Universidad Zaragoza

Simulación Distribuida de Redes de Petri

Redes y Sistemas Distribuidos

Miniproytecto

Autor

Marius Sorin Crisan - 721609

ESCUELA DE INGENIERÍA Y ARQUITECTURA 2020-2021

Índice

1. Introducción

En este trabajo se lleva a cabo el diseño, implementación y validación de un simulador distribuido de redes de petri. Se trata de una versión conservativa de simulación para eventos discretos. Para evitar los bloqueos se utiliza el algoritmo de Chandy-Misra-Bryant [1].

2. Desarrollo

En esta sección se va a describir la arquitectura del sistema una vez desplegado en distribuido así como las diferentes estructuras de datos utilizadas para lograr el objetivo. También se detallan las etapas/estados de la simulación que se lleva a cabo en un nodo del sistema distribuido y como este interacciona con el resto de nodos.

Para que el algoritmo de Chandy-Misra-Bryant funcione correctamente es necesario que se cumplan las siguientes premisas:

- No hay creación dinámica de procesos lógicos (PLs)
- Los PLs intercambian eventos con estampillas de tiempos.
- Los PLs envían eventos con estampillas de tiempo crecientes. Es decir, si un PL envía un evento con estampilla de tiempo t, las estampillas de los eventos futuros no podrán tener una estampilla de tiempo menor que t.
- La red garantiza la entrega fiable y preserva el orden de envío.

2.1. Arquitectura

Cada *PL* se encarga de ejecutar una subred, la cual está definida en el fichero **.network.json*. Los *PLs* se envían entre si eventos en el caso de que estos modifiquen transiciones que pertenecen a la subred del *PL* receptor. Para ello, se utiliza una comunicación punto a punto (*tcp*). Con esto se consigue la cuarta premisa, entrega fiable y conservación del orden de envío.

Los eventos generados al disparar una transición están compuestos por el identificador de la transición a disparar, el instante de tiempo del disparo y la constante a aplicar a la *LEF*. Respecto a la versión centralizada, se han añadido dos campos: uno para indicar el remitente del evento en caso de que sea necesario enviarlo a otro *PL* y otro para especificar si se trata de un evento nulo. Las estructuras que almacenan una transición y una subred son las mismas que para el simulador centralizado (Fig. 1: clases *Transition* y *Lefs*).

La clase *Engine* es la estructura principal del simulador. Esta contiene la información del nodo local (*Node*), la subred a simular (*Lefs*), el reloj local, una lista de eventos locales pendientes de ser procesador, los resultados de la simulación (lista con tuplas {}*índice de la transición local disparada, instante del disparo*{}) y un diccionario que asocia el índice de una transición con el nodo al que

pertenece. Este diccionario se utiliza al disparar una transición para saber si un evento dado debe ser enviado a otro nodo o solamente hay que añadirlo a la lista de eventos locales pendientes.

La información del nodo (clase *Node*) está compuesta por el nombre del nodo, el puerto en el que escucha eventos enviados por otros nodos y una diccionario con el resto de nodos o *PLs* que forman el sistema distribuido. Para cada nodo remoto (clase *Partner*) se almacena su IP, puerto tcp, el *lookahead* utilizado para los eventos nulos enviados al nodo remoto, una cola FIFO que almacena todos los eventos recibidos de dicho nodo y dos tiempos: uno para saber la última vez que se envió un evento nulo para no enviar repetidos y el otro para mantener constancia del tiempo del último evento enviado a dicho nodo.

Tanto la información del nodo local como la del resto de nodos, se almacena en el fichero **network.json*, común para todos los nodos del sistema. Además, este fichero también contiene la distribución de transiciones por nodo (Fig. 1: clase *Engine*, atributo *MapTransitionNode*). Esta configuración permite asignar mediante fichero los *PLs* a diferentes máquinas, según los recursos de la infraestructura, sin tener que modificar el código. Además, también se puede cambiar la ubicación de una transición de un nodo a otro. Esto podría ser especialmente útil en una aplicación real ya que ante la caída de uno de los nodos se podría cambiar la ejecución de la correspondiente subred a otro nodo y que el sistema siga funcionando.

Respecto a la depuración del sistema, se ha configurado un sistema de logging para todos los componentes de un mismo nodo. De esta forma, se puede registrar información sobre la ejecución del sistema con cuatro niveles de logging diferentes (*trace, info, warnning y error*).

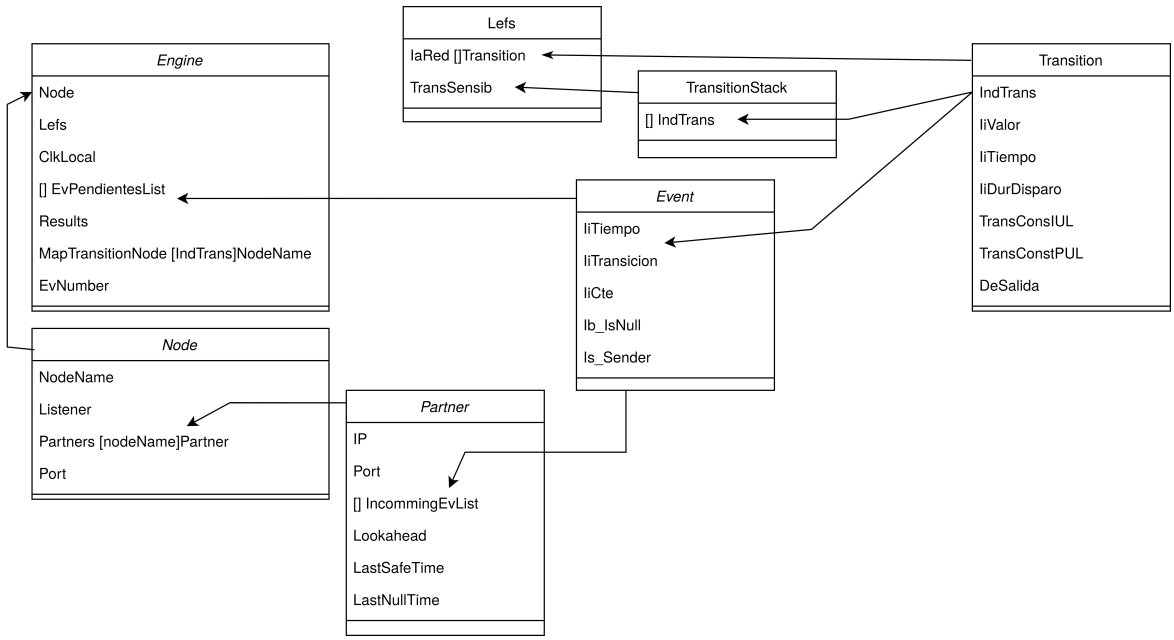

Figura 1: Diagrama de clases del simulador distribuido.

En cuanto a la arquitectura del sistema, cada subred se ejecuta en un *PL*. Cada proceso lógico está compuesto por la rutina principal (motor de simulación) que simula la subred y envía eventos a

nodos remotos. Además, cada *PL* cuenta con una *gorutina* que recibe los eventos de otros nodos y los envía mediante un canal al motor de simulación para que sean tratados.

2.2. Comportamiento

En la figura 2 se muestra el diagrama de actividades del motor de simulación ejecutado en cada nodo. Cada color corresponde a un nivel de abstracción. En amarillo se muestran las funciones invocadas directamente desde el bucle del motor de simulación. En verde se representan las funciones invocadas desde el nivel superior. Por último, en rojo aparecen acciones más específicas como podría ser enviar o recibir un evento.

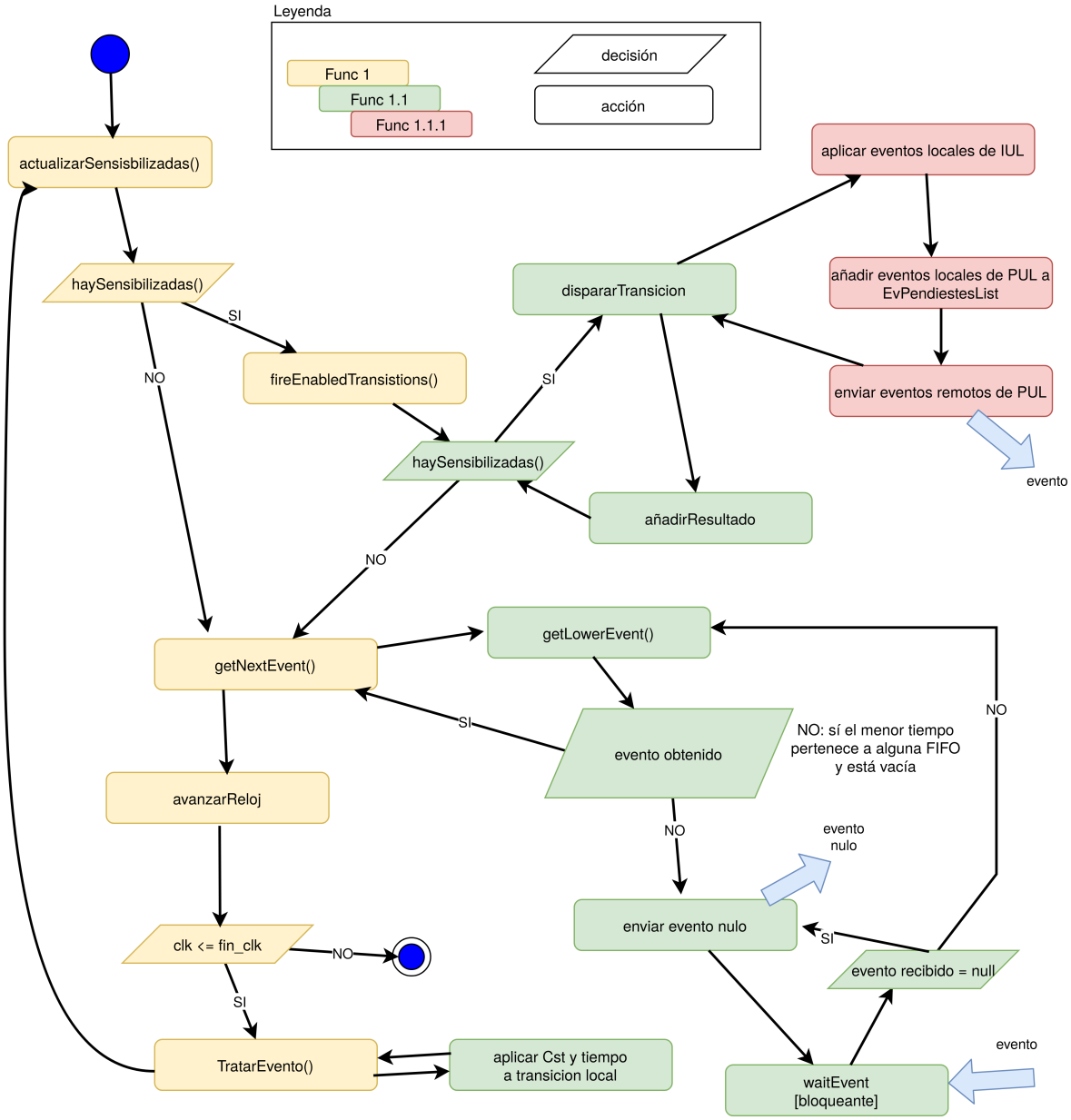

Figura 2: Diagrama de actividades de la simulación distribuida en un nodo. Las flechas en azul corresponden a eventos enviados/recibidos hacia/desde otros nodos.

La simulación comienza actualizando la lista de transiciones que se sensibilizan con el reloj inicial.

En caso de que haya alguna transición sensibilizada, se dispara. Es decir, se procesan los eventos inmediatos que aparecen en *TransConsIUL* (Fig. 1 clase *Transition*). Después, se añaden a lista de eventos pendientes (*EvPendientesList*: Fig. 1 clase *Engine*) todos los eventos locales no inmediatos. Si la transición disparada modifica los *tokens* de alguna transición perteneciente a otra subred, se busca en el diccionario *MapTransitionNode* (Fig. 1 clase *Engine*) el nodo al que pertenece dicha transición y se envía el evento correspondiente a dicho nodo.

La siguiente etapa consiste en obtener el evento con menor estampilla de tiempo entre los eventos locales pendientes y los recibidos de otros nodos. Para ello, cada nodo remoto cuenta con un reloj que indica el tiempo del último evento recibido. Si el menor tiempo corresponde al primer evento local se devuelve el evento local. En caso contrario, se devuelve el evento de la cola FIFO del nodo con menor tiempo. Si dicha cola está vacía, el proceso se bloquea hasta recibir un nuevo evento de la *gorutina* que recibe los eventos enviados por otros nodos. Este proceso se repite hasta obtener un evento para el proceso remoto con menor tiempo. Antes de comenzar la espera de un nuevo evento se envía un mensaje null a todos los procesos para evitar un bloqueo global.

Por último, se actualiza el reloj local y si este es menor que el reloj final de la simulación se trata el evento obtenido previamente y se vuelven a actualizar las transiciones sensibilizadas repitiendo así todo el proceso de nuevo. En caso contrario, se envía un evento que indica que la simulación ha terminado, finaliza la simulación local y se muestran los resultados.

3. Pruebas

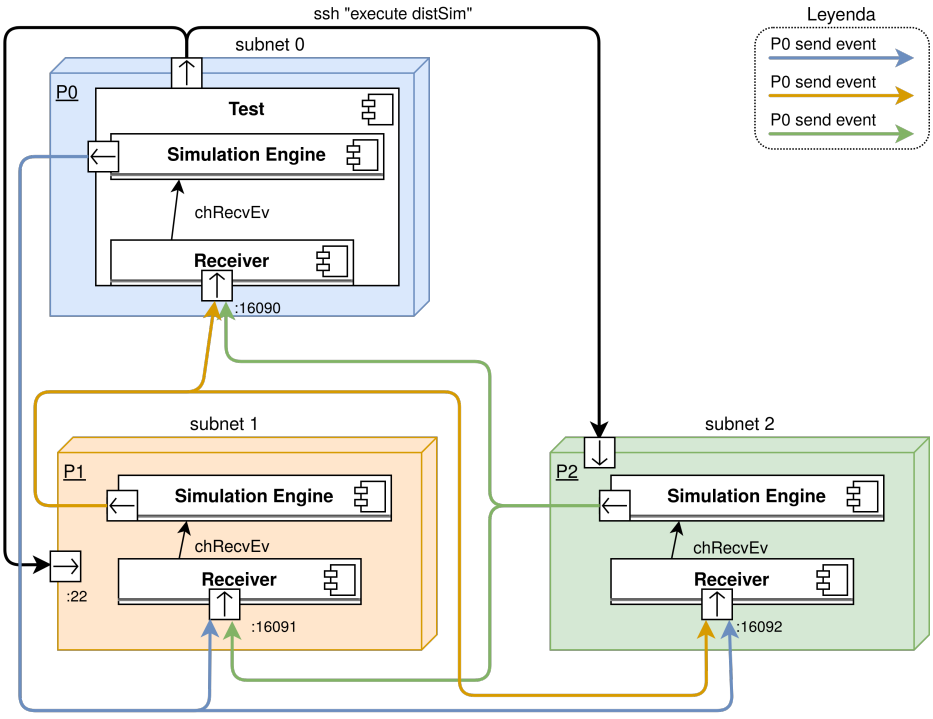

Figura 3: Arquitectura del sistema distribuido al simular tres subredes en tres máquinas.

Para ejecutar las pruebas, desde el fichero de test se lanza la subred principal en la misma máquina que se están ejecutando los tests. A continuación, se establece una conexión *ssh* con cada uno de los otros nodos del simulador y se empieza la ejecución del resto de subredes en función de la distribución especificada en el fichero *network.json*. En la figura 3 se muestra el despliegue del sistema distribuido al simular una red de Petri de la figura 4, pero ejecutando cada rama concurrente en una máquina.

El correcto funcionamiento del simulador distribuido se verifica ejecutando la red de Petri de la figura 4. Con el objetivo de simplificar la depuración la red se divide en dos subredes: el nodo raíz ejecuta las transiciones 0 y 1 mientras que el otro nodo ejecuta las dos transiciones de las ramas concurrentes. La verificación consiste en comparar los ficheros de *log* de ambos nodos con la salida obtenida del simulador centralizado.

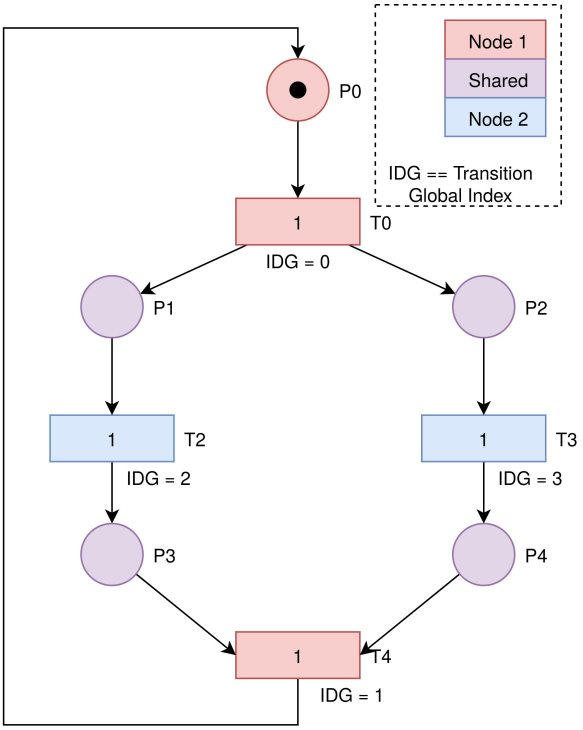

Figura 4: Red de Petri utilizada para la verificación del simulador distribuido.

En la figura 5 se muestra el diagrama de secuencia que se obtiene al ejecutar la red de Petri anterior en dos nodos. En rojo se muestran los eventos enviados entre nodos como resultado del disparo de una transición. En azul aparecen los eventos *NULL* enviados para evitar bloqueos. Las flechas que salen de un nodo y no llegan al otro representan los eventos *NULL* no enviados ya que no aportan información adicional al receptor (estampilla de tiempo menor que la última estampilla enviada a dicho nodo). Por último, las acciones *GNE* representan la obtención del evento de menor tiempo junto con el posible bloqueo hasta la recepción de eventos remotos.

Por último, se ejecuta una batería de pruebas (Tabla 2) con el objetivo de comprobar la influencia en el tiempo de simulación de aspectos como el número de ramas, número de transiciones por ramas o el valor del *lookahead*. Para que los resultados sean consistentes, se simulan 100 ciclos en cada red

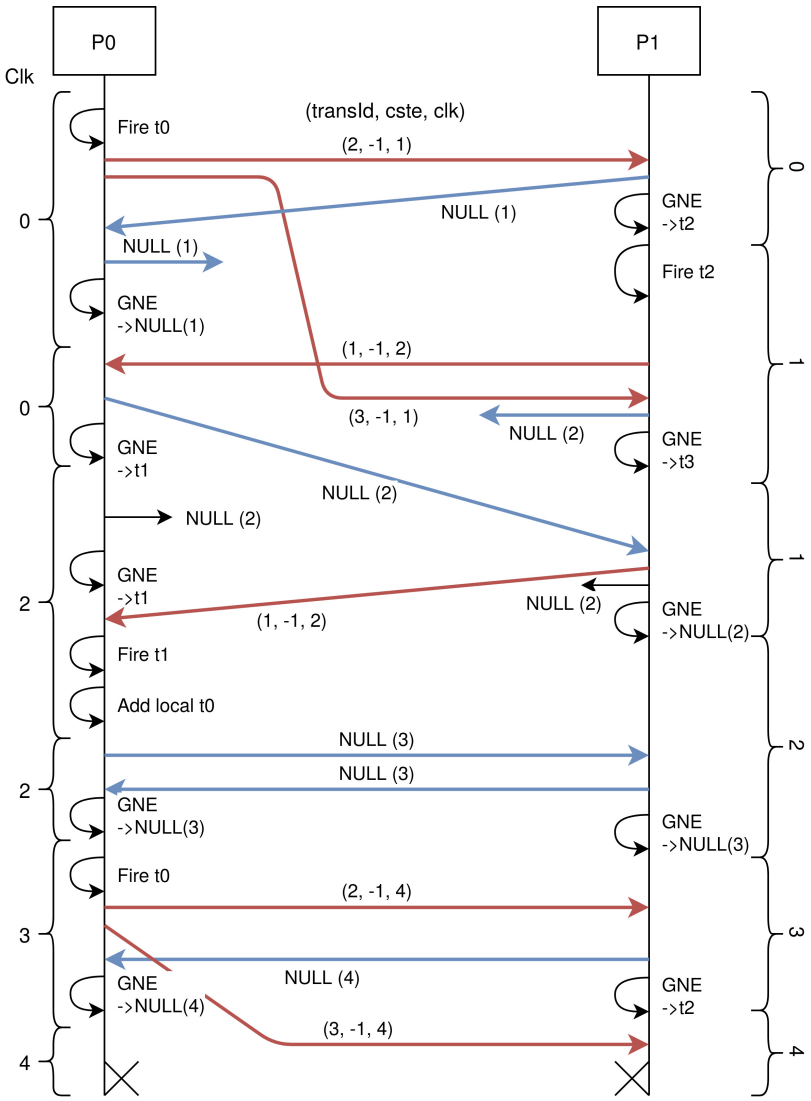

Figura 5: Diagrama de secuencia al ejecutar una red de Petri con dos ramas y 2 lugares por rama (Fig.: 4).

de *Petri*. Aún así, puede que alguno de los resultados estén influidos por los tiempos de espera que se introducen cuando hay varios intentos para conectarse con un nodo remoto.

La única variación entre los primeros dos experimentos son las máquinas que se utilizan para simular la red de *Petri* de la figura 4. El tiempo de simulación del segundo experimento es 3 segundos mayor que el del primero. Además, las máquinas 1 y 2 se procesan menos de la mitad de eventos por segundo que en la máquina 1. Esto indica, que en el primer experimento la máquina 1 sufre menos bloques que las máquinas 1 y 2 en el segundo experimento ya que no tiene que esperar eventos de la otra rama concurrente. Teniendo en cuenta los resultados anteriores, si el número de transiciones por ramas no es muy elevado es mejor procesar dichas ramas en una misma subred ya que se evitan muchos bloqueos esperando eventos innecesarios que llegan de ramas concurrentes.

En el tercer experimento, se simula una red de *Petri* con cinco ramas y seis subredes (subred con *fork* y *join* + 5 subredes concurrentes). El tiempo de disparo de todas las transiciones es 3 y el

Exprimento	P0	P1	P2	P3	P4	P5	Tiempo Simulación
2 ramas - 2 procesos (1 trans./rama)	19.4	14.0	-	-	-	-	4.89
2 ramas - 3 procesos (1 trans./rama)	18.7	6.4	6.5	-	-	-	7.77
5 ramas - 6 procesos (3 trans./rama)	6.5	3.7	3.7	4.0	4.0	4.4	5.35
5 ramas - 6 procesos (3 trans./rama) Rama 5 lenta	10.3	5.0	5.17	5.17	5.18	5.24	5.22
5 ramas - 6 procesos (3 trans./rama) Lookahead = 3	6.9	4.0	3.87	4.11	4.13	3.97	5.17

Tabla 1

Tabla 2: Resultados obtenidos al ejecutar las pruebas en las máquinas del laboratorio 102. Las columnas *Px* indican los eventos procesados por segundo en la máquina *Px*.

lookahead es 1.

El cuarto experimento se distingue del anterior en que todas las transiciones excepto las de la rama 5 tienen un tiempo de disparo de 1. Al reducir el tiempo de disparo, se procesan más eventos por segundo ya que se disparan más veces las transiciones en el mismo periodo de simulación. A pesar de que la rama 5 tiene tiempos de disparo mayores, ejecuta el mismo número de eventos por segundo que las otras. Esto se debe a que el *lookahead* es el mismo que el del resto de ramas. Por lo tanto, envía los mismos eventos que el resto de ramas pero muchos de ellos son eventos nulos.

La red de *Petri* de la última prueba tiene las misma características que la del experimento 3 (tiempo de disparo = 3) pero se ha incrementado el valor del *lookahead* de 1 a 3. El tiempo de simulación debería ser mucho menor en este caso que en el experimento 3 ya que se envían menos eventos entre nodos. Esta incoherencia en los resultados puede que se deba a que los tiempos de espera introducidos al fallar una conexión son muy grandes respecto al tiempo de simulación e influyen demasiado en los resultados.

4. Conclusiones

En este proyecto se ha convertido un simulador centralizado de redes de *Petri* en un simulador de eventos discretos distribuido y conservativo. Esto ha ayudado a afianzar los conocimiento impartidos en la asignatura sobre simulación de eventos discretos y redes de *Petri*.

Además, durante el desarrollo del proyecto se ha comprobado de nuevo la dificultad que implica la depuración de un sistema distribuido y la importancia que tienen las fases de análisis y diseño del sistema y sobretodo la fase de testing del sistema distribuido.

5. Referencias

[1] K. Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, 5:440 – 452, 10 1979.