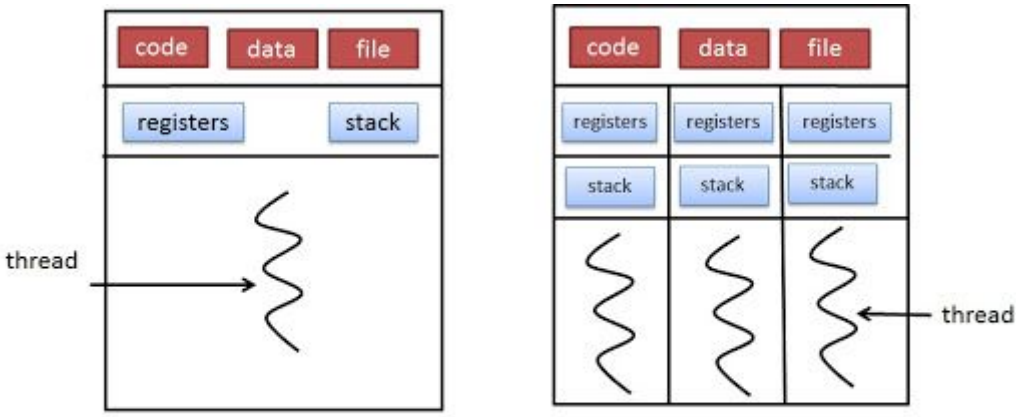
UNC Universidad Nacional de Córdoba

FCEFyN Facultad de Ciencias Exactas Físicas y Naturales



Single-threaded Process Multithreaded Process

Programación Concurrente

Trabajo Práctico de Laboratorio Nº2
Modelo de Procesamiento y Almacenamiento con Redes de Petri

Docentes:

- Dr. Ing. Orlando Micolini
- Ing. Luis Orlando Ventre
- Ing. Mauricio Ludemann

Autores:

Matrícula	Apellido y Nombre	Carrera
41279796	Bonino, Francisco Ignacio	Ing. en Comp.
40502859	Coronati, Federico Joaquín	Ing. en Comp.
39129465	Luna, Lihué Leandro	Ing. en Comp.
41232347	Merino, Mateo	Ing. en Comp.

Grupo Code Busters

Año 2020

Índice

Objetivo

Plantear y analizar un modelo matemático de un sistema de procesamiento y almacenamiento de tareas en memorias de 8 direcciones. Analizar las propiedades de la red de Petri que modela el sistema y verificar si existen problemas de concurrencia; de ser así, proponer una solución a los mismos.

Luego, modelar la red de Petri y un monitor de concurrencia para administrar la ejecución del programa concurrente de acuerdo al modelo matemático previamente establecido. Añadir un archivo Log para llevar un registro de la evolución de la red para luego corroborarlo.

Modelar lo mencionado con objetos en Java y presentar diagramas de clase y de secuencia que esclarezcan el código Java.

Red de Petri del Sistema

La red de Petri que se proporciona en un principio es la que se muestra a continuación:

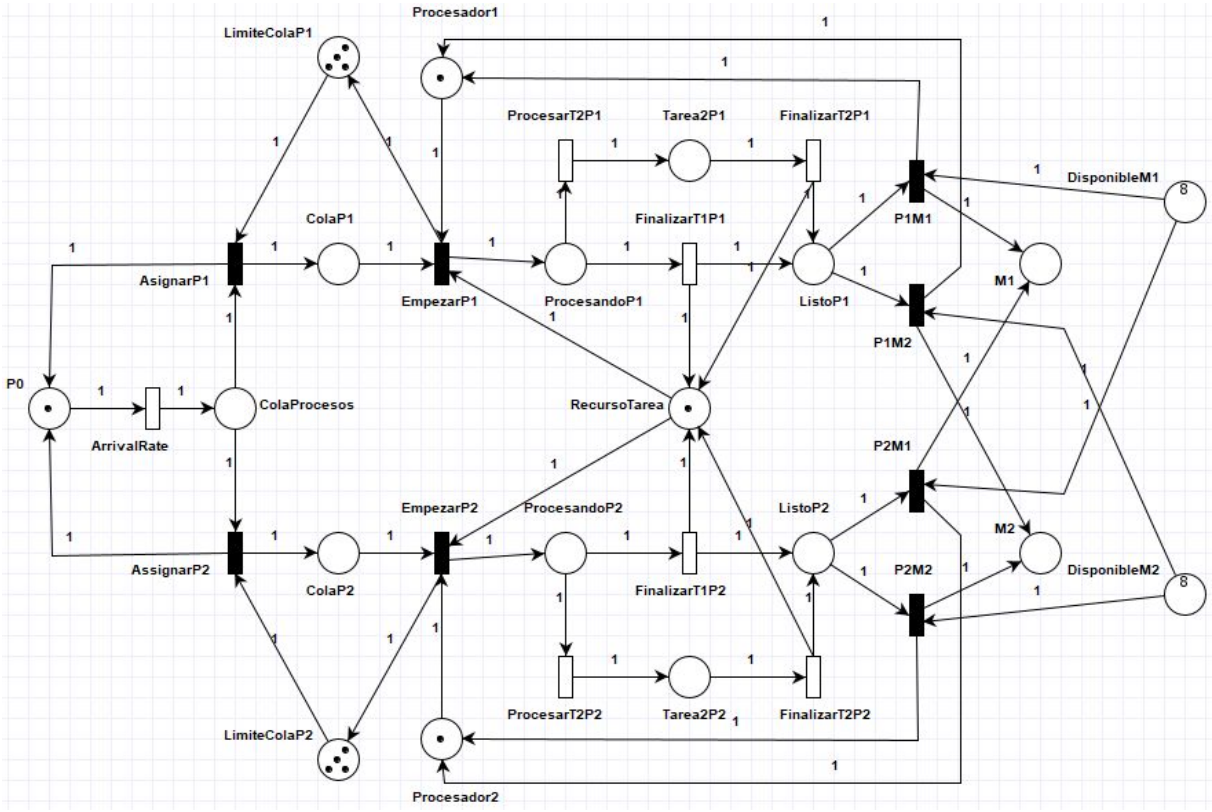


Figura 1: Red de Petri proporcionada en el enunciado

Cabe aclarar que la Red de Petri dada en el enunciado del trabajo posee **puntos muertos** (o "**deadlocks**"), lo que significa que luego de disparar ciertas transiciones, la red alcanza un estado final en el que ya no se puede avanzar más, porque habrá transiciones que quedan esperando tokens que nunca llegan. Para evitar tener estos puntos muertos se procede a realizar algunas modificaciones en la red.

Para trabajar en el análisis de la red, se emplea el software **Platform Independent Petri Net Editor (PIPE),** el cual nos permite modelar la red de Petri suministrada y simular la evolución de sus transiciones, además de poder calcular automáticamente las propiedades como invariantes y su clasificación, que se detallan más adelante.

Modificaciones en la red

Haciendo uso de PIPE se determina que la red está acotada, es decir, que hay una cantidad de tokens constante (no se crean ni se pierden tokens). También se indica que no es segura y que tiene un deadlock:

Bounded	true
Safe	false
Deadlock	true

Las plazas M1 y M2 eran las causantes de dicho deadlock. Cuando los tokens de las plazas DisponibleM1 y DisponibleM2 se entregan a las plazas M1 y M2, representando un recurso de disponibilidad de memoria, llegamos a un punto en el que estos recursos se vacían completamente, quedando 8 tokens en cada una de las plazas M1 y M2 pero nunca se vacían las memorias y, por consiguiente, no son devueltos los tokens de disponibilidad de memoria. De esta forma, al haber consumido todos los recursos, quedaba muerta toda la red de Petri, es decir, **sin posibilidad de evolucionar**.

Se decidió que lo más óptimo sería vaciar las memorias inmediatamente. Esto se logra agregando transiciones temporizadas de la forma:

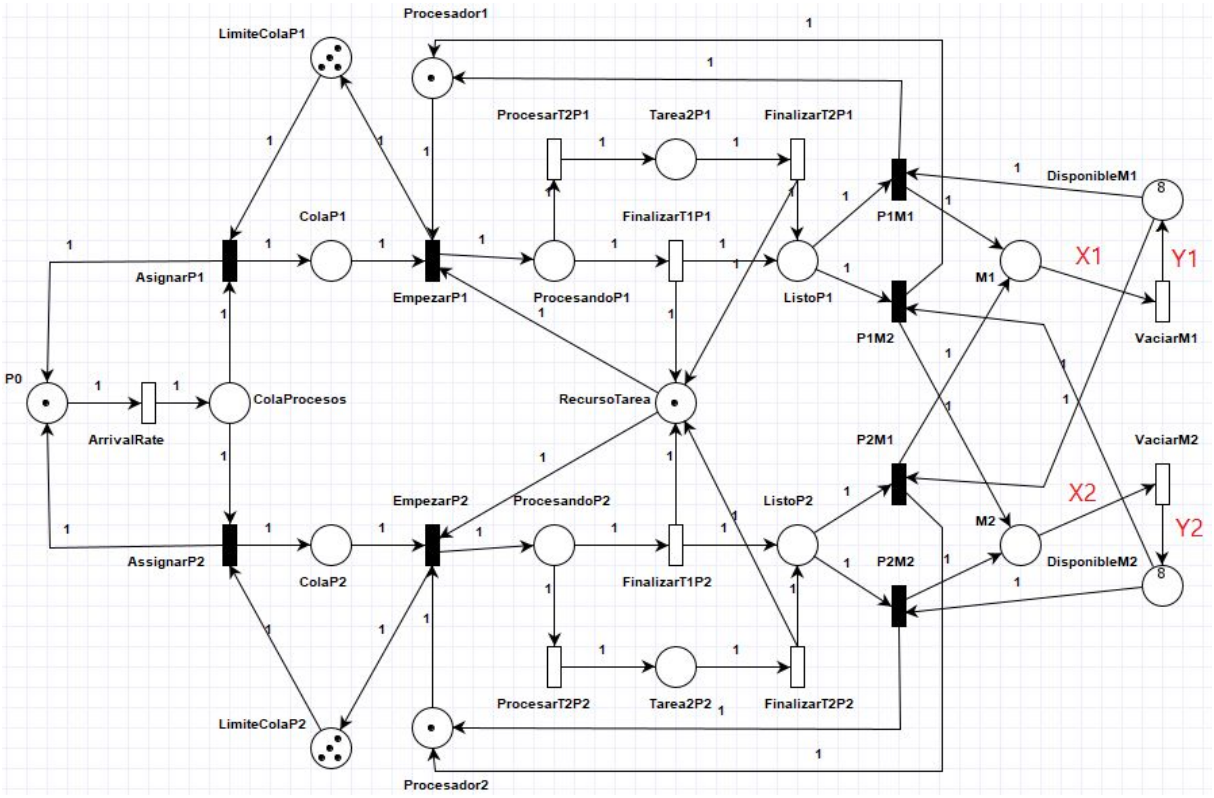


Figura 2: Red de Petri con transiciones de vaciado de memorias

El equipo de trabajo decidió que lo mejor y más sencillo de implementar es que, ni bien aparezca un token en las plazas M1 o M2, se sensibilicen sus transiciones de salida *VaciarM1* y *VaciarM2* respectivamente y poder vaciar las memorias de inmediato. Con esta modificación final, la red queda de la forma que se muestra en la figura 3, salvándose del deadlock.

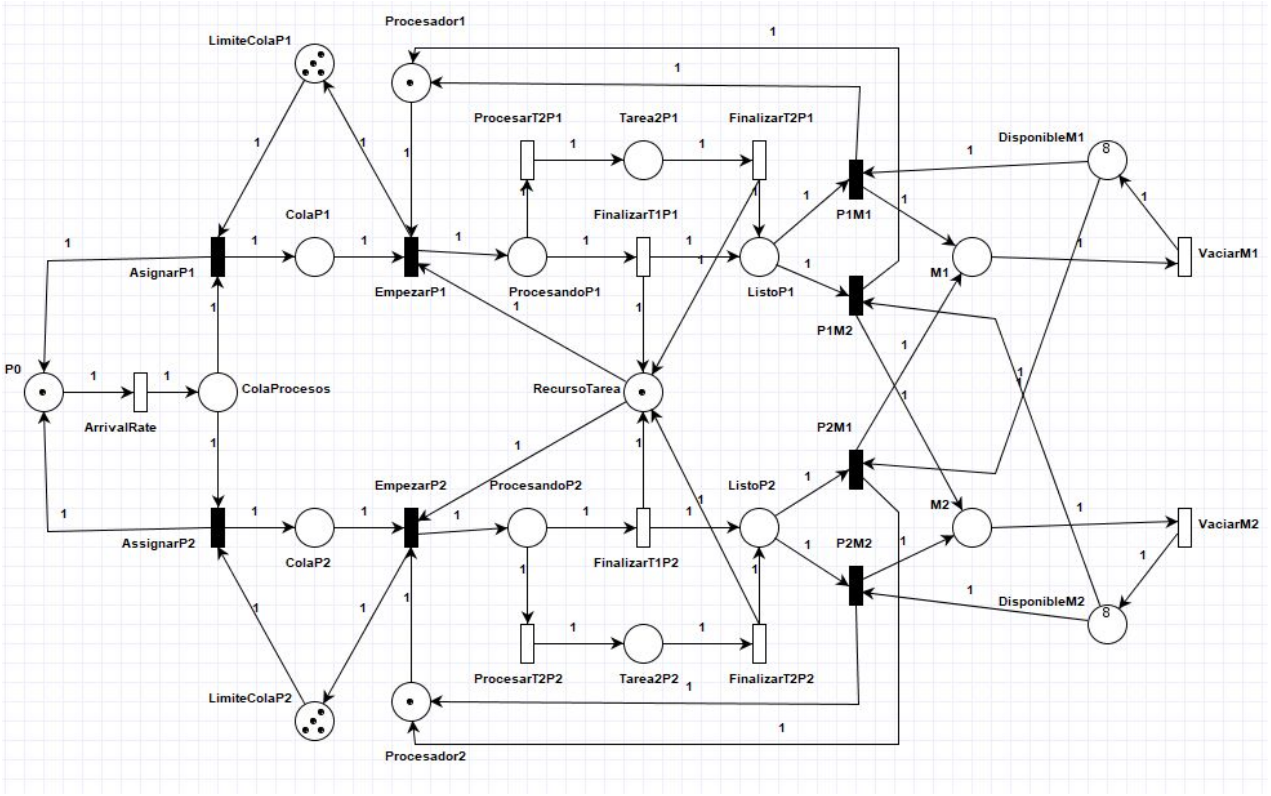


Figura 3: Red de Petri con las transiciones de vaciado de memorias funcionando

El hecho de modificar la red agregando arcos de peso 1 de entrada/salida a las transiciones que vacían las memorias nos garantiza el explícito cumplimiento de los invariantes de transición, y de esta manera la red fluye “aprovechando” sus propiedades.

Una vez solucionado el punto muerto o deadlock, se realizan nuevamente los análisis para esta red modificada.

Clasificación de la red

Petri net classification results

State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	false

Matrices Características de la Red

Matriz de incidencia de salida (I^+)

Forwards incidence matrix I^+

	ArrivalRate	AsignarP1	AssignarP2	EmpezarP1	EmpezarP2	FinalizarT1P1	FinalizarT1P2	FinalizarT2P1	FinalizarT2P2	P1M1	P1M2	P2M1	P2M2	ProcesarT2P1	ProcesarT2P2	VaciarM1	VaciarM2
ColaP1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ColaP2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ColaProcesos	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DisponibleM1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
DisponibleM2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
LimiteColaP1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
LimiteColaP2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ListoP1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
ListoP2	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
M1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
M2	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
P0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Procesador1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
Procesador2	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
ProcesandoP1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
ProcesandoP2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
RecursoTarea	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
Tarea2P1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Tarea2P2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Matriz de incidencia de entrada (I^-)

Backwards incidence matrix I^-

	ArrivalRate	AsignarP1	AssignarP2	EmpezarP1	EmpezarP2	FinalizarT1P1	FinalizarT1P2	FinalizarT2P1	FinalizarT2P2	P1M1	P1M2	P2M1	P2M2	ProcesarT2P1	ProcesarT2P2	VaciarM1	VaciarM2
ColaP1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
ColaP2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ColaProcesos	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DisponibleM1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
DisponibleM2	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0
LimiteColaP1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LimiteColaP2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ListoP1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
ListoP2	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
M1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
M2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
P0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Procesador1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Procesador2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ProcesandoP1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
ProcesandoP2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
RecursoTarea	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Tarea2P1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Tarea2P2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

Matriz de flujo ($I = I^+ - I^-$)

Combined incidence matrix I

	ArrivalRate	AsignarP1	AssignarP2	EmpezarP1	EmpezarP2	FinalizarT1P1	FinalizarT1P2	FinalizarT2P1	FinalizarT2P2	P1M1	P1M2	P2M1	P2M2	ProcesarT2P1	ProcesarT2P2	VaciarM1	VaciarM2
ColaP1	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
ColaP2	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
ColaProcesos	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DisponibleM1	0	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	1	0
DisponibleM2	0	0	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	1
LimiteColaP1	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
LimiteColaP2	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ListoP1	0	0	0	0	0	1	0	1	0	-1	-1	0	0	0	0	0	0
ListoP2	0	0	0	0	0	0	1	0	1	0	0	-1	-1	0	0	0	0
M1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	-1	0
M2	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	-1
P0	-1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Procesador1	0	0	0	-1	0	0	0	0	0	1	1	0	0	0	0	0	0
Procesador2	0	0	0	0	-1	0	0	0	0	0	0	1	1	0	0	0	0
ProcesandoP1	0	0	0	1	0	-1	0	0	0	0	0	0	0	-1	0	0	0
ProcesandoP2	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	-1	0	0
RecursoTarea	0	0	0	-1	-1	1	1	1	1	0	0	0	0	0	0	0	0
Tarea2P1	0	0	0	0	0	0	0	-1	0	0	0	0	0	1	0	0	0
Tarea2P2	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	1	0	0

Marcado inicial

$\mu_0 = (0,0,0,8,8,4,4,0,0,0,0,1,1,1,0,0,1,0,0)$

Transiciones sensibilizadas en el estado inicial: ArrivalRate

Análisis de los Invariantes

Invariantes de plaza

Por definición los invariantes de plaza son propiedades estructurales de la red que indican un conjunto de plazas en las cuales la cantidad de tokens permanece constante. En nuestra red, esos invariantes son los siguientes:

IP1) M(ColaProcesos) + M(P0) = 1

IP2) M(ListoP1) + M(Procesador1) + M(ProcesandoP1) + M(Tarea2P1) = 1

IP3) M(ListoP2) + M(Procesador2) + M(ProcesandoP2) + M(Tarea2P2) = 1

IP4) M(ProcesandoP1) + M(ProcesandoP2) + M(RecursoTarea) + M(Tarea2P1) + M(Tarea2P2) = 1

IP5) M(ColaP1) + M(LimiteColaP1) = 4

IP6) M(ColaP2) + M(LimiteColaP2) = 4

IP7) M(DisponibleM1) + M(M1) = 8

IP8) M(DisponibleM2) + M(M2) = 8

P-Invariants

ColaP1	ColaP2	ColaProcesos	DisponibleM1	DisponibleM2	LimiteColaP1	LimiteColaP2	ListoP1	ListoP2	M1	M2	P0	Procesador1	Procesador2	ProcesandoP1	ProcesandoP2	RecursoTarea	Tarea2P1	Tarea2P2
0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1

Interpretación de los Invariantes de Plaza

IP1: Se asignan procesos a las colas P1 o P2. Este invariante de plaza es interpretado de tal manera que, las tareas se asignan “de a una”. La red está estructuralmente organizada para que cada determinado tiempo se pueda asignar una tarea (disparando “AsignarP1” o “AsignarP2”). Este conflicto *efectivo* debe ser resuelto con alguna política determinada, que detallaremos más adelante. En cualquier marcado de la red, la suma de tokens M(ColaProcesos) + M(P0) = 1.

IP2: Este invariante de plaza representa que, si se toma el *RecursoTarea* y *Procesador1,* y se accede a la plaza “ProcesandoP1”, se va a poder realizar la tarea 2 ó la tarea 1, es decir que no se pueden ejecutar de forma concurrente. El recurso *Procesador1* se toma para “decidir” qué tarea realizar, y luego se devuelve junto con el *RecursoTarea* para notificar que el procesador se liberó y que se puede realizar una nueva tarea.

IP3: Análogo al caso anterior pero con Procesador 2.

IP4: Este invariante de plaza indica que el *RecursoTarea* puede tomarse para trabajar con el procesador 1 ó con el procesador 2, es decir que no se puede trabajar con los dos al mismo tiempo, por lo tanto, representa la **exclusión mutua** entre dichos procesadores.

IP5: Este invariante representa un límite estructural que condiciona la cantidad máxima de hilos que pueden estar esperando para utilizar el recurso Procesador1. Si la plaza *ColaP1* llega a su límite de tokens (4), ya no se encolarán más tareas hasta desocupar dicha cola.

IP6: Análogo al caso anterior pero con ColaP2.

IP7: Representa la disponibilidad física de lugares en la memoria M1, siendo que *DisponibleM1* contiene los 8 lugares disponibles de memoria en el estado inicial de la red, es decir, la **"memoria vacía"**. A medida que la red evoluciona se van ocupando y liberando estos 8 lugares (que constituyen las plazas *DisponibleM1 y M1)* disparando las transiciones correspondientes, que representan la escritura y el vaciado de la memoria.

IP8: Análogo al caso anterior pero con la memoria M2.

Verificación Matemática de los Invariantes de Plaza

La verificación matemática de los invariantes de plaza se puede realizar de manera sencilla a partir de la matriz de incidencia ***I***, teniendo en cuenta que si se extraen las filas correspondientes a las plazas involucradas en un invariante de plaza, y se hace la suma vectorial de estas filas, el resultado debería ser un vector fila cuyos elementos son todos ceros, ya que implicaría disparar **todas las transiciones adyacentes** a las plazas seleccionadas y el resultado implica que la cantidad de tokens en cada plaza **se mantiene invariante**.

Como ejemplo se detalla el invariante IP5 a continuación.

	ArrivalRate	AsignarP1	AssignarP2	EmpezarP1	EmpezarP2	FinalizarT1P1	FinalizarT1P2	FinalizarT2P1	FinalizarT2P2	P1M1	P1M2	P2M1	P2M2	ProcesarT2P1	ProcesarT2P2	VaciarM1	VaciarM2
ColaP1	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
ColaP2	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
ColaProcesos	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DisponibleM1	0	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	8	0
DisponibleM2	0	0	0	0	0	0	0	0	0	0	-1	0	-1	0	0	0	8
LimiteColaP1	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
LimiteColaP2	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
ListoP1	0	0	0	0	0	1	0	1	0	-1	-1	0	0	0	0	0	0
ListoP2	0	0	0	0	0	0	1	0	1	0	0	-1	-1	0	0	0	0
M1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	-8	0
M2	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	-8
P0	-1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Procesador1	0	0	0	-1	0	0	0	0	0	1	1	0	0	0	0	0	0
Procesador2	0	0	0	0	-1	0	0	0	0	0	0	1	1	0	0	0	0
ProcesandoP1	0	0	0	1	0	-1	0	0	0	0	0	0	0	-1	0	0	0
ProcesandoP2	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	-1	0	0
RecursoTarea	0	0	0	-1	-1	1	1	1	1	0	0	0	0	0	0	0	0
Tarea2P1	0	0	0	0	0	0	0	-1	0	0	0	0	0	1	0	0	0
Tarea2P2	0	0	0	0	0	0	0	0	-1	0	0	0	0	0	1	0	0

En la tabla se pueden apreciar las filas "ColaP1" y "LimiteColaP1" destacadas en color rojo, que son las plazas asociadas al invariante IP5. Al sumar estas filas de forma vectorial, se realizan cambios en el marcado gracias a las transiciones adyacentes a estas plazas, que se indican en la tabla con rectángulos amarillos ("AsignarP1" y "EmpezarP1"). Se verifica que al sumar estos vectores el resultante es un **vector nulo**, indicando que no habrá variaciones en la cantidad de tokens de las plazas en cuestión.

Invariantes de transición

Por definición, los invariantes de transición son propiedades estructurales de la red que indican una secuencia de transiciones que luego de ser disparadas da como resultado el marcado que se tenía antes de disparar la secuencia, lo que nos asegura la vivacidad de la red en esa secuencia. En este caso, los invariantes comienzan y terminan en el marcado inicial $\mathbf{m_i}$. A continuación se listan los invariantes de transición de la red. Identificamos a los invariantes de transición como IT1, IT2, y así sucesivamente.

T-Invariants

ArrivalRate	AsignarP1	AssignarP2	EmpezarP1	EmpezarP2	FinalizarT1P1	FinalizarT1P2	FinalizarT2P1	FinalizarT2P2	P1M1	P1M2	P2M1	P2M2	ProcesarT2P1	ProcesarT2P2	VaciarM1	VaciarM2
1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	1	0
1	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	1
1	0	1	0	1	0	1	0	0	0	0	1	0	0	0	1	0
1	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	1
1	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0
1	1	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1
1	0	1	0	1	0	0	0	1	0	0	1	0	0	1	1	0
1	0	1	0	1	0	0	0	1	0	0	0	1	0	1	0	1

Interpretación de los Invariantes de Transición

IT1: Si disparamos el arribo de tareas, la asignación a la ColaP1, EmpezarP1, FinalizarT1P1, y almacenamos el resultado de la tarea 1 en M1, para luego vaciar la memoria M1 (solo vaciamos 1 dato), volvemos al marcado inicial de la red, evitando el deadlock que se tenía en la red inicialmente propuesta.

IT2: Análogo al caso anterior pero almacenando la tarea 1 del procesador 1 en la memoria M2.

IT3: Análogo al caso IT1 pero empleando al procesador 2, registrando en la memoria M1.

IT4: Análogo al caso IT2 pero empleando al procesador 2, registrando en la memoria M2.

IT5: Análogo al caso IT1 pero la tarea que se realiza es la tarea 2.

IT6: Análogo al caso IT2 pero la tarea que se realiza es la tarea 2.

IT7: Análogo al caso IT3 pero la tarea que se realiza es la tarea 2.

IT8: Análogo al caso IT4 pero la tarea que se realiza es la tarea 2.

Verificación Matemática de los Invariantes de Transición

Para verificar que los invariantes de transición son correctos, antes de empezar con el código del trabajo práctico, se realizó una clase externa al proyecto en código Java dedicada exclusivamente a verificar estas propiedades. El procedimiento llevado a cabo para verificar los invariantes de transición es el siguiente.

Teniendo en cuenta la ecuación de estado

$$m_{i+1} = m_i + I \cdot S$$

Donde m_{i+1} es el estado subsiguiente, m_i es el estado actual, **I** es la matriz de incidencia y **S** es el vector de disparo de transiciones.

Si hacemos corresponder **I** a la matriz de incidencia de la red de petri, y **S** al vector que corresponde al invariante de transición IT1, el producto matricial da como resultado un vector donde todos los elementos son ceros, de modo que el estado subsiguiente sea el mismo que el actual.

Es decir

$$Si\ (S = ITx) \Rightarrow I \cdot S = \overline{0},$$

Por lo tanto

$$m_{i+1} = m_i + \overline{0} = m_i$$

Entonces se verifica que ITx es un **invariante de transición** de la Red de Petri.

Diagrama de Clases

Antes de comenzar el código en Java del proyecto, se decidió comenzar por el **diagrama de clases en UML**, utilizando la herramienta **Astah UML** porque la misma permite exportar el código Java, ahorrandonos un poco de tiempo luego de terminar de organizar las clases. Es importante aclarar que el diagrama que se observa en la figura 4 es un primer boceto, planteado previo a la escritura del código y sufrirá modificaciones a lo largo del desarrollo del código, según las ideas que se van presentando a lo largo del trabajo.

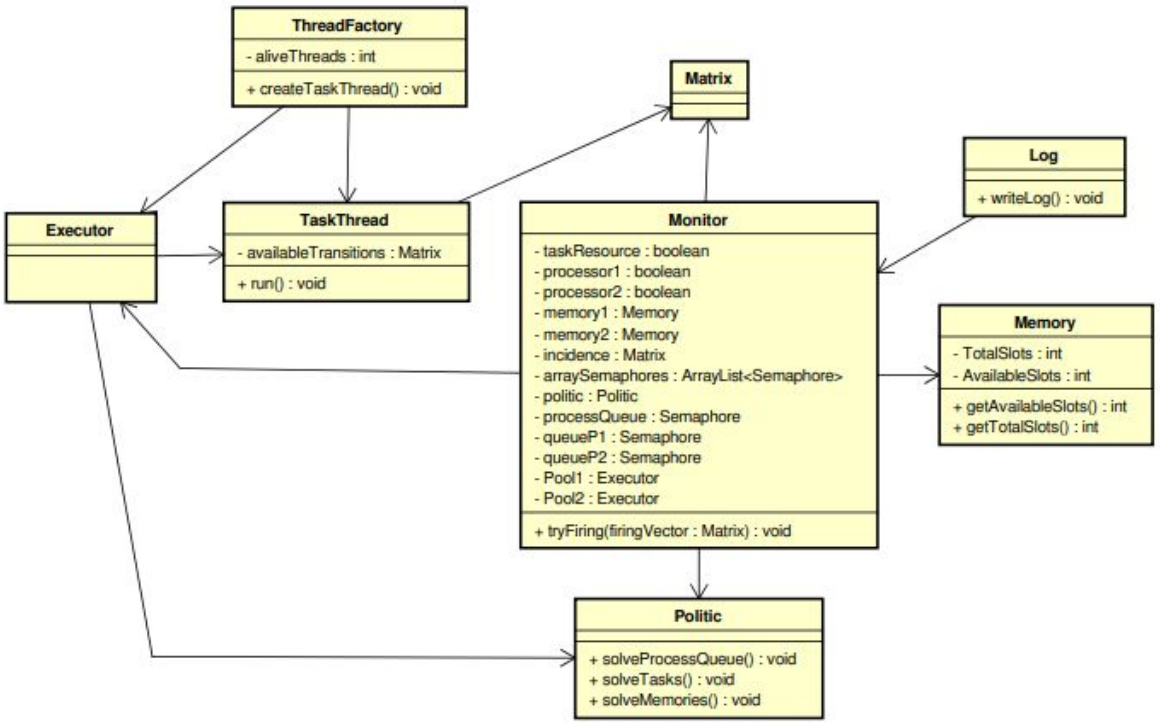


Figura 4: Diagrama de clases Inicial

Nota: La clase **Matrix** no forma parte del paquete de Java, ni tampoco es una clase del proyecto, sino que forma parte de un paquete extra llamado "**Jama**", que posee diferentes clases para realizar operaciones algebraicas, que se decidió usar en el proyecto por simplicidad a la hora de realizar los cálculos matriciales.

Luego de semanas de debate acerca del trabajo y tomando ciertos criterios para la creación de hilos y asignación de transiciones para cada uno de ellos, y además con un primer avance del código escrito, se realizó el siguiente diagrama de clases avanzado, que se muestra en la figura 5, que podría sufrir modificaciones pero sólo en detalles. Nótese que en el diagrama no se muestran todos los métodos del código sino que solamente aparecen aquellos que se consideraron de importancia para ilustrar el comportamiento general del proyecto y no aquellos que son de más bajo nivel para implementar en los algoritmos de resolución.

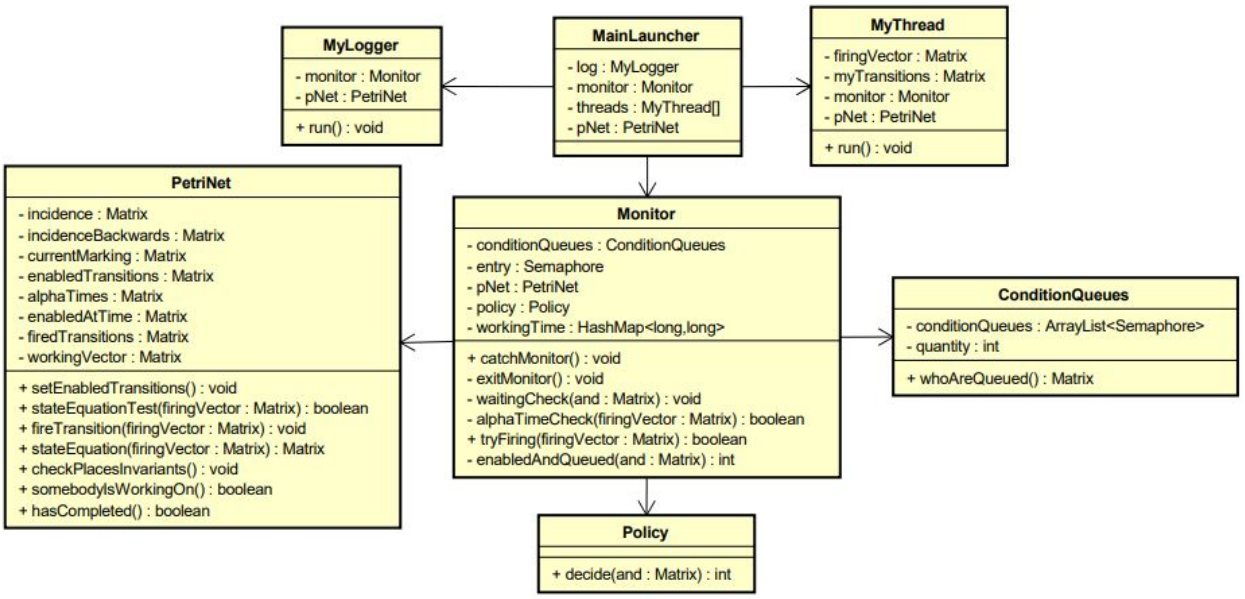


Figura 5: Diagrama de clases avanzado

Breve descripción de los métodos más importantes

Monitor → catchMonitor(): Un hilo toma la sección crítica del monitor y el resto se encola en la cola de entrada.

Monitor → exitMonitor(): Se libera la sección crítica para que otro hilo pueda acceder al monitor.

Monitor → tryFiring(): El método principal de la clase. Como su nombre indica, un hilo al llamar a este método booleano intenta disparar teniendo su vector de disparo, en donde el monitor evalúa sus condiciones y en base a eso, retorna *true* si el hilo pudo disparar su transición o *false* si debe esperar a que transcurra el tiempo alpha. Dentro del método se chequea, en caso de que el hilo deba esperar a una cola de condición, si ya hay alguien trabajando en su transición (en caso de ser temporizada), junto con otros cálculos. Es el método a través del cual el monitor administra la ejecución de la red de Petri.

Monitor → alphaTimeCheck(): Este método chequea el tiempo alpha de las transiciones temporizadas. Retorna *true* si el tiempo alpha para el disparo de la transición del hilo ya transcurrió, y *false* en caso contrario teniendo que ir a hacer un sleep o *tiempo de trabajo*.

Policy → decide(): En este método se decide qué transición será elegida para despertar a los hilos que tenga encolados. Para esto, se recorre el vector resultado de la operación 'AND' para ver cuántas transiciones hay con hilos esperando en sus colas de condición. Se almacena el índice de estas transiciones en un arreglo y se hace una elección aleatoria con distribución uniforme entre todos los índices que se hayan guardado.

PetriNet → setEnabledTransitions(): Este método recorre la matriz de incidencia 'backwards' o de salida, chequeando cada columna (transición) para ver si está sensibilizada (el peso de cada arco es menor o igual a la cantidad de tokens de la plaza).

PetriNet → fireTransition(): Este método se encarga de actualizar el estado de la red en general, considerando el nuevo vector de marcado y las transiciones sensibilizadas en base al mismo. Además, chequea los invariantes de plaza.

PetriNet → stateEquationTest(): Calcula la ecuación de estado, devolviendo *true* si el vector de firing da un resultado correcto para el disparo de una transición, y *false* en caso de no hacerlo.

PetriNet → checkPlacesInvariants(): En este método se chequea si se respetan los invariantes de plaza de la red. Para hacerlo, se itera en la matriz de invariantes de plaza comparándola con las plazas del marcado actual, verificando si la cantidad de tokens encontrados se corresponde con la cantidad de tokens que debería tener el invariante de plaza.

Diagrama de secuencia de disparo con política

En el diagrama de la figura 6, se muestra el disparo exitoso de una transición, en un momento en el que se encuentran varias transiciones sensibilizadas y sus hilos esperando en las respectivas colas de condición, por lo tanto es necesario establecer una política para seleccionar qué hilo debería disparar su transición.

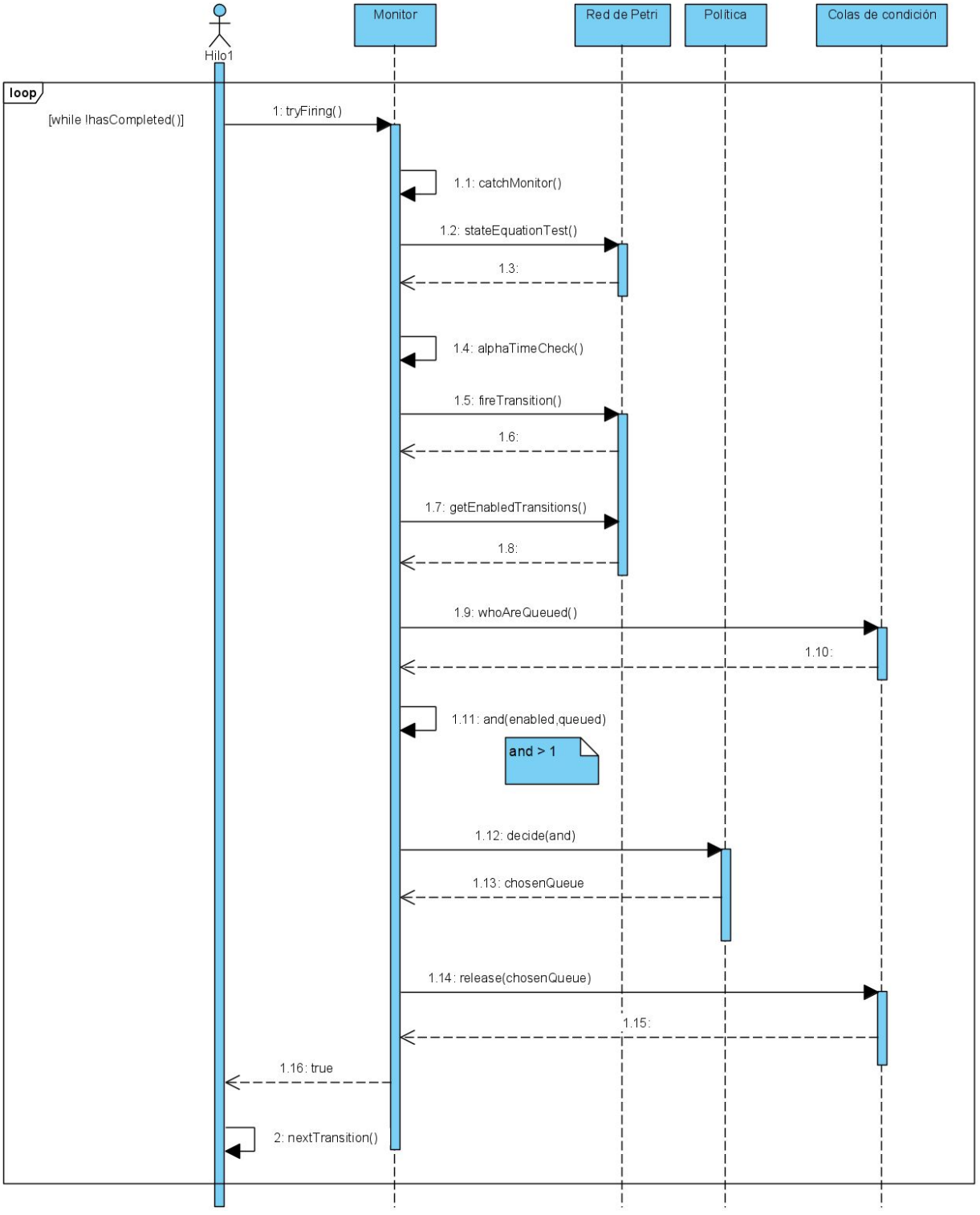


Figura 6: Diagrama de secuencia de disparo con política

Se optó por elegir una política de **decisión aleatoria de distribución uniforme**, ya que el equipo de trabajo consideró que de esta manera se mantendría un cierto equilibro en las cargas de los procesadores y también en la escritura de las memorias, con poco costo computacional. Después de varias ejecuciones se pudo observar que el equilibrio es aceptable, con algunas ejecuciones más disparejas que otras dependiendo de cuantas tareas se solicita, ya que cuanto mayor sea el número de tareas más equilibradas resultan ser las cargas.

Criterio de los hilos

Para definir y justificar la cantidad de hilos, se tuvieron en cuenta algunas consideraciones.

En primer lugar, asumiendo que los hilos deben conocer qué transición deben disparar antes de tomar el monitor, se optó por definir una secuencia de transiciones a disparar por los hilos, para evitar aleatoriedad a la hora de que los hilos tengan que decidir cuál transición deben disparar, y una vez finalizada la secuencia de cada hilo, comienzan nuevamente a disparar desde la primera, y así sucesivamente hasta cumplir con la condición de corte, estipulada como la finalización de 1000 tareas, es decir, una vez que se dispara exitosamente 1000 transiciones sumando los disparos de las 4 transiciones que implican finalizar tareas. Esta secuencia también la llamamos “caminos” que toman los hilos, y se ilustran en la figura 7.

Por lo tanto, se definieron un conjunto de hilos, “hardcodeados” al inicio de la ejecución de la clase Main Launcher, en base a los **invariantes de transición** que tiene la red. Se asignaron 8 hilos correspondientes a dichos invariantes, y un hilo extra dedicado exclusivamente a disparar la transición 0, es decir, Arrival Rate. La tabla indica los caminos que recorre cada hilo y en la figura 7 se ve lo mismo pero de manera gráfica. En total quedan 9 hilos instanciados. Los numeros se destacan en colores para luego representarlos en la figura 7.

ID	Letra	Transiciones
0	a	0
1	b	1, 3, 5, 9, 15
2	c	1, 3, 5, 10, 16
3	d	2, 4, 6, 11, 15
4	e	2, 4, 6, 12, 16
5	f	1, 3, 13, 7, 9, 15
6	g	1, 3, 13, 7, 10, 16
7	h	2, 4, 14, 8, 11, 15
8	i	2, 4, 14, 8, 12, 16

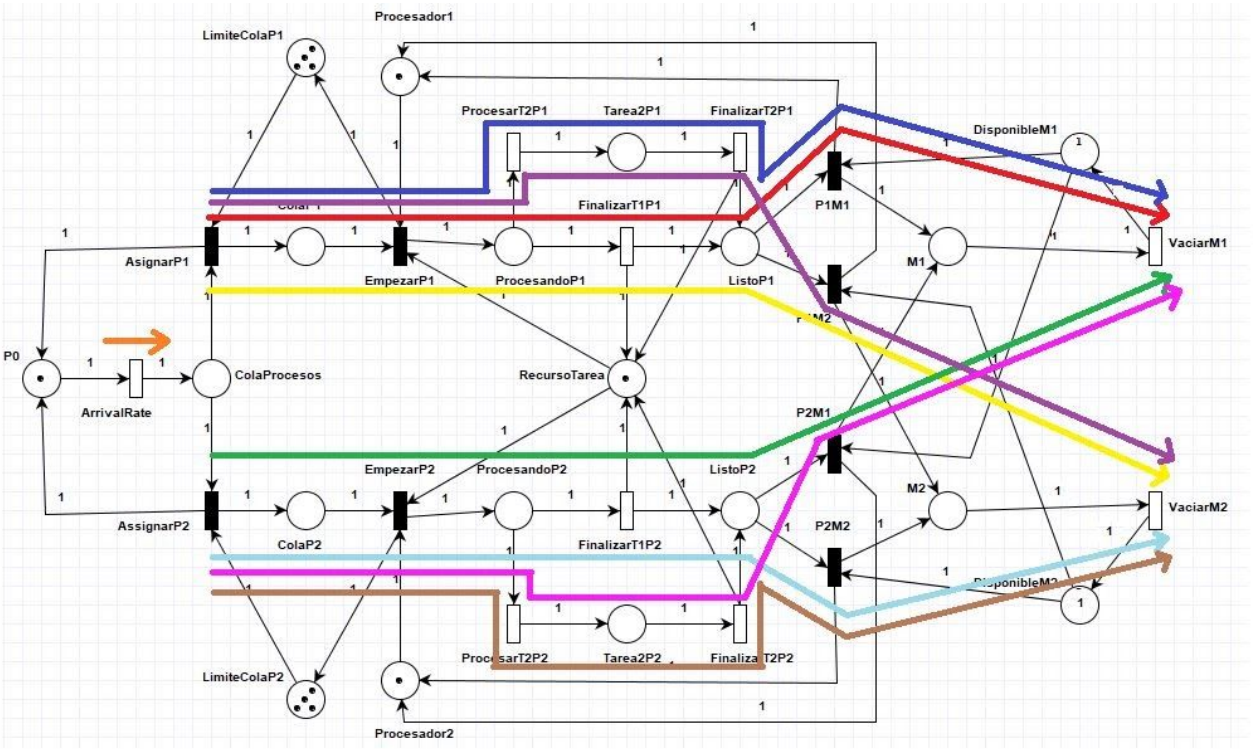


Figura 7: Caminos que recorren los hilos al ejecutar transiciones

Luego de probar con distintos criterios para la creación de hilos y asignación de transiciones, se decidió asignar de esta manera el flujo de la red (instanciando un hilo por cada invariante, y un hilo extra para la transición inicial que es la más demandada), ya que garantiza un correcto funcionamiento de la misma, equilibrando la performance del código en cuanto a cantidad de hilos en ejecución y aprovechar la concurrencia.

Desarrollo del Trabajo

Para llevar a cabo el desarrollo de todo el trabajo necesario para este proyecto, en primer lugar el equipo de trabajo comenzó por leer el enunciado completo y hacer un análisis de la red para hacerle las modificaciones que ya se comentaron anteriormente.

Luego de agregar las transiciones de vaciado de memoria, se decidió no avanzar en el código hasta verificar los invariantes de transición, de manera de fijar mejor los conceptos algebraicos sobre la ecuación de estado y los invariantes de plazas y transiciones. En esta etapa se hizo una clase extra al proyecto, que posteriormente fue dejada sin efecto, para familiarizarse con la clase Matrix de la librería Jama y sus métodos, y poder operar con los invariantes de modo de entenderlos más detalladamente.

A continuación se planteó el primer diagrama de clases antes ilustrado, para comenzar a definir los criterios que usamos en el trabajo. Se dedicaron muchas horas de trabajo sólo a conversar con qué criterio se crean los hilos en función de qué red de Petri se estuviera analizando. En principio la idea era que la cantidad de hilos fuera variable y que tuvieran un conjunto de transiciones asociadas a ellos, no necesariamente en un orden determinado, para decidir qué transición se dispara dentro del monitor, al que llamamos **“Criterio de Colores”**. Sin embargo, se decidió dejar sin efecto este criterio y “hardcodear” algunos hilos correspondientes a los invariantes de transición para la red proporcionada para simplificar el trabajo y poder terminarlo dentro de la fecha límite.

Se debatió acerca de la política, para conseguir un balance efectivo de las cargas según se solicita en la consigna, y se decidió que lo más efectivo sería hacer una distribución aleatoria uniforme, de modo de tener un relativo balance a un bajo costo computacional.

En esta etapa se **codificó** todo lo debatido en las etapas anteriores, mientras a su vez se iban diagramando las ideas en los diagramas de clase y secuencia definitivos, de modo de tener una base a la hora de codificar pero pudiendo añadir las nuevas ideas al diagrama conforme iban surgiendo.

Luego, se hizo hincapié en implementar el archivo Log, mediante una clase “viva” heredada de Thread, llamada MyLogger, que mientras no se cumpla la condición de corte haga un relevo de las tareas realizadas y el balance de cargas hasta el momento. Estos relevos se realizan cada 1 segundo, agregando un relevo final cuando finaliza la ejecución de las 1000 tareas.

Por último, se trabajó en la verificación de los invariantes, implementando un método para verificar los invariantes de plazas durante la ejecución, concretamente después de cada disparo exitoso, mientras que para los invariantes de transición se tuvo que buscar la forma de verificarlos después de la ejecución a través de la secuencia de transiciones disparada, y para ello se emplearon **expresiones regulares**, realizando experimentos en las páginas web *Regex101.com* y *Debuggex.com*, las cuales nos brindaron la posibilidad de ver cuáles cadenas se encuentran en la secuencia de transiciones disparadas para posteriormente hacer la verificación solicitada en código Java.

La **verificación de los invariantes de transición** se hace en dos partes; primero, se analiza el resultado de las transiciones disparadas (en el Log) después de la

ejecución del programa luego de finalizar 1000 tareas. Dicho resultado, un vector de transiciones disparadas, se lo utiliza en un algoritmo hecho en ***Python*** el cual lo recorre y va "sacando" cada invariante en base a una expresión regular que abarca todos los invariantes, esto se realiza iterativamente hasta sacar todos los invariantes completos que tiene dicho arreglo, ya que quedan un par de transiciones las cuales no se completó su invariante dado que, al completarse las 1000 tareas, no necesariamente se disparan las "últimas" transiciones de la red (las que vacían las memorias y completan invariantes).

La segunda parte consta de, con todas las transiciones que quedaron "sin completar ningún invariante" **se arma un vector de disparo** que las contiene a todas, y se resuelve la ecuación de estado con dicho vector, y se corrobora que el resultado de la ecuación (un vector de marcado) sea igual al marcado final, es decir, el marcado $\mathbf{m_i}$ que queda luego de la ejecución del programa.

A continuación se ilustra una captura del gráfico que representa la **expresión regular** empleada en la verificación de los invariantes, en la web **Debuggex**.

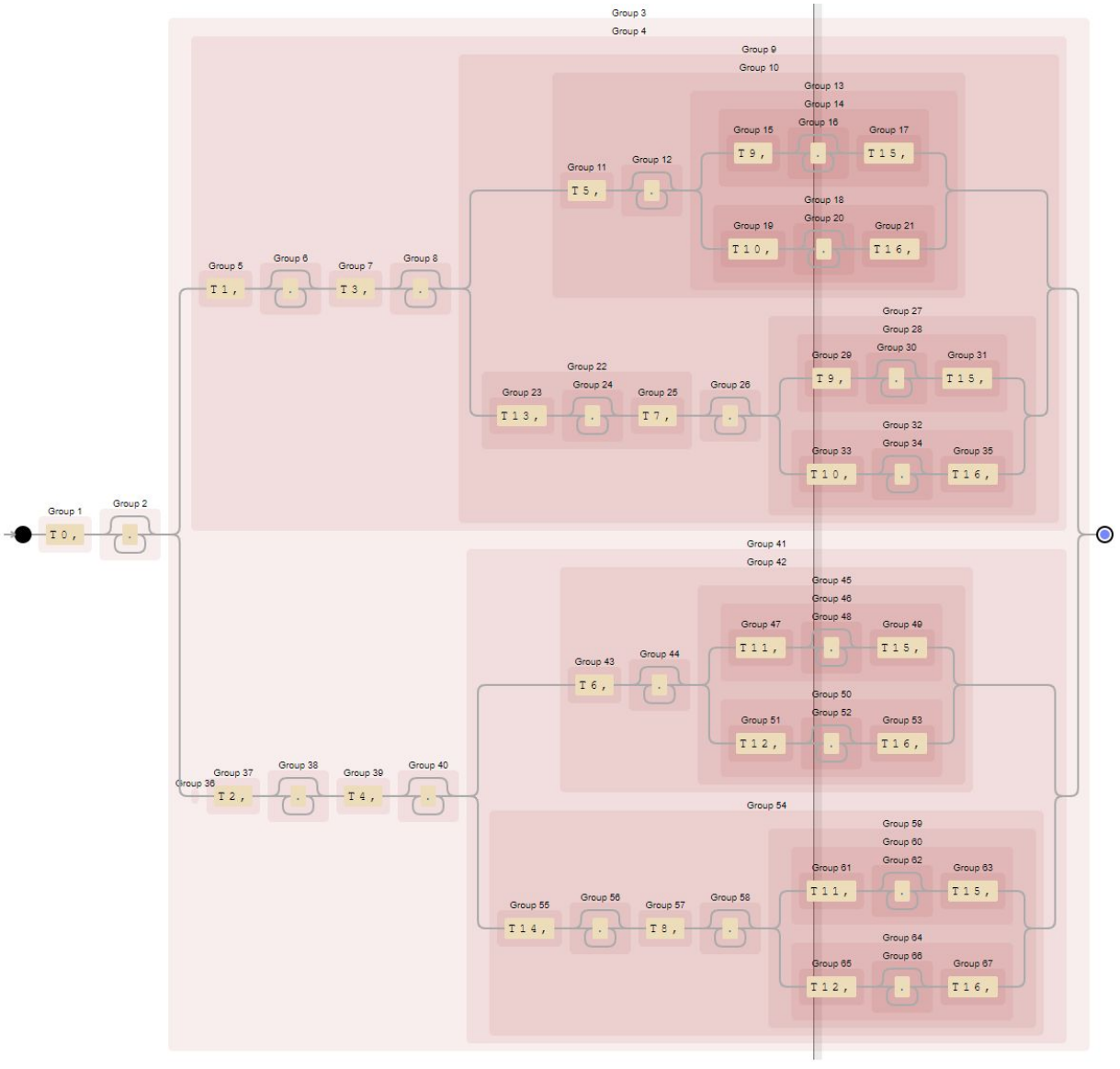


Figura 8: Expresión Regular planteada en Debuggex

Conclusiones de Diseño

Resultados de Ejecución

Las siguientes figuras son los resultados obtenidos luego de la ejecución del programa.

```
Feb 09, 2021 6:08:04 PM MyLogger run
INFO:
Cantidad de escrituras en memoria 1: 507.0
Cantidad de escrituras en memoria 2: 492.0
Carga del procesador 1: 501.0
Carga del procesador 2: 506.0
Cantidad de ejecuciones de T1 en procesador 1: 247.0
Cantidad de ejecuciones de T2 en procesador 1: 250.0
Cantidad de ejecuciones de T1 en procesador 2: 268.0
Cantidad de ejecuciones de T2 en procesador 2: 235.0
Feb 09, 2021 6:08:04 PM MyLogger run
INFO:
El tiempo de ejecucion fue de: 21 segundos.
```

Figura 9: Ejecución con Alphas x1

```
Feb 09, 2021 6:09:41 PM MyLogger run
INFO:
Cantidad de escrituras en memoria 1: 511.0
Cantidad de escrituras en memoria 2: 488.0
Carga del procesador 1: 506.0
Carga del procesador 2: 501.0
Cantidad de ejecuciones de T1 en procesador 1: 242.0
Cantidad de ejecuciones de T2 en procesador 1: 260.0
Cantidad de ejecuciones de T1 en procesador 2: 242.0
Cantidad de ejecuciones de T2 en procesador 2: 256.0
Feb 09, 2021 6:09:41 PM MyLogger run
INFO:
El tiempo de ejecucion fue de: 50 segundos.
```

Figura 10: Ejecución con Alphas x3

```
Feb 09, 2021 6:12:21 PM MyLogger run
INFO:
Cantidad de escrituras en memoria 1: 504.0
Cantidad de escrituras en memoria 2: 495.0
Carga del procesador 1: 506.0
Carga del procesador 2: 501.0
Cantidad de ejecuciones de T1 en procesador 1: 260.0
Cantidad de ejecuciones de T2 en procesador 1: 243.0
Cantidad de ejecuciones de T1 en procesador 2: 243.0
Cantidad de ejecuciones de T2 en procesador 2: 254.0
Feb 09, 2021 6:12:21 PM MyLogger run
INFO:
El tiempo de ejecucion fue de: 152 segundos.
```

Figura 11: Ejecución con Alphas x10

La figura 9 muestra los resultados de la ejecución del programa con los tiempos alpha según el siguiente vector:

{ 2, 0, 0, 0, 0, 5, 5, 5, 5, 0, 0, 0, 0, 7, 7, 4, 4 }

Se consideró que los tiempos de las transiciones están en el orden de los **milisegundos** y que el menor de ellos es ArrivalRate, ya que representa que las tareas llegan rápido en comparación con lo que tardan en procesarse y almacenarse. Todas las transiciones se trataron como temporizadas con la salvedad de que las inmediatas tienen un alfa igual a 0. Luego de probar con esos tiempos se los multiplica por 3 y luego por 10, para mostrar los resultados en las figuras 10 y 11, respectivamente.

El **balance de las cargas** se verificó en diferentes ejecuciones. El Log se encarga de dar un informe parcial que se muestra cada 1 segundo. El desbalance oscila entre 2% y 10% debido a la aleatoriedad uniforme. Sin embargo, se considera que este es un balance aceptable.

Conclusiones Finales

Este trabajo práctico nos pareció uno de los más desafiantes que hemos hecho hasta el momento a lo largo de la carrera.

Dentro de los **inconvenientes** con los que nos encontramos, en primer lugar, comenzamos usando la versión 5.0 del software **Platform Independent Petri Net Editor (PIPE)**, permitía colocar pesos variables en los arcos mediante ecuaciones, pero no permitía hacer un análisis de las propiedades de la red, por ser una versión en desarrollo, motivo por el cual se optó por utilizar la versión 4.3.0 del software.

En un principio se invirtió mucho tiempo en definir un criterio de creación de hilos y asignación de transiciones a los mismos, aproximadamente una semana entera, porque el objetivo era conseguir un criterio que funcione correctamente independientemente de qué red de petri se estuviera utilizando, por lo tanto la cantidad de hilos y las transiciones asociadas a ellos no estarían definidas de antemano sino que dependería de las propiedades de la red, como los invariantes. Este criterio se bautizó como “**Criterio de los Colores**”, porque a la hora de explicarlo se facilita mucho pensar que existen hilos de distintos tipos o “colores”. El principal inconveniente de este método es que difiere un poco con las explicaciones teóricas dadas por los docentes y requeriría quizás de mucho tiempo de codificación. Los docentes nos aconsejaron que busquemos algún otro criterio más simple aunque no se ajuste a todas las redes de petri.

Se nos complicó bastante el desarrollo de la **expresión regular** porque no tenemos experiencia con ellas. Tuvimos que buscar herramientas auxiliares como Regex101 y Debuggex. Sería más fácil chequearlo durante la ejecución, sin embargo es cierto que el objetivo de analizarlas luego de la ejecución implica ignorar la implementación del código y sólo centrarse en las transiciones que se dispararon, haciendo más fácil el reconocimiento de errores en la evolución de la red independientemente de la implementación que cada grupo haya decidido llevar a cabo.