Universidad Nacional de Córdoba

Facultad de Cs. Exactas, Físicas y Naturales

Programación Concurrente

Trabajo Práctico Final

Docentes:

Ventre, Luis Orlando — Ludemann, Mauricio

Estudiantes:

- Amallo, Sofía 41.279.731
- Auet, Luca 42.336.357
- Ojeda, Gastón 40.248.481
- Segura, Gaspar 40.416.697

Índice

Introducción

La programación concurrente es una técnica de programación que permite la ejecución simultánea de múltiples tareas dentro de un mismo programa. Sin embargo, la concurrencia también puede generar problemas como la competencia por recursos compartidos y la posibilidad de condiciones de carrera, lo que puede afectar negativamente el rendimiento y la fiabilidad del sistema. Por esta razón, es necesario utilizar herramientas adecuadas para modelar y analizar la concurrencia en sistemas complejos.

Generalmente, los programas concurrentes deben colaborar para llegar a un objetivo común, para lo cual la sincronización entre procesos es crucial.

Entre estas herramientas, las redes de Petri y los monitores son ampliamente utilizados en el ámbito de la programación concurrente. Las redes de Petri son una técnica de modelado matemático que se utiliza para describir y analizar sistemas distribuidos y concurrentes. Por su parte, los monitores son estructuras de datos que se utilizan para gestionar el acceso concurrente a recursos compartidos, mediante la sincronización de los procesos que acceden a ellos.

En este trabajo práctico, se explorará el uso de redes de Petri y monitores para la programación concurrente, describiendo su aplicación y analizando sus ventajas y limitaciones. Se abordarán temas como la modelización de sistemas concurrentes con redes de Petri, la implementación de monitores para la gestión de recursos compartidos, y la utilización de estas herramientas en la resolución de problemas de concurrencia.

Consigna

En la Figura 1 se observa una red de Petri con el modelado de un sistema de producción (industrial, informático, cyber physical systems, etc). Las plazas {P25, P26, P28} representan recursos compartidos en el sistema. Las plazas {P1, P4, P13, P16} son plazas idle, que corresponden a buffers del sistema. El resto de las plazas son plazas donde se realizan actividades relacionadas con el proceso.

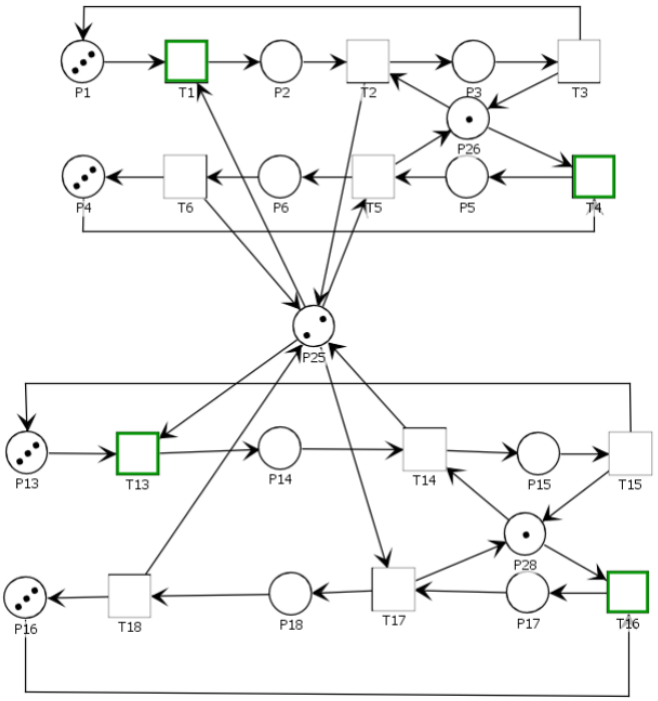


Figura 1

Es necesario implementar un monitor de concurrencia para la simulación y ejecución del modelo. Una vez implementado el monitor de concurrencia, con el sistema funcionando, se deberá implementar la semántica temporal. También se deben implementar políticas que resuelvan los conflictos relacionados a mantener balanceada la carga en los diferentes invariantes de transición.

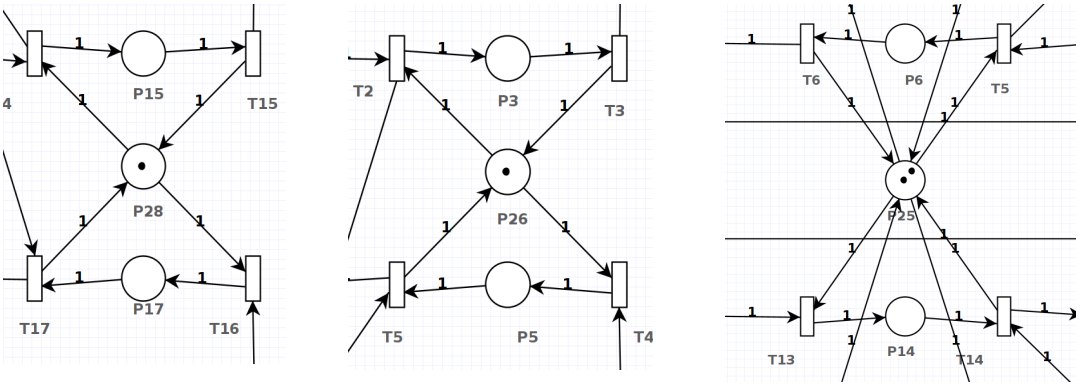
Desarrollo

Análisis de la red original

Para comenzar, a partir de la red original con la cual se debe desarrollar el trabajo, se lleva a cabo un análisis para así conocer las características de la topología de la misma.

En primer lugar, se analiza si la red posee conflictos estructurales. Estos se dan cuando una plaza tiene más de una transición de salida. Lo llamamos conflicto porque una vez que la plaza entrega el token a una de las transiciones, en caso de que las dos estén sensibilizadas, la otra podría quedar no sensibilizada y no se podrá saber a cuál transición se le entregará el token.

En esta red, identificamos los siguientes conflictos estructurales:



Debido a la cantidad de tokens que poseen las plazas, se puede establecer que estos conflictos eran efectivos ya que disparar una transición inhabilitaba disparar la otra, por lo que la red solo podía avanzar por uno de los caminos.

A partir de esto, se realizó un análisis de la red para obtener sus propiedades. Para llevarlo a cabo se hizo uso del software PIPE:

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	true

Las características que se tienen en cuenta son:

- **Acotada (Bounded):** una red de Petri es acotada cuando la misma tiene limitada la cantidad máxima de tokens posibles. Esto implica que existe un número finito de marcados alcanzables a partir del marcado inicial. Es decir, que si comenzamos con un número finito de recursos en las plazas y dejamos que la misma evolucione, no alcanzaremos un número infinito de recursos. El caso contrario sería una situación irreal, ya que no podríamos simular de manera práctica dicha cantidad infinita de recursos.

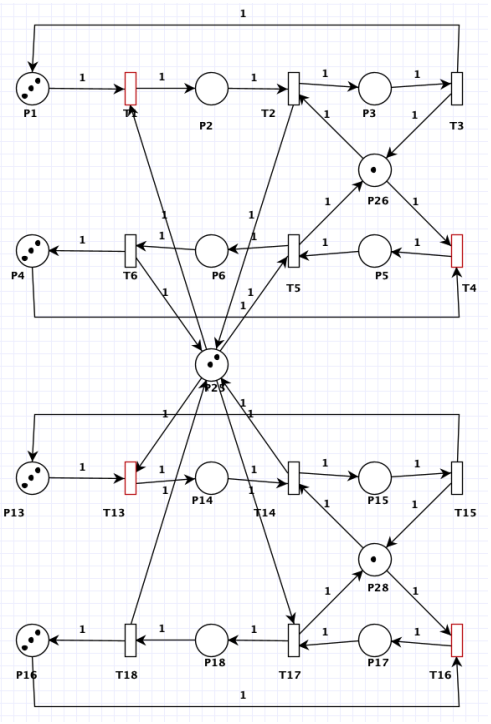
- **Segura (Safe):** una red es segura cuando la cantidad de tokens que puede tener una plaza es solo uno. Garantiza que nunca se producirá una condición de carrera entre las diferentes transiciones.

- **Deadlock (Interbloqueo):** una red posee deadlock cuando ninguna transición puede dispararse debido a que todas las plazas de las que consumen recursos no tienen la cantidad suficiente y por lo tanto no pueden sensibilizarse.

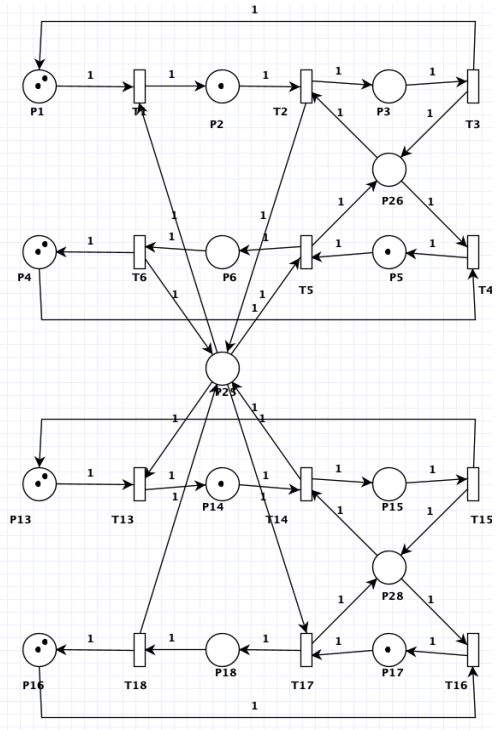
A partir de estas características, podemos definir que nuestra red es acotada, no es segura, lo que implica que debemos implementar un mecanismo de sincronización, el Monitor, que se encargue de gestionar el acceso a los recursos de las plazas para evitar condiciones de carrera, y posee deadlock, lo que implica que existen casos puntuales donde el marcado de la red queda en un estado en el cual no se pueden disparar más transiciones. Por lo tanto, la red debe ser modificada con el fin de eliminar esta última propiedad.

Considerando la red original, podemos ver como los conflictos estructurales identificados anteriormente se vuelven efectivos al momento de realizar la secuencia de disparo T1 - T4 - T13 - T16 llevando así a la red a un estado de deadlock.

Marcado Inicial



Marcado luego de la secuencia de disparo (Deadlock)



Soluciones

Debido a que la red posee deadlock, ésta debe ser modificada. Las soluciones que se plantean no pueden cambiar los invariantes de transición.

Estos invariantes se obtienen a partir de la estructura de la red:

T-Invariants

T1	T13	T14	T15	T16	T17	T18	T2	T3	T4	T5	T6
0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1

Donde cada fila corresponde a un invariante:

$Inv1 = T1 - T2 - T3$

$Inv2 = T4 - T5 - T6$

$Inv3 = T13 - T14 - T15$

$Inv4 = T16 - T17 - T18$

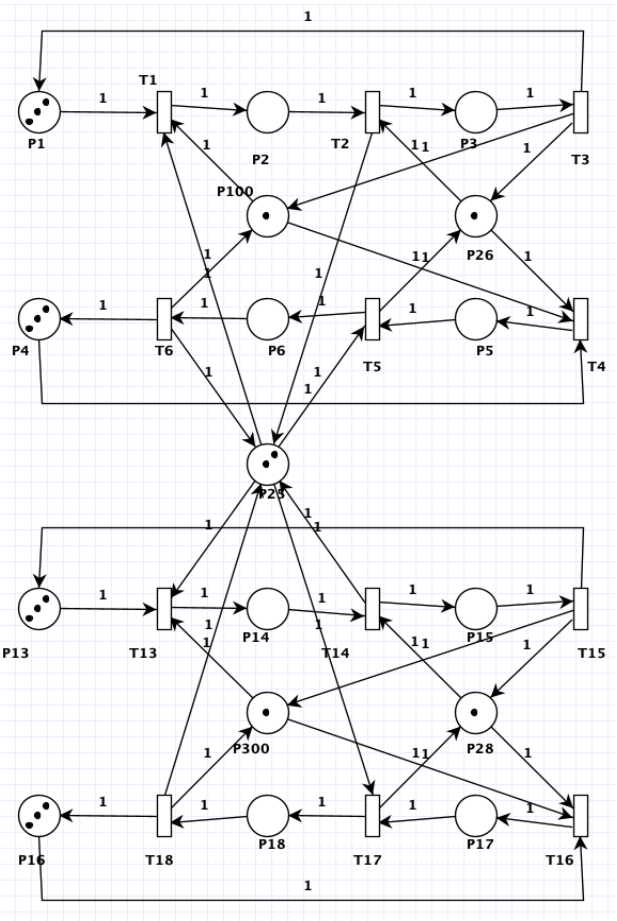
Por lo tanto, para desarrollar una solución sin modificar los invariantes de transición, solo se van a agregar plazas y tokens con sus respectivos arcos.

A partir de la red original, se analiza el patrón de los estados previos al bloqueo, es decir, que sucede en la red para que esta llegue a un estado en el cual no puede disparar ninguna otra transición.

Luego de observar el comportamiento, se encuentra que el deadlock se produce cuando dos de las líneas de producción del mismo lado de la red (las dos por encima o las dos por debajo de P25) intentan avanzar en simultáneo lo cual produce que consuman un recurso cada una que luego la otra necesita para avanzar, lo cual hace que se bloqueen.

Solución 1:

La primera solución planteada consiste en agregar las plazas P100 y P300, colocando un token a cada una. Estas plazas cumplen la función de limitar el avance de la red, garantizando que solo una de las líneas de producción de cada lado (arriba y abajo) se ejecute desde el inicio hasta el final del invariante, por lo cual no se produce el deadlock.



Se chequea que la red no tenga deadlock:

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

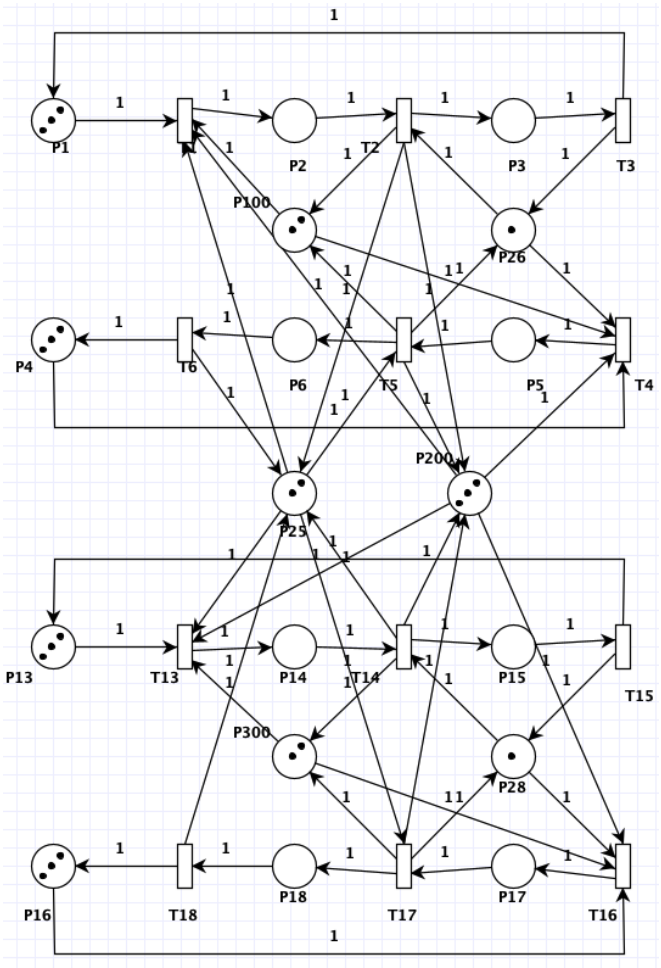
Y a su vez que se mantenga los invariantes de transición:

T-Invariants

T1	T13	T14	T15	T16	T17	T18	T2	T3	T4	T5	T6
0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1

Solución 2:

A partir de la solución anterior, se intenta buscar un mayor paralelismo sin comprometer que se produzca deadlock en la red. Se agrega una plaza más con 3 tokens, y a su vez se agrega 1 token más tanto a P100 como a P300.



La red sigue sin tener deadlock:

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

Y mantiene los mismos invariantes de transición:

T-Invariants

T1	T13	T14	T15	T16	T17	T18	T2	T3	T4	T5	T6
0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1

Comparación de soluciones:

La solución 2 posee mayor paralelismo respecto a la solución 1 ya que permite que se ejecuten 3 líneas de producción en paralelo a diferencia de solo 2.

Para corroborar estos resultados se realiza un análisis de todos los marcados posibles de cada red haciendo uso de una de las funciones que provee PIPE, y se exporta a una tabla. En esta, se eliminan las columnas de las plazas que representan recursos, buffer y restricciones para analizar sólo las plazas en las que se realiza trabajo.

Plaza	Promedio Tokens		
	Original	Solución 1	Solución 2
P2	0,99	0,2	0,64
P3	0	0,2	0,18
P5	0,99	0,2	0,53
P6	0	0,2	0,18
P14	0,99	0,2	0,64
P15	0	0,2	0,18
P17	0,99	0,2	0,53
P18	0	0,2	0,18
Total	3,96	1,6	3,06

A partir de esta tabla podemos determinar con exactitud que la Solución 2 efectivamente posee mayor paralelismo. Se observa que la red original pareciera tener un mayor paralelismo, pero esto no es real ya que la red posee deadlock, y podemos observar que algunas de las plazas tienen un promedio de 0.

Solución Final

Luego de tomar la decisión de la red que se va a utilizar para llevar adelante el desarrollo, se procede a analizar las propiedades finales de esta red.

Se soluciono el problema de deadlock que poseía la red original, y esto se logró agregando 3 plazas más junto a sus respectivos tokens manteniendo un buen nivel de paralelismo.

Invariantes de plaza:

P-Invariant equations

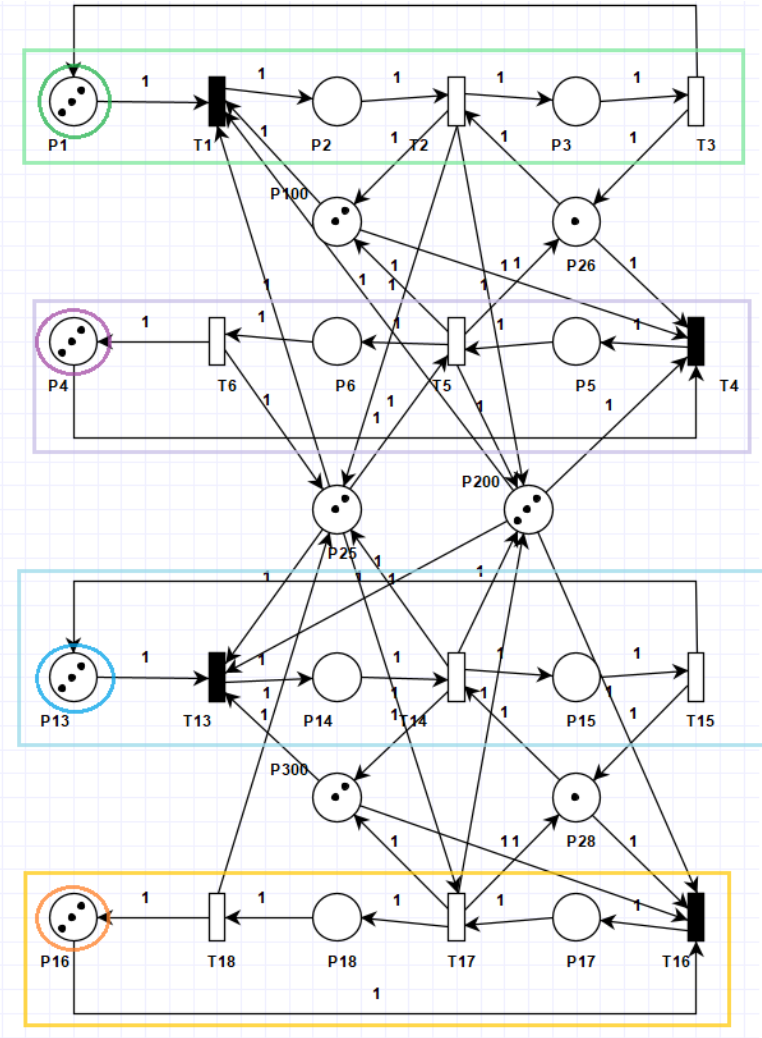
M(P1) + M(P2) + M(P3) = 3
M(P100) + M(P2) + M(P5) = 2
M(P13) + M(P14) + M(P15) = 3
M(P16) + M(P17) + M(P18) = 3
M(P14) + M(P17) + M(P2) + M(P200) + M(P5) = 3
M(P14) + M(P18) + M(P2) + M(P25) + M(P6) = 2
M(P26) + M(P3) + M(P5) = 1
M(P15) + M(P17) + M(P28) = 1
M(P14) + M(P17) + M(P300) = 2
M(P4) + M(P5) + M(P6) = 3

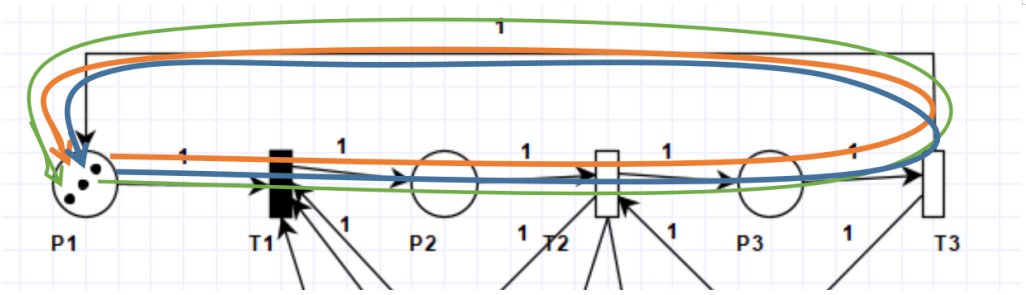
Invariantes de transición:

T-Invariants

T1	T13	T14	T15	T16	T17	T18	T2	T3	T4	T5	T6
0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1

Interpretamos que en el modelo cada invariante de transición representa una línea de producción, en la cual cada transición corresponde a una etapa de la misma. Los tokens tanto en las plazas encerradas en círculos (P1, P4, P13 y P16) que representan a los trabajadores, como los tokens que se encuentran entre cada linea de produccion (P100, P26, P25, P200, P300 y P28) que representan maquinarias, materias primas, entre otros, son los recursos que posee nuestra red.





Implementación

Para llevar adelante la implementación de esta red de Petri se utilizó el lenguaje de programación Java. Para poder lograr la simulación de este sistema se hizo uso de una conjunto de clases. A continuación se detallan las mas importantes junto a su explicación correspondiente:

Monitor:
Se encarga de sincronizar el acceso de los hilos a los recursos. De esta manera, los disparos de la red se hacen de manera tal que esta siempre es consistente. Para ello cuenta con un mutex, que le permite a un único hilo estar “dentro” del mismo y acceder a sus métodos y variables.

Para llevar a cabo estos disparos cuenta con una serie de colas de condición que consisten en un arreglo de semáforos (1 para cada transición). Los hilos que quieren disparar una transición que no está sensibilizada se van a dormir en estas colas hasta que puedan ser despertados para continuar con la ejecución.
Para decidir qué hilo continua con la ejecución, este le consulta a la política cuál debe hacerlo debido a las condiciones definidas en ella.

Policy:
Para mantener balanceada la red, la política lleva un registro de las transiciones disparadas y asocia un contador a cada invariante de transición. Cuando esta es consultada para saber qué transición debe dispararse, y por lo tanto a que hilo despertar, decide por la transición sensibilizada que pertenezca al invariante que menos veces fue disparado.

PetriNet:
Para modelar la red como tal, se utilizó álgebra lineal aplicada a Redes de Petri. Se tiene un vector que representa el marcado actual, donde cada elemento es una plaza, y los valores asociados son la cantidad de tokens que posee. Además, se tiene la matriz de incidencia donde se relacionan las plazas con las transiciones, respecto a si las transiciones consumen o entregan tokens a dichas plazas cuando se actualiza la red. Por ejemplo: si la plaza i está involucrada en la transición j,

entonces el elemento (i,j) de la matriz es igual a -1 si la plaza i es un lugar de entrada de la transición j, o es igual a 1 si la plaza i es un lugar de salida de la transición j. Si la plaza i no está involucrada en la transición j, entonces el elemento (i,j) de la matriz es igual a 0.
Haciendo uso de la ecuación fundamental: $m_k = m_i + W * s$ donde $m_k$es el marcado luego de disparar una transición, m_i el marcado actual de la red, W la matriz de incidencia y s el vector de disparo, podemos simular la evolución de la red.

Se creó una clase auxiliar Matrix en la cual se ejecutan todas las operaciones matriciales necesarias para simular el avance de la red.

Además, en cada disparo que se realiza se chequea que los invariantes de plaza cumplan. En caso de que esto no suceda, es porque algo salió mal y el programa emite un error.

Worker:
Esta clase representa un hilo de ejecución que dispara las transiciones de una red de Petri en un orden específico y de manera continua mientras se ejecuta.
Recibe como argumento un arreglo de enteros que representa al invariante de transición que se va a encargar de disparar.
En cada iteración del ciclo, el hilo llama al método fire() del objeto monitor el cual se encargará de manejar la concurrencia para disparar la transición correspondiente.

Logger:
Esta clase implementa un logger thread safe, lo que significa que varios procesos pueden escribir en el mismo archivo de registro sin interferir entre sí. Se utiliza una cola concurrente (ConcurrentLinkedQueue) para almacenar los mensajes de registro entrantes, que es donde cada uno de los Workers deposita el mensaje que quieren que se loguee.

Esta clase utiliza el patrón Singleton para garantizar que solo es creado una vez. Además, se ejecuta en un hilo aparte que el resto del programa y por lo tanto no ralentiza el proceso.

Diagramas

Diagrama de Clases

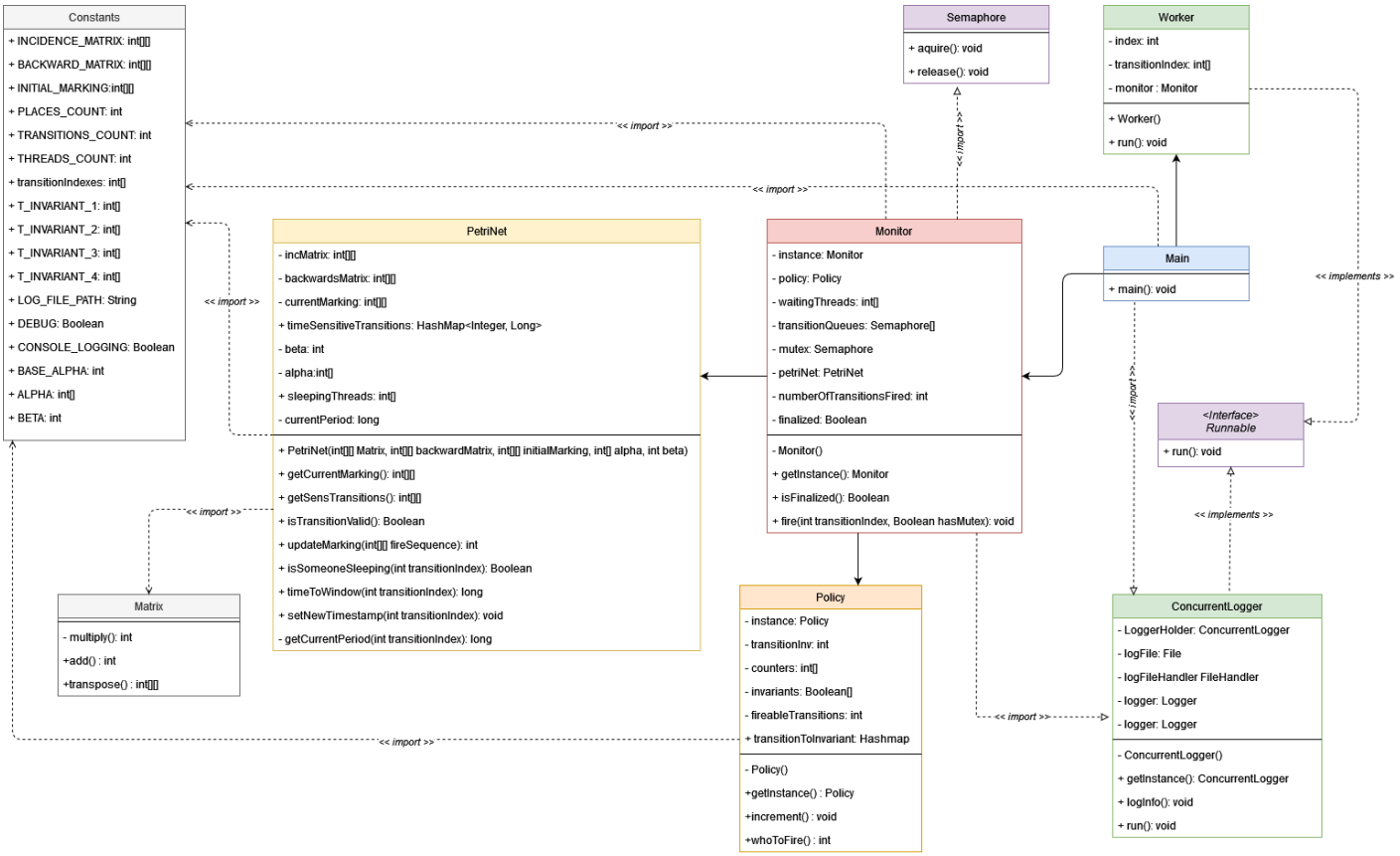
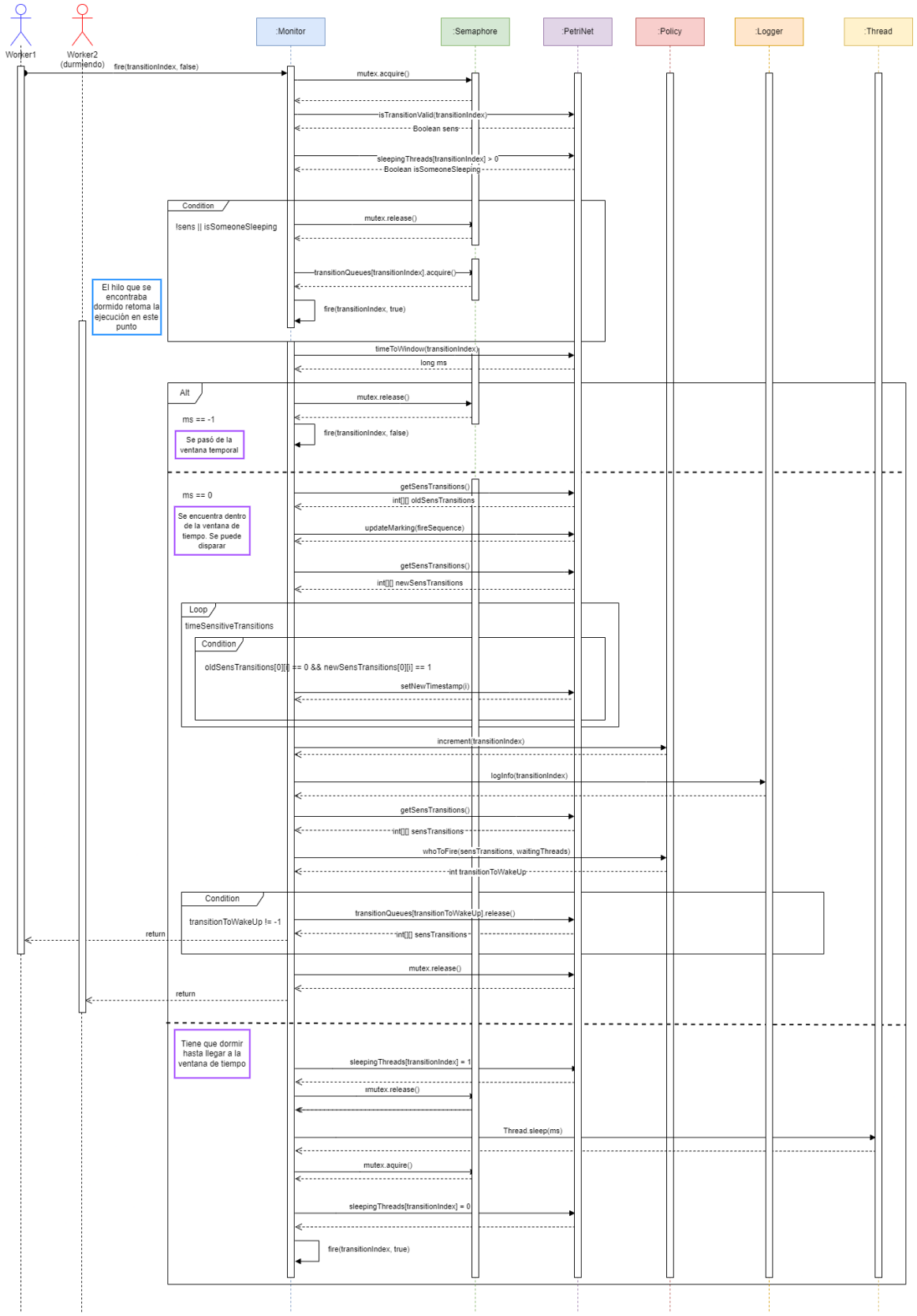


Diagrama de Secuencia

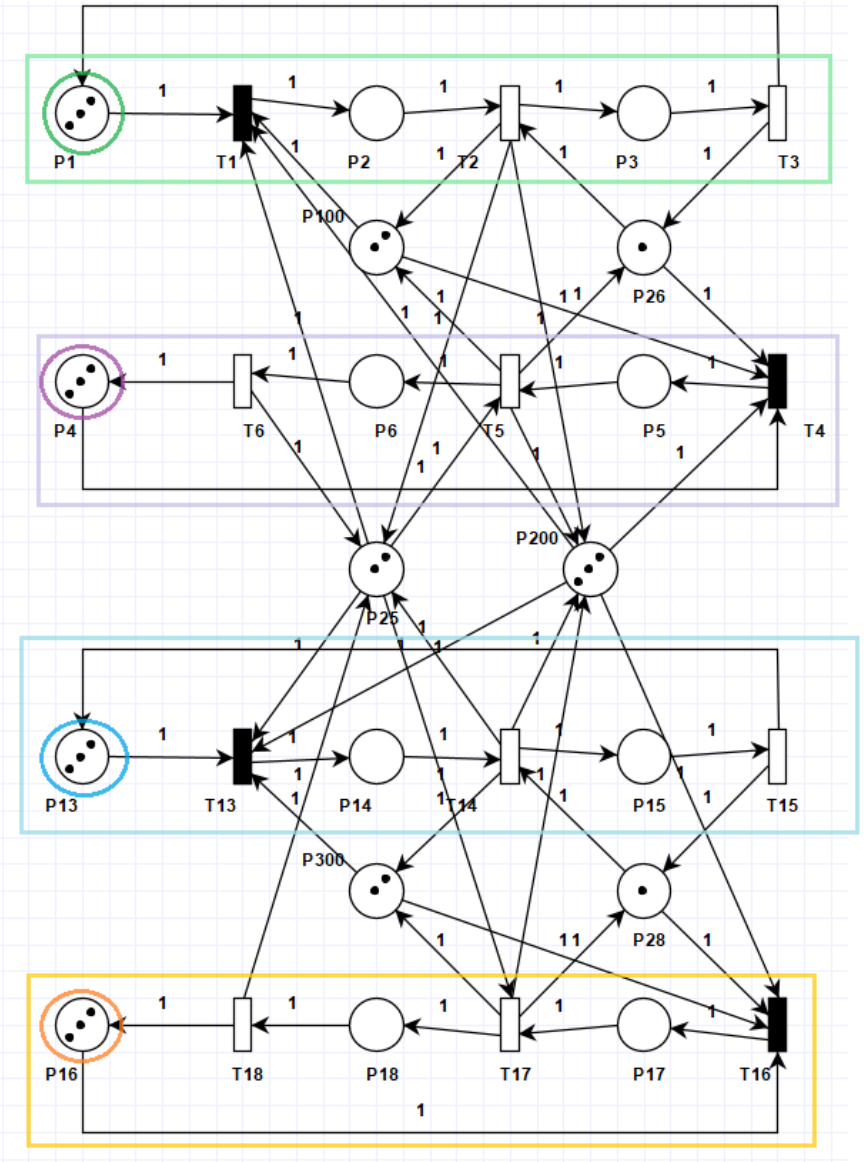
Representa el disparo exitoso y la activación de un hilo en espera, del monitor.



Regex

A partir de los logs generados por el programa realizado se procede a verificar el cumplimiento de los invariantes de transición haciendo uso de expresiones regulares.

La red original consta de 12 transiciones:



De la cual sus invariantes de transición son:

T-Invariants

T1	T13	T14	T15	T16	T17	T18	T2	T3	T4	T5	T6
0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0
1	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1

De forma más clara:

$Inv1 = T1 - T2 - T3$

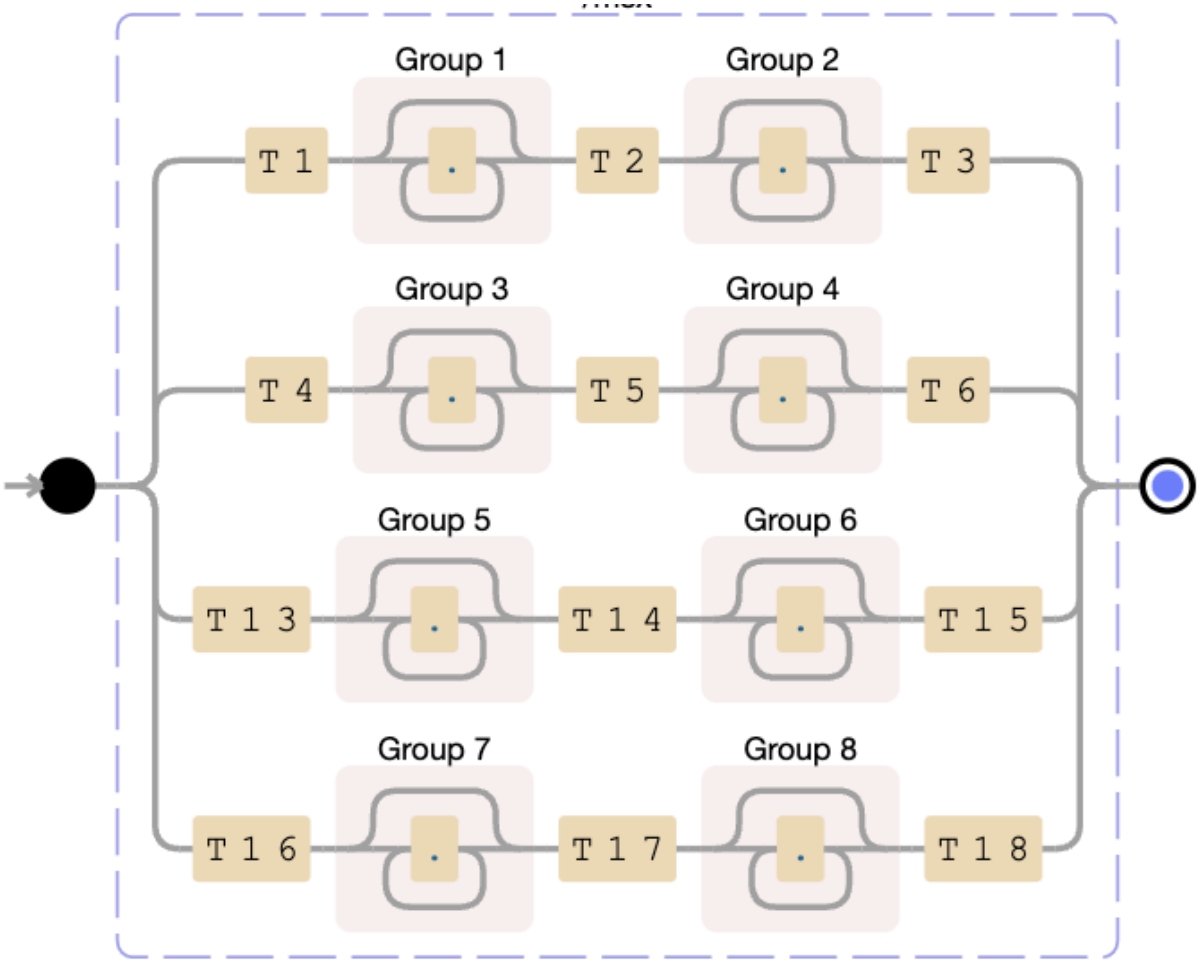
$Inv2 = T4 - T5 - T6$

$Inv3 = T13 - T14 - T15$

$Inv4 = T16 - T17 - T18$

Para analizar estos invariantes, se creó una expresión regular que posee la siguiente sintaxis:

$T1\ (.*?)T2\ (.*?)T3\ |T4\ (.*?)T5\ (.*?)T6\ |T13\ (.*?)T14\ (.*?)T15\ |T16\ (.*?)T17\ (.*?)T18$



Como se puede apreciar, debido a que los invariantes no tienen ninguna transición en común, se trata de 4 expresiones independientes.

Si analizamos solo la primer expresión $T1\ (.*?)T2\ (.*?)T3$ y la desglosamos en cada una de sus partes:

$T1$:

- **T**→ Coincide con un carácter ‘T’.
- **1**→ Coincide con un carácter ‘1’.
- → Coincide con un carácter ‘ ’.

(. *?):

- **()** → Grupo de captura 1. Agrupa varios tokens y crea un grupo de captura para extraer una subcadena.
- **.** → Coincide con cualquier carácter excepto los saltos de línea.
- ***** → Coincide con 0 o más de el token anterior.
- **?** → Hace que el cuantificador precedente sea ‘lazy’, haciendo que coincida con el menor número posible de caracteres.

$T2$:

- **T**→ Coincide con un carácter ‘T’.
- **1**→ Coincide con un carácter ‘2’.
- → Coincide con un carácter ‘ ’.

(. *?):

- **()** → Grupo de captura 2. Agrupa varios tokens y crea un grupo de captura para extraer una subcadena.
- **.** → Coincide con cualquier carácter excepto los saltos de línea.
- ***** → Coincide con 0 o más de el token anterior.
- **?** → Hace que el cuantificador precedente sea ‘lazy’, haciendo que coincida con el menor número posible de caracteres.

$T3$:

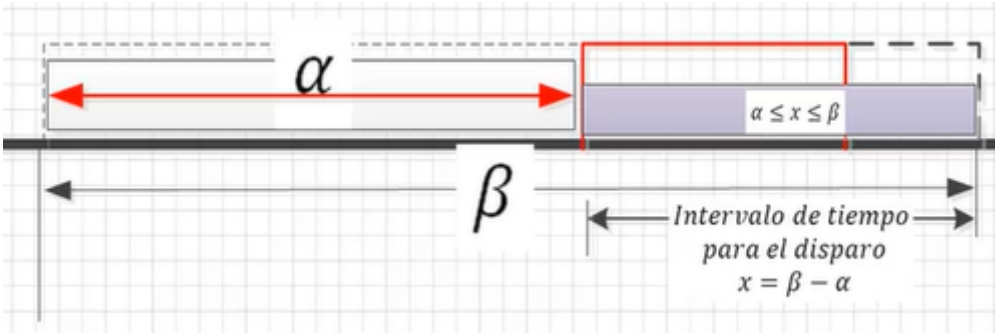
- **T**→ Coincide con un carácter ‘T’.
- **1**→ Coincide con un carácter ‘3’.
- → Coincide con un carácter ‘ ’.

Análisis de Tiempos

Para realizar el análisis de tiempos contrastamos dos parámetros: el tiempo total de ejecución y el tiempo total que los hilos pasaban “en espera”. Este último consiste en una variable que acumula el tiempo total que los hilos tuvieron que dormir.

```
totalTimeSleeping += ms;
    Thread.sleep(ms);
```

Donde `ms` se obtiene de la función `petriNet.timeToWindow(transitionIndex)`.
Si la ejecución de nuestro programa fuera secuencial, es decir, que los hilos se suceden uno a otro, de tal forma que el llamado a la función sleep fuera bloqueante, el tiempo total de ejecución sería igual a `totalTimeSleeping`.
Como explicamos anteriormente, el objetivo del monitor es sincronizar el acceso a los recursos, y proteger la integridad de la red controlando que un solo hilo modifique la misma. Cuando un hilo se va a ir a dormir, libera el mutex del monitor para que otro hilo ingrese al mismo. De esta forma, se consigue que la ejecución no sea secuencial, y para comprobarlo realizamos una serie de ejecuciones en diferentes sistemas. Estas se llevaron a cabo incrementando el valor de alpha, que es la variable que indica cuánto tiempo tiene que esperar el hilo para entrar en su “ventana de tiempo”, que es el periodo en el cual está habilitado para disparar una transición.

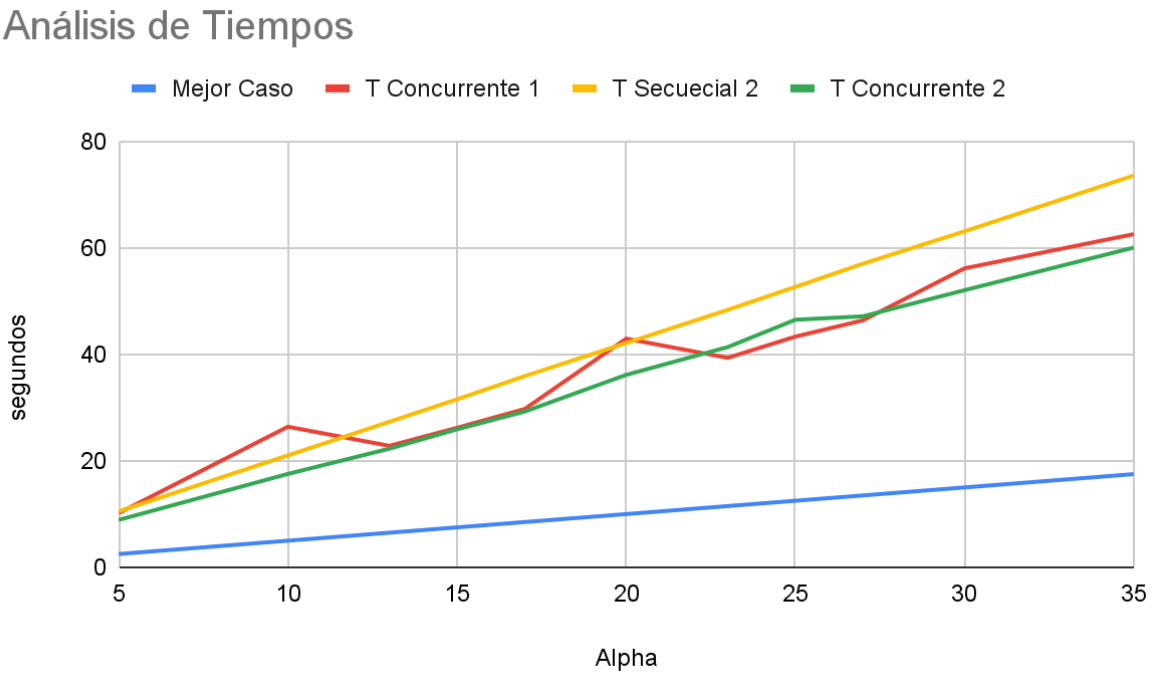


En el siguiente gráfico se puede visualizar una comparación entre el tiempo que le tomaría una ejecución secuencial vs una ejecución concurrente a dos sistemas con diferentes arquitecturas.

α	Mejor Caso	Peor Caso	T Sec 1	T Con 1	Mejora 1	T Sec 2	T Con 2	Mejora 2
5	2500	10000	10590	10186	3,81%	10505	8901	15,27%
10	5000	20000	20810	26401	-26,87%	21020	17536	16,57%
13	6500	26000	27781	22812	17,89%	27326	22281	18,46%
15	7500	30000	32010	26203	18,14%	31560	25924	17,86%
17	8500	34000	36261	29734	18,00%	35904	29245	18,55%
20	10000	40000	50799	42940	15,47%	42120	36141	14,20%
23	11500	46000	48967	39334	19,67%	48346	41356	14,46%
25	12500	50000	53500	43321	19,03%	52650	46490	11,70%

27	13500	54000	57132	46382	18,82%	57024	47132	17,35%
30	15000	60000	64170	56142	12,51%	63090	52040	17,51%
35	17500	70000	74445	62562	15,96%	73570	60038	18,39%
Prom					12,04%			16,39%

PC 1: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz (Sofi)
PC 2: Intel i3 540 (4) @ 3.059GHz (Horacio)



Dichas pruebas se realizaron en computadoras con procesadores distintos, para comprobar si había diferencias considerables.
Puede observarse que en el caso de la computadora #1 (rojo y azul) habían valores de alpha que hacían que la ejecución demore más, a diferencia de la computadora #2 donde se mantienen valores relativamente lineales. Lo importante que se pudo observar es la tendencia de la ejecución concurrente a demorar menor tiempo que la secuencial. En promedio la mejora fue de un 16%.

Conclusión

En conclusión, podemos afirmar que el uso de redes de Petri ha demostrado ser una herramienta valiosa para modelar y analizar problemas de programación concurrente y brindar soluciones. Este enfoque nos ha permitido representar de manera visual y clara la interacción de procesos y recursos en un sistema, facilitando la identificación de posibles problemas y soluciones.

Además, al abordar los problemas de concurrencia que fueron surgiendo como interbloqueos y condiciones de carrera hemos aprendido a aplicar conceptos y técnicas específicas para garantizar la correcta ejecución y coordinación de los procesos.

Utilizar estas herramientas para llevar adelante nuestro trabajo, nos permitió comprender cómo a partir de estas se puede optimizar la implementación de concurrencia, reduciendo la probabilidad de errores y mejorando el rendimiento general y la confiabilidad de nuestros programas.