PROGRAMACIÓN CONCURRENTE

Trabajo Práctico Final

GRUPO:

Error 404

INTEGRANTES:

Chagay Vera, Adriel Ian
Klincovitzky, Sebastian
Severini Montanari Alejo
Vega Cuevas Silvia Jimena
Wortley Agustina Daniela

DOCENTES:

Ventre, Luis Orlando
Micolini, Orlando
Ludemann, Mauricio

Introducción

En este trabajo se realizó una simulación de un sistema con dos procesadores, con tareas asociadas y dos memorias, y mediante el uso de hilos, concurrencia, análisis de invariantes, monitor y semáforos, se completo un sistema funcional.

El trabajo se realizó en lenguaje Java, en el IDE Eclipse, la Red de Petri se analizó en el simulador PIPE.

Simulación con deadlock

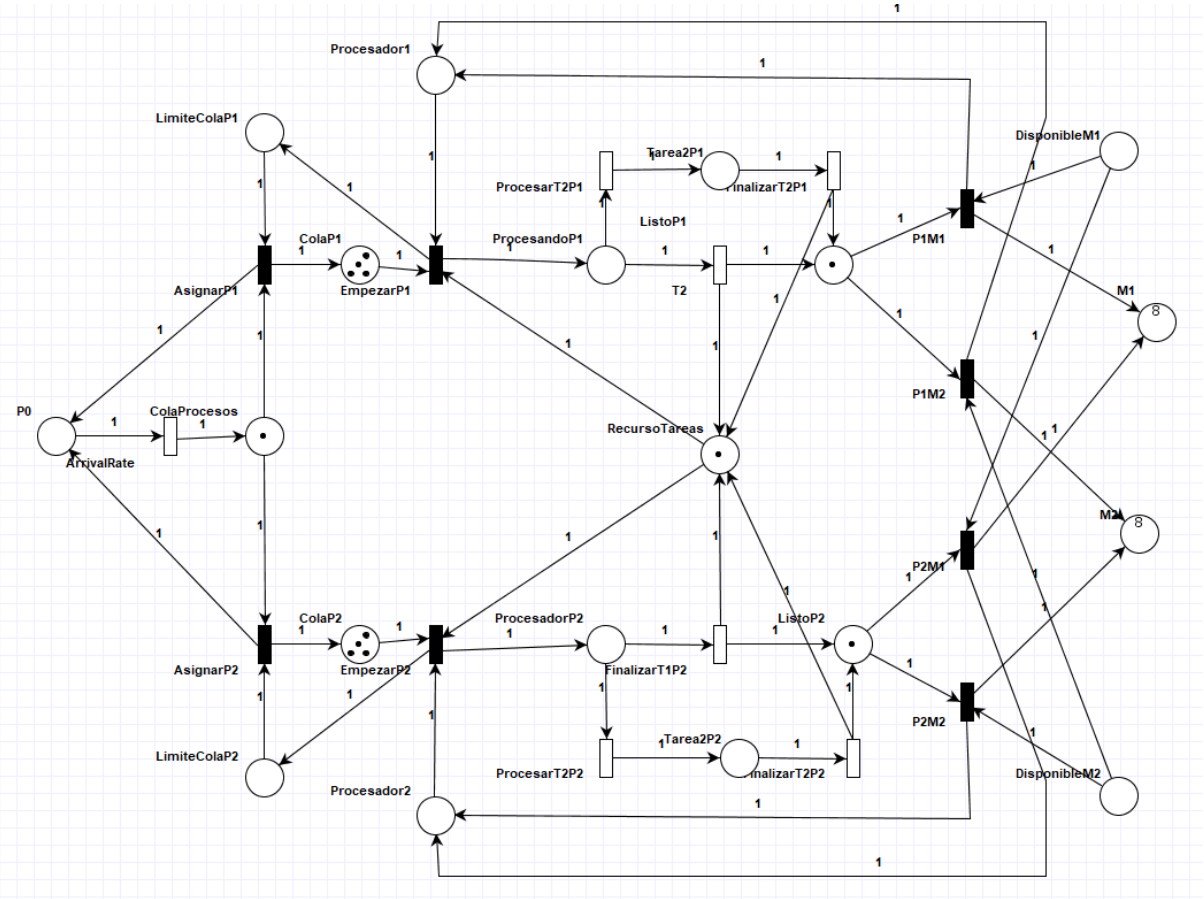


Fig. 1.1

Como se puede ver en la figura 1.1, esta red tiene deadlock, ya que cuando se llena la memoria, el programa no puede seguir avanzando, debido a que DisponibleM1 y DisponibleM2 no sensibilizan más las transiciones P1M1, P1M2, P2M1, P2M2. Esto provoca que Procesador1 y Procesador2, no recuperen el token, y por lo tanto se llenan las colas y el programa no puede avanzar más.

Esto tiene sentido en el modelado de una situación real, ya que si la memoria se llena, el procesador ya no podría trabajar más.

Simulación sin deadlock

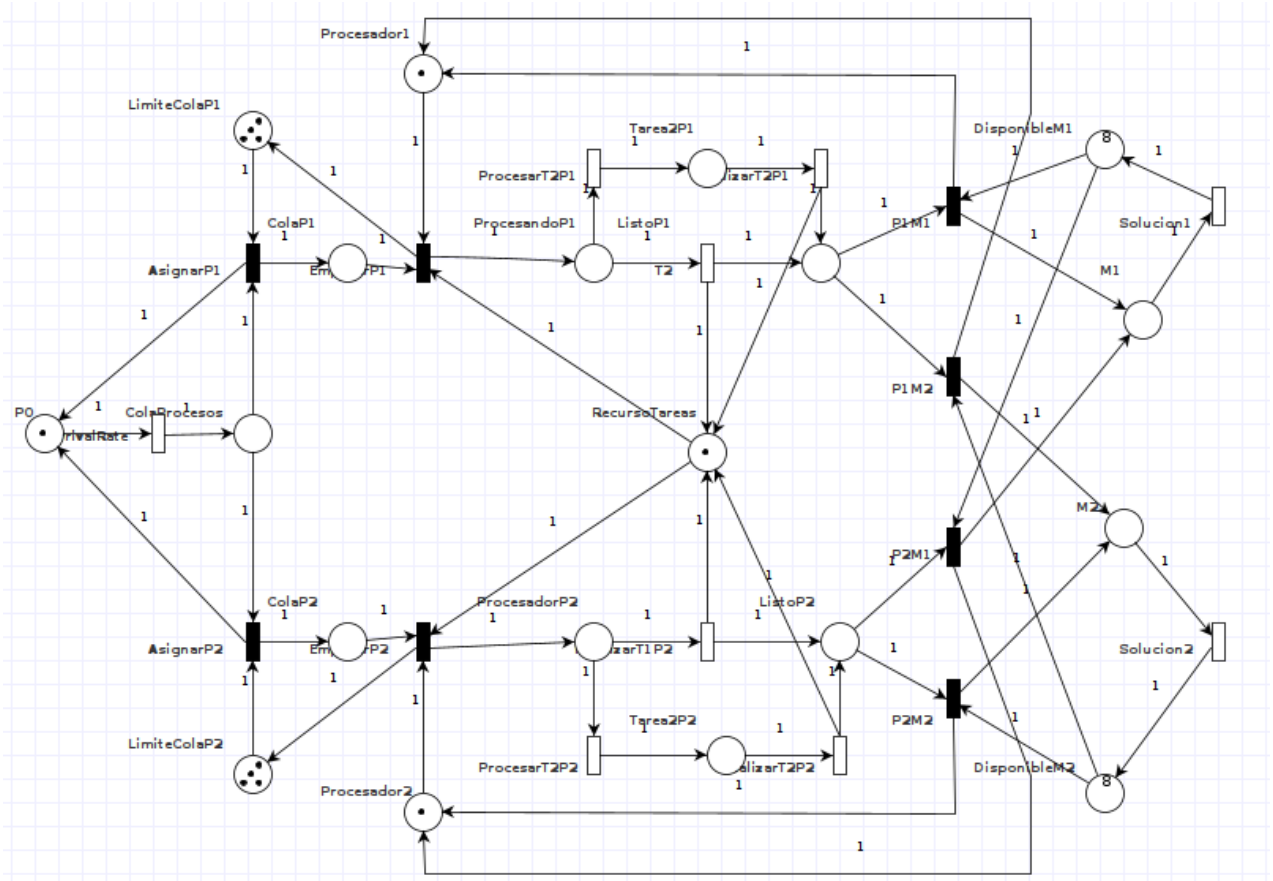


Fig. 1.2

Como se comentó previamente, el problema está en que se llena la memoria, por lo que ya no puede avanzar más. Con esto entendemos que para solucionar el deadlock, lo que se debe hacer es liberar espacios de memoria (y evidentemente volver a aumentar la disponibilidad).

Se proponen dos soluciones igualmente válidas:

- La primera es vaciar las memorias con una transición 1 a 1, con lo que podríamos ir liberando espacios de memoria a medida que se van llenando. Esta se puede ver en la transición Solucion1.
- La segunda es que, mediante una transición 7 a 7, se vacíe la memoria una vez que se llene. Esta se puede ver en la transición Solucion2.

Finalmente se decidió realizar el vaciado con un peso de arco de 7, ya que consideramos ir limpiando la memoria de a uno es un desperdicio de recursos, y que tampoco es lo más conveniente esperar a que se llene la memoria, porque podría traer inconvenientes.

Cabe destacar que las transiciones agregadas son de color blanco, dado que el vaciado de las memorias tardaran almenos 70 milisegundos.

Propiedades (Post-modificación)

- **Bounded:** como la máxima cantidad de tokens que puede haber en la red de petri en todo momento para cada plaza es <=8 podemos decir que la misma, está limitada.
- **Liveness:** para esta red todas las transiciones se encuentran vivas, es decir, para toda marca de la red de petri, existe una secuencia tal que genere el disparo de la transición. Por lo tanto la RdP es viva.
- **Safe:** esta propiedad no se cumple, dado que en algunas plazas, los tokens superan la unidad.
- **Deadlock Free:** esta RdP al cumplir con la propiedad de liveness, no posee deadlocks. Anteriormente si presentaba deadlock, pero este fue salvado anteriormente (y presenta su respectiva explicación).
- **Reversible:** Decimos que es reversible, porque existe al menos una combinación de transiciones con el que la red de petri vuelve a su estado inicial.

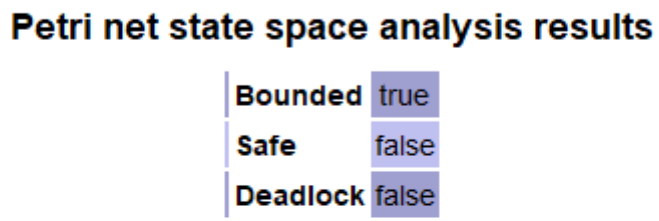


Fig. 1.3

El programa PIPE no es capaz de procesar la red entera, pero al dejar las transiciones que se agregaron para realizar la solución al problema de color negro (es decir, sin tiempo) esto cambia, siendo ahora posible su procesamiento, de esta forma se indica efectivamente que la modificación realizada evita de manera inequívoca el deadlock dado que la condición de tiempo en las transiciones no cambia este hecho. Esta situación es conocida, dado que en un primer momento mientras se estudiaban formas de arreglar el deadlock, esta fue una primera versión de la solución.

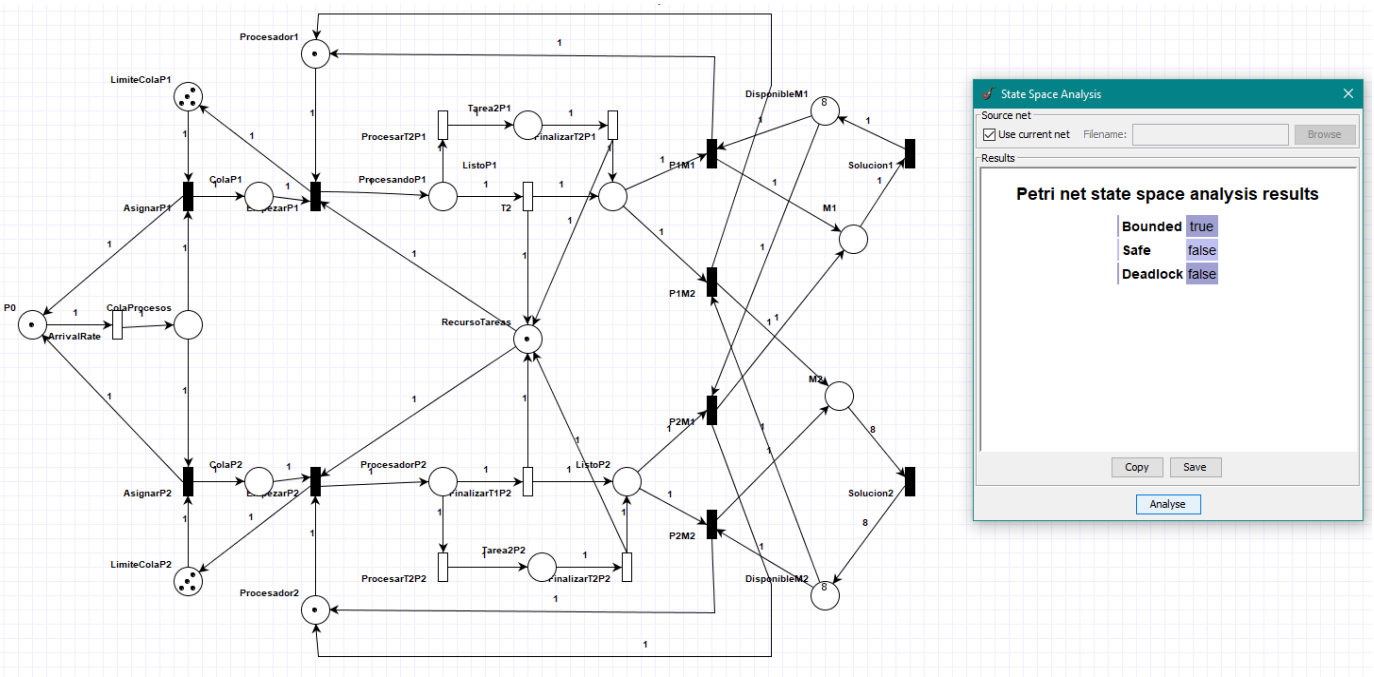


Fig. 1.4

Monitor de Concurrencia

Para el trabajo, se hace uso de un monitor de concurrencia, para que solamente un hilo pueda modificar la Red de Petri.

En nuestro caso decidimos usar un monitor donde los hilos para entrar al monitor deben pasar por un cola de entrada, y en caso de que la transición que desean ejecutar no se encuentra sensibilizada, van a una cola de condición (no tiene cola de cortesía).

La forma que fue resuelto, se utiliza semáforo binario llamado “mutex”, que garantiza la exclusión mutua en la Red de Petri.

Cuando un hilo no se puede disparar, éste libera el mutex y se guarda en una cola de espera.

Cuando un hilo si se puede disparar, luego de modificar el marcado en la Red de Petri y de que se actualice la información sobre las transiciones sensibilizadas, chequea la cola de espera. Lo que hace, es recuperar un vector que indica todas las transiciones que se encuentran en la cola de espera, y se le hace pasar por una AND con el vector de transiciones sensibilizadas, recuperando de esa forma solo aquellas transiciones que permanecen en la cola y que están ahora si, sensibilizadas. Posteriormente la política decide cuál de ellos es liberado y el hilo que se acaba de disparar se retira sin liberar el semáforo mutex, luego el que acaba de salir de la cola empieza su ejecución sin pasar por la entrada, de esta forma aquellos que se encuentran en la cola tienen prioridad sobre la entrada, ya que este no se liberará, hasta que ninguno encuentre uno en la cola.

Para el caso de los hilos temporales, una vez que han adquirido el “mutex”, el programa revisa si se encuentran en la ventana temporal. En caso positivo se disparan normalmente, caso negativo, liberan el semáforo y duermen el tiempo necesario hasta alcanzar la ventana, luego intentan nuevamente.

Para llevar una cuenta del tiempo, existe un vector que contiene TimeStamps. Cada vez que existe un disparo exitoso, y la red se ha actualizado, este vector se actualiza. Lo que hace es verificar las transiciones sensibilizadas, y comparar con el estado anterior, donde si la transición no se encontraba sensibilizada, y ahora lo está, significa que se acaba de sensibilizar, y por lo tanto se actualiza su posición en el vector de tiempo, con el TimeStamp actual.

Luego, para comprobar la ventana, se hace el siguiente cálculo: Tiempo= alfa-tiempoActual-TimeStamp, donde tiempoActual es el TimeStamp actual, y TimeStamp, es aquel guardado en el vector, la resta entre estos dos, nos informa cuánto tiempo ha pasado desde que la transición se ha sensibilizado. Como el resultado debe ser si o si positivo, al restarlo con alfa sabemos que si Tiempo es positivo, todavía no se alcanzó la ventana, por el contrario si es negativo, ya a superado a alfa. Llegado a este punto, hace una última comparación con beta (con tiempoActual-TimeStamp), si beta es mayor, se encuentra efectivamente en la ventana temporal, si beta es menor, ya se ha pasado.

Políticas

Para el trabajo era necesario implementar políticas que resuelvan conflictos relacionados a mantener la carga en los procesadores balanceada y mantener la carga en las memorias balanceadas.

En base a un vector con las tareas que se encuentran en la cola de espera y que están sensibilizados se elige de manera aleatoria una tarea para ser despertada.

Estas políticas llevan el registro de un contador para controlar la cantidad de disparos totales en la RdP.

Colas

Existe la clase cola, donde se puede encontrar un array de semáforos donde cada uno corresponde a una transición. De esta forma cada hilo que no pueda disparar su transición, ya sea porque ésta no está sensibilizada y/o su ventana de tiempo no es compatible, al conseguir el mutex tendrá que adquirir el semáforo correspondiente.

Cuando una transición logra dispararse se realiza una búsqueda en la cola de espera para ver qué hilos se encuentran y las políticas se encargan de liberar uno de estos de forma aleatoria.

Cantidad de Hilos:

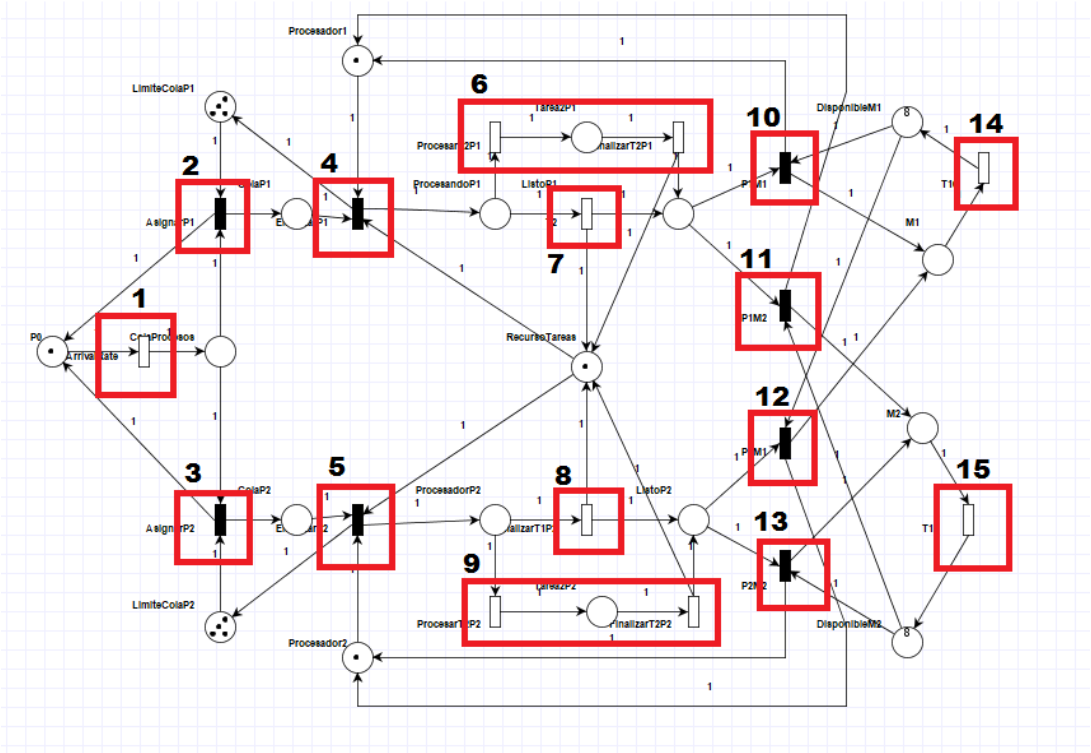


Fig 1.6 : hilos y transiciones otorgadas como tareas a cada uno

Como grupo se optó por utilizar un hilo para cada transición de la RdP, excepto en dos casos (Hilo 6 e Hilo 9) en donde cada hilo ejecuta 2 transiciones distintas. Se eligió de esta forma por distintos motivos:

Primero, para evitar que una transición que estuviera sensibilizada, no pueda ser disparada porque el hilo está intentando disparar otra transición que también está sensibilizada. Por ejemplo, si la transición que ejecuta el hilo 2 y la transición que ejecuta el hilo 4 fueran ejecutadas por un mismo hilo, la primera vez que éste adquiriese el monitor y disparara la primer transición (en caso de que ésta se encontrara sensibilizada) se estaría “bloqueando” la posibilidad de que se vuelva a disparar la primer transición hasta que la segunda transición sea disparada (lo cual sucedería la segunda vez que el hilo consiguiera entrar al monitor). Distinto es el caso del hilo 6 y el hilo 9, para ejemplificar veamos el hilo 6. En caso de que el hilo entrara al monitor y ejecutase la primer transición, dado que ese sector está limitado por la plaza “RecursoTareas”, nunca podrá ingresar otro hilo a ese sector hasta que se ejecute la segunda transición, por lo que aquí no se presenta el problema anterior, y conseguimos resolver la Red de Petri usando menor cantidad de hilos .

La segunda razón, es que de esta forma, se consigue que la automatización de la red resuelva los conflictos, es decir, el hilo 2 y 3 están en conflicto, ya que ambos se sensibilizan al mismo tiempo con el recurso de “Cola Procesos” (Sin tener en cuenta los límites de cola), y ejecutar uno quita la sensibilización al otro, y como las distintas transiciones son ejecutadas por distintos hilos,

el monitor utilizado en el trabajo con la cola de entrada y cola de condición, no existirá un problema en la ejecución provocado por este conflicto.

En el trabajo, para que los hilos 6 y 9 pudieran tener 2 transiciones cada uno, la forma en que resolvió, fue que por constructor, los hilos reciben las transiciones por separado, y cada vez que salen del monitor, y antes de volver a entrar, intercambiara la transición que intentan disparar.

Aclaración en la estructura de los hilos :

Como se ha visto hasta el momento, solo algunas transiciones son temporales y sólo 2 hilos poseen más de una transición que intentan ejecutar, sin embargo, en la aplicación, tratando de hacer el programa de la forma más estándar posible, se decidió, que todos los hilos serían iguales, y se puede observar lo siguiente:

- Todos los hilos tienen un alfa asignado, solo que aquellas transiciones que no debían ser temporales, tienen un alfa = 0 y beta del mayor tamaño posible, por lo que siempre pueden encontrar dentro de la ventana.
- Todos los hilos cambian de transición cada vez que entran al monitor, solo que si solo se supone que disparen una transición, su transición secundaria es la misma que la principal, por lo que el cambio realmente no tiene efecto.

Conclusiones tras 1000 tareas completadas:

Entendiendo las tareas, como las tareas de tipo1 y tipo2 explicadas en el enunciado, se limitó el programa a 1000 disparos de las transiciones correspondientes a dichas tareas.

A continuación se mostraran 2 imágenes con resultados obtenidos en 2 ejecuciones distintas para comentar al respecto:

```
[00:04:39:236]: Tares ejecutadas en Procesador 1: 500
        - Tareas Tipo 1 ejecutadas: 320
        - Tareas Tipo 2 ejecutadas: 180
[00:04:39:236]: Tares ejecutadas en Procesador 2: 500
        - Tareas Tipo 1 ejecutadas: 265
        - Tareas Tipo 2 ejecutadas: 235
[00:04:39:237]: Tares guardadas en Memoria 1: 499
[00:04:39:237]: Tares guardadas en Memoria 2: 500
```

Fig 1.7

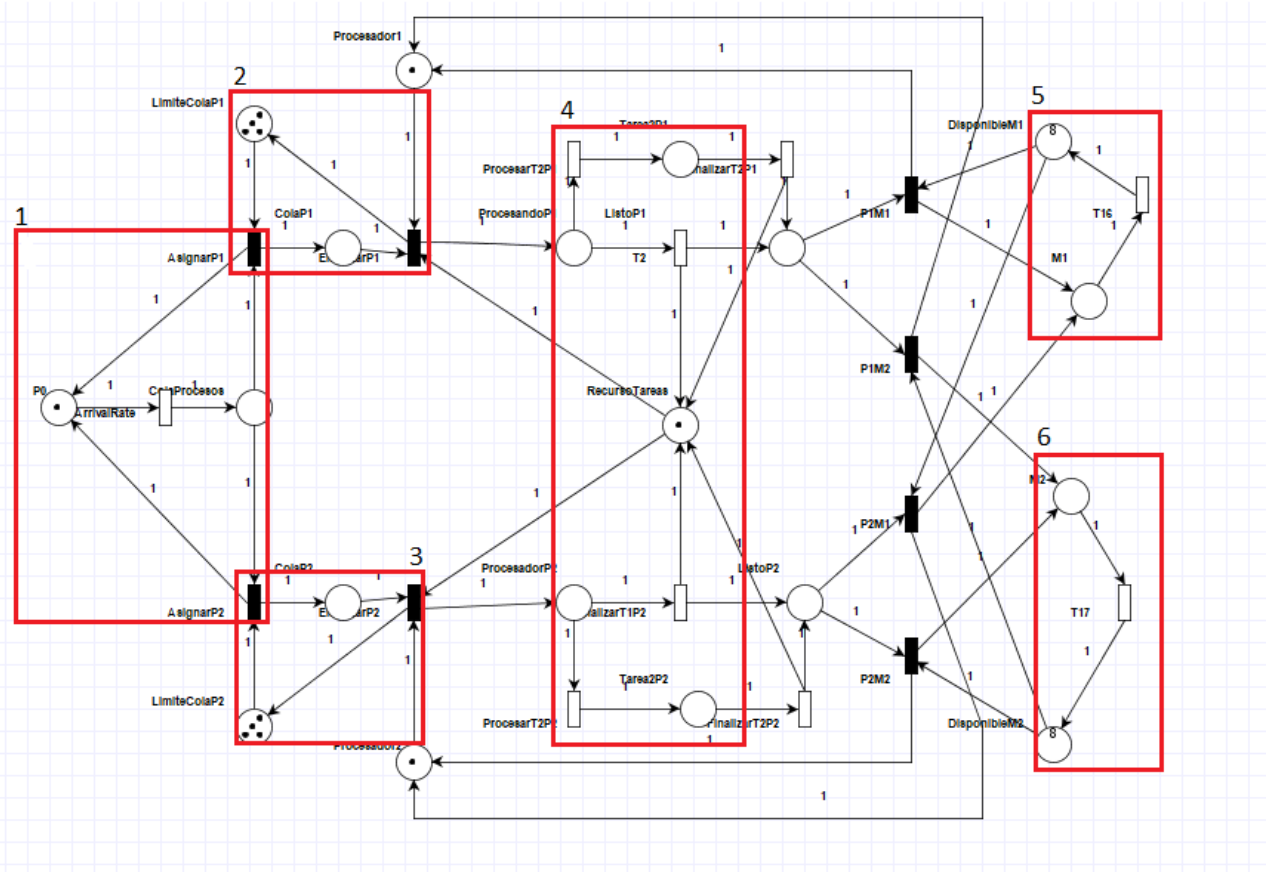
```
[00:17:06:165]: Tares ejecutadas en Procesador 1: 500
        - Tareas Tipo 1 ejecutadas: 254
        - Tareas Tipo 2 ejecutadas: 246
[00:17:06:165]: Tares ejecutadas en Procesador 2: 500
        - Tareas Tipo 1 ejecutadas: 249
        - Tareas Tipo 2 ejecutadas: 251
[00:17:06:165]: Tares guardadas en Memoria 1: 500
[00:17:06:165]: Tares guardadas en Memoria 2: 499
```

Fig 1.8

En primer lugar, como se puede observar, en las distintas ejecuciones cada procesador se ejecutó exactamente 500, quedando en evidencia, que la política implementada cumplió con éxito su tarea de balancear la carga en los procesadores.

En segundo lugar, podemos ver nuevamente qué tan equitativa es la política, al ver los resultados en la memoria, donde cada una guardo aproximadamente la misma cantidad de tareas. La razón para que tras 1000 tareas procesadas, solo se vean 999 tareas guardadas, se debe a que tras terminar con la última tarea, el programa finaliza, por lo que no se le permite al hilo que quiera guardarla continuar.

Invariantes de plaza:



P-Invariants

ColaP1	ColaP2	ColaProcesos	DisponibleM1	DisponibleM2	LimiteColaP1	LimiteColaP2	ListoP1	ListoP2	M1	M2	P0	Procesador1	Procesador2	ProcesadorP2	ProcesandoP1	RecursoTareas	Tarea2P1	Tarea2P2
0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1
0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

The net is covered by positive P-Invariants, therefore it is bounded.

Primer invariante de plazas:
P0 + ColaProcesos = 1

Este invariante de plazas nos indica que cada proceso se puede asignar a un solo procesador a la vez.

Segundo y tercer invariantes de plazas:
ColaP1 + LimiteColaP1 = 4
ColaP2 + LimiteColaP2 = 4

Estos invariantes de plazas nos indican que tanto en la cola para el Procesador 1 como en la cola para el Procesador 2, sólo podrá haber como máximo 4 procesos esperando.

Cuarto invariante de plazas:
ProcesandoP1 + Tarea2P1 + RecursoTareas + ProcesandoP2 + Tarea2P2 = 1

Este invariante de plazas nos indica que solo puede estar funcionando un procesador a la vez.

Quinto y sexto invariantes de plazas:
DisponibleM1 + M1 = 8
DisponibleM2 + M2 = 8

Estos invariantes de plazas nos indican que sólo pueden haber 8 elementos en cada memoria.

Séptima y octava invariantes de plazas:
Procesador1+ ListoP1 + ProcesandoP1 + Tarea2P1 = 1
Procesador2+ ListoP2 + ProcesandoP2 + Tarea2P2 = 1

Estos invariantes de plazas nos indican que el procesador no podrá tomar una nueva tarea, hasta que finalice por completo de procesar la tarea que acaba de realizar, acomodandola en un espacio de la memoria.
No se encuentra especificado en el gráfico, porque en lugar de ayudar a la comprensión la perjudicaría.

Los invariantes se encuentran comprobados en el archivo que genera el log. Al comienzo se tiene:

```
[09:13:25:035]:  P1(0)  +  P6(4) = 4
[09:13:25:036]:  P2(1)  +  P11(0) = 1
[09:13:25:036]:  P4(8)  +  P10(0) = 8
[09:13:25:037]:  P0(0)  +  P5(4) = 4
[09:13:25:037]:  P7(0)  +  P12(1) +  P15(0) +  P17(0) = 1
[09:13:25:037]:  P8(0)  +  P13(1) +  P14(0) +  P18(0) = 1
[09:13:25:038]:  P3(8)  +  P9(0) = 8
[09:13:25:038]:  P14(0)  +  P15(0) +  P16(1) +  P17(0) +  P18(0) = 1
```

Al final de la ejecución se observa que se mantuvo sin cambios:

```
[09:15:12:577]:  P1(4)  +  P6(0) = 4
[09:15:12:577]:  P2(1)  +  P11(0) = 1
[09:15:12:577]:  P4(5)  +  P10(3) = 8
[09:15:12:577]:  P0(4)  +  P5(0) = 4
[09:15:12:577]:  P7(0)  +  P12(1) +  P15(0) +  P17(0) = 1
[09:15:12:577]:  P8(1)  +  P13(0) +  P14(0) +  P18(0) = 1
[09:15:12:577]:  P3(6)  +  P9(2) = 8
[09:15:12:577]:  P14(0)  +  P15(0) +  P16(1) +  P17(0) +  P18(0) = 1
```

Invariantes de transición:

Para encontrar los invariantes de transición se utilizó el archivo “disparos.txt” en donde se encuentran todos los disparos realizados en las 1000 ejecuciones de la red de petri.

Petri net invariant analysis results

T-Invariants

T0	T1	T10	T11	T12	T13	T14	T15	T16	T2	T3	T4	T5	T6	T7	T8	T9
7	7	7	0	0	7	0	0	1	7	0	0	0	0	0	0	7
7	7	7	0	0	0	7	1	0	7	0	0	0	0	0	0	7
7	0	0	7	0	0	0	0	1	0	7	7	0	7	0	0	0
7	0	0	7	0	0	0	0	1	0	7	7	0	0	7	7	0
7	7	0	0	0	7	0	0	1	7	0	0	7	0	0	0	0
7	7	0	0	0	0	7	1	0	7	0	0	7	0	0	0	0
7	0	0	0	7	0	0	1	0	0	7	7	0	7	0	0	0
7	0	0	0	7	0	0	1	0	0	7	7	0	0	7	7	0

The net is covered by positive T-Invariants, therefore it might be bounded and live.

Aclaración: la falta de nombres de los invariantes en la imagen anterior, se debe a que PIPE presenta un bug a la hora de procesar los invariantes con sus nombres completos.

Para Procesar el resultado de los disparos, se hace uso de un script realizado en Python, el cual levanta los invariantes de transición de un archivo .txt y basándose en éstos recorre el log de disparos para ir sacando aquellos que pertenezcan a las secuencias de los invariantes, al finalizar, el script muestra en pantalla la cantidad de invariantes completos que se encontraron y los disparos del log que sobraron o que no alcanzaron para completar una secuencia completa de los invariantes de transición.

Como se puede ver en la matriz, los resultados ofrecidos por Pipe solo procesa caminos “puros”, es decir, si la red pudiera tomar dos caminos con los que llegar indistintamente al mismo resultado, primero procesa yendo exclusivamente por un camino, y posteriormente por el otro, esto provoca que al revisar el resultado de “disparos.txt”, sobren algunas transiciones, sumado a que las últimas iteraciones del programa, no terminan de cumplir con un invariante.

Conclusión

Durante el desarrollo de este trabajo práctico fue posible ver de manera práctica y en mayor detalle el funcionamiento de una Red de Petri. Al comienzo fue planteada una red de petri con deadlock utilizando transiciones temporales e instantáneas, su ecuación de estado y su vector de transiciones sensibilizadas. Para ello se desarrolló una herramienta de software programada en Java donde se puede ver el avance de la Red de Petri controlado mediante políticas impuestas por los desarrolladores.

Es posible hacer esta implementación utilizando cierto nivel de concurrencia, donde distintos hilos intentan disparar transiciones en la RdP. Estos hilos pudieron ser gestionados por el uso de un Monitor; que es donde se controla que hilo accede o no a los recursos, en este caso la RdP, y por consecuencia efectuar el disparo de determinadas transiciones. La formalidad matemática propia de las Redes de Petri proporcionan herramientas de análisis para analizar los posibles estados a los que el sistema modelado pudiera alcanzar.

De esta forma se puede estudiar y comprender su comportamiento. Existen otras herramientas computacionales aparte de las que se utilizaron durante este trabajo práctico que permiten analizar este tipo de sistemas, las cuales están basadas en análisis estadísticos y ofrecen soluciones con ciertos grados de incertidumbre.

Finalmente podemos afirmar que las RdP son efectivamente una herramienta muy útil para modelar sistemas de una forma universal que es la matemática y su importante labor en la concurrencia informática dentro del vasto campo de la ingeniería.