



Programación Concurrente

04/08/2023

Trabajo Práctico Final

Sistema doble de Procesamiento de imágenes

Integrantes:

Mayorga, Federica

Palombarini, Giuliano Matias

Raimondi, Marcos

Segura, Gaston Marcelo



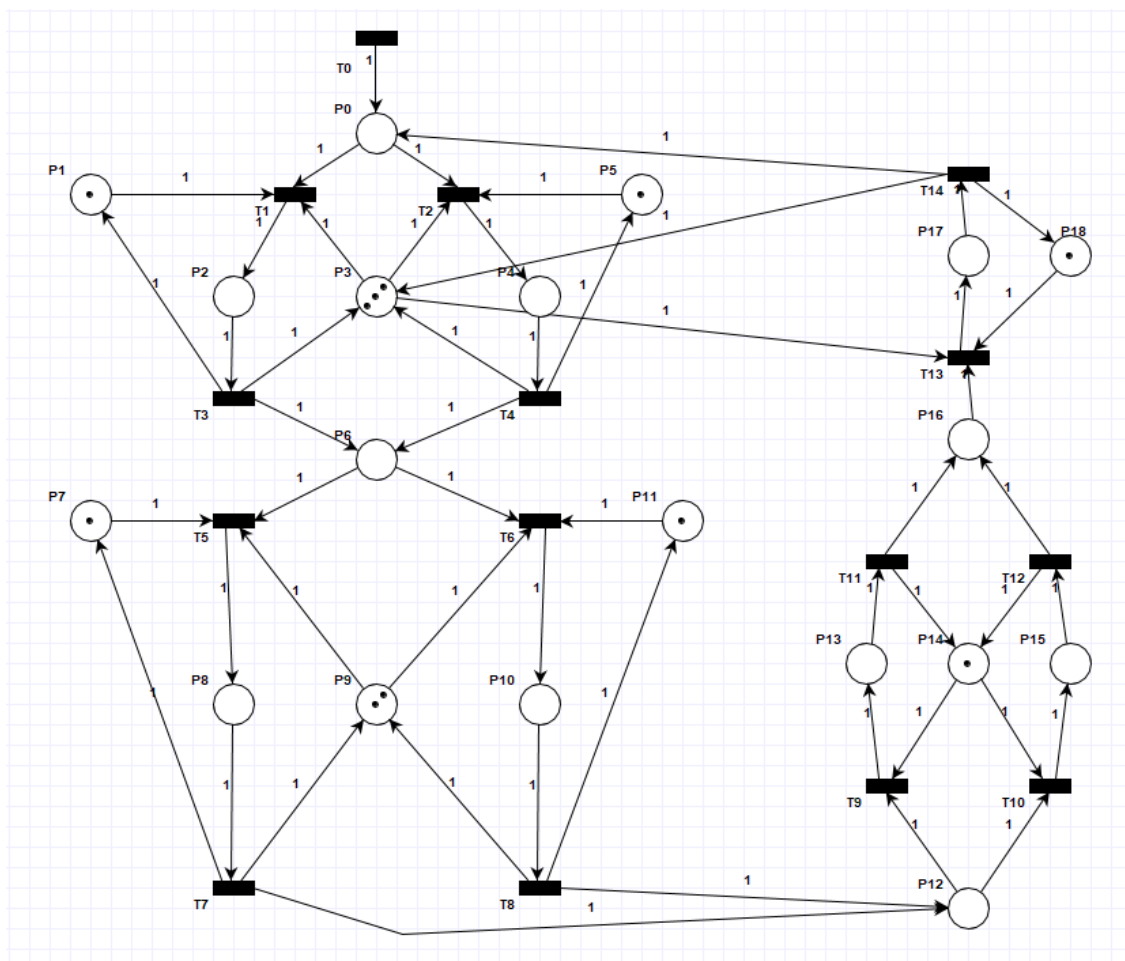
Índice

Índice.....	2
Enunciado.....	3
Desarrollo.....	4
Implementación.....	4
Simulación de la RdP en PIPE.....	4
Invariantes de Transición.....	4
Invariantes de Plaza.....	5
Propiedades de la Red.....	6
DeadLock.....	7
Vivacidad.....	7
Seguridad.....	7
Limitada.....	7
Tabla de Estados del Sistema.....	7
Tabla de Eventos del Sistema.....	8
Diagramas UML.....	9
Diagrama de Clases.....	9
Diagrama de Secuencia.....	10
Cantidad de hilos necesarios y su responsabilidad.....	12
Caso 1.....	15
Caso 2.....	17
Expresión Regular.....	18
Especificaciones.....	19
Resultados.....	23
Análisis de tiempos.....	26
Tiempo considerando una secuencialización del programa.....	26
Tiempo considerando una semi secuencialización del programa.....	26
Tiempo considerando la concurrencia del programa.....	27
Conclusiones.....	30
Anexo.....	31

Enunciado

En la imagen se observa una Red de Petri (RdP) que modela un sistema doble de procesamiento de imágenes.

El sistema tiene distintos bloques o secuencias de procesos. Primero se almacenan las imágenes en un contenedor (buffer) de entrada de imágenes, y se cargan las imágenes al siguiente contenedor para su procesamiento. En segundo lugar, se ajusta la calidad de las mismas y se almacenan en un contenedor de imágenes mejoradas en calidad. En tercer lugar, las imágenes son recortadas y depositadas en un contenedor de imágenes en estado final. Por último, estas son exportadas fuera del sistema.





Desarrollo

1. Con la herramienta de simulación Platform Independent Petri net Editor (PIPE) se observó el comportamiento de la RdP modelo. Esta información servirá para analizar las propiedades de la RdP.
2. Se realizaron tablas con los estados y eventos del sistema.
3. Se realizó un diagrama de clase para el enunciado y un diagrama de secuencia para el monitor a implementar.
4. Se determinó la cantidad de hilos a implementar para la ejecución del sistema, en dos casos planteados. Realizando gráficos para observar las responsabilidades de cada hilo.
5. Se implementaron las dos políticas planteadas, en los conflictos.
6. Se realizan cálculos de tiempo para diferentes casos y el análisis de los mismos.

Implementación

Simulación de la RdP en PIPE

Se utiliza PIPE para realizar la simulación de la RdP planteada, ya que esta herramienta nos facilita el cálculo de los Invariantes de Transición (IT) y los Invariantes de Plaza (IP), como así también nos permite ir comprobando el funcionamiento de la red.

A continuación, el análisis de los IT e IP, que luego serán de utilidad para determinar la cantidad de hilos a implementar y la responsabilidad de los mismos.

Invariantes de Transición

$$IT_1 = \{T2, T4, T6, T8, T10, T12, T13, T14\}$$

$$IT_2 = \{T2, T4, T6, T8, T9, T11, T13, T14\}$$

$$IT_3 = \{T2, T4, T5, T7, T10, T12, T13, T14\}$$

$$IT_4 = \{T2, T4, T5, T7, T9, T11, T13, T14\}$$

$$IT_5 = \{T1, T3, T6, T8, T10, T12, T13, T14\}$$

$$IT_6 = \{T1, T3, T6, T8, T9, T11, T13, T14\}$$

$$IT_7 = \{T1, T3, T5, T7, T10, T12, T13, T14\}$$

$$IT_8 = \{T1, T3, T5, T7, T9, T11, T13, T14\}$$

→ Estos invariantes representan los procesos por los que debe pasar una imagen en el sistema: carga, ajuste, recorte, exportación, etc. Cada invariante representa un “camino” posible que puede recorrer la imagen para pasar por todos esos procesos.

Invariantes de Plaza

$$IP_1 : P1 + P2 = 1$$

→ Este invariante indica que el proceso de carga “izquierdo” puede cargar como máximo una imagen por vez.

$$IP_2 : P4 + P5 = 1$$

→ Este invariante indica que el proceso de carga “derecho” puede cargar como máximo una imagen por vez.

$$IP_3 : P7 + P8 = 1$$

→ Este invariante indica que el proceso de ajuste “izquierdo” puede ajustar como máximo una imagen por vez.

$$IP_4 : P10 + P11 = 1$$

→ Este invariante indica que el proceso de ajuste “derecho” puede ajustar como máximo una imagen por vez.

$$IP_5 : P17 + P18 = 1$$

→ Este invariante indica que el proceso de exportado puede exportar como máximo una imagen por vez.

$$IP_6 : P8 + P9 + P10 = 2$$

→ Este invariante indica que el proceso de ajuste puede ajustar simultáneamente hasta dos imágenes por vez.

$$IP_7 : P_{13} + P_{14} + P_{15} = 1$$

→ Este invariante indica que el proceso de recorte se realiza de a una imagen por vez. Es decir, la rama “izquierda” y “derecha” de este proceso no se pueden ejecutar simultáneamente.

$$IP_8 : P_2 + P_3 + P_4 + P_{17} = 3$$

→ Este invariante indica que se puede cargar y exportar imágenes simultáneamente. El conjunto $\{IP_1, IP_2, IP_4, IP_7\}$ indican que se puede cargar dos imágenes en paralelo y simultáneamente exportar otra imagen.

$$IP_9 : P_0 + P_2 + P_4 + P_6 + P_8 + P_{10} + P_{12} + P_{13} + P_{15} + P_{16} + P_{17} = N$$

→ Este invariante en el cual ‘N’ representa el número de tokens que pasan por P_0 (este invariante deja de serlo si se considera la transición fuente y ‘N’ tiende a infinito) si ‘N’ es finito este invariante representa el recorrido de las imágenes por los distintos estados posibles en los que se puede encontrar: en buffer de entrada, cargando, en contenedor, en contenedor de mejoradas, ajustando, recortando, en contenedor de listas y exportando.

Propiedades de la Red

Considerando que el análisis de las propiedades de la RdP se realizó uniando T16 con P0, eliminando T0 y marcando las transiciones como inmediatas, se obtuvieron los siguientes resultados:

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

Figura 1: Resultado del análisis sobre propiedades de la RdP, obtenido con PIPE.



DeadLock

La red presentada no tiene deadlock, esto implica que nunca va a terminarse inesperadamente por un bloqueo. Es decir, las imágenes siempre serán procesadas en su totalidad.

Vivacidad

La red es viva, lo que quiere decir que en cualquier marcado posible siempre existe una secuencia para disparar cualquiera de las transiciones. Por lo tanto, decimos que las imágenes siempre podrán ser procesadas por el sistema en algún momento.

Seguridad


La RdP no es segura, esto quiere decir que puede haber más de un token en las plazas. Esto se observa en algunas plazas de recursos, representando la existencia de múltiples recursos o en plazas que representan buffers o contenedores de imágenes, indicando que pueden almacenar más de una.

Limitada

La red es limitada ya que no posee alguna plaza con cantidad de tokens que aumenten ilimitadamente, con lo cual el conjunto de marcas alcanzables es finito. Que sea limitado nos da la idea de que estamos representando un sistema real, en este caso el procesamiento de imágenes. Pero considerando la RdP original que contiene una transición fuente no sería limitada.

Tabla de Estados del Sistema

Plaza	Estado
P0	Buffer de entrada de imágenes
P1	Recurso de carga
P2	Imagenes en carga
P3	Recursos de entradas y salidas
P4	Imagenes en carga
P5	Recurso de carga



P6	Buffer de imágenes a procesar
P7	Recurso de ajuste
P8	Imágenes en ajuste de calidad
P9	Recursos de acceso a ajuste
P10	Imágenes en ajuste de calidad
P11	Recurso de ajuste
P12	Buffer de imágenes mejoradas en calidad
P13	Imágenes a recortar tamaño definitivo
P14	Recurso de acceso a recorte
P15	Imágenes a recortar tamaño definitivo
P16	Buffer de imágenes en estado final
P17	Imágenes a exportar fuera del sistema
P18	Recurso de exportación

Tabla de Eventos del Sistema

Transición	Evento
T0	Carga de imagen al buffer de entrada
T1	Toma de imagen a cargar
T2	Toma de imagen a cargar
T3	Carga de imagen a procesar
T4	Carga de imagen a procesar
T5	Toma de imagen a ajustar
T6	Toma de imagen a ajustar

T7	Carga de imagen ajustada
T8	Carga de imagen ajustada
T9	Toma de imagen a recortar
T10	Toma de imagen a recortar
T11	Carga de imagen recortada
T12	Carga de imagen recortada
T13	Toma de imagen a exportar
T14	Exportar imagen

Diagramas UML

Diagrama de Clases

A continuación se observa el diagrama de clases en dos partes debido a su extensión. Para mayor calidad de imagen se encuentra en la sección de entregas un link con el mismo.

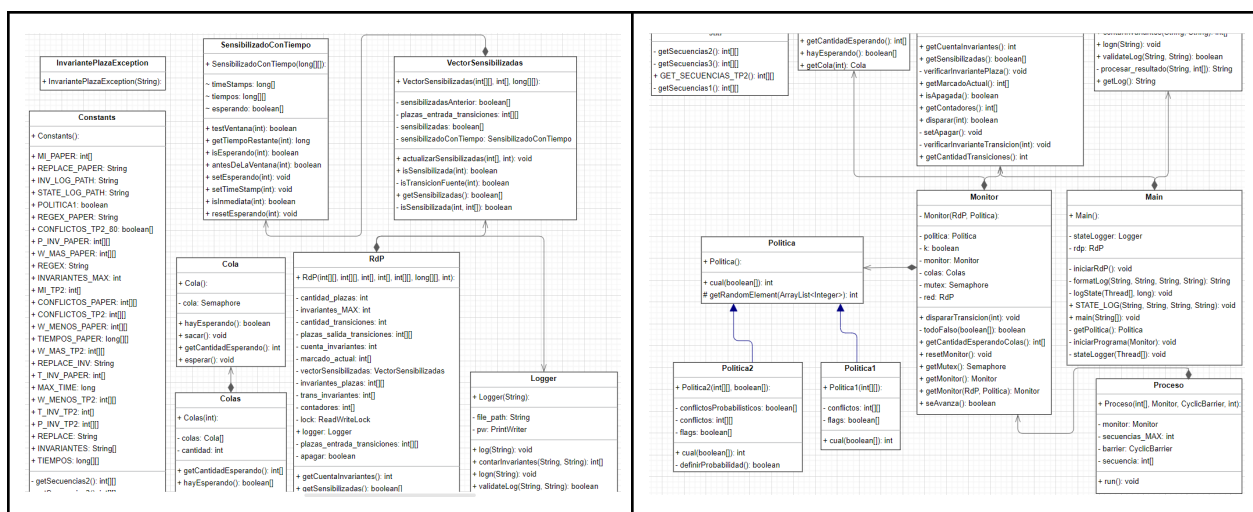


Figura 2: Diagrama de Clases obtenido.

Diagrama de Secuencia

Para el Diagrama de Secuencia se observa el disparo exitoso de una transición sensibilizada, mostrando el uso de la política. Ver en la sección “Entregas” para mayor calidad de imagen.

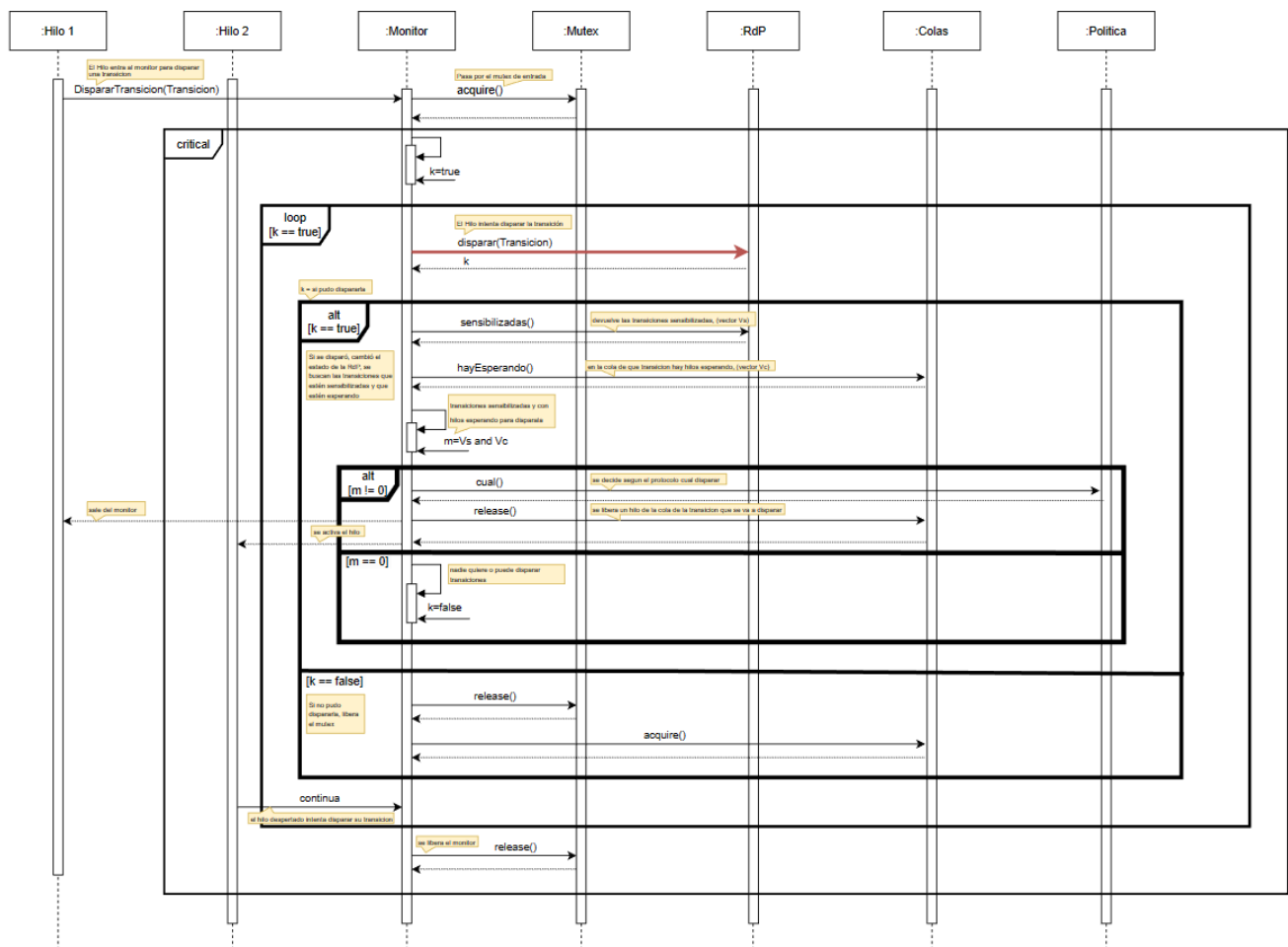


Figura 3: Diagrama de Secuencia de un disparo de transición exitoso.

En el Diagrama de Secuencia anterior, puede verse en RdP el método disparar(Transición) más en detalle:

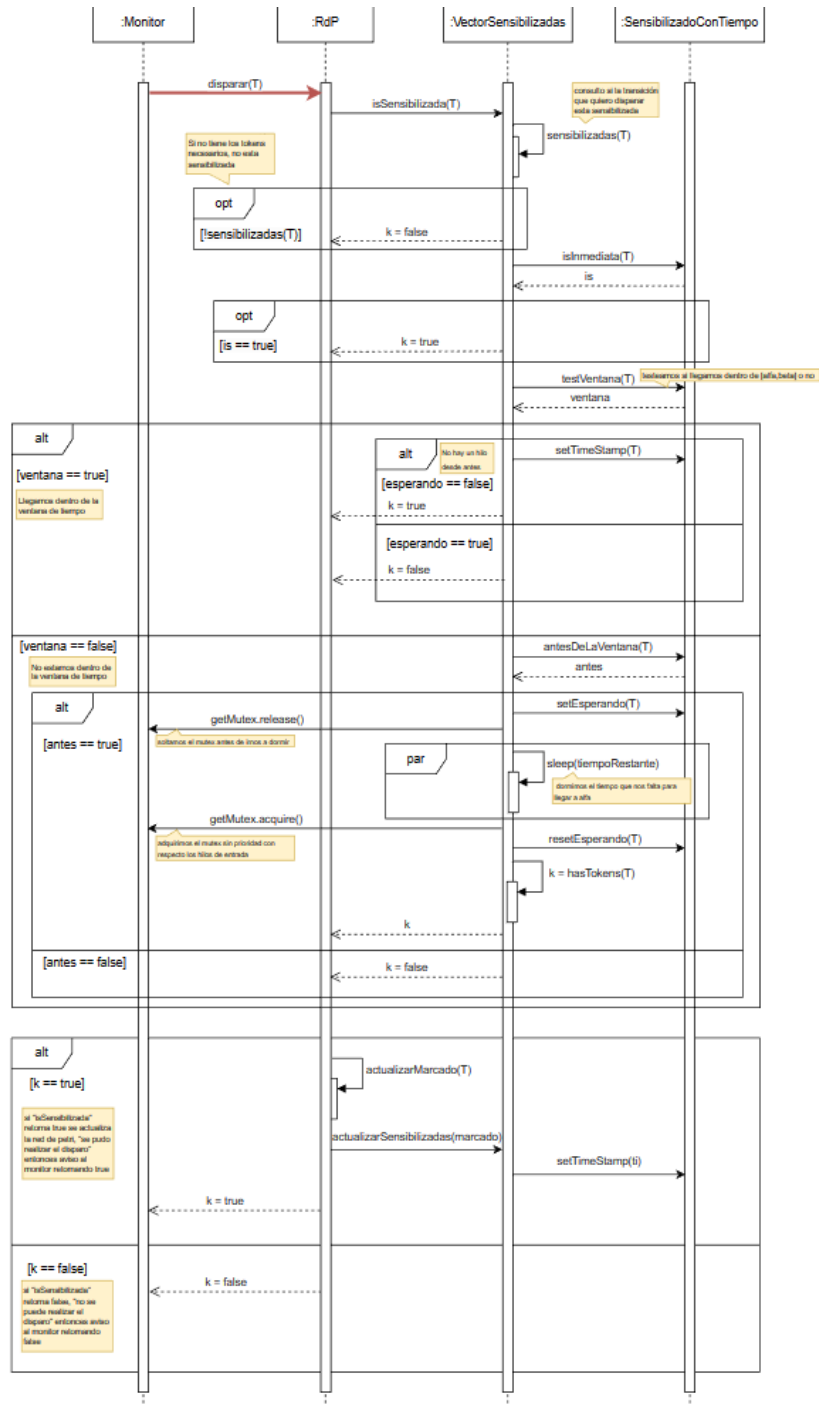


Figura 4: Detalle del diagrama de secuencia del disparo de transición exitoso.

Cantidad de hilos necesarios y su responsabilidad

Para poder indicar la cantidad de hilos necesarios para la ejecución del programa, se aplican los algoritmos propuestos por la referencia ofrecida por la cátedra.

Sobre la RdP, se obtuvieron los IT explicitados anteriormente. A estos, les obtuvimos el conjunto de **plazas asociadas**, denominados como PI_i .

$$PI_1 = \{P0, P3, P4, P5, P6, P9, P10, P11, P12, P14, P15, P16, P17, P18\}$$

$$PI_2 = \{P0, P3, P4, P5, P6, P9, P10, P11, P12, P13, P14, P16, P17, P18\}$$

$$PI_3 = \{P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P17, P18\}$$

$$PI_4 = \{P0, P3, P4, P5, P6, P7, P8, P9, P12, P13, P14, P16, P17, P18\}$$

$$PI_5 = \{P0, P1, P2, P3, P6, P9, P10, P11, P12, P14, P15, P16, P17, P18\}$$

$$PI_6 = \{P0, P1, P2, P3, P6, P9, P10, P11, P12, P13, P14, P16, P17, P18\}$$

$$PI_7 = \{P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P17, P18\}$$

$$PI_8 = \{P0, P1, P2, P3, P6, P7, P8, P9, P12, P13, P14, P16, P17, P18\}$$

El siguiente paso fue, sobre estos PI_i , quitar las plazas con restricciones, recursos o idle (fuente).

$$PEliminadas = \{P1, P5, P7, P11, P18 \cup P3, P9, P14 \cup P0, P6, P12, P16\}$$

Obteniendo así las **plazas de acción** asociadas a su respectivo IT, denominadas como PA_i .

$$PA_1 = \{P4, P10, P15, P17\}$$

$$PA_2 = \{P4, P10, P13, P17\}$$

$$PA_3 = \{P4, P8, P15, P17\}$$

$$PA_4 = \{P4, P8, P13, P17\}$$

$$PA_5 = \{P2, P10, P15, P17\}$$

$$PA_6 = \{P2, P10, P13, P17\}$$

$$PA_7 = \{P2, P8, P15, P17\}$$

$$PA_8 = \{P2, P8, P13, P17\}$$

De estas PA , se puede realizar el conjunto de plazas de acción y ver su **marcado** en la evolución de la RdP. De las sumas de estos marcados, se tiene interés en la de mayor valor.

P2	P4	P8	P10	P13	P15	P17	SUMA
1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	1
...
1	1	1	1	1	0	1	6
1	1	1	1	0	1	1	6

Tabla: La tabla enumera todos los marcados posibles, pero debido a la extensión de la misma solamente se muestran los 4 primeros estados con su suma y los 2 últimos estados.

Entonces la *cantidad máxima de hilos activos simultáneos* en el sistema serán **6**. Lo cual es un resultado lógico, ya que la plaza 14 al tener un solo token, podremos tener un hilo en P15 o en P13, mientras que en las restantes plazas de acción podremos tener hasta un hilo de manera simultánea en cada una.

A continuación, se aplica el algoritmo para asignar la **responsabilidad** a los **hilos del sistema**.

Se utilizan los IT explicitados anteriormente. Se obtiene el conjunto de las plazas de cada segmento, denominado como PS_i . Y luego a estos PS_i , se les obtiene el marcado de segmento.

$$PS_A = \{P2\}$$

$$MS_A = \{0\} \mid \{1\}$$

$$PS_B = \{P4\}$$

$$MS_B = \{0\} \mid \{1\}$$

$$PS_C = \{P8\}$$

$$MS_C = \{0\} \mid \{1\}$$

$$PS_D = \{P10\}$$

$$MS_D = \{0\} \mid \{1\}$$

$$PS_E = \{P13\}$$

$$MS_E = \{0\} \mid \{1\}$$

$$PS_F = \{P15\}$$

$$MS_F = \{0\} \mid \{1\}$$

$$PS_G = \{P17\}$$

$$MS_G = \{0\} \mid \{1\}$$

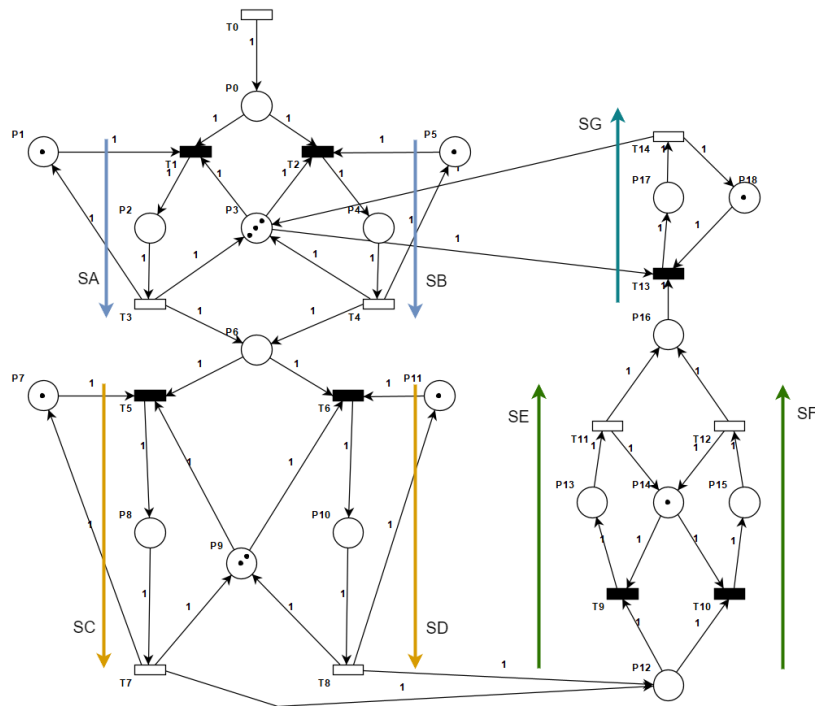



Figura 5: muestra los hilos asociados a cada segmento.



El marcado máximo de cada uno de estos segmentos, será la cantidad de hilos necesarios para ese segmento. Si sumamos *la cantidad máxima de hilos del sistema*, tenemos que nuestro sistema va a necesitar **7** hilos.

Caso 1

Hasta este momento no se ha contemplado la asignación en la responsabilidad de ejecución de los IT según si estos presentan conflicto y/o uniones. Para estos casos particulares, esta responsabilidad se fracciona en diferentes segmentos. Un segmento antes del conflicto y otros dos segmentos después.

Como ventaja, se elimina la decisión de solución del conflicto por parte del hilo, y ahora es tomada por el componente responsable, la política.

Para la RdP de nuestro sistema, se observó que en el conflicto entre nuestro *Buffer de imágenes mejoradas en calidad* (P12) y las *Tomas de imágenes a recortar* (T9, T10), siendo la instancia previa de transiciones instantáneas, no se respetaban las políticas (expuestas más adelante).

Para solucionar esto, se decidió realizar el segmentado propuesto y agregar dos hilos más al análisis anterior, encargados de disparar las transiciones T9 y T10, obteniendo entonces una *nueva cantidad máxima de hilos del sistema* de **10**.

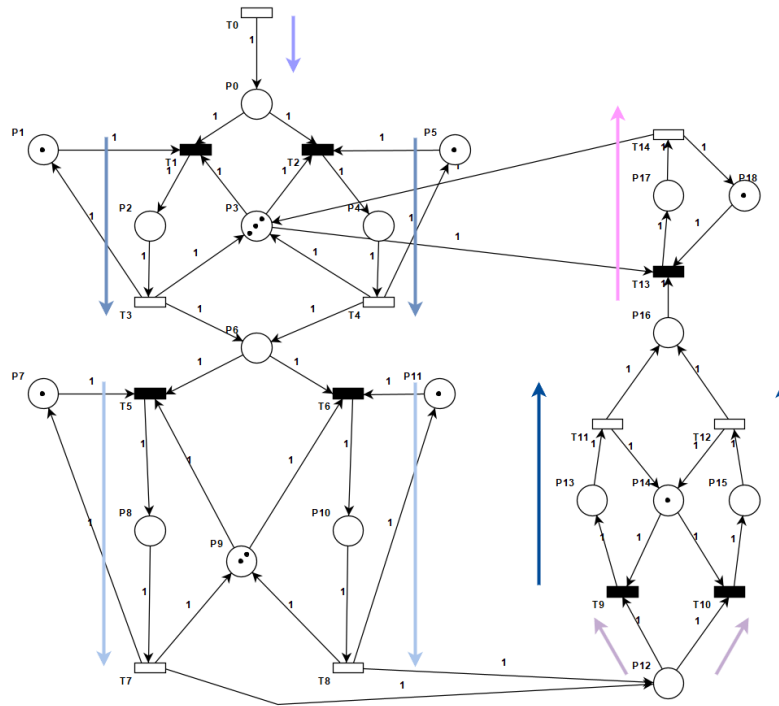


Figura 6: muestra una nueva cantidad de hilos asociados al sistema.

Analizando las alternativas de implementación y considerando que, para las secciones donde nos encontramos con estructuras que comparten transiciones en conflicto estructural y otras con una plaza de unión, que no fueron completamente considerados en el análisis anterior de responsabilidad de los hilos, se decidió explorar la posibilidad de agregar más hilos. Como vemos en la siguiente figura, se han agregado hilos en las salidas de nuestro *Buffer de entrada de imágenes* y también en el *Buffer de imágenes a procesar*.

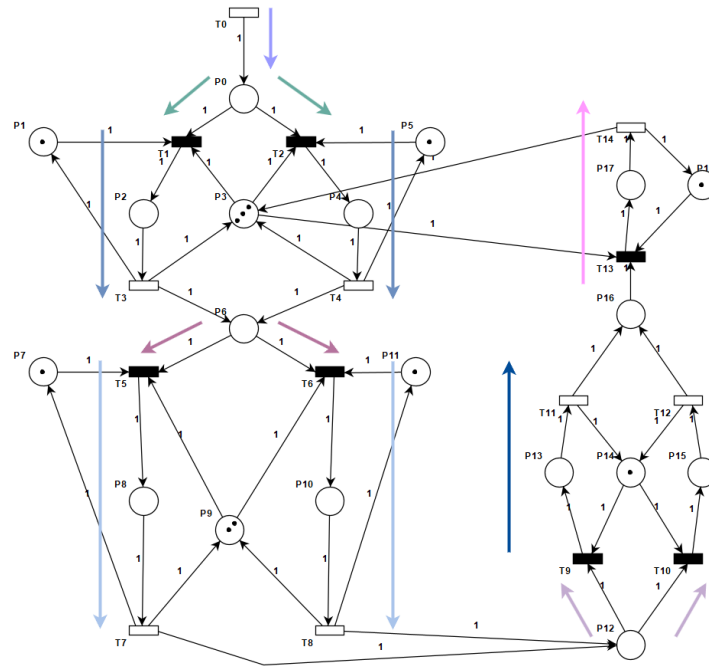


Figura 7: muestra la cantidad final de hilos del sistema.

Realizando distintas combinaciones en la cantidad de hilos, y la responsabilidad de sus segmentos, se proponen **14** hilos como *nueva cantidad máxima de hilos del sistema*, que será desarrollado más adelante en la sección de Análisis de Tiempo. Podemos resaltar desde ya que se nota una mejoría de rendimiento.

Esta alternativa nace para que las políticas sean aplicables a todas las secciones de la RdP y no solo a la sección que involucra a T9 y T10.

Caso 2

En la RdP un invariante de transición se presenta con otro en un join en la plaza P16, y desde el enunciado se propone el caso de agregar un hilo por token simultáneo en esta plaza. Pero siendo que la secuencia de disparos posterior a esta plaza puede ser manejada únicamente por un hilo, ya que la plaza P18 posee solo un token por ser un *recurso de exportación*, no se encuentra mayor necesidad de agregar este hilo extra.

Expresión Regular

Para realizar el análisis de la expresión regular (regex, por su nombre abreviado en inglés) se utilizaron las siguientes páginas: [regex101](#) y [Debuggex](#)

A continuación se muestra la expresión regular obtenida, en lenguaje python:

```
(?P<I0>T0)(?P<T0>(?:\d+)*?)(?: (?P<I1>T1)(?P<T1>(?:\d+)*?)(?P<I3>T3) | (?P<I2>T2)(?P<T2>(?:\d+)*?)(?P<I4>T4))(?P<T3T4>(?:\d+)*?)(?: (?P<I5>T5)(?P<T5>(?:\d+)*?)(?P<I7>T7) | (?P<I6>T6)(?P<T6>(?:\d+)*?)(?P<I8>T8))(?P<T7T8>(?:\d+)*?)(?: (?P<I10>T10)(?P<T10>(?:\d+)*?)(?P<I12>T12) | (?P<I9>T9)(?P<T9>(?:\d+)*?)(?P<I11>T11))(?P<T11T12>(?:\d+)*?)(?P<I13>T13)(?P<T13>(?:\d+)*?)(?P<I14>T14)(?P<T14>(?:\d+)*?)
```

La expresión regular utiliza metacaracteres (como . ^ \$ { } [] \ | ()) y cuantificadores (como * + ?). Los metacaracteres son símbolos especiales que no son leídos por la regex como caracteres comunes. En cambio, los cuantificadores se utilizan para cuantificar, expresar numéricamente una cantidad de repeticiones, de estos caracteres.

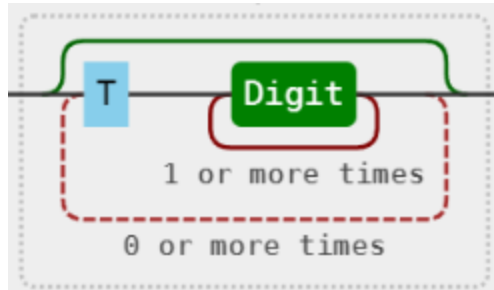
A continuación, se explica brevemente los caracteres utilizados en la expresión regular:

T0 → Busca el carácter 'T' y luego el carácter '0'.

(?P<name>) → Define un grupo captador referenciado con un nombre <name>.

(?:) → Define un grupo no captador, al que se puede aplicar cuantificadores. Es un grupo de caracteres que interesa encapsular, pero no capturar.

(?:T(?:\d+))*?) → Define un grupo no captador que coincide (match) con cero o más caracteres de 'T' y a continuación un carácter de número decimal de al menos un dígito. Al finalizar con *?, esto hace coincidencia con el grupo de caracteres anteriores con un comportamiento vago o lazy, es decir, el mínimo.

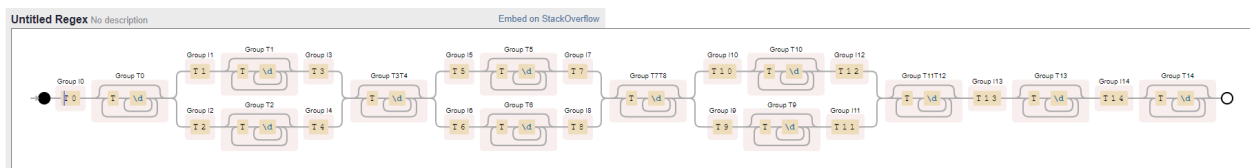


Los grupos con nombres referenciados, que pueden verse en la representación visual de la regex, son los que luego serán utilizados para coincidir con los patrones repetitivos.

A continuación, nuestros grupos:

```
\\g<T0>\\g<T1>\\g<T2>\\g<T3T4>\\g<T5>\\g<T6>\\g<T7T8>\\g<T9>\\g<T10>\\g<T11T12>\\g<T13>\\g<T14>
```


Representación visual de la regex:



La bifurcación de caminos, que puede observarse en la representación visual de la regex, indican los posibles caminos de transiciones que pueden ocurrir para obtener un invariante de transición.

Especificaciones

Para la implementación del problema anteriormente enunciado, se observa el control de los problemas de concurrencia y condiciones de sincronización a resolver mediante la implementación de un monitor de concurrencia.




Partiendo de los diagramas, se desarrollaron en profundidad las clases y métodos de cada una, que son detallados a continuación:

En la clase **Main** en primer lugar se crea y se obtiene la política a utilizar a través de *getPolitica()*, en segundo lugar inicia la rdp con el método *iniciarRdP()* que le otorga todos los parámetros necesarios para su creación, finalmente se obtiene el monitor a utilizar mediante *getMonitor(rdp, politica)*. Con el método *iniciarPrograma(monitor)* se crean e inician los hilos, en función de la cantidad de secuencias que tenemos, y la *CyclicBarrier*, la cual permite la sincronización de dos o más hilos (en este caso, son los 14 hilos utilizados según las secuencias nombradas al inicio del documento más el hilo main) en un determinado punto, el cual en este caso es cuando se finaliza el disparo de su secuencia. El método *stateLogger(threads)* crea un nuevo hilo, del tipo *Daemon*, denominado *stateLoggerThread* que contiene información sobre cada hilo y del sistema en general.

La clase **Proceso**, la cual implementa la interfaz *Runnable*, y se crea al pasarla como parámetro a los nuevos hilos. Contiene el método *run()* sobrescrito el cual va a disparar la secuencia asignada a cada hilo, sumar el contador *contadorSecuencia*, y al terminar de dispararla ejecutar el *await* ese hilo.

Debido a que cada variable de condición del monitor son contenedores de subprocesos que esperan una determinada condición, son representadas a través de las transiciones de la red. Estas transiciones pueden ser disparadas o no, por lo tanto están a su vez representadas por semáforos de tipo *mutex*. Como ya se sabe, los semáforos tienen asociados una cola, la cual contiene a los hilos que esperan que se de la condición para disparar la transición. Por lo tanto se puede decir que hay una serie de colas dentro de una cola:

Clase **Cola**: en su constructor crea un semáforo con *permits=0*, ya que al inicio del programa no se tienen hilos esperando, entonces directamente se van a dormir al no poder dispararse, y además está establecido con *fairness* (justicia), para que el siguiente hilo que obtenga el recurso sea el que más tiempo lleva esperando por él. Contiene los métodos: *sacar()*, que ejecuta el *release()* a la cola, y *esperar()*, que ejecuta el *acquire()* a la cola. Otros métodos que contiene son: *hayEsperando()*, el cual devuelve si hay o no un hilo esperando en la cola, y *getCantidadEsperando()* que devuelve el tamaño de la cola de hilos que están esperando.




Clase **Colas**: en su constructor permite crear, en función del valor que se coloca como parámetro, la cantidad de Cola. Se dice que Colas está compuesta por Cola. Contiene los métodos: *getCola(index)*, el cual devuelve la Cola que se pide por parámetro, y *hayEsperando()*, el cual devuelve un vector que indica si hay o no hilos esperando en cada cola de la red.

La clase **Constants** contiene valores (constantes) que se utilizan por las demás clases, como son: matrices, tiempos, regex, políticas, path, invariantes. Posee el método *GET_SECUENCIAS_TP2()* que devuelve la secuencia que utiliza 14 hilos, o la de 8 hilos o la utilizada en el paper, según cual se encuentre seleccionada.

La clase **Logger** crea el archivo log en la ubicación que es pasada por parámetro al constructor, y a través del método *validateLog(regex,replace)*, busca con la regex la combinación de todos los invariantes de transición que permitan corroborar el funcionamiento de la red, para lo cual no debería quedar ninguna transición en el log al ir iterando constantemente y reemplazando con los grupos definidos. Con el método *log(string)* se añade al log la transición disparada (como string), y con el método *getlog()* se puede obtener el archivo log.

Clase **Monitor**: su constructor utiliza la red y la política pasada por parámetro, crea un semáforo con un solo permit, ya que existe solo un único monitor, y además crea la cantidad de colas en base a la cantidad de transiciones que hay en la red. Utiliza el patrón singleton, por lo tanto esta clase tiene sólo una instancia. Necesita de una red de petri y de la política, además tendrá asociadas las Colas. Posee los métodos: *getmonitor()*, el cual crea el monitor en caso de que no se lo haya hecho y lo retorna, *todoFalso(m)* que devuelve true o false según si el arreglo que se le pasa está formado por todos elementos falsos o no (o sea, si se tiene alguna transición sensibilizada con algún hilo dispuesto a dispararla), *seAvanza()* informa si la rdp está apagada o no (o sea, lee el valor de la variable apagar), *dispararTransicion(transicion)* que en primer lugar intenta adquirir el mutex para luego, en caso de ser posible, realizar el disparo de la transición, y por último, el método *getMutex()* que retorna el mutex.

La clase **Politica** es la superclase de *Politica1* y *Politica2*. Contiene el método *cual(transiciones)*, que devuelve solo una de todas las transiciones que cumplen estar




sensibilizadas y tener un hilo esperando en ellas; la elección de cuál transición devolver (de todas las que cumplen las condiciones) la realiza a través del método *getRandomElement(list)* que selecciona una de ellas de manera aleatoria.

La clase **Politica1** extiende de la clase *Politica*, su constructor recibe una matriz con los conflictos de la red, para poder tomar la elección de cuál transición disparar según lo que corresponda, y crea un array con las flags para cada conflicto. Contiene el método *cual(transiciones)* que sobrescribe el mismo de la superclase, y que en caso de haber conflicto efectivo, va alternando las flags para que las transiciones se disparen por partes iguales.

La clase **Politica2** extiende de la clase *Politica*, su constructor recibe una matriz con los conflictos de la red, para poder tomar la elección de cuál transición disparar según lo que corresponda, y crea un array con las flags para cada conflicto. Contiene el método *cual(transiciones)* que sobrescribe el mismo de la superclase, y que en caso de haber conflicto efectivo, para las transiciones T9 y T10 aplica la política del 80% de la carga sobre el lado izquierdo utilizando el método *definirProbabilidad()*, que lo realiza de manera aleatoria, y para el resto de conflictos utiliza la *Politica1*.

Clase **RdP**: su constructor recibe como parámetro las plazas de salida y de entrada de las transiciones, marcado inicial, invariantes de plaza y de transición, tiempos de cada transición y cantidad máxima de invariantes. Se utiliza *ReentrantReadWriteLock()* para proteger la variable *apagar*, la cual puede ser leída o modificada usando varios métodos e indica si se ha alcanzado la cantidad de invariantes solicitada por la consigna, entonces al implementar el lock puede ser leída siempre y cuando no esté siendo escrita. Sus métodos son: *disparar(transicion)* el cual devuelve true o false según si se puede realizar el disparo de la transición (puede ser con o sin tiempo) determinada como parámetro, y luego actualiza el marcado; al actualizar el marcado, se tienen dos métodos para ir verificando el funcionamiento de la red: *verificarInvariantePlaza()* que verifica que se vayan cumpliendo para todos los marcados de la red los invariantes de plaza (lanza una excepción en caso de que no ocurra esto), y *verificarInvarianteTransicion(transicion)* que verifica si se disparó T14 para que con ello aumente en uno el contador *cuenta_invariantes* (que luego se utiliza para saber si llegamos al disparo de los 200 invariantes requeridos). Además se tiene *getSensibilizadas()* que devuelve el vector de sensibilizadas que indica el estado (si está sensibilizada o no) de cada transicion, *isApagada()* devuelve el valor de la variable *apagar*, *setApagar()* la cual utiliza *writeLock()* para poder setear la variable *apagar* en true de manera segura, *getMarcadoActual()*, *getCantidadTransiciones()*, *getCuentaInvariantes()*, *getContadores()*, los cuales realizan la acción indicada por su denominación.




Clase **SensibilizadoConTiempo**: su constructor recibe como parámetro una matriz con el tiempo asignado a cada transición. Sus métodos son: *setTimeStamp(transicion)* que setea el tiempo en la cual la transición es sensibilizada por tokens, *isInmediata(transicion)* que indica si la transición enviada como parámetro es inmediata o no, *testVentana(transicion)* que indica si el hilo ha llegado dentro de la ventana de tiempo [alfa,beta] o no, *isEsperando(transicion)* en el cual se ve si ya hay o no un hilo antes que este durmiendo en esa transicion, *resetEsperando(transicion)* en el cual setea esperando como false, *setEsperando(transicion)* en el cual setea a esperando como true, *antesDeLaVentana(transicion)* en donde se verifica si efectivamente se llega antes del valor de tiempo alfa o no, *getTiempoRestante(transicion)* devuelve el tiempo que falta dormir para poder llegar al tiempo alfa y con ello poder entrar a la ventana de tiempo.

La clase **VectorSensibilizadas** contiene los métodos *actualizarSensibilizadas()* que es llamado por el metodo disparar de la clase RdP en el momento que se ha podido disparar la transicion y se ha cambiado el marcado, entonces se debe actualizar este vector de transiciones y si alguna transicion cambio de estado (o sea pasó de estar sensibilizada a no estarlo, o viceversa) se setea su timestamp (el cual es el tiempo en el que la transicion se sensibilizo por tokens), *isSensibilizada(transicion)* verifica si la transicion esta sensibilizada por tokens y por tiempo, y *getSensibilizadas()* devuelve un vector que indica si las transiciones están sensibilizadas o no.

Resultados

Dentro del archivo log.txt, se pueden ver reflejadas las transiciones disparadas, este archivo es el analizado con la expresión regular. También se genera otro archivo de estado del sistema stateLog.txt, en él un hilo Daemon imprime cada cierto tiempo un resumen del estado del sistema (Monitor, Red de Petri, hilos, entre otros) a modo de registro.



```

[INFO ] 30-07-23 17:06:37 [Thread-14] TIME: 2507
[INFO ] 30-07-23 17:06:37 [Thread-14] MARCADO: [1, 0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 85, 1, 0, 0, 0, 1, 0]
[INFO ] 30-07-23 17:06:37 [Thread-14] CONT TRANS: [184, 92, 91, 91, 90, 88, 91, 87, 90, 71, 21, 70, 21, 91, 90]
[INFO ] 30-07-23 17:06:37 [Thread-14] CONT INV: 90
[INFO ] 30-07-23 17:06:37 [Thread-14] COLAS: [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0]
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-0 TIMED_WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-1 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-2 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-3 TIMED_WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-4 TIMED_WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-5 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-6 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-7 RUNNABLE
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-8 RUNNABLE
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-9 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-10 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-11 TIMED_WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-12 WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Thread-13 TIMED_WAITING
[INFO ] 30-07-23 17:06:37 [Thread-14] THREAD: Alive: 14 , Running: 2
[INFO ] 30-07-23 17:06:37 [Thread-14] EOM: -----

```

Figura 9: registro del estado del sistema.

En la imagen se observa un ejemplo del log de estado:

- TIME: Tiempo en ms que pasó desde el inicio del programa.
- MARCADO: Marcado actual de la red de petri.
- CONT TRANS: Cuantas veces se disparó cada transición.
- CONT INV: Cuantos invariantes se ejecutaron.
- COLAS: Cuantos hilos hay esperando en cada cola.
- THREAD: Estado de cada hilo que dispara una secuencia.

Se muestra un ejemplo de salida de consola de una ejecución del programa:


Política 1	Política 2
<pre> Verificando Invariantes... LOG VALIDATION SUCCESS Cuentas de Invariantes: 0. T0T1T3T5T7T10T12T13T14 = 33 1. T0T1T3T6T8T10T12T13T14 = 21 2. T0T2T4T6T8T10T12T13T14 = 32 3. T0T2T4T5T7T10T12T13T14 = 13 4. T0T1T3T5T7T9T11T13T14 = 33 5. T0T1T3T6T8T9T11T13T14 = 16 6. T0T2T4T5T7T9T11T13T14 = 20 7. T0T2T4T6T8T9T11T13T14 = 32 TOTAL = 200 </pre>	<pre> Verificando Invariantes... LOG VALIDATION SUCCESS Cuentas de Invariantes: 0. T0T1T3T5T7T10T12T13T14 = 9 1. T0T1T3T6T8T10T12T13T14 = 9 2. T0T2T4T6T8T10T12T13T14 = 8 3. T0T2T4T5T7T10T12T13T14 = 6 4. T0T1T3T5T7T9T11T13T14 = 40 5. T0T1T3T6T8T9T11T13T14 = 41 6. T0T2T4T5T7T9T11T13T14 = 45 7. T0T2T4T6T8T9T11T13T14 = 42 TOTAL = 200 </pre>

Figura 8: salidas de ejecuciones con las diferentes políticas.

Se observa que se realiza la validación del log de los invariantes de transición y que posteriormente se cuenta la cantidad de veces que se ejecutó cada invariante.

En el primer caso de la política 1, se observa que cada invariante se ejecuta aproximadamente la misma cantidad de veces.

En el caso de la política 2, se observa que los primeros 4 invariantes que incluyen las transiciones T10-T12, se cumplieron menor cantidad de veces que los demás. Esto se debe a que en la política 2 se prioriza la rama T9-T11 el 80% de las veces. Comparando la cantidad de invariantes que se ejecutaron que contienen las transiciones T9-T11 (168, 84%) con las que contienen T10-T12 (32 veces, 16%), se observa que se cumple aproximadamente la relación 80-20 con respecto al total de invariante ejecutados (200). La distribución de los 4 primeros invariantes es uniforme respecto de ese 20%.



Si se aumenta la cantidad de invariantes a ejecutar, los porcentajes resultantes se acercan más al valor esperado.

Análisis de tiempos

Tiempo considerando una secuencialización del programa

Si el sistema fuera secuencializado completamente, es decir, que un hilo se encargue de ejecutar todas las transiciones necesarias para el procesamiento de 200 imágenes, se puede llegar a el tiempo aproximado que demoraría su ejecución.

En total se obtiene un tiempo teórico aproximado de:

$$\begin{aligned} \text{tiempo teórico} &= [(T_0) + (T_1 + T_3) + (T_5 + T_7) + (T_9 + T_{11}) + (T_{13} + T_{14})] * \text{Imágenes} \\ \text{tiempo teórico} &= (10 + (0 + 20) + (0 + 20) + (0 + 20) + (0 + 20)) \times 200 = 18[s] \end{aligned}$$

Este cálculo temporal no tiene en cuenta el tiempo agregado en el resultado total de las ejecuciones, ya que al estar secuencializada la toma de decisiones dentro del monitor, indefectiblemente hay un tiempo de handling que se produce por el paso de los hilos dentro del mismo.

Para este caso, con el sistema completamente secuencializado, se observa el siguiente tiempo de ejecución (Tiempo Total) y tiempo de handling:

```
Secuencia 0      FINALIZO, disparo: 1800      Transiciones y 200  Secuencias, observando un Tiempo promedio con Handling: 124[ms]
Tiempo Total: 24998 ms
```

$$\text{tiempo de handling} = (\text{tiempo total} - \text{tiempo})/\text{secuencia}$$

$$\text{tiempo de handling} \simeq (25 - 18)/200 \simeq 0,035 [s]$$

Tiempo considerando una semi secuencialización del programa

Otro caso a considerar es el semi secuencializado de los procesos, en el que se considera el promedio de tiempo que tarda en procesar una imagen en cierto proceso,

teniendo en cuenta el paralelismo en los procesos. Se puede calcular un tiempo aproximado de ejecución:

$$\text{tiempo teórico} = [(T0) + \frac{(T3|T4)}{2} + \frac{(T7|T8)}{2} + (T11) + (T14)] * \text{Imágenes}$$

$$\text{tiempo teórico} = (10 + \frac{20}{2} + \frac{20}{2} + 20 + 20) \times 200 = 14 [s]$$

Para este caso, con el sistema semi secuencializado, se observa el siguiente tiempo de ejecución (Tiempo Total) y tiempo de handling:

```
Secuencia 0      FINALIZO, disparo: 1800      Transiciones y 200      Secuencias, observando un Tiempo promedio con Handling: 93[ms]
Tiempo Total: 18796 ms
```

$$\text{tiempo de handling} = (\text{tiempo total} - \text{tiempo})/\text{secuencia}$$

$$\text{tiempo de handling} \simeq (19 - 14)/200 \simeq 0,025[s]$$

Tiempo considerando la concurrencia del programa

En los casos anteriores no se tuvo en cuenta la concurrencia del programa, ya que para que una imagen sea procesada y le dé lugar a la próxima imagen, la primera debe terminar todo su proceso.

Permitiendo la concurrencia se observa que se puede tener imágenes en diferentes etapas en simultáneo, y esto se vería reflejado en los tiempos de ejecución del programa. Este tiempo se debe a que cuando se está en un estado de máxima concurrencia, es decir, se tiene una imagen en cada etapa del proceso, se obtiene una imagen a la salida cada 20 [ms]. Este tiempo considera que sin importar que haya imágenes en los procesos de forma simultánea, no van a poder ser exportadas antes de ese tiempo que dura el proceso de exportación.

$$\text{tiempo teórico} = (T14) * \text{Imágenes}$$

$$\text{tiempo teórico} = (20) [ms] * 200 = 4 [s]$$

Considerando que este caso planteado es el ideal, será tomado como una cota mínima temporal. Por la razón de que este no tiene en cuenta el tiempo de handling, que sucede dentro del monitor y el tiempo que demora cuando la red no está en un estado

ideal, es decir, una imagen simultánea por proceso.

Para este caso, con el sistema en concurrencia, se observa el siguiente tiempo de ejecución (Tiempo Total) y tiempo de handling para las secuencias:

```
Secuencia 0    FINALIZO, disparo: 200  Transiciones y 200  Secuencias, observando un Tiempo promedio con Handling: 15[ms]
Secuencia 13   FINALIZO, disparo: 400  Transiciones y 200  Secuencias, observando un Tiempo promedio con Handling: 31[ms]
Secuencia 7    FINALIZO, disparo: 102  Transiciones y 102  Secuencias, observando un Tiempo promedio con Handling: 61[ms]
Secuencia 4    FINALIZO, disparo: 101  Transiciones y 101  Secuencias, observando un Tiempo promedio con Handling: 62[ms]
Secuencia 9    FINALIZO, disparo: 155  Transiciones y 155  Secuencias, observando un Tiempo promedio con Handling: 40[ms]
Secuencia 11   FINALIZO, disparo: 155  Transiciones y 155  Secuencias, observando un Tiempo promedio con Handling: 40[ms]
Secuencia 10   FINALIZO, disparo: 45   Transiciones y 45   Secuencias, observando un Tiempo promedio con Handling: 139[ms]
Secuencia 3    FINALIZO, disparo: 99   Transiciones y 99   Secuencias, observando un Tiempo promedio con Handling: 63[ms]
Secuencia 5    FINALIZO, disparo: 102  Transiciones y 102  Secuencias, observando un Tiempo promedio con Handling: 61[ms]
Secuencia 1    FINALIZO, disparo: 99   Transiciones y 99   Secuencias, observando un Tiempo promedio con Handling: 63[ms]
Secuencia 12   FINALIZO, disparo: 45   Transiciones y 45   Secuencias, observando un Tiempo promedio con Handling: 139[ms]
Secuencia 8    FINALIZO, disparo: 98   Transiciones y 98   Secuencias, observando un Tiempo promedio con Handling: 64[ms]
Secuencia 2    FINALIZO, disparo: 101  Transiciones y 101  Secuencias, observando un Tiempo promedio con Handling: 62[ms]
Secuencia 6    FINALIZO, disparo: 98   Transiciones y 98   Secuencias, observando un Tiempo promedio con Handling: 64[ms]

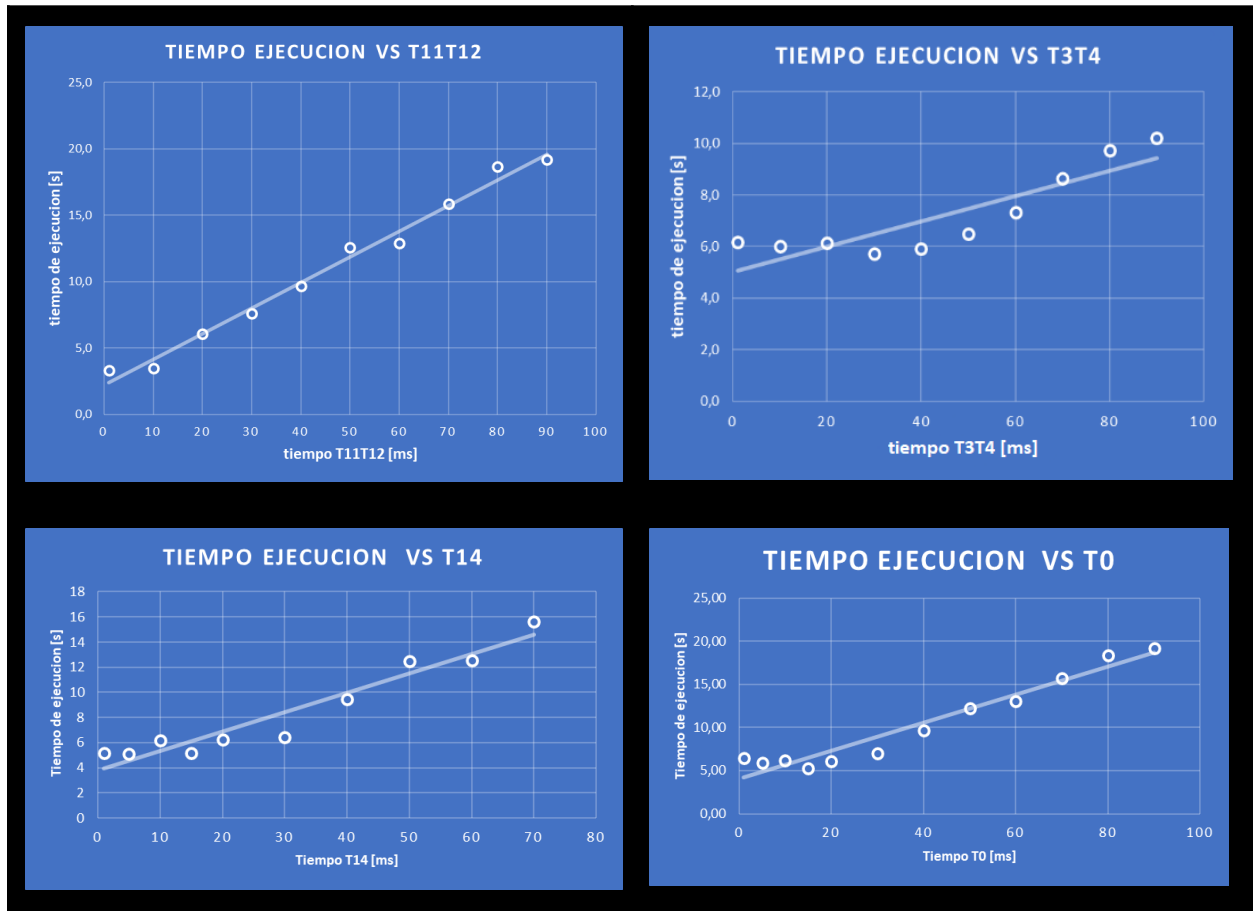
Tiempo Total: 6307 ms
```

$$\text{tiempo de handling} = (\text{tiempo total} - \text{tiempo})/\text{secuencia}$$

$$\text{tiempo de handling} \simeq (6,3 - 4)/200 \simeq 0,0115[s]$$

Tanto en el caso secuencializado, semi secuencializado y en el concurrente se nota una diferencia entre los mismos, llegando a advertir que el tiempo de ejecución va disminuyendo. También se observa que el tiempo de handling va disminuyendo, debido a que se realiza un mejor uso del monitor para la sincronización de disparos, y esto se debe a que se realizan los disparos más eficientemente y por ende más fluidos.

Se hace un análisis de los tiempos de ejecución variando el tiempo de algunas transiciones determinadas para observar su influencia sobre el tiempo total:



Se puede observar que las transiciones más críticas para los tiempos elegidos son las transiciones T14 y T11-T12, esto se debe a que estas son ejecutadas por un solo hilo, formando un cuello de botella. La transición T0 no es limitante a pesar de que es ejecutada por un solo hilo ya que el tiempo de ejecución está limitado aún más por las transiciones siguientes. El conjunto T3-T4 está a cargo de dos hilos por lo que tampoco afectan significativamente al tiempo total.



Conclusiones

A partir de los resultados obtenidos podemos concluir que el sistema trabaja de manera concurrente y procesa las imágenes eficientemente gracias a ello.

En la determinación de la cantidad de hilos se tomó la decisión de agregar más de los que recomendaba el paper de estudio de las responsabilidades de hilos, con el fin de cumplir con el requerimiento de la política 80% - 20%. En caso de que no fuese requerido implementar esta política sería innecesario agregar hilos extra ya que se desperdiciarán recursos.

En cuanto a la implementación de las clases vivas, se optó por implementar la interfaz Runnable de Java frente a extender de la clase Thread, ya que con la interfaz Runnable se cubrían las necesidades implementando el método run.

Se advierte que en la implementación del monitor se expone públicamente el mutex de entrada, esto transgrede la responsabilidad asignada al monitor como controlador de la concurrencia y podría traer problemas si su uso es inadecuado. Como mejora a futuro se propone que la sección de mandar a dormir a los hilos (liberando y recuperando el mutex del monitor correspondientemente), sea efectuada por el monitor y no por la clase vectorSensibilizadas.

Con la elaboración de este trabajo práctico se pudo comprender la implementación y funcionamiento de un monitor completo, mecanismo de alto nivel de abstracción para el manejo, control y sincronización de eventos concurrentes, reconociendo que sus beneficios están ligados también a una mayor complejidad en comparación con otras herramientas de menor nivel de abstracción.

Con lo que se afirma que el uso de Redes de Petri evidencia ser muy útil a la hora de modelar, representar y analizar problemas de programación concurrente, dando una forma de resolución a las dificultades halladas con un fundamento matemático.



Anexo

Dentro de este Trabajo Práctico se entrega:

- Un archivo de imagen con el [diagrama de clases](#).
- Un archivo de imagen con el [diagrama de secuencias](#).
- El código fuente de la resolución del enunciado. Incluyendo las librerías y extensiones necesarias.
- Un archivo log.txt con una breve descripción de los datos mostrados.
- El presente informe en formato pdf, de estructura formal.