborado.

- Las transiciones normales o retardadas usadas en los modelos anteriores usan las semánticas de servidor SS (Single Server o servidor único) en el caso de la transición T1, y DS (Delay Server o servidor de retraso) en el caso de la transición T5.

Nos centraremos en la última restricción, ya que implicará cambios en el modelado del multiprocesador. El módulo Transition que se ha desarrollado para OMNeT++ únicamente podrá usar la semántica de servidor único o SS. Esto implicará un cambio en el modelado de la transición T1 del modelo.

6.5 Representación del Multiprocesador usando la semántica SS.

En el modelo inicial en la transición T1 se usaba la semántica de servidor retardado o DS. Al poder usar únicamente la semántica de servidor retardado o SS debido a que el módulo desarrollado para OMNeT++ sólo es compatible con ésta, la transición T1 tendrá que ser sustituida como se muestra en la imagen de abajo:

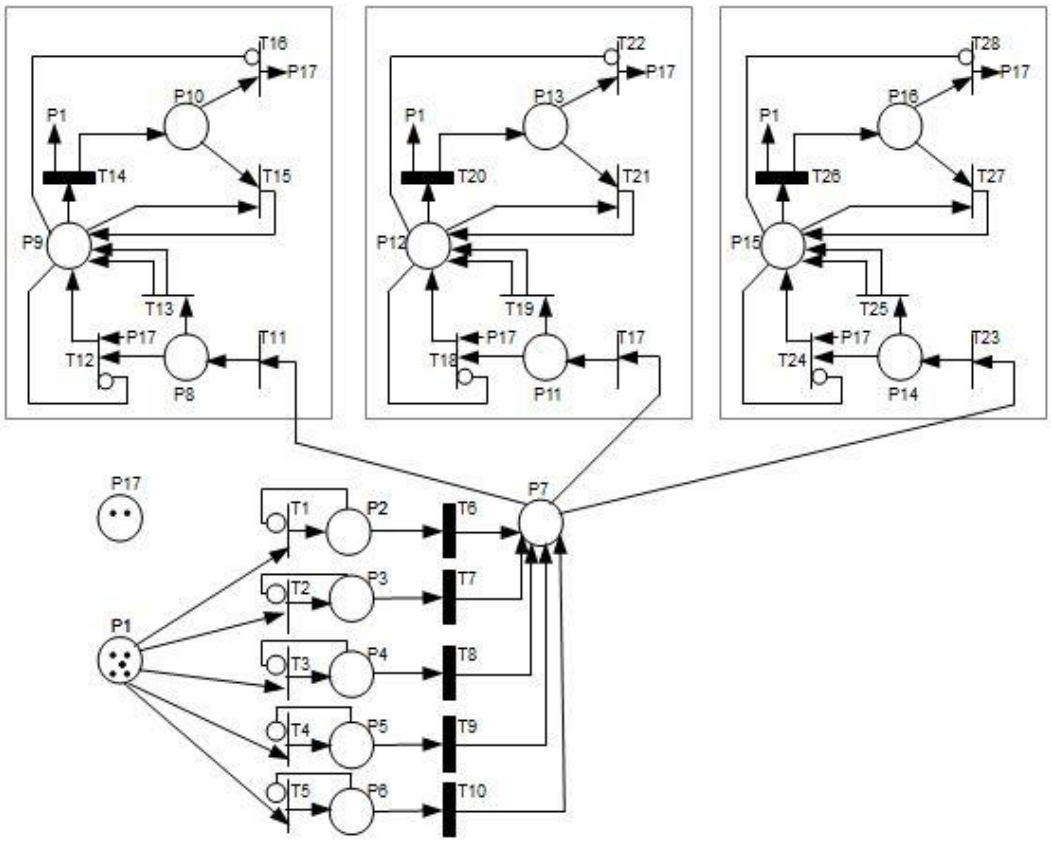

Ilustración 23 - Modelo del multiprocesador usando la semántica SS

La transición T1 ha sido sustituida por una subred de Petri. Las transiciones inmediatas sucesoras al lugar P1 (T1, T2, T3, T4 y T5) únicamente dejarán pasar una marca o token a sus lugares sucesores (P2, P3, P4, P5 y P6), debido a que tienen un arco inhibidor procedente de los lugares sucesores como entrada. El tiempo de retardo en el disparo de las transiciones T6, T7, T8, T9 y T10 simulará el tiempo que pasa hasta que se produzca un fallo de lectura en la caché local.

Esta subred estará diseñada exclusivamente para simular las lecturas de memoria caché de 5 procesadores. Para que este modelo pueda funcionar correctamente con una cantidad mayor (o menor) de procesadores será necesario añadir (o quitar) al modelo tantas subredes de este tipo como procesadores se quieran añadir (o quitar):

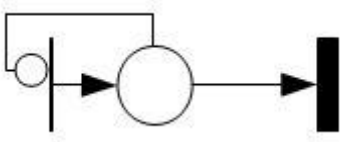

Ilustración 24 - Subred caché multiprocesador SS

6.6 Modelado del multiprocesador.

A continuación se muestra la descripción Ned del modelo multiprocesador. Se ha usado el modelo que usa únicamente la semántica de servidor SS en las transiciones retardadas. Debido a su extensión, no se ha incluido la descripción completa, pero se han comentado las partes más importantes. Para ver la descripción NED completa del modelo, abrir el fichero PetrinetInm4.ned del proyecto.

```
network PetrinetInm4
{
    @display("bgb=1408,883");
    types:
        channel Arc extends ned.IdealChannel
        {int multiplicity = default(1);}
    submodules:
       place1: Place {@display("p=116,641;t=tokens: $tokens");}
      #Definición de los módulos Place 2,3,4,5,6,7,8,9
      #10,11,12,13,14,15,16,17 Ver código.
      transition6: Transition {@display("p=498,525");}
      #Definición de los módulos Transition Ver código.
      transition1: InmediateTr {@display("p=213,525;t=p: $priority");}
      #Definición de los módulos InmediateTr 1,2,3,4,5,11,12,13,15,16
      #17,18,19,21,22,23,24,25,27,28. Ver código.
      solveConflict: SolveConflict {@display("p=39,308");}
    connections:
     // MÓDULO MEMORIA MEMORIA 1
     //transition 11
      place7.outputport++ --> Arc --> transition11.inputport++;
      transition11.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place7.outputportin++;
      transition11.outputport++ --> Arc --> place8.inputport++;
      //transition 12
      place8.outputport++ --> Arc --> transition12.inputport++;
      transition12.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place8.outputportin++;
      place9.outputport++ --> Arc { @display("t=0,t;ls=,1,s");
               multiplicity = 0; } --> transition12.inputport++;
      transition12.inputportout++ --> Arc { @display("ls=,0");
               multiplicity = 0; } --> place9.outputportin++;
      place17.outputport++ --> Arc --> transition12.inputport++;
      transition12.inputportout++ --> Arc { @display("ls=,0"); } -->
                    place17.outputportin++;
      transition12.outputport++ --> Arc --> place9.inputport++;
      //transition 13
```

```
        place8.outputport++ --> Arc --> transition13.inputport++;
        transition13.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place8.outputportin++;
        place9.outputport++ --> Arc --> transition13.inputport++;
        transition13.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place9.outputportin++;
        transition13.outputport++ --> Arc { @display("t=2,t;ls=,1,s");
                        multiplicity = 2; } --> place9.inputport++;
        //transition 14
        place9.outputport++ --> Arc --> transition14.inputport++;
        transition14.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place9.outputportin++;
        transition14.outputport++ --> Arc --> place10.inputport++;
        transition14.outputport++ --> Arc { @display("ls=,0"); } -->
                        place1.inputport++;

        //transition 15
        place10.outputport++ --> Arc --> transition15.inputport++;
        transition15.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place10.outputportin++;
        place9.outputport++ --> Arc --> transition15.inputport++;
        transition15.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place9.outputportin++;
        transition15.outputport++ --> Arc --> place9.inputport++;

        //transition 16
        place10.outputport++ --> Arc --> transition16.inputport++;
        transition16.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place10.outputportin++;
        place9.outputport++ --> Arc { @display("t=0,t;ls=,1,s");
                 multiplicity = 0; } --> transition16.inputport++;
        transition16.inputportout++ --> Arc { @display("ls=,0");
                multiplicity = 0; } --> place9.outputportin++;
        transition16.outputport++ --> Arc { @display("ls=,0"); } -->
                        place17.inputport++;

        //conexiones transiciones inmediatas - solveconflict en MÓDULO 1
        transition11.solveconflict <--> Arc { @display("ls=,0"); } <-->
                        solveConflict.inmediatetrans++;
        //...
        //conexiones de transition 12,13,14,15,16 con SolveConflict


        //MÓDULO MEMORIA 2
        //Las conexiones serán idénticas a las del BLOQUE 1, pero usando los módulos
        //Place 11, 12 y 13 y los módulos Transition 17, 18, 19, 20, 21 y 22,
        //Todas las transiciones inmediatas se conectarán al módulo SolveConflict
        //Para más información ver el código del proyecto.

        //MÓDULO MEMORIA 3
        //Las conexiones serán idénticas a las del BLOQUE 1 y 2, pero usando los módulos
        //Place 14, 15 y 16 y los módulos Transition 23, 24, 25, 26, 27 y 28,
        //Todas las transiciones inmediatas se conectarán al módulo SolveConflict
        //Para más información ver el código del proyecto.

        //CONEXIONES EXTERNAS

        //arcos a transiciones inmediatas
        place1.outputport++ --> Arc --> transition1.inputport++;
        transition1.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place1.outputportin++;
        //...
        //conexiones place 1 a transition 2,3,4,5 en los dos sentidos

        //arcos de transiciones inmediatas a place sucesores
        transition1.outputport++ --> Arc --> place2.inputport++;
        //...
        //conexiones transition 2,3,4,5 a place 3,4,5,6

        //arcos de places a transiciones retardadas
        place2.outputport++ --> Arc --> transition6.inputport++;
```

```
        transition6.inputportout++ --> Arc { @display("ls=,0"); } -->
                        place2.outputportin++;
        //...
        //conexiones place 3,4,5,6 a transition 7,8,9,10 en los dos sentidos



        //arcos inhibidores a transiciones inmediatas
        place2.outputport++ --> Arc { @display("t=0,t;ls=,1,s");
                multiplicity = 0; } --> transition1.inputport++;
        transition1.inputportout++ --> Arc { @display("ls=,0");
                multiplicity = 0; } --> place2.outputportin++;
        //...
        //arcos inhibidores place 2,3,4,5,6 a transition 2, 3, 4, 5 en los dos sentidos

        //arcos a place 7
        transition6.outputport++ --> Arc --> place7.inputport++;
        //...
        //arcos desde transition 7,8,9,10 a place 7

        //Conexiones transiciones inmediatas con SolveConflict
        transition1.solveconflict <--> Arc { @display("ls=,0"); } <-->
        solveConflict.inmediatetrans++;
        //...
        //conexiones de transition 2,3,4,5 con SolveConflict
}
```

En esta topología de red se usarán los cuatro módulos desarrollados: Place, Transition, InmediateTr y SolveConflict. Las conexiones entre módulos se han dividido en cuatro bloques: las conexiones externas, las que se han realizado en la "caja" del bloque de memoria 1, las que se han realizado en la "caja" del bloque de memoria 2, y las que se han realizado dentro de la caja del bloque de memoria 3. Para las conexiones se ha definido el tipo de canal Arc (en referencia a los arcos de las redes de Petri), que es una extensión de IdealChannel y que además incluye un parámetro llamado multiplicity, cuyo valor por defecto será 1, que servirá para especificar la multiplicidad del arco.

Dependiendo de la ubicación, cada módulo Place y sus marcas tendrán un significado diferente. Los Módulos Transition servirán para simular tiempos de acceso a la memoria caché de cada procesador o a la memoria compartida. Todos los módulos Módulos InmediateTr se conectarán a un único módulo SolveConflict, que será el encargado de gestionar los disparos de todos ellos.

Nota 1: En la definición de las conexiones habrá que tener en cuenta que para añadir los arcos de unión entre un módulo Place precedente y un módulo Transition (o InmediateTr) se usarán conexiones unidireccionales, mientras que para los arcos de unión entre un módulo Transition y un módulo Place sucesor a éste, se usarán conexiones bidireccionales y la conexión además tendrá que realizarse en ambos sentidos (de Transition a Place y de Place a Transition). Para más información ver la conectividad de las puertas de los módulos Transition, InmediateTr y Place.

Nota 2: Para realizar una conexión entre un módulo InmTransition y el módulo SolveConflict bastará con hacer una única conexión debido a que las puertas que intervendrán en ambos módulos son bidireccionales (inout). Para más información ver la conectividad de los módulos InmediateTr y SolveConflict.

Nota 3: Las conexiones que usen el parámetro *@display("ls=, 0")* estarán siendo ocultadas por OMNeT++ en la representación de la topología de la simulación. Se ha decidido ocultarlas para reflejar con mayor claridad la dirección real de los arcos del modelo.

La topología resultante del multiprocesador en OMNeT++ será:

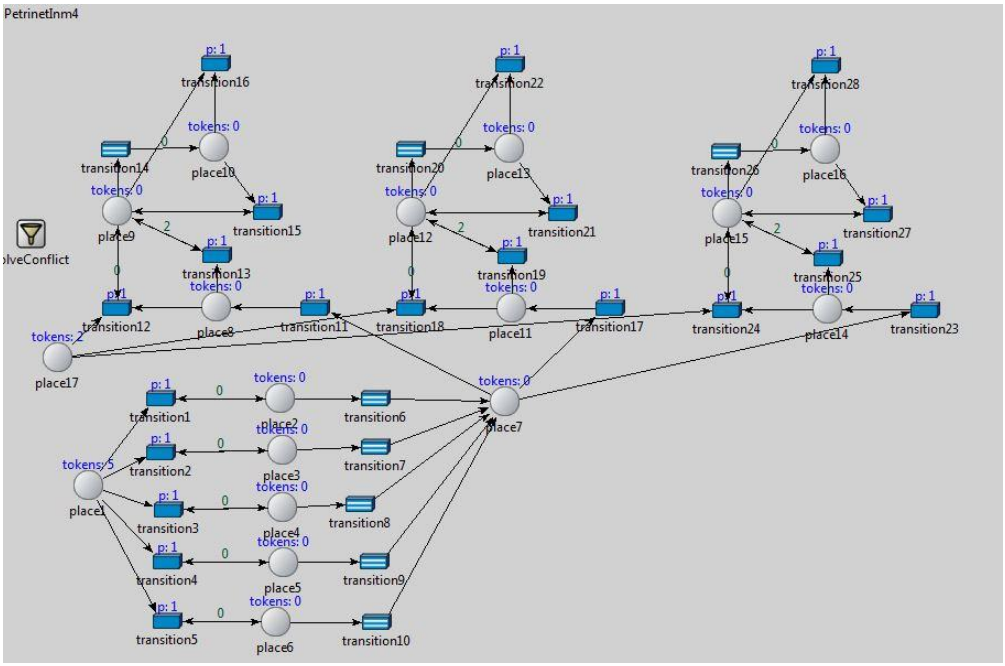

Ilustración 25 - Topología del modelo multiprocesador en OMNeT++

6.7 Simulación del multiprocesador.

A continuación se muestra la configuración en el fichero omnetpp.ini del modelo del multiprocesador. Debido a su extensión, no se ha incluido la descripción completa, pero se han comentado las partes más importantes. Para ver la descripción NED completa del modelo, abrir el fichero PetrinetInm4.ned del proyecto.

```
[Config PetrinetInm4]
network=PetrinetInm4
#tokens iniciales de cada lugar
*.place1.tokens = 5
*.place17.tokens = 2
*.place2.tokens = 0
#los módulos place 3,4,5,6,7,8,9,10,11,12,13,14,15,16
#tendrán el mismo número inicial de tokens que place 2
#para más información, ver código

#prioridad de disparo módulo Inmediate Transition
*.transition1.priority=1
#los módulos transition 2,3,4,5,11,12,13,14,15,16,17,18,19,21,22,23
#24,25,27,28 también tendrán la misma prioridad, para más información
#ver código
```

```
#distribuciones en los módulos Transition
*.transition6.firingTime = exponential(0.5s)
#los módulos transition 7,8,9,10 también usarán la misma media

*.transition14.firingTime = exponential(3s)
#los módulos transition 20,26 también usarán la misma media

#estadística TokenTimeAvg a grabar en módulos Place
#se grabará sólo el escalar tmavg (promedio temporal)
*.place2.TokenTimeAvg.result-recording-modes = +timeavg
*.place3.TokenTimeAvg.result-recording-modes = +timeavg
*.place4.TokenTimeAvg.result-recording-modes = +timeavg
*.place5.TokenTimeAvg.result-recording-modes = +timeavg
*.place6.TokenTimeAvg.result-recording-modes = +timeavg
*.place17.TokenTimeAvg.result-recording-modes = +timeavg

#estadística Productivity a grabar en módulos Transition
#se grabará sólo el escalar last (último resultado registrado)
*.transition11.Productivity.result-recording-modes = +last
*.transition17.Productivity.result-recording-modes = +last
*.transition23.Productivity.result-recording-modes = +last

#estadística NotEmptyAvg a grabar en módulos Place
#se grabará sólo el escalar tmavg (promedia temporal)
*.place9.NotEmptyTimeAvg.result-recording-modes = +timeavg
*.place12.NotEmptyTimeAvg.result-recording-modes = +timeavg
*.place15.NotEmptyTimeAvg.result-recording-modes = +timeavg
```

En esta configuración tenemos lo siguiente:

- El módulo Place1 comenzará con 5 tokens, Place2 con 2 tokens y el resto con 0.
- La prioridad de todas las transiciones inmediatas será 1.
- El tiempo de retardo de las transiciones retardadas que simulan el acceso a la caché (módulos transition 6, 7, 8, 9 y 10) se calculará a partir de una distribución exponencial con media de 0,5 segundos.
- El tiempo de retardo de las transiciones retardadas que simulan el acceso a la caché (módulos transition 14, 20 y 26) se calculará a partir de una distribución exponencial con media de 3 segundos.

En cuanto a la configuración de la grabación de las estadísticas:

- Se grabará el promedio temporal de los tokens (estadística TokenTimeAvg) en los módulos Place 2, 3, 4, 5 y 6. Para ello se usará la opción de grabación timeavg que producirá un escalar. Esto nos permitirá saber la fracción del tiempo en que cada uno de los cinco procesadores estará accediendo a su caché local. También se grabará el promedio temporal de los tokens en el módulo Place 17, que nos permitirá saber el promedio de buses libres en el modelo.

- Se grabará la productividad de los disparos (estadítica Productivity) de las transiciones 11, 17 y 23. Para ello se recogerá el último resultado registrado en la simulación con la opción de grabación last que producirá un escalar. La estadística Productivity nos permitirá saber el número de veces que se ha intentado el acceso a cada módulo de la memoria compartida divido por el tiempo de simulación.

- Se grabará el promedio temporal en que un lugar no permanezca vacío (estadística NotEmptyAvg) de los módulos place 9, 12 y 15. Para ello se usará la opción de grabación timeavg que producirá un escalar. Esto nos permitirá saber para cada módulo de memoria compartida, la fracción del tiempo en que se está accediendo a ésta (si hay tokens en Place 9, 12 o 15 significará que se está accediendo a un módulo de memoria compartida).

6.8 Resultados de la simulación en OMNeT++.

En este apartado se usarán los parámetros de rendimiento estimados en la simulación del modelo de red de Petri para OMNeT++, y las Leyes de Little y de la utilización vistas en el apartado de la teoría de las redes de colas, para poder obtener datos útiles acera del funcionamiento del sistema.

6.8.1 Ley de Little.

En primer lugar nos fijaremos en la Ley de Little y su aplicación en el modelo. Esta ley será aplicable al sistema entero o a los subsistemas del sistema. En este caso nos interesará aplicarla al subsistema que abarca toda la memoria compartida, es decir las cajas de los tres módulos de memoria. A este subsistema llegarán clientes, en este caso procesadores (tokens), y pasado un tiempo se marcharán. La fórmula de la Ley de Little será la siguiente:

$$L = \lambda W$$

Dónde L será el promedio temporal del número de clientes en el sistema, λ la productividad del sistema y W la esperanza del tiempo que los clientes permanecen en el sistema. Nos interesará conocer estos tres parámetros para el subsistema de la memoria compartida.

6.8.1.1 λ o productividad del sistema.

La productividad del sistema será la tasa de clientes (procesadores) que entran al sistema igual a la tasa de clientes que salen. Se han grabado los siguientes datos de las productividades (estadística Productivity) en los módulos transition 11, 17 y 19.

Productividad de transition 11: 0.2090

Productividad de transition 17: 0.2010

Productividad de transition 19: 0.2029

Si sumamos estas tres productividades tendremos como resultado la productividad del subsistema de la memoria compartida.

Productividad del subsistema: 0.6129

6.8.8.2 L o promedio temporal de clientes en el sistema.

L es el promedio temporal del número de clientes (procesadores) en un sistema. Para calcular este parámetro en el subsistema de la memoria compartida, la forma más sencilla será calcular el promedio temporal del número de procesadores fuera del subsistema, es decir, que estén leyendo de sus memorias caché locales.
Promedio temporal de clientes fuera del sistema:

0.0643+ 0.0544 + 0.06321 + 0.0615 + 0.0679 = 0.3113 segundos

Para saber el promedio temporal de clientes dentro se usará la siguiente fórmula, dónde N es el número de clientes.

L(dentro del sistema) = N – L(fuera del sistema)

Para 5 clientes (procesadores) en el sistema, habrá que calcular 5 – L (fuera):

L = 5 - 0.3113 = 4.6887 segundos

6.8.8.3 W o Esperanza del tiempo que los clientes permanecen en el sistema.

W será la esperanza de tiempo que los clientes (en este caso procesadores) permanecerán dentro del subsistema. En este caso se no han obtenido datos de la esperanza, por lo que ésta se obtendrá a partir de la siguiente fórmula:

$$W = L / \lambda$$

Con L nos referiremos al promedio de clientes dentro del subsistema y con λ a la productividad del subsistema.

W= 4.6887 / 0.6129 = 7.65 segundos

6.8.2 Ley de Utilización.

Esta ley a diferencia de la Ley de Little sólo será aplicable a un subsistema compuesto por una única estación y con un único servidor. En este caso se aplicará a cada caja de los módulos de memoria. La fórmula de la Ley de Utilización será la siguiente:

$$U = \lambda S$$

Dónde U será la fracción de tiempo que cada módulo está dando servicio a los clientes, λ la productividad de cada módulo y S la esperanza del tiempo de servicio a los clientes en cada módulo.

6.8.2.1 Λ o productividad de cada módulo.

Se calculará haciendo el promedio temporal del número de los servicios dados a los clientes (procesadores), en cada módulo. Se podrá conocer sabiendo las productividades de las transiciones de acceso a las cajas de los módulos de memoria:

Productividad módulo memoria 1 = Productividad transition 11 = 0.2090

Productividad módulo memoria 2 = Productividad transition 17 =0.2010

Productividad módulo memoria 3 = Productividad transition 23-> 0.2029

6.8.2.2 o esperanza del tiempo de servicio a los clientes.

S es la esperanza del tiempo que los clientes (procesadores) permanecerán dentro de cada subsistema (en este caso, bloque de memoria).

Su valor habrá sido previamente prefijado en el archivo de configuración omnetpp.ini a 3 segundos (exponential (3s)).

6.8.2.3 U o utilización de cada módulo.

U será la fracción del tiempo que cada módulo está dando servicio a los clientes (procesadores).
Se podrá obtener de la estadística NotEmptyAvg de los lugares P9, P12 o P15, o calcularlo aplicando la ley de la utilización U= λ *S

Cálculo U: 0.2090 * 3 = 0.6271; Place9 NotEmptyAvg=0.6467

Cálculo U: 0.2010 * 3=0.6031; Place12 NotEmptyAvg=0.6176

Cálculo U: 0.2029 * 3=0.6086; Place15 NotEmptyAvg=0.6126

Vemos que los resultados son muy parecidos.

7. Referencias y bibliografía.

OMNeT ++:

- Tutorial modelo TicToc: www.omnetpp.org/doc/omnetpp/tictoc-tutorial
- Introducción en 10 minutos: www.omnetpp.org/pmwiki/index.php?n=Main.OmnetppInNutshell
- Manual: www.omnetpp.org/doc/omnetpp/manual/usman.html
- Guía de usuario: www.omnetpp.org/doc/omnetpp/UserGuide.pdf
- API: www.omnetpp.org/doc/omnetpp/api/index.html

Evaluación del rendimiento de sistemas informáticos y teoría de las redes de Petri:

- Introduction to Computer System Performance Evaluation. Krishna Kant. McGraw-Hill, 1992.
- Apuntes de la asignatura Evaluación del Rendimiento de Sistemas Informáticos: http://www.sc.ehu.es/ccwalirx/docs/siee-apunteak.pdf
- The Art of Computer Systems Performance Analysis. Wiley, 1991.
- http://en.wikipedia.org/wiki/Petri_net
- http://en.wikipedia.org/wiki/Discrete_event_simulation

Ubicación del Proyecto:

- http://www.sc.ehu.es/ccwalirx/usr/15IgnacioGarbizu.zip