Programación Concurrente

Trabajo Final Integrador

Facultad de Ciencias Exactas, Físicas y Naturales
Universidad Nacional de Córdoba

Preparado por:

- *Badariotti, Juan*
- *Montero, Felipe*

Profesores:

- *Ventre, Luis Orlando*
- *Ludemann, Mauricio*

Índice

Introducción

Los avances tecnológicos están correlacionados con un incremento en la complejidad de los sistemas informáticos e industriales en general. Los sistemas embebidos, redes de sistemas embebidos (IoT), sistemas distribuidos, sistemas de multiprocesamiento simétrico y servidores que prestan servicios remotos, son algunos ejemplos de sistemas que presentan un rápido desarrollo y que se han vuelto cada vez más complejos. Esto conlleva a que estas arquitecturas o sistemas tengan requerimientos y especificaciones de calidad cada vez más complejos.

La verificación de un sistema generalmente se realiza a través del testing [1], por ejemplo pruebas unitarias y de integración. El problema radica en que es imposible garantizar la ausencia de errores a través del testeo de un sistema. Es posible encontrar y corregir errores, pero es imposible asegurar la ausencia o no aparición de futuros errores. Una solución a este problema es la verificación de un sistema a través de la comparación contra un modelo [1], [2]. Para esto, el sistema es desarrollado como un modelo preciso, el cual incluye todos los mecanismos críticos, funcionalidades y subsistemas del sistema en cuestión. De esta forma, el modelo nos permite estudiar el sistema en una mayor cantidad de casos o situaciones posibles, que de otra forma sería imposible llevar a cabo en estudios de campo o realidad. Sin embargo, esta aproximación también tiene sus limitaciones o problemas. Un ejemplo de problema con la verificación mediante modelado es "*state-explosion*" [1], en donde el modelo se vuelve tan complejo y tiene tantos estados que se vuelve imposible verificarlos todos. Otro problema puede ser el mismo sistema de modelado, en donde un modelo incorrecto, inexacto o impreciso genera resultados incorrectos. Por esta razón es de suma importancia elegir un sistema de modelado adecuado, que cumpla con estándares y que permita modelar sistemas complejos de forma general y precisa.

Las redes de petri (PN) son un sistema de modelado ampliamente utilizado para modelar sistemas distribuidos [1]. Una red de petri consiste en una determinada cantidad de plazas, transiciones y arcos. Cada plaza puede contener tokens o marcas. Los arcos conectan plazas con transiciones y transiciones con plazas. Las transiciones mueven los tokens de una plaza a otra, a través de los arcos. Un ejemplo simple de red de petri se observa en la Figura N° 1.

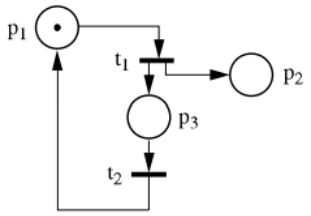

Figura N° 1. Ejemplo de red de petri. p1, p2 y p3 son plazas; t1 y t2, transiciones. Las flechas corresponden a arcos. Fuente [2].

Entre las ventajas de las redes de petri se como sistema de modelado se pueden mencionar [2]:

- Facilidad y semántica apropiada para modelar sistemas concurrentes y/o distribuidos.
- Soporte gráfico simple para la representación de sistemas.
- Facilidad de expresar conceptos básicos de comunicación, como la espera (wait) y sincronización.
- Capacidad de modelar semántica temporal y procesos estocásticos.
- Independencia de cualquier lenguaje de programación o implementación física.

En el presente trabajo se ejemplifica el uso de las redes de petri para el modelado y verificación de un sistema concurrente. Como ejemplo se utiliza un sistema de procesamiento de imágenes capaz de realizar tareas de carga, filtrado y recorte de forma paralela. En base a las especificaciones del sistema se modela el mismo como una red de petri. Las propiedades del sistema son estudiadas a partir de las propiedades de la red. Finalmente, en base a los resultados obtenidos del análisis de la red, se modela el sistema utilizando el lenguaje de programación Java.

El modelo en Java hace uso de un monitor[3] para gestionar el acceso a los recursos de la red (el acceso a las secciones críticas). Los hilos solicitan el acceso a recursos al monitor y este decide, en base la política[3,4] implementada, si otorgar el acceso al recurso solicitado. El monitor solo decide si dar acceso a los recursos y permitir el trabajo de los hilos, pero estos realizan su tarea fuera del monitor. Esto permite que los hilos trabajen de forma concurrente.

El desarrollo del modelo en Java se dividió en dos iteraciones. En primer lugar se desarrolló el modelo en base a una red de petri no temporalizada, es decir sin transiciones temporales. En este caso la duración de las tareas se simuló dentro de cada hilo/tarea, de forma ajena a la red, al monitor y la política. Finalizado este modelo se procedió con la segunda iteración, que consistió en modelar el sistema en base a una red de petri temporizada. Los tiempos se incluyeron en las transiciones. Se implementó una semántica de tiempo débil[4], en la cual la red impone limitaciones de tiempo sobre los disparos, como un intervalo de tiempo asociado a las transiciones que se modelan como temporales.

Propiedades de la Red

La red de petri que modela el sistema de procesamiento de imágenes se muestra en la Figura N° 2. Las plazas P1, P3, P5, P7, P9, P11, P15 representan recursos compartidos en el sistema. La plaza P0 es una plaza idle que corresponde al buffer de entrada de imágenes al sistema. En las plazas P2, P4 se realiza la carga de imágenes en el contenedor para procesamiento. La plaza P6 representa el contenedor de imágenes a procesar. Las plazas P8, P10, P11, P13 representan los estados en los cuales se realiza un ajuste de la calidad de las imágenes. Este ajuste debe hacerse en dos etapas secuenciales. Con la plaza P14 se modela el contenedor de imágenes mejoradas en calidad, listas para ser recortadas a su

tamaño definitivo. En las plazas P16, P17 se realiza el recorte. En la plaza P18 se depositan las imágenes en estado final. La plaza P19 representa el proceso por el cual las imágenes son exportadas fuera del sistema.

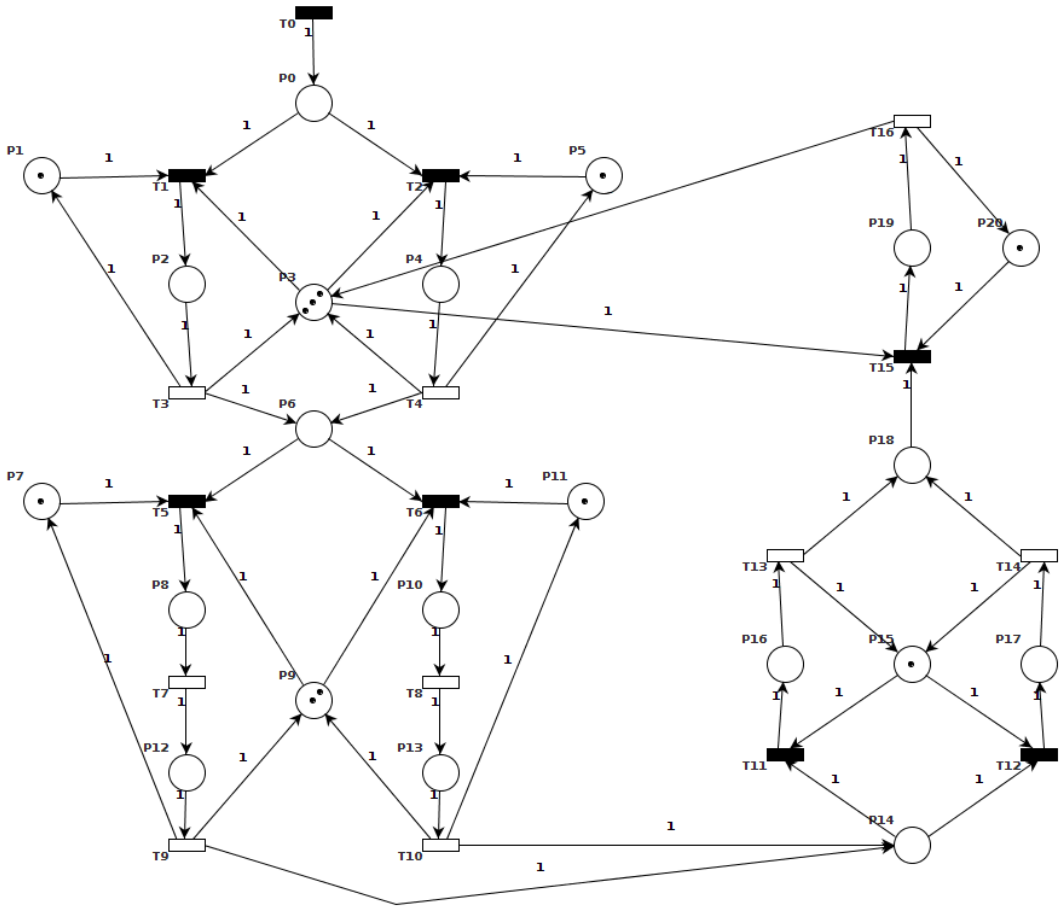

Figura N° 2. Red de petri que modela el sistema de procesamiento de imágenes.

El marcado de la plaza P0 depende del disparo de la transición T0, la cual representa la entrada de imágenes al sistema, es decir un evento externo. Por lo tanto se trata de una red no autónoma. Se observa que todas las transiciones tienen más de un lugar de entrada, por lo que la red no representa una máquina de estados. Las plaza P0 tiene 2 transiciones de salida, la plaza P18 tiene 2 transiciones de entrada y las plazas P6, P9 y P15 tiene 2 transiciones de entrada y 2 de salida, por lo que la red no es un grafo de marcado. Las plazas P0, P3, P6, P9, P14 y P18 pueden presentar un marcado mayor a 1, por lo que la red no es segura.

Se observan conflictos estructurales en las plazas P0, P6 y P15, dado que, como se mencionó anteriormente, tienen 2 transiciones de salida. El conflicto en la plaza P15 es efectivo, dado que esta plaza no puede tener más de un token, en cambio, el conflicto en las plazas P0, P6 y P14 puede no ser efectivo si ingresa más de una imágen al sistema, es decir si T0 se dispara más de una vez.

Las plazas P1, P5, P7, P11 y P20 son limitadores. Limitan a las plazas P2, P4, P8, P10, P12, P13 y P19 a un máximo de 1 token. Las plazas P3, P9 y P15 son recursos compartidos. La plaza P15, además, modela un caso de exclusión mutua. Por lo tanto, las secuencias de disparo T_1T_3, T_2T_4, $T_5T_7T_9$, $T_6T_8T_{10}$, $T_{11}T_{13}$, $T_{12}T_{14}$ y $T_{15}T_{16}$ representan secciones críticas.

Invariantes y Álgebra de la Red

Para analizar los invariantes de plaza y de transición de la red se utilizó el software TIme petri Net Analyzer (TINA) [5]. El mismo permite modelar una red de petri, realizar análisis de invariantes, simular la evolución de la red y obtener el árbol de alcanzabilidad a partir de un marcado inicial, entre otras funcionalidades.

Para poder realizar el análisis de invariantes se modificó la red, con fin de convertirla en una red acotada y viva. Esto es debido a que en el modelo original (Figura N° 2) la transición 0 genera los tokens que ingresan a la plaza 0 (el ingreso de imágenes al sistema) de forma indefinida, por lo que la plaza P0 puede, en principio, tener infinitos tokens. De la misma forma, la transición 16 elimina tokens (las imágenes procesadas) de la red, por lo que la misma podría quedar sin elementos (tokens, imágenes) para procesar, es decir, bloqueada. La red modificada se muestra a continuación (Figura N° 3).

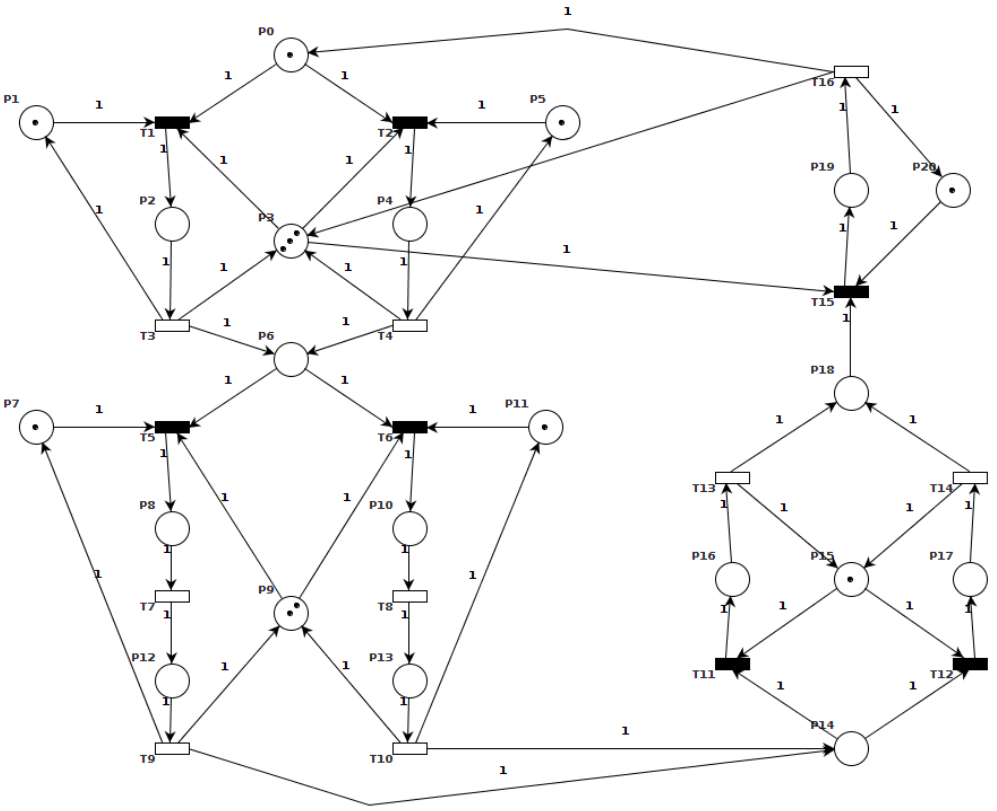

Figura N° 3. Red de petri modificada para cálculo de invariantes.

En la Figura N° 3 se observa un token inicial en la plaza 0. El mismo es auxiliar y cumple la función de mantener la red desbloqueada al momento de realizar el análisis de invariantes.

Se determinaron los siguientes invariantes de plaza, a partir de la red de petri modificada:

IP1: P1 + P2 = 1 (1)
IP2: P4 + P5 = 1 (2)
IP3: P2 + P3 + P4 + P19 = 3 (3)
IP4: P7 + P8 + P12 = 1 (4)
IP5: P10 + P11 + P13 = 1 (5)
IP6: P8 + P9 + P10 + P12 + P13 = 2 (6)
IP7: P15 + P16 + P17 = 1 (7)
IP8: P19 + P20 = 1 (8)

Existe un noveno invariante de plaza, para la red modificada (Figura N° 3), el cual depende del marcado inicial en P0, el cual es:

IP9: P0 + P2 + P4 + P6 + P8 + P10 + P12 + P13 + P14 + P16 + P17 + P18 + P19 = n (9)

Donde n corresponde al marcado inicial en P0. Para el marcado inicial de la Figura N° 3, n es igual a 1.

Los invariantes de transición determinados a partir de la red son:

IT1 = {T1, T3, T5, T7, T9, T11, T13, T15, T16} (10)
IT2 = {T1, T3, T5, T7, T9, T12, T14, T15, T16} (11)
IT3 = {T1, T3, T6, T8, T10, T11, T13, T15, T16} (12)
IT4 = {T1, T3, T6, T8, T10, T12, T14, T15, T16} (13)
IT5 = {T2, T4, T5, T7, T9, T11, T13, T15, T16} (14)
IT6 = {T2, T4, T5, T7, T9, T12, T14, T15, T16} (15)
IT7 = {T2, T4, T6, T8, T10, T11, T13, T15, T16} (16)
IT8 = {T2, T4, T6, T8, T10, T12, T14, T15, T16} (17)

Cada invariante de transición representa el recorrido (o procesamiento) por el que tiene que pasar una imágen para ser procesada completamente y ser exportada del sistema.

Para modelar la evolución de la red y poder desarrollar el modelo del sistema en el lenguaje de programación Java, se obtuvo la matriz de incidencia de la red y a partir de esta, la ecuación de estado[4]. La matriz de incidencia se determinó utilizando el software Petrinator [6]. La misma se muestra en la Tabla N° 1.

Tabla N° 1. Matriz de incidencia de la red de petri (Figura N° 1).

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
P0	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
P3	0	-1	-1	1	1	0	0	0	0	0	0	0	0	0	0	-1	1
P4	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	1	1	-1	-1	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0
P8	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0	0
P9	0	0	0	0	0	-1	-1	0	0	1	1	0	0	0	0	0	0
P10	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	1	1	-1	-1	0	0	0	0
P15	0	0	0	0	0	0	0	0	0	0	0	-1	-1	1	1	0	0
P16	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0	0
P17	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0
P18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	-1	0
P19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-1
P20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1

A partir de la matriz de incidencia se puede obtener el estado (marcado) de la red para cualquier disparo de transición con la ecuación de estado[3,4]:

$$M_{i+1} = M_i + I \times \sigma \qquad (18)$$

Donde M_i corresponde a un marcado de la red (estado), I a la matriz de incidencia (Tabla N° 1), M_{i+1} al marcado luego de realizar el disparo y σ al vector de disparo. Es un vector columna de la forma:

$$\sigma = [t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}]^T \qquad (19)$$

Donde cada t_i corresponde a la transición disparada en el estado M_i para llegar al estado M_{i+1}.

Cantidad Máxima de Hilos Simultáneos

A partir del algoritmo desarrollado por Micolini y Ventre [3] se calculó la cantidad máxima de hilos activos simultáneos para el sistema modelado por la red. Para el análisis se utilizó la red modificada (Figura N° 3), debido a que el algoritmo hace uso de los invariantes de plaza y de transición, los cuales fueron obtenidos a partir de esta red. En primer lugar se obtuvieron los invariantes de transición (ecuaciones 10-17). Luego, se determinaron los

conjuntos de plazas asociadas a cada invariante de transición, es decir, las plazas involucradas en la secuencia de disparo de cada invariante. Los conjuntos determinados son:

PI1: {P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20} (20)
PI2: {P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20} (21)
PI3: {P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20} (22)
PI4: {P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20} (23)
PI5: {P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20} (24)
PI6: {P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20} (25)
PI7: {P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20} (26)
PI8: {P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20} (27)

Luego, en base a la descripción de las plazas presentada anteriormente (sección Propiedades de la Red), se determinaron los siguientes conjuntos de plazas que no son plazas de acción (Tabla N° 2):

Tabla N° 2. Conjunto de plazas idle, recursos y restricción.

Conjunto	Plazas
Plazas idle	P0, P6, P14, P18
Plazas de recursos	P1, P3, P5, P7, P9, P11, P20
Plazas de restricción	P15

Al sustraer los conjuntos de plazas idle, de recursos y de restricción a las plazas de cada invariante de transición (ecuaciones 20-27) se obtuvieron los conjuntos de plazas de acción de cada invariante:

PA1: {P2, P8, P12, P16, P19} (28)
PA2: {P2, P8, P12, P17, P19} (29)
PA3: {P2, P10, P13, P16, P19} (30)
PA4: {P2, P10, P13, P17, P19} (31)
PA5: {P4, P8, P12, P16, P19} (32)
PA6: {P4, P8, P12, P17, P19} (33)
PA7: {P4, P10, P13, P16, P19} (34)
PA8: {P4, P10, P13, P17, P19} (35)

Finalmente el conjunto de plazas de acción son todas plazas que pertenecen al conjunto de de plazas de acción de todos los invariantes de transición:

PA = {P2, P4, P8, P10, P12, P13, P16, P17, P19} (36)

La cantidad máxima de hilos activos se determina tomando el máximo de la suma de tokens de las plazas de acción, para todos los marcados posibles. Debido a que los marcados

posibles dependen del marcado inicial y dado que el número de tokens en la plaza P0 es variable (dado que representa las imágenes que ingresan al sistema), se determinaron los marcados posibles (es decir, se obtuvieron los árboles de alcanzabilidad) para diferentes cantidades de tokens iniciales en la plaza P0. Se evaluaron marcados iniciales donde la plaza P0 tiene desde 1 hasta 15 tokens. Los resultados se muestran en la Tabla N° 3.

Tabla N° 3. Máximo de suma de tokens de plazas de acción para diferentes marcados iniciales en P0.

m_0(P0)	Max[Suma(PA)]
1	1
2	2
3	3
4	4
5	5
6	6
7	6
8	6
9	6
10	6
11	6
12	6
13	6
14	6
15	6

Se puede observar que el máximo de la suma de marcas de las plazas de acción aumenta a medida que aumenta el marcado inicial en P0, hasta un máximo de 6. Esto significa que si el invariante de plaza IP9 (ecuación 9) es menor a 6, el sistema (la red) tiene recursos sin usar o desperdiciados. Si IP9 es mayor o igual a 6, entonces el sistema puede trabajar a su máxima capacidad, sin desperdiciar recursos. Esto significa que el máximo de hilos activos simultáneos para el sistema es 6.

Cabe aclarar que, como se mencionó anteriormente, el análisis se realizó a partir de la red modificada, por lo que no tiene en cuenta la transición T0, la cual modela el evento externo del ingreso de imágenes al sistema. Si la ocurrencia de este evento se incluye en el modelo y se le asigna cierta temporalidad (Figura N° 2), se debe incluir dicha acción como un hilo más. Por lo tanto al incluir T0, la máxima cantidad de hilos activos simultáneos es 7.

Responsabilidad de Hilos y Segmentos de la Red

De acuerdo al procedimiento descrito en [3], sección 4.2, se analizó la red con fin de asignar responsabilidades específicas a los hilos y segmentar la red. Se puede observar (Figura N°2) que la red presenta diferentes forks (conflictos) y joins (uniones). El primer fork ocurre en la plaza P0, de donde surgen 2 segmentos. Estos convergen en un join, en la plaza P6, que al mismo tiempo es un nuevo fork. Nuevamente se generan 2 segmentos que se unen en el join de la plaza P14. La plaza P14 también es un fork de dos segmentos, pero que solo pueden ser ejecutados en exclusión mutua, por la presencia de la plaza P15. Finalmente, estos segmentos se unen en un último join, en la plaza P18, que da origen al último segmento de la red. Por lo tanto los segmentos de la red son S_A, S_B, S_C, S_D, S_E, S_F, S_G, S_H. Las plazas asociadas a cada segmento son:

PS_A: {P0} (37)
PS_B: {P2} (38)
PS_C: {P4} (39)
PS_D: {P8, P13} (40)
PS_E: {P10, P13} (41)
PS_F: {P16} (42)
PS_G: {P17} (43)
PS_H: {P19} (44)

Y las transiciones asociadas a cada segmento son:

TS_A: {T0} (45)
TS_b: {T1, T3} (46)
TS_C: {T2, T4} (47)
TS_D: {T5, T7, T9} (48)
TS_E: {T6, T8, T10} (49)
TS_F: {T11, T13} (50)
TS_G: {T12, T14} (51)
TS_H: {T15, T16} (52)

Los roles de los segmentos son:

S_A: Ingreso de imágen (Importer).
S_B y S_C: Carga de la imágen (Loaders).
S_D y S_E: Filtrado de la imágen (Filters).
S_F y S_G: Recorte de la imágen (Resizers).
S_H: Exportado de la imágen (Exporter).

Se realizó el mismo procedimiento que se utilizó para determinar la máxima cantidad de hilos simultáneos (referencia [3], sección 4.1), para determinar la cantidad máxima de hilos por segmento, es decir, se aplicó el algoritmo para encontrar el máximo de hilos activos simultáneos a cada segmento. El resultado se muestra en la Tabla N° 4.

Tabla N° 4. Máximo de suma de plazas de acción para cada segmento (m_0(P0) = 15).

Segmento	Max[Suma(PS_i)]
S_A	1
S_B	1
S_C	1
S_D	1
S_E	1
S_F	1
S_G	1
S_H	1

Los resultados de la Tabla N° 4 indican que a cada segmento le corresponde un hilo de ejecución. Si bien se tienen 8 segmentos en total, cabe destacar que, debido al marcado de la plaza P15 (m(P15) = 1) S_E y S_F no pueden estar activos simultáneamente (sección Cantidad Máxima de Hilos Simultáneos).

Una representación gráfica de los hilos y segmentos de la red se muestra en la Figura N° 4.

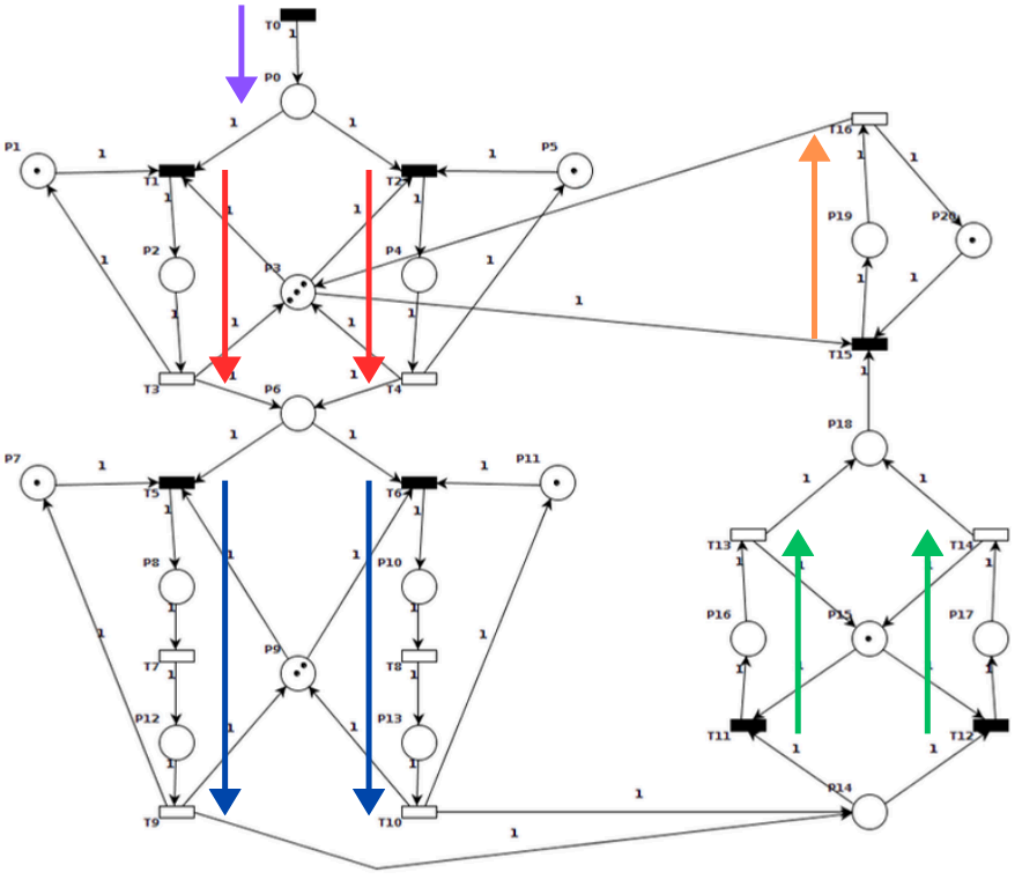

Figura N° 4. Red de petri segmentada en hilos de acción. Hilo violeta: Importer. Hilos rojos: Loaders. Hilos azules: Filters. Hilos verdes: Resizers. Hilo naranja: Exporter.

Implementación

Como se mencionó en la introducción, la implementación del modelo en Java se dividió en dos etapas: modelado de la red sin transiciones temporales y modelado de la red incluyendo transiciones temporales. En este caso las transiciones temporalizadas fueron T0, T3, T4, T7, T9, T8, T10, T13, T14 y T16.

El programa se implementó de forma configurable. Se incluyó la posibilidad de dar prioridad a un segmento frente a su par, es decir otorgarle un exceso de carga. Esto significa que entre los pares S_B-S_c, S_D-S_E y S_F-S_G se da la opción de seleccionar un hilo que procese más imágenes que su par. Esto se implementó en la política del monitor. También se implementó la posibilidad de configurar los intervalos de tiempo de las transiciones, antes de la ejecución, mediante un archivo de configuración.

La política implementada en el monitor fue *signal and continue* y las transiciones temporales se implementaron con semántica de tiempo débil.

Se implementaron las siguientes clases:

- Main
- TaskFactory
- Task
- Importer
- Loader
- Filter
- Resizer
- Exporter
- PetriNet
- Monitor
- Politic
- Queues
- Log

Además, se implementaron las siguientes clases de excepciones:

- InvalidMarkingException
- TaskInterruptedException
- TransitionsMismatchException
- TransitionTimedOutExcpetion

Clases

A continuación se describe la funcionalidad de cada clase, los métodos más relevantes de cada una y de ser necesario, las variables implementadas en la clase.

Main

La clase Main corresponde al ejecutor del programa. Instancia al monitor y al factory de hilos. Configura el tiempo de las transiciones de la red de petri del monitor. Solicita al factory la creación de los hilos y luego espera a que se cumplan 200 disparos de T16, para luego interrumpir los hilos y finalizar el programa. Esperar 200 disparos de T16 es equivalente a esperar a que la suma de ocurrencias de los invariantes IT1 a IT8 sea 200.
En Main se implementaron las siguientes funciones adicionales:

- `void parseConfigFile(String[] args)`: Parsea el archivo de configuración *.ini* con distintos parámetros de configuración del programa. El archivo de configuración tiene las siguientes secciones:
    - [Parametros]: Configuraciones generales. Tiene los siguientes campos:
        - tokens=valor: Marcado inicial en p0.
        - maxTinvariates=valor: Cantidad de invariantes de transición a cumplir para finalizar la ejecución.
    - [TiempoTareas]: Tiempos de sleep de cada tarea, utilizados en el método doTask, para simular la duración de una tarea. Sus campos son:
        - segmento=sleep: Corresponde a los String “A”-”H”. Se asigna el tiempo de sleep en mili segundos.
    - [Transiciones]: Valores de alfa y beta para asignar transiciones temporales. Los tiempos configurados como 0,0 corresponden a transiciones no temporales. Los campos son:
        - transición=alfa,beta: Valor del 0 al 16 con el intervalo de temporalidad.
    - [Prioridad]: Corresponde a los parámetros de la relación de carga sobre el segmento a priorizar. Tiene los siguientes campos:
        - segmento=”A”-”F”. Asigna el segmento al que se le da prioridad.
        - Carga=valor. Corresponde al porcentaje de carga del segmento prioritario. Es un número decimal que en caso de ser 0, no se le asigna prioridad a ningún hilo. En caso contrario, debe ser mayor o igual 0,5 y menor o igual 1,0.

Un ejemplo de cómo se estructuran las secciones en el archivo de configuración se muestra en la Tabla N° 5.

Tabla N° 5. Ejemplo de formato de sección del archivo de configuración .ini.

[Sección 1]	
clave 1	valor
clave 2	valor

- `void usage()`: Indica cómo ejecutar el programa y los parámetros a incluir en el archivo de configuración:

  > java programa.jar `configFile`.ini

Donde *`configFile`.ini* es el archivo discutido anteriormente.

TaskFactory

Esta clase incorpora el patrón de diseño "*Factory*"[7] al programa. La instancia de esta clase es la encargada de crear los hilos (tareas). El propósito es que la creación de los hilos se haga solamente desde el factory. EL método más relevante de la clase es:

- `Thread newTask(String taskType, int[] transitions, Monitor monitor)`: Recibe el nombre de la tarea (el tipo, "Importer", "Loader", "Filter", "Resizer" o "Exporter"), las transiciones disparadas por las mismas y una referencia al monitor. Crea una instancia de la tarea solicitada y un hilo para ejecutar la misma. Retorna la referencia al hilo.

- `Map<Task, Thread> getTasks()`: Devuelve un mapa de las tareas creadas y sus hilos de ejecución correspondientes.

Además la clase TaskFactory fue implementada como Singleton[7], para garantizar un único objeto creador de hilos.

Task

Es la clase padre de la cual todas las tareas heredan. Implementa la interface "Runnable" y posee el método abstracto "doTask", que cada tarea derivada debe implementar. En el método "run" se ejecuta el método "doTask".

Importer, Loader, Filter, Resizer y Exporter

Estas clases son derivadas de la clase Task. Cada clase sobreescribe el método "doTask":

- void doTask(): El método ejecuta un ciclo *while* hasta ser interrumpido por la clase Main. Dentro del bucle se le solicita al monitor el disparo de una transición. Las transiciones a disparar se guardan en un array, el cual es recorrido en cada iteración del ciclo.

PetriNet

Corresponde a la red de petri que modela el sistema. La instancia de esta clase se crea dentro del monitor y es implementada como singleton. Esto es para garantizar una única red de petri y para evitar posibles conflictos entre los hilos que ejecutan las transiciones de la red. Las variables de la clase son:

- `RealMatrix incidenceMatrix`: Matriz de dos dimensiones. Es la matriz de incidencia implementada como RealMatrix para soportar la operación algebraica de multiplicación de matriz por vector.

- `RealVector marking`: Vector de marcado de la red. Implementado como RealVector para soportar las operaciones algebraicas de multiplicación de matriz por vector y suma vectorial.

- `int[] enabledByTokens`: Array que indica si una transición está habilitada o no. Un 1 corresponde a una transición habilitada y un 0 a una transición no habilitada.

- `long[] transitionsTimeStamps`: Array para almacenar el instante de tiempo en que cada transición es habilitada por tokens. El instante de tiempo es utilizado para determinar cuando la transición se habilita temporalmente.

- `ArrayList<Pair<Long, Long>> alphaBeta`: Lista de valores del intervalo temporal, alfa y beta, de cada transición. Implementa la semántica de tiempo débil en la temporización de las transiciones.

- `enum Status {ENABLED, NO_TOKENS, BEFORE_WINDOW, AFTER_WINDOW}`: Enum utilizado para indicar si una transición habilitada fue disparada o por qué razón no se pudo disparar.

- `Status[] transitionsStatus`: Array que almacena el estado de disparo recién mencionado, para cada transición.

Los métodos más importantes de la clase son:

- `boolean isEnabled(int transition)`: Verifica si una transición está habilitada para ser disparada. Debe estar habilitada tanto por marcado, como por tiempo. Devuelve true o false y guarda el estado de la transición en el array transitionStatus.

- `RealVector createTransitionVector(int transition)`: Crea un vector de disparo a partir de una única transición. El objetivo es actualizar el vector de marcado (estado) luego de disparar una transición, utilizando la ecuación de estado (ecuación 18).

- `boolean holdsPlaceInvariants()`: Verifica que se cumplan los invariantes de plaza (ecuaciones 1-9) de la red. En caso contrario termina la ejecución.

- `void updateEnabledTransitions()`: Actualiza el array de transiciones habilitadas por tokens. Se utiliza luego de realizar el disparo de una transición. Para determinar las transiciones habilitadas utiliza la matriz de incidencia y el vector de marcado.

- `boolean fire(int transition)`: Dispara una transición. Verifica si la transición está habilitada y de ser así, actualiza el vector de marcado y el array de transiciones habilitadas. Devuelve true o false.

Monitor

Es el encargado de gestionar los recursos del sistema. Cuando un hilo quiere disparar una transición, el disparo es solicitado al monitor y es este quien tiene acceso directo a la red de petri y gestiona el acceso a la misma. Implementa colas de condición para cada transición y una cola de ingreso, para garantizar que los hilos no tengan acceso al monitor de forma simultánea. Las variables implementadas en el monitor fueron:

- `Semaphore mutex`: Este semáforo representa la cola de ingreso al monitor. Se inicializa con un valor de 1, para que funcione como un mutex.

- `List<Double> counterList`: Lista que almacena la cuenta de disparos por transición. La lista es utilizada por la política, para decidir qué transición bloqueada y habilitada disparar.

- `PetriNet petriNet`: Esta es la instancia de la red de petri que, como se mencionó anteriormente, existe en el monitor.

- `Queues transitionQueues`: Objeto que tiene una lista de semáforos que implementan las colas de condición.

- `Politic politic`: Objeto que modela la política del monitor.

- `Log log`: Log que registra las transiciones disparadas.

El método más relevante importado en el monitor es:

- `void fireTransition(int transition)`: Gestiona el disparo de una transición (el acceso a recursos del sistema). La toma y liberación del mutex ocurre en este método, así como el ingreso y salida de las colas de condición y también la decisión de la política de qué transición habilitar.

Politic

Objeto que modela la política implementada en el monitor. Toma la decisión sobre qué transición que se encuentre bloqueada en una cola de condición disparar. El método donde se implementa la toma de decisión es:

- `int selectTransition(int[] enabledTransitions, List<Double> counterList)`: Recibe el listado de transiciones habilitadas y el listado con la cuenta de disparos por transición. Toma la decisión sobre cuál transición habilitada disparar. Si no se ha configurado dar prioridad a un hilo, simplemente elige la transición habilitada con la menor cantidad de disparos realizados. De esa forma se mantiene un balance equitativo entre la carga que reciben los hilos. Si hay un hilo configurado como prioritario frente a su par, no se selecciona el hilo de menor prioridad a menos que se supere el exceso de carga establecido para el hilo prioritario.

Queues

Esta clase implementa las colas de condición del monitor. Como se mencionó anteriormente, existe una cola de condición por transición. Las variables relevantes de la clase son:

- `ArrayList<Semaphore> queues`: Lista de semáforos donde cada semáforo representa una cola de condición.

- `int[] blockedList`: Array que indica si hay hilos esperando en una cola de condición. Un 1 significa que la cola tiene por lo menos un hilo esperando. Un 0 significa que la cola está vacía.

Los métodos principales de la clase son:

- `void acquire(int transition)`: Ingresa un hilo a la cola de condición indicada por transition.

- `void release(int transition)`: Libera un hilo de la cola de condición indicada por transition.

Log

Log utilizado en el monitor para registrar el disparo de una transición en un archivo de resultados. Los métodos relevantes de la clase son:

- `void logMessage(String message)`: Permite escribir un mensaje (String) en el archivo de log.

- `void logTransition(int transition)`: Registra el disparo de una transición en el archivo de log, utilizando el método logMessage.

- `void addTransition(int transition)`: Registra el disparo de una transición en una lista interna, `List<String> transitionList`, sin escribir en el archivo de log.

- `void writeLog()`: Escribe todas las transiciones almacenadas en la lista transitionList en el archivo de log.

Pruebas y Métodos

Ejecución

Se realizaron diferentes pruebas para validar el funcionamiento de la implementación. Se estudió el funcionamiento del sistema con transiciones no temporizadas y temporizadas y se compararon ambos resultados. Como se mencionó en la descripción de la clase Main, el programa está configurado para funcionar hasta que se cumpla el procesamiento de 200 imágenes (200 invariantes de transición totales completados).

En las pruebas sin transiciones temporales se simuló el tiempo de duración de cada tarea como un sleep entre cada intento de disparar una transición, es decir un sleep implementado en el método doTask de cada Task. En las pruebas con transiciones

temporales todos los tiempos de tarea implementados en el método doTask se configuraron en 1 ms.

Al finalizar la ejecución todas las transiciones disparadas quedan registradas en el log de resultados. Se creó un script de Bash que permite ejecutar el programa repetidas veces.

Test Unitarios

Se realizaron test unitarios sobre los métodos principales de PetriNet y Monitor como pre-validación del código desarrollado. Se comprobó que la red de petri mantenga sus invariantes de plaza, no alcance a estados inválidos y no dispare transiciones no habilitadas.

Análisis de Resultados

Se crearon dos scripts en Python para analizar los resultados de ejecución del programa. El primero cuenta los disparos de todas las transiciones en todos los logs encontrados en el directorio donde se almacenan los mismos. Estos resultados se utilizaron para estudiar la distribución de carga de trabajo entre los diferentes hilos y para verificar el funcionamiento de la asignación de prioridades implementado en la política.

El segundo script analiza un log y verifica que todas las transiciones disparadas pertenezcan a algún invariante de transición. Cuenta la cantidad de invariantes completados y muestra, en caso de existir, las transiciones que no pertenecen a algún invariante. Para realizar la búsqueda de invariantes se utilizó la siguiente expresión regular:

(T0)(.*?)((T1)(.*?)(T3)(.*?)|(T2)(.*?)(T4)(.*?))((T5)(.*?)(T7)(.*?)(T9)(.*?)|(T6)(.*?)(T8)(.*?)(TA)(.*?))((TB)(.*?)(TD)(.*?)|(TC)(.*?)(TE)(.*?))(TF)(.*?)(TG)

Una representación gráfica de esta expresión se muestra en la Figura N° 5.

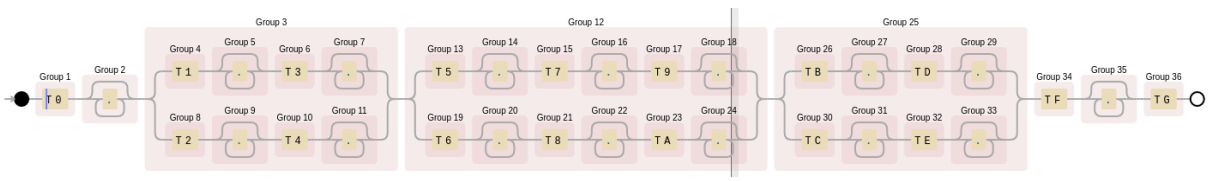

Figura N° 5. Expresión regular utilizada para encontrar los invariantes de transición en los archivos de log.

Resultados

Transiciones no Temporales

A continuación se muestran los resultados de 100 ejecuciones del programa sin transiciones temporales, con un tiempo de sleep de 1 ms para todas las tareas y sin asignar prioridad a ningún hilo.

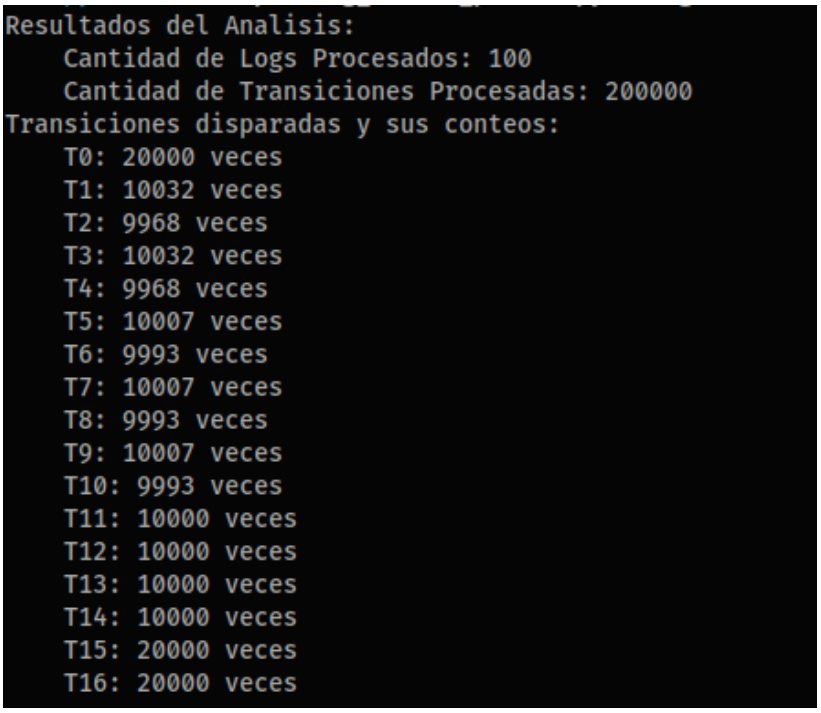

Figura N° 6. Resultados de 100 ejecuciones. Transiciones no temporales. 1 ms sleep por tarea. Sin prioridades.

Se observa un exceso de carga del 0,32 % del hilo importer izquierdo con respecto al derecho y un 0,07 % de exceso de carga del filter izquierdo con respecto al derecho. Se considera la diferencia no significativa, por lo que se puede afirmar que el sistema funciona de forma balanceada.

A continuación se muestran los resultados de asignarle prioridad al hilo resizer izquierdo (segmento F, transiciones 11 y 13). Se asigna una carga del 80 % y se utilizan los mismos tiempos que en el caso anterior.

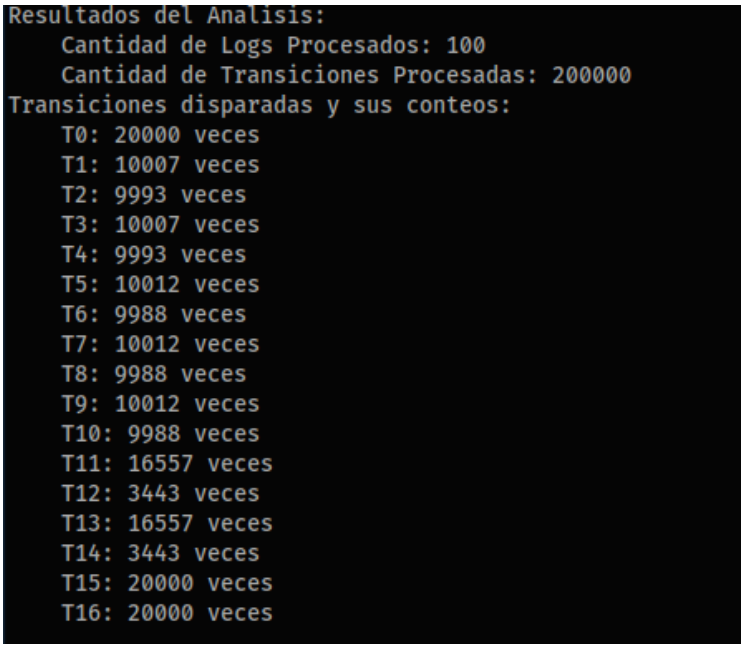

Figura N° 7. Resultados de 100 ejecuciones. Transiciones no temporales. 1 ms sleep por tarea. Prioridad del segmento F con carga del 80 %.

En la Figura N° 7 se observa una carga del 82,79 % del segmento F con respecto al G. El resultado se considera lo suficientemente cercano al 80 % configurado como para confirmar el correcto funcionamiento de la política.

Se configuraron los siguientes tiempos a las tareas, sin asignar prioridades: Importer, 3 ms; Loaders, 10 ms; Filters, 2 ms; Resizers, 4 ms; Exporter, 15 ms. Los resultados se observan en la Figura N° 8.

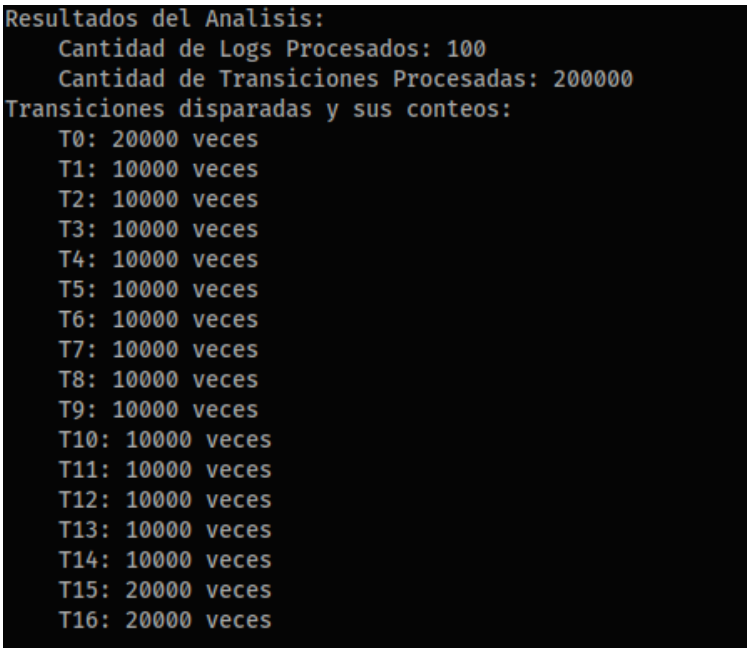

Figura N° 8. Resultados de 100 ejecuciones. Transiciones no temporales. Importer: 3 ms sleep. Loaders: 10 ms sleep. Filters: 2 ms sleep. Resizers 4 ms sleep. Exporter: 15 ms sleep. Sin prioridades.

En este caso se observa que la ejecución es completamente “fair” y que la carga es perfectamente balanceada entre los pares de tareas. Al utilizar los mismos tiempos, pero asignando prioridad al segmento F (80 % de carga) se observa lo siguiente:

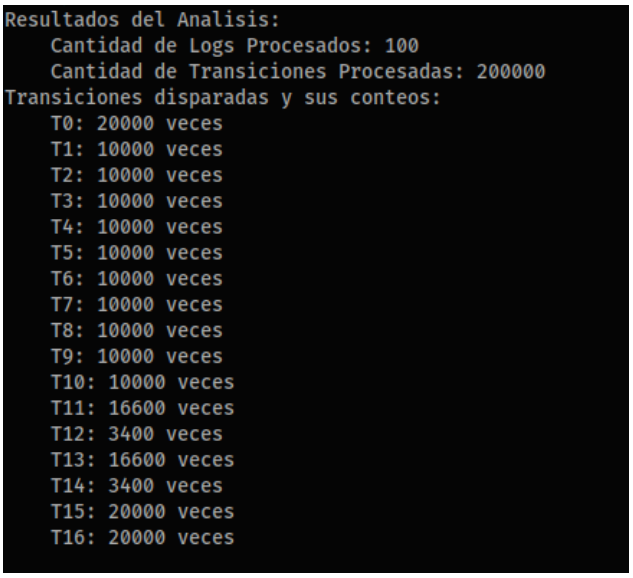

Figura N° 9. Resultados de 100 ejecuciones. Transiciones no temporales. Importer: 3 ms sleep. Loaders: 10 ms sleep. Filters: 2 ms sleep. Resizers 4 ms sleep. Exporter: 15 ms sleep. Prioridad del segmento F con carga del 80 %.

En la Figura N° 9 se observa un 83 % de carga sobre el segmento F. Los resultados son coincidentes con la configuración establecida e indican que la modificación de tiempos de tareas no afectan a la configuración de la política.

Finalmente se muestran los resultados de asignar diferentes tiempos de tarea al par de hilos Filters. Los tiempos configurados fueron: Importer, 3 ms; Loaders, 10 ms; Filter-1, 2 ms; Filter-2, 10 ms; Resizers, 4 ms; Exporter, 15 ms. En la Figura N° 10 se muestran los resultados sin asignar prioridades.

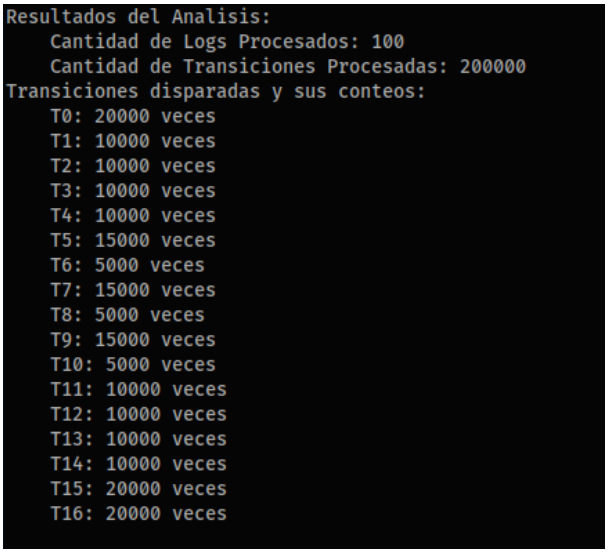

Figura N° 10. Resultados de 100 ejecuciones. Transiciones no temporales. Importer: 3 ms sleep. Loaders: 10 ms sleep. Filter1: 2 ms sleep. Filter2: 10 ms sleep. Resizers 4 ms sleep. Exporter: 15 ms sleep. Sin prioridades.

Se observa que el segmento Filter más rápido (con sleep de 2 ms) recibe un 75 % de la carga. Esto significa que este hilo recibe el exceso de carga mientras el otro hilo espera en el sleep.

En la Figura N° 11 se muestran los resultados asignándole prioridad al segmento F, con 80 % de la carga.

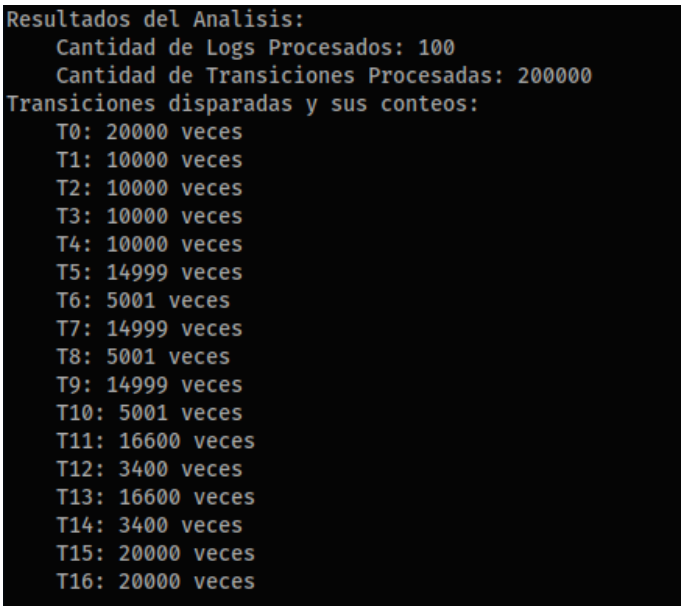

Figura N° 11. Resultados de 100 ejecuciones. Transiciones no temporales. Importer: 3 ms sleep. Loaders: 10 ms sleep. Filter1: 2 ms sleep. Filter2: 10 ms sleep. Resizers 4 ms sleep. Exporter: 15 ms sleep. Prioridad del segmento F con carga del 80 %.

Se observa que la política le otorga un 83 % de la carga al hilo prioritario y nuevamente el hilo Filter más rápido procesa un 75 % de la carga total.

Transiciones Temporales

Como se mencionó anteriormente, en todos los casos con transiciones temporales, los tiempos de tareas asignados en cada Task, en el método doTask, son los siguientes:

- Importer: 1 ms
- Loaders: 1 ms
- Filters: 1 mm
- Resizers: 1 ms
- Exporter: 1 ms

A continuación se muestran los resultados de ejecutar el programa con las transiciones temporales configuradas con los intervalos que se muestran en la Tabla N° 6, sin asignar prioridad a ningún hilo.

Tabla N° 6. Configuración de transiciones temporales.

T	[alpha,beta]
0	1,100
3	10,20
4	10,20
7	2,5
8	2,5
9	3,6
10	3,6
13	7,15
14	7,15
16	5,10

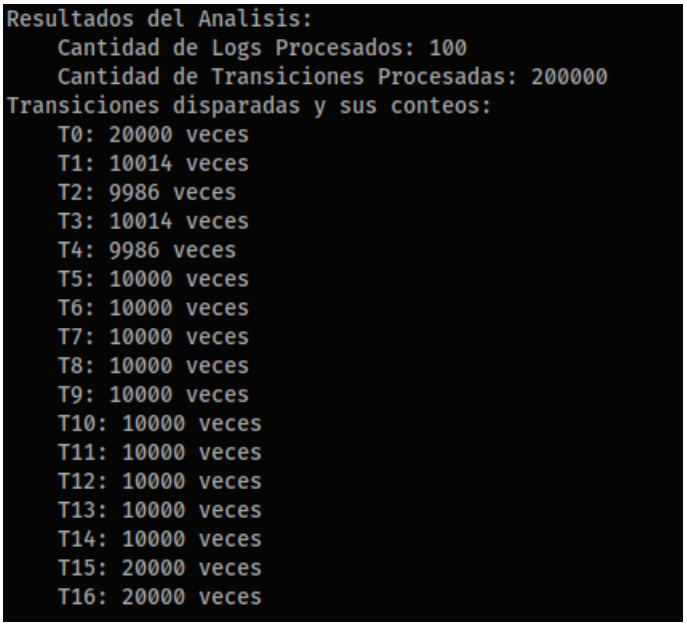

Figura N° 12. Resultados de 100 ejecuciones. Transiciones temporales. Tiempos configurados según Tabla N° 6. Sin prioridades.

Se observa un resultado similar al observado en la Figura N° 8. Todos los pares de hilos están configurados con los mismos intervalos temporales, por lo que la carga es balanceada entre los pares.

Al utilizar la misma configuración de tiempos (Tabla N° 6), pero asignando prioridad al segmento F, con carga del 80 % se obtiene el siguiente resultado.

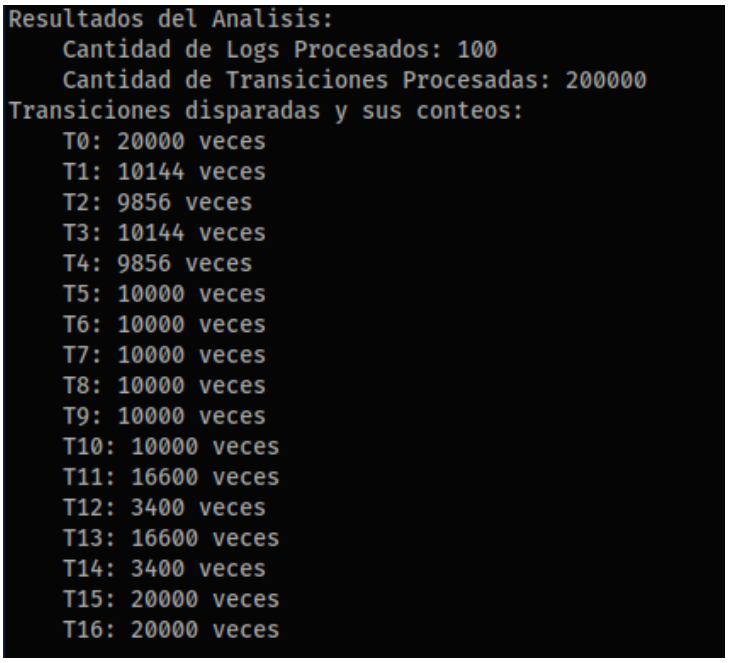

Figura N° 13. Resultados de 100 ejecuciones. Transiciones temporales. Tiempos configurados según Tabla N° 6. Prioridad del segmento F con carga del 80 %.

En la Figura N° 13 se observa que el segmento F procesa el 83 % de la carga total. El resultado es similar al observado en la Figura N° 9 y confirma que el funcionamiento de la política es independiente de la configuración temporal de la red.

A continuación se muestran los resultados obtenidos al configurar las transiciones temporales con los tiempos mostrados en la Tabla N° 7. En este caso se configura el segmento D para funcionar más rápido que el E.

Tabla N° 7. Configuración de transiciones temporales.

T	[alpha,beta]
0	1,100
3	10,20
4	10,20
7	2,5
8	10,15
9	3,6
10	8,12
13	7,15
14	7,15
16	5,10

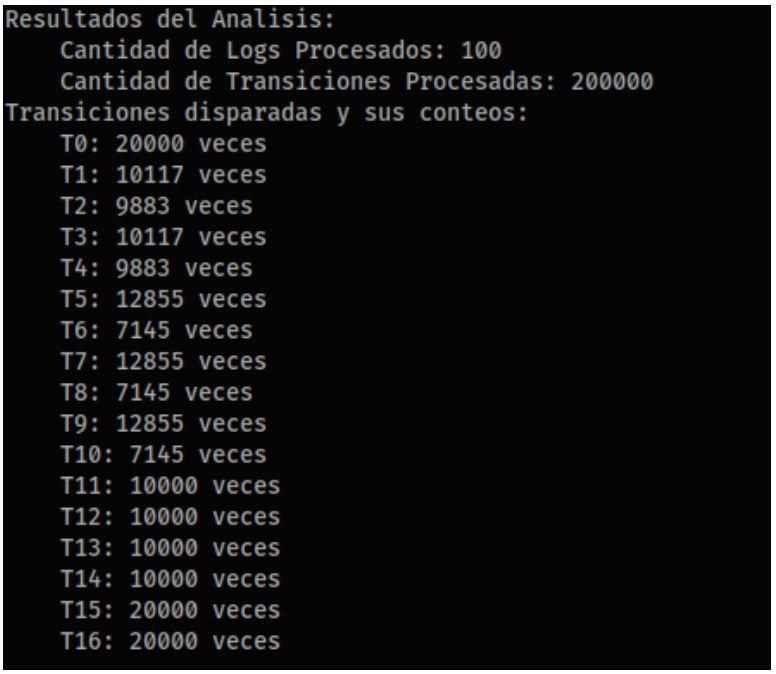

Figura N° 14. Resultados de 100 ejecuciones. Transiciones temporales. Tiempos configurados según Tabla N° 7. Sin prioridades.

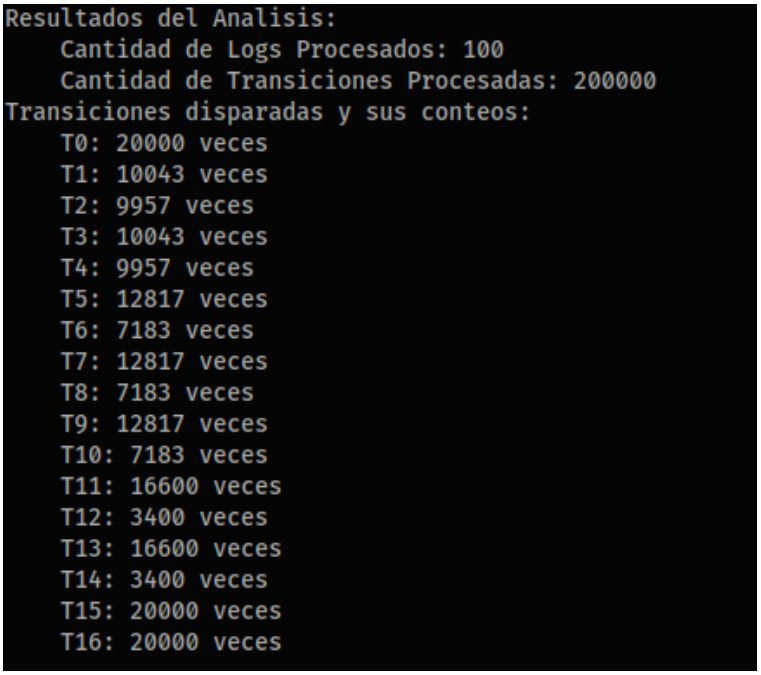

Figura N° 15. Resultados de 100 ejecuciones. Transiciones temporales. Tiempos configurados según Tabla N° 7. Prioridad del segmento F con carga del 80 %.

En las Figuras N° 14 y N° 15 se observa como el segmento D, que trabaja más rápido, procesa un porcentaje mayor de la carga total. Esto es debido a que el hilo continúa trabajando en los intervalos en que el segmento E está esperando por la ventana temporal de la transición. En la Figura N° 15 se observa el efecto de la prioridad configurada en la política, la cual otorga un 83 % de la carga al segmento F.

Finalmente, se muestra el efecto de hacer más lento al hilo priorizado por la política. La configuración de las transiciones temporales para este caso se muestra en la Tabla N° 8

Tabla N° 8. Configuración de transiciones temporales.

T	[alpha,beta]
0	1,100
3	10,20
4	10,20
7	2,5
8	2,5
9	3,6
10	3,6
13	15,25
14	6,12
16	5,10

```
Resultados del Analisis:
    Cantidad de Logs Procesados: 100
    Cantidad de Transiciones Procesadas: 200000
Transiciones disparadas y sus conteos:
    T0: 20000 veces
    T1: 10021 veces
    T2: 9979 veces
    T3: 10021 veces
    T4: 9979 veces
    T5: 10000 veces
    T6: 10000 veces
    T7: 10000 veces
    T8: 10000 veces
    T9: 10000 veces
    T10: 10000 veces
    T11: 16600 veces
    T12: 3400 veces
    T13: 16600 veces
    T14: 3400 veces
    T15: 20000 veces
    T16: 20000 veces
```

Figura N° 15. Resultados de 100 ejecuciones. Transiciones temporales. Tiempos configurados según Tabla N° 8. Prioridad del segmento F con carga del 80 %.

En la Figura N° 15 se observa un resultado similar al de la Figura N° 13. Esto significa que disminuir la tasa de ejecución del segmento prioritario no afecta la carga procesada por el mismo, establecida por la política. En este caso, el segmento G, aunque trabaje más rápido, no es seleccionado por la política si su carga procesada supera el umbral configurado.

En todos los casos discutidos en la sección de Resultados se obtuvo el siguiente resultado al analizar el cumplimiento de los invariantes de transición, con el script basado en la expresión regular (Figura N° 5):

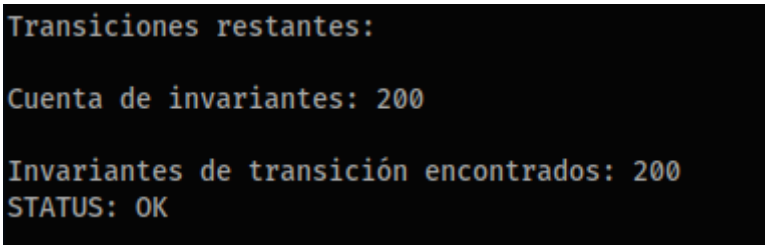

Figura N° 16. Resultados de contar y analizar el cumplimiento de los invariantes de transición.

La Figura N° 16 muestra que en todos los casos analizados se completaron 200 invariantes de transición, como fue configurado el programa. En ningún caso se observaron transiciones restantes, que no pertenecieran a alguno de los invariantes.

Discusión

Los resultados presentados indican que, en principio, el sistema funciona correctamente. En todos los casos en los que no se asignó prioridad a algún hilo y no se modificó la duración de tarea de un segmento, con respecto a su par (los pares de Loaders, Filters y Resizers), el sistema funcionó de manera balanceada y asignó la misma carga (cantidad de imágenes) a cada miembro de los pares de tareas. Esto significa que la política implementada en el monitor logra distribuir de manera equitativa la carga entre cada segmento izquierdo y derecho, en las 3 secciones de procesamiento de la red.

La asignación de prioridades implementada en la política funcionó de la manera esperada. En todos los casos se observó que la carga recibida por el hilo prioritario fue entre 82 % y 83 % cuando se configuró para recibir el 80 % del total. Este comportamiento no se modificó aún cuando se hizo funcionar al hilo prioritario más lentamente que a su par (Figura N° 16). Esto significa que en este caso, si la carga del hilo prioritario está por debajo de la establecida, la velocidad de trabajo de este determinará la velocidad de trabajo de todo el sistema, debido a que el hilo no prioritario nunca será asignado para trabajar.

En los casos en los que se desbalancea la relación de tiempo de trabajo de un par se segmentos (Figuras N° 10, 11, 14 y 15) el hilo más rápido recibe una mayor carga. Esto significa que, si bien la política intenta mantener la carga balanceada entre cada par de segmentos, no traba o detiene al sistema, esperando al hilo más lento, sino que le asigna más trabajo al hilo más rápido, con fin de mantener al sistema funcionando de forma continua.

En términos generales, el comportamiento global del sistema, es decir, el comportamiento con o sin prioridades descrito hasta ahora es similar utilizando transiciones temporales o tareas con duración fija. Sin embargo, el uso de transiciones temporales tiene ventajas frente a la aproximación de sleep predeterminado en cada iteración de trabajo de los hilos. En primer lugar, permiten asignar temporalidad a diferentes partes de la tarea realizada por el hilo (temporalidad diferente a cada transición disparada). Además, al describir la temporización mediante un intervalo, le otorga mayor flexibilidad y realismo a la duración de la tarea. Esto debido a que, una vez que se habilita la ventana temporal de disparo, una misma transición no siempre es disparada en el mismo instante de tiempo. Esto simula de manera más adecuada la duración de una misma tarea, en diferentes circunstancias. Por último, el modelado mediante transiciones temporales, permite implementar un método de detección o corrección de fallos en el sistema. Esto es debido a que una transición deja de estar habilitada una vez que se excede el tiempo de su ventana temporal, lo cual puede significar que una determinada tarea no pudo realizarse o demoró más de un tiempo crítico mínimo establecido.

Conclusiones

Se confirma lo mencionado en la introducción: las redes de Petri son una herramienta esencial para simular y analizar el funcionamiento de sistemas complejos como el presentado en la consigna. Sin lugar a dudas, la realización de este trabajo afianza significativamente la comprensión de los conceptos teóricos vistos en clase.

A medida que se completó y mejoró la implementación del monitor de concurrencia y los distintos agentes de la red, se logró una comprensión mucho más clara del funcionamiento de la red y cómo los distintos parámetros iniciales afectan su evolución. El uso del archivo de configuración, los logs de transiciones y los scripts de python para el análisis de disparos e invariantes fueron parte clave de este proceso.

En los resultados presentados se destaca la eficacia de las políticas del Monitor a la hora de dividir equitativamente la carga del sistema en la red no temporizada y al buscar dinámicamente la mayor eficiencia priorizando los hilos más rápidos en la red temporizada.

La implementación de la política que da prioridad con carga diferenciada a distintos segmentos también nos da una noción de los casos prácticos donde pueden aplicarse a nodos con diferentes recursos disponibles para el procesamiento o la búsqueda de una mayor eficiencia general en el sistema.

En el trabajo conjunto para este proyecto se aplicaron buenas prácticas de programación, como el uso de ramas de GitHub en la implementación de las distintas funcionalidades sin comprometer la estabilidad del código funcional, la implementación de tests unitarios y el uso de herramientas como Maven para la compilación y construcción del proyecto.

Referencias

1. Rob Bamberg. Non-Deterministic Generalised Stochastic Petri Nets Modelling and Analysis. University of Twente. 2012.

2. Petri Nets. Fundamental Models, Verification and Applications. Michael Diaz. John Wiley & Sons, Inc. 2009.

3. O. Micolini, L.O. Ventre. Algoritmos para determinar cantidad y responsabilidad de hilos en sistemas embebidos modelados con Redes de Petri S3PR. Memorias del Congreso Argentino en Ciencias de la Computación - CACIC. 2021.

4. O. Micolini, M. Cebollada, M Eschoyez, L. O. Ventre, M. I. Schild. Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos. XXII Congreso Argentino de Ciencias de la Computación (CACIC). 2016.

5. Sitio oficial de TIme petri Net Analyzer: https://projects.laas.fr/tina/index.php.

6. Petrinator 1.0.0. Desarrollado por Joaquín Rodriguez Felici y Leandro Assón. Universidad Nacional de Córdoba. Sin sitio oficial.

7. Head First Design Patterns. E. Freeman, E. Freeman, K Sierra, B. Bates. O'Reilly Media Inc. 2004.