UNIVERSIDAD NACIONAL DE CÓRDOBA

Facultad de Ciencias Exactas, Físicas y Naturales

Trabajo Práctico Final - "Red de Petri / Agencia de Viajes"

Programación Concurrente

Grupo: *Threading Bad*

- RODRIGUEZ, MATEO (43.967.398)
- TRACHTA, AGUSTIN (43.271.890)

1 Introducción

El presente informe detalla el trabajo realizado para abordar un problema de programación concurrente mediante el uso de redes de Petri y monitores. La programación concurrente se ha convertido en una disciplina esencial en el desarrollo de sistemas modernos, ya que permite aprovechar la capacidad de procesamiento de múltiples hilos de ejecución para mejorar la eficiencia y el rendimiento de las aplicaciones.

Sin embargo, la programación concurrente también presenta complejidades y desafíos únicos, como condiciones de carrera, bloqueos y problemas de sincronización. En este contexto, las redes de Petri emergen como una poderosa herramienta para modelar y analizar sistemas concurrentes, brindando una representación gráfica y formal de los estados y transiciones del sistema.

El objetivo principal de este informe es presentar el enfoque adoptado para resolver un problema específico de programación concurrente utilizando redes de Petri para modelar el sistema y monitores para gestionar la sincronización y el acceso seguro a los recursos compartidos. A lo largo del informe se hacen referencia a algunas definiciones que pasaremos a detallar ahora:

2 ¿Qué es una Red de Petri?

Una Red de Petri (RdP) es un modelo matemático y gráfico utilizado para describir y analizar sistemas concurrentes y distribuidos.

3 ¿Por qué utilizamos RdP?

Permiten modelar y visualizar casos de la vida real con paralelismo, concurrencia, sincronización e intercambio de recursos. Además, tienen un formalismo matemático y gráfico que nos permite determinar el disparo de una RdP y el siguiente marcado. Las RdP nos permiten separar la lógica de la política. La lógica es lo que puedo hacer, y la política es lo que me conviene hacer.

4 Propiedades de la Red

A continuación, se presentan las propiedades analizadas en la red de Petri, validadas mediante la herramienta `PIPE`. La figura inferior ilustra la red modelada:

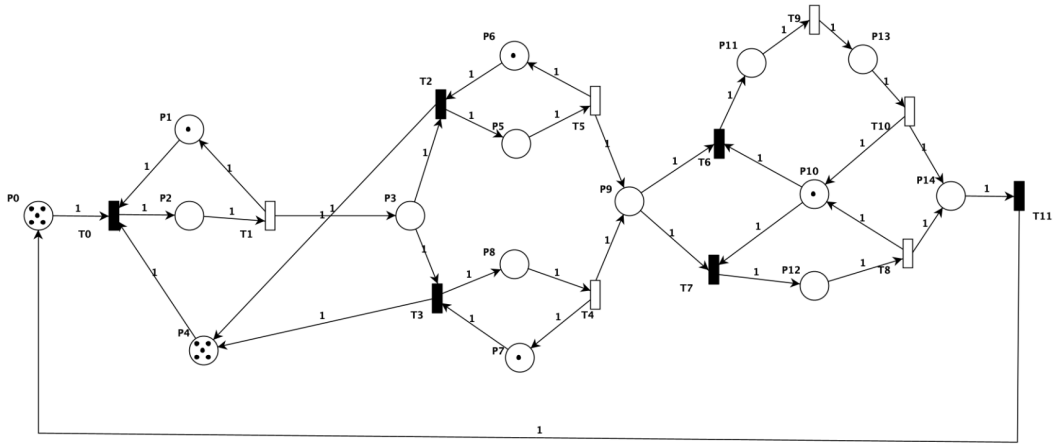


Figura: Red de Petri modelada.

Adicionalmente, se muestran los resultados del análisis del espacio de estados de la red:

Petri Net State Space Analysis Results

Bounded	true
Safe	false
Deadlock	false

4.1 Red Limitada

Una red de Petri es **limitada** si existe un límite superior para el número de tokens que pueden acumularse en cada plaza en cualquier marcado. Esto es fundamental para garantizar que los recursos representados por los tokens no se desborden. En este caso, la limitación se traduce en que la cantidad de clientes que pueden ingresar y ser atendidos está restringida por la capacidad del sistema (espacio físico, recursos humanos, etc.).

4.2 Red Segura

La red se considera **segura** si, en cualquier marcado alcanzable, cada plaza contiene como máximo un token. En nuestro modelo, se observa que algunas marcas pueden contener hasta 5 tokens, lo que indica que la red no es segura. Esto es coherente con el hecho de que se permite la entrada de hasta 5 clientes, lo que puede ser aceptable según la lógica del sistema, pero implica que la seguridad en el sentido estricto (máximo 1 token por plaza) no se cumple.

4.3 Sin Deadlock y Vivacidad

Un **deadlock** ocurre cuando la red alcanza un estado en el que ninguna transición es habilitada, lo que impide cualquier avance en la ejecución del sistema. El análisis realizado indica que la red no presenta deadlock, ya que en cada estado alcanzable existe al menos una transición habilitada. Esto garantiza que, sin importar el camino de ejecución, el sistema siempre puede continuar avanzando. Este comportamiento se relaciona con la **vivacidad** de la red: se asegura que, para cada transición, exista una secuencia de disparos que permita su eventual ejecución. En nuestro modelo, la vivacidad se corrobora al verificar que siempre hay transiciones disponibles para disparar, evitando estados bloqueados.

La validación de estas propiedades mediante `PIPE` nos proporciona confianza en el correcto comportamiento concurrente del sistema, asegurando que, a pesar de permitir múltiples tokens en algunas plazas, se mantienen las restricciones necesarias para evitar inconsistencias y bloqueos.

5 Invariante de Transición

Es un vector conformado por números enteros asociados a una secuencia de disparos. Son útiles para determinar propiedades estructurales de una RdP en forma analítica. Un invariante de transición es el conjunto mínimo de transiciones que, cuando las dispare, vuelvo al estado inicial. Esto nos indica que algo se hizo. Si me fijo cuántos invariantes se completaron, voy a entender cuántos ciclos se completaron.

Petri net invariant analysis results

T-Invariants

T0	T1	T10	T11	T2	T3	T4	T5	T6	T7	T8	T9
1	1	0	1	0	1	1	0	0	1	1	0
1	1	1	1	0	1	1	0	1	0	0	1
1	1	0	1	1	0	0	1	0	1	1	0
1	1	1	1	1	0	0	1	1	0	0	1

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

6 Invariante de Plaza

Un invariante de plaza es el conjunto de plazas en donde la suma de sus tokens se mantiene constante a lo largo de todos los marcados de la red.

P-Invariants

P0	P1	P10	P11	P12	P13	P14	P2	P3	P4	P5	P6	P7	P8	P9
0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	0	1	0	0	1	1
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0

The net is covered by positive T-invariants, therefore it is bounded.

6.1 Bounded and live

El programa quiere decir que, además de estar acotada (bounded), la red siempre tiene la posibilidad de disparar alguna transición (live), por lo que no entra en un estado de bloqueo definitivo.

6.2 Ecuaciones de Invariante de Plaza

A continuación, se listan las ecuaciones de invariante de plaza derivadas de la red:

$$M[P1] + M[P2] = 1$$

$$M[P10] + M[P11] + M[P12] + M[P13] = 1$$

$$M[P0]+M[P11]+M[P12]+M[P13]+M[P14]+M[P21]+M[P3]+M[P5]+M[P8]+M[P9] = 5$$

$$M[P2] + M[P3] + M[P4] = 5$$

$$M[P5] + M[P6] = 1$$

$$M[P7] + M[P8] = 1$$

7 Tabla de Estados en la Red de Petri

En la siguiente tabla se detallan las plazas de la red de Petri y el estado que representan:

Plaza	Estado
P0	Buffer de entrada de Clientes
P1	Recursos compartidos del sistema
P2	Cliente listo para entrar
P3	Cliente esperando a ser atendido
P4	Recursos compartidos del sistema
P5	Gestión de las Reservas
P6	Recursos compartidos del sistema
P7	Gestión de las Reservas
P8	Recursos compartidos del sistema
P9	Agente por cancelar la reserva
P10	Agente por generar reserva
P11	Agente por confirmar reserva
P12	Agente por pagar la reserva
P13	El cliente está por retirarse
P14	(Lugar adicional según el modelo)

Table 1: Tabla de estados en la red de Petri.

8 Tabla de Eventos en la Red de Petri

A continuación se listan las transiciones y el evento asociado en la red de Petri:

Transiciones	Evento
T0	Input de Clientes
T1	Cliente entra al negocio
T2	Cliente es atendido
T3	Cliente es atendido
T4	Cliente pasa a la Confirmación
T5	Cliente pasa a la Confirmación
T6	Agente confirma la reserva
T7	Agente rechaza la reserva
T8	Cliente pasa a retirarse
T9	Se paga la reserva
T10	Cliente pasa a retirarse
T11	Cliente se retira de la agencia

Table 2: Tabla de eventos en la red de Petri.

9 Determinación de la cantidad de hilos

Para determinar la cantidad máxima de hilos activos simultáneamente en el sistema, se aplicó el algoritmo propuesto por el profesor, el cual se basa en el análisis de los invariantes de transición (IT) de la red de Petri. Este método permite descomponer la ejecución en trayectorias independientes, facilitando la asignación de hilos según la concurrencia en cada segmento. A continuación, se detalla el procedimiento seguido:

9.1 Obtención de los Invariantes de Transición (IT)

Se han identificado los siguientes IT de la tabla mencionada en el apartado 5:

1. $\{T0, T1, T3, T4, T7, T8, T11\}$
2. $\{T0, T1, T3, T4, T6, T9, T10, T11\}$
3. $\{T0, T1, T2, T5, T7, T8, T11\}$
4. $\{T0, T1, T2, T5, T6, T9, T10, T11\}$

9.2 Obtención del Conjunto de Plazas Asociadas a cada IT

Para cada IT, se identifica el conjunto de plazas involucradas, lo que permite analizar cómo se distribuyen los recursos y estados en el sistema.

1. Para el IT1: $\{P0, P1, P2, P3, P4, P7, P8, P9, P10, P12, P14\}$
2. Para el IT2: $\{P0, P1, P2, P3, P4, P7, P8, P9, P10, P11, P13, P14\}$
3. Para el IT3: $\{P0, P1, P2, P3, P4, P5, P6, P9, P10, P12, P14\}$
4. Para el IT4: $\{P0, P1, P2, P3, P4, P5, P6, P9, P10, P11, P13, P14\}$

9.3 Determinación de las Plazas de Acción

Dentro de cada IT, se elimina el conjunto de plazas que representan restricciones, recursos o estados idle, dejando únicamente aquellas plazas asociadas a acciones críticas. Así se obtienen:

1. Para el IT1: $\{P2, P8, P12\}$
2. Para el IT2: $\{P2, P8, P11, P13\}$
3. Para el IT3: $\{P2, P5, P12\}$
4. Para el IT4: $\{P2, P5, P11, P13\}$

9.4 Construcción del Conjunto de Estados del Conjunto de Plazas de Acción (PA)

Se define el conjunto PA como la unión de todas las plazas de acción obtenidas:

$$PA = \{P2, P5, P8, P11, P12, P13\}$$

A partir del árbol de alcanzabilidad de la red de Petri, se extraen todos los marcados (estados) posibles correspondientes a este conjunto.

9.5 Cálculo del Valor Máximo de Hilos Simultáneos

Para cada marcado en el conjunto PA se calcula la suma total de tokens. El valor máximo obtenido en estas sumas representa la cantidad máxima de hilos que pueden estar activos simultáneamente en el sistema. En nuestro caso, se determinó que:

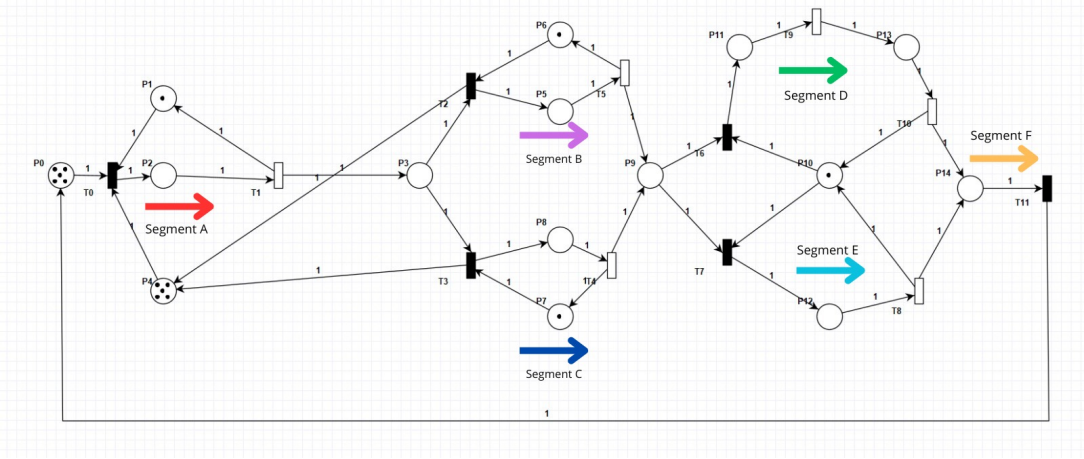
$$\textbf{Valor máximo de hilos simultáneos} = 4$$

Este resultado indica que, en el punto de mayor concurrencia, hasta 4 hilos pueden ejecutarse de forma simultánea, lo que permite aprovechar de manera óptima los recursos disponibles.

Referencia: Los algoritmos y el procedimiento descrito se basan en el informe proporcionado por el profesor (ver Referencias al final del informe).

10 Red de Petri final

La red de Petri final utilizada en el programa consta de 6 segmentos.



11 Justificación de la elección de los segmentos

La elección de los segmentos en la red de Petri se basó en la identificación de estructuras lineales, forks y joins en la red. Cada segmento se definió con el objetivo de maximizar la concurrencia y optimizar el uso de los hilos disponibles:

1. **Segmento A:** Es una secuencia de transiciones sin bifurcaciones, por lo que se ejecuta de manera secuencial sin interferencias.

2. **Segmentos B, C, D, E:** Estos segmentos contienen transiciones que generan caminos en paralelo (forks). Al dividir la ejecución en hilos independientes, se evita que un solo hilo tenga que decidir qué camino seguir, delegando la decisión a la política de ejecución del sistema.

3. **Segmento D:** Aunque es parte de un fork, en su parte final no comparte transiciones con otros segmentos, por lo que actúa también como un segmento lineal.

4. **Segmento F:** Representa el punto de convergencia de caminos en la red (join). Para evitar bloqueos, se permite la ejecución paralela de los segmentos anteriores, asegurando que todas las dependencias se resuelvan antes de continuar con la ejecución final.

12 Determinación de hilos máximos por segmento

A partir de la segmentación de la red de Petri y la identificación de los subconjuntos de plazas asociadas a cada segmento, se determina la cantidad máxima de hilos que pueden ejecutarse simultáneamente. La segmentación permite dividir la ejecución en partes independientes, asegurando que no haya interferencias en la ejecución concurrente.

Cada segmento tiene un conjunto de plazas asociadas, denominadas PS_i, que representan los estados alcanzables en dicho segmento. A continuación, se presentan los segmentos con sus respectivas plazas y la cantidad máxima de hilos necesarios para su ejecución:

- $PS_a = \{P2\} \Rightarrow \text{MAX} = 1$
- $PS_b = \{P5\} \Rightarrow \text{MAX} = 1$
- $PS_c = \{P8\} \Rightarrow \text{MAX} = 1$
- $PS_d = \{P11, P13\} \Rightarrow \text{MAX} = 1$
- $PS_e = \{P12\} \Rightarrow \text{MAX} = 1$
- $PS_f = \{P14\} \Rightarrow \text{MAX} = 1$

Dado que cada segmento requiere un hilo para su ejecución, la suma de los hilos necesarios en todos los segmentos nos da la cantidad total de hilos que el sistema puede necesitar en toda su ejecución:

$$\textit{Hilos totales del sistema} = 1 + 1 + 1 + 1 + 1 + 1 = 6$$

Sin embargo, esto no significa que en todo momento haya 6 hilos ejecutándose en simultáneo. El primer algoritmo determinó que la **máxima cantidad de hilos activos simultáneamente** en la red de Petri es **4**, lo que significa que en el punto de mayor concurrencia del sistema, pueden llegar a ejecutarse hasta 4 hilos a la vez. La segmentación realizada permite distribuir la carga de trabajo de manera eficiente, asegurando que cada hilo ejecute su parte sin interferencias y maximizando el rendimiento del sistema.

13 Implementación en Java

En esta sección se describe la implementación en Java del sistema, haciendo especial énfasis en cómo se tradujo el modelo teórico de la red de Petri a una solución práctica y concurrente. El sistema se organiza en varios paquetes, entre los que se destacan: `monitor`, `petrinet`, `utils` y `pool`. A continuación, se detalla el funcionamiento del paquete `petrinet`, que constituye el núcleo del modelo.

13.1 Paquete `petrinet`

El paquete `petrinet` es el núcleo de la simulación de la red de Petri y se encarga de modelar su comportamiento teórico en un entorno concurrente. En este paquete se definen las clases principales que gestionan el estado de la red, las reglas de disparo (transiciones) y la ejecución de segmentos de la red. A continuación se describen sus componentes:

13.1.1 Clase `Places`

La clase `Places` representa el conjunto de plazas de la red de Petri y gestiona el conteo de tokens en cada una de ellas. Sus responsabilidades principales son:

- **Manejo del estado:** Mantener un mapa (`HashMap`) que asocia cada identificador de plaza con la cantidad de tokens presentes.
- **Sincronización:** Utiliza un bloqueo global (proporcionado por `TransitionNotifier.lock`) para asegurar la exclusión mutua en las operaciones de lectura y escritura, evitando condiciones de carrera.
- **Operaciones básicas:**
  - `addPlace(int placeId, int initialTokens)`: Agrega una plaza con un número inicial de tokens.
  - `getTokenCount(int placeId)`: Devuelve la cantidad de tokens en una plaza, protegiendo la operación mediante bloqueo.
  - `addTokens(int placeId, int count)` y `removeTokens(int placeId, int count)`: Modifican la cantidad de tokens y notifican a los hilos en espera mediante la condición `transitionsEnabled`.
  - `checkInvariants()`: Recorre todas las plazas para verificar que no existan violaciones (por ejemplo, que no haya valores negativos), manteniendo la integridad del modelo.

13.1.2 Clase `Transition`

La clase `Transition` encapsula la definición de una transición en la red de Petri, es decir, una regla que define cómo se modifican los tokens al dispararse:

- **Precondiciones y Postcondiciones:** Se modelan como mapas que asocian el identificador de cada plaza con la cantidad de tokens requeridos (precondiciones) o a añadir (postcondiciones).
- **Transiciones Temporales:** Permite definir transiciones con retardo. Si una transición es temporal, se introduce un retraso (delay) en milisegundos antes de aplicar las modificaciones en el estado.
- **Verificación de habilitación:** El método `isEnabled(Places places)` comprueba que en cada plaza se cumplen las precondiciones (tokens suficientes) para permitir el disparo.
- **Disparo de la transición:** El método `fire(Places places)` valida la habilitación, espera el retardo en caso de ser temporal, y procede a remover tokens de las plazas de entrada y añadir tokens en las plazas de salida, modificando así el estado de la red.

13.1.3 Clase `Segment`

La clase `Segment` representa un segmento de ejecución de la red de Petri, es decir, un conjunto de transiciones agrupadas que se ejecutan como una unidad:

- **Agrupación de Transiciones:** Cada segmento contiene una lista de transiciones que, según la lógica del modelo, deben dispararse en conjunto.
- **Implementación de `Runnable`:** Permite que cada segmento sea gestionado por un planificador central (a través del monitor), ejecutándose en hilos independientes.
- **Sincronización interna:** Se utiliza un flag `isRunning` para evitar que el mismo segmento se ejecute de forma concurrente en más de un hilo.
- **Comprobación de habilitación:** El método `hasEnabledAndAllowedTransition()` verifica, consultando la política definida en el monitor, que al menos una transición del segmento está habilitada y permitida.
- **Ejecución:** En el método `run()`, el segmento itera sobre sus transiciones y, para cada una habilitada, solicita al monitor que dispare la transición mediante el método `fireTransition`.

13.1.4 Interacción y Sincronización Global

El diseño del paquete `petrinet` se integra de forma coordinada con otros componentes del sistema:

- La clase `Places` actúa como repositorio global del estado de la red, sobre el cual operan las transiciones.
- La clase `Transition` utiliza `Places` para leer y modificar el estado, respetando las condiciones de sincronización mediante bloqueos.
- Los segmentos, representados por la clase `Segment`, agrupan transiciones y se gestionan a través del monitor, que decide la ejecución basada en la política de resolución de conflictos.
- Se utiliza un `Logger` para trazar eventos y facilitar la depuración, lo que ayuda a verificar el correcto funcionamiento y a identificar posibles errores.

13.2 Paquete `monitor`

El paquete `monitor` se encarga de coordinar la ejecución de las transiciones de la red de Petri, implementando un mecanismo central de sincronización y resolución de conflictos. Para ello, se define la clase `Monitor` que expone el método único `fireTransition`, y se utiliza un Scheduler para la gestión de segmentos de la red. Asimismo, se implementan diferentes políticas (`BalancedPolicy` y `PriorityPolicy`) para resolver conflictos y asignar prioridades en la ejecución de transiciones. A continuación, se describen los componentes principales de este paquete.

13.2.1 Clase `Scheduler`

La clase `Scheduler` es un hilo dedicado que supervisa continuamente los segmentos definidos en la red de Petri y los envía al administrador de hilos (`PoolManager`) cuando detecta que alguno de ellos tiene transiciones habilitadas para disparar. Sus características principales son:

- **Monitoreo de segmentos:** Recorre la lista de segmentos y verifica, mediante el método `canBeScheduled()`, si alguno de ellos tiene al menos una transición habilitada.
- **Sincronización eficiente:** Utiliza un `ReentrantLock` y una condición (`transitionsEnabled`) definida en la clase `TransitionNotifier` para esperar que haya algun movimiento de tokens en las plazas del sistema cuando no hay segmentos listos, evitando ciclos de espera activa.
- **Terminación controlada:** Proporciona un método `stop()` que permite finalizar la ejecución del Scheduler de forma segura, notificando a cualquier hilo que se encuentre en espera.

El método `run()` del Scheduler itera sobre los segmentos y, si ninguno está listo, se bloquea esperando una señal. Esta señal se envía desde otros componentes cuando el estado de las transiciones cambia (al agregar en las plazas).

13.2.2 Clase `Monitor`

La clase `Monitor` actúa como el coordinador central del sistema y se encarga de:

- **Fijar la sincronización:** El método `fireTransition` es el único método público y se utiliza para disparar transiciones. Este método es sincronizado para asegurar un acceso seguro y exclusivo al estado compartido.
- **Consulta de habilitación y políticas:** Antes de disparar una transición, el Monitor consulta si la transición está habilitada (mediante el método `isEnabled` de la clase `Transition`) y, además, verifica que la política activa permita su ejecución.
- **Actualización de contadores:** Tras el disparo exitoso de una transición, se actualizan los contadores internos mediante el método `updateCounters` de la política, lo que permite llevar un registro del balance o la prioridad en la ejecución.
- **Gestión del Scheduler:** El Monitor es responsable de iniciar y detener el Scheduler, integrándolo con el administrador de hilos (`PoolManager`) para garantizar que los segmentos se ejecuten en función de la disponibilidad de transiciones.

Dentro del método `fireTransition`, se realiza el siguiente flujo:

1. Se recupera la transición asociada al identificador.
2. Se vuelve a verificar que la transición esté habilitada en función de los tokens presentes en las `Places`.
3. Se consulta nuevamente la política para confirmar que el disparo es permitido.

4. Se dispara la transición, aplicando las precondiciones y postcondiciones sobre las `Places`.

5. Se actualizan los contadores de la política y se comprueban los invariantes del sistema.

6. Se suma un contador sobre la transicion 0 de tal forma que al llegar a 187 (es decir, 186 cilos, o invariantes de transicion completados) todos los hilos son notificados y se termina el sistema.

13.2.3 Interfaces y Políticas

El Monitor utiliza interfaces para definir la estructura de la política y la forma de coordinar el disparo de transiciones:

- **Interfaz `MonitorInterface`:** Define el método `fireTransition(int transition)` que es el punto de entrada para disparar transiciones.
- **Interfaz `Policy`:** Establece dos métodos: `allowTransition(int transitionId, Places places)` para decidir si una transición puede dispararse, y `updateCounters(int transitionId, Places places)` para actualizar el estado interno tras el disparo.

Se implementan dos políticas principales:

1. **BalancedPolicy:**
   - **Objetivo:** Garantizar un equilibrio en la atención entre el agente de reservas superior (transición 2) y el inferior (transición 3). De igual forma, balancear el número de confirmaciones y cancelaciones.
   - **Implementación:** Se utilizan contadores internos para cada tipo de operación. Por ejemplo, la política permite disparar la transición asociada al agente superior solo si su contador es menor o igual al del agente inferior.
2. **PriorityPolicy:**
   - **Objetivo:** Priorizar el procesamiento del agente superior y la confirmación de reservas. Se requiere que el agente superior genere al menos el 75% de las reservas y que las confirmaciones representen al menos el 80% del total.
   - **Implementación:** Se calculan ratios basados en los contadores internos y se permite o bloquea el disparo de ciertas transiciones en función de estos porcentajes.

Ambas políticas se integran en el Monitor, de modo que al disparar una transición se consulta la política activa. Esto garantiza que la lógica de resolución de conflictos y asignación de prioridades se mantiene separada de la lógica de disparo y la actualización del estado, facilitando la modularidad y el mantenimiento del código.

13.2.4 Interacción Global en el Paquete `monitor`

La interacción entre el Scheduler, el Monitor y las políticas es fundamental para el correcto funcionamiento del sistema:

- El `Scheduler` monitorea continuamente los segmentos de la red y, cuando alguno tiene transiciones habilitadas, los envía al administrador de hilos para su ejecución.
- El `Monitor`, mediante el método `fireTransition`, se encarga de coordinar el disparo de las transiciones, integrando la consulta a la política y la actualización del estado.
- Las políticas (`BalancedPolicy` y `PriorityPolicy`) permiten gestionar de forma dinámica los conflictos y asegurar que se cumplan los criterios de balance y prioridad definidos, actualizando internamente los contadores que determinan si se permite el disparo de una transición.

En conjunto, el paquete `monitor` garantiza que la ejecución de la red de Petri se realice de forma ordenada, segura y eficiente, permitiendo la resolución de conflictos y la maximización del paralelismo a través de la coordinación de hilos y la aplicación de políticas específicas.

13.3 Paquete `pool`

El paquete `pool` se encarga de la gestión y administración del pool de hilos, permitiendo la ejecución concurrente de tareas. Este paquete implementa dos componentes principales: un `ThreadFactory` personalizado y un administrador del pool de hilos (`PoolManager`).

13.3.1 Clase `MyThreadFactory`

La clase `MyThreadFactory` implementa la interfaz `ThreadFactory` y se utiliza para crear hilos con un nombre específico, facilitando la identificación y depuración de los mismos. Sus características principales son:

- **Nombre base para hilos:** Permite asignar un nombre base a los hilos creados, seguido de un contador incremental, lo que genera nombres únicos (por ejemplo, `"ThreadPool-1"`, `"ThreadPool-2"`, etc.).
- **Atomicidad:** Utiliza un objeto `AtomicInteger` para asegurar que el contador de hilos se incremente de forma segura en entornos concurrentes.
- **Personalización:** Se pueden configurar propiedades adicionales de los hilos (como el estado daemon) en el método `newThread`, adaptándose a las necesidades del sistema.

13.3.2 Clase `PoolManager`

La clase `PoolManager` encapsula la lógica para la administración del pool de hilos, utilizando un `ExecutorService` con un número fijo de hilos. Sus responsabilidades incluyen:

- **Inicialización del pool:** Configura un pool de hilos fijo mediante `Executors.newFixedThreadPo` utilizando el `MyThreadFactory` para la creación de hilos.

- **Envío de tareas:** Proporciona el método `submitTask(Runnable task)` para enviar tareas al pool de hilos, permitiendo la ejecución concurrente de las mismas.
- **Cierre del pool:** Implementa métodos para la terminación controlada del pool (`shutdown()`), esperando a que las tareas en ejecución finalicen, o bien para un cierre inmediato (`shutdownNow()`), cancelando las tareas en curso.

En conjunto, el paquete `pool` proporciona una infraestructura robusta para la gestión de hilos, lo que es crucial para la ejecución concurrente del sistema. La integración de un `ThreadFactory` personalizado con el `PoolManager` asegura que las tareas se ejecuten de forma ordenada y se puedan rastrear fácilmente durante la ejecución, lo que resulta fundamental para la depuración y optimización de la aplicación.

13.4 Paquete `utils`

El paquete `utils` proporciona herramientas de apoyo fundamentales para el correcto funcionamiento del sistema. Entre ellas se incluyen un Logger para la depuración y registro de eventos, la construcción de la red de Petri a través de la clase `PetriNet`, y un mecanismo de notificación de cambios en la red, implementado en la clase `TransitionNotifier`.

13.4.1 Clase `Logger`

La clase `Logger` es una implementación sencilla y thread-safe de un registrador (logger). Se implementa como un singleton, lo que garantiza que exista un único punto de registro en la aplicación. Entre sus características se destacan:

- **Formato de registro:** Cada entrada de log sigue el formato:

  `[YYYY-MM-DD HH:MM:SS] [Thread-Name] [LEVEL] Message`

- **Métodos de registro:** Se proporcionan métodos para registrar mensajes de distintos niveles (INFO, DEBUG, WARN, ERROR). Cada método utiliza el mismo método privado `log` para formatear y escribir la entrada en el archivo `petri_net.log`.
- **Sincronización:** Los métodos de registro son sincronizados para asegurar que múltiples hilos puedan escribir en el archivo de log sin interferencias.

13.4.2 Clase `TransitionNotifier`

La clase `TransitionNotifier` es esencial para la coordinación de la red de Petri. Esta clase centraliza dos elementos críticos para la sincronización:

- **ReentrantLock:** Se utiliza para proteger las operaciones que modifican el estado de la red, evitando condiciones de carrera.
- **Condition `transitionsEnabled`:** Esta condición permite notificar a los hilos que están esperando cambios en el estado de la red (por ejemplo, cuando se agregan o quitan tokens en las `Places`). Cada vez que se actualiza el estado de la red, se llama a `signalAll()` para despertar a los hilos que esperan que alguna transición se habilite.

De esta forma, `TransitionNotifier` actúa como un mecanismo central de señalización, facilitando la coordinación entre el Monitor, el Scheduler y los segmentos que esperan poder disparar sus transiciones.

13.4.3 Clase `PetriNet`

La clase `PetriNet` encapsula el proceso de construcción de la red de Petri. Su función es:

- **Inicializar el modelo:** Crea las instancias de `Places`, define las transiciones (con sus precondiciones, postcondiciones y, en algunos casos, retardo temporal) y agrupa estas transiciones en segmentos de ejecución.
- **Integrar componentes:** Establece la relación entre las `Places`, las `Transition`, los segmentos y el Monitor, asegurando que el modelo se construya de acuerdo a la especificación.
- **Selección de políticas:** Permite elegir entre distintas políticas de resolución de conflictos (por ejemplo, `BalancedPolicy` o `PriorityPolicy`) que se aplicarán en el Monitor.

13.5 Clase `Main`

La clase `Main` es el punto de entrada de la aplicación y coordina la ejecución de la simulación de la red de Petri. A continuación se describe el flujo principal implementado en el método `main`:

1. **Inicialización y registro:** Se obtiene la instancia del `Logger` (implementado como singleton) para comenzar a registrar los eventos de la ejecución, y se registra el inicio de la simulación.
2. **Construcción de la red de Petri:** Se crea una instancia de la clase `PetriNet` (del paquete `utils`), la cual se encarga de construir el modelo: inicializar las `Places`, definir las `Transitions` (incluyendo sus precondiciones, postcondiciones y retardo para transiciones temporales) y agruparlas en `Segments`. Además, se configura el `Monitor` y se selecciona la política a utilizar (en este ejemplo se opta por `PriorityPolicy`, aunque también se podría utilizar `BalancedPolicy`).
3. **Configuración de la concurrencia:** Se extraen los segmentos, las `Places` y el `Monitor` del objeto `PetriNet`. A continuación, se crea un `PoolManager` utilizando un `MyThreadFactory` personalizado que asigna nombres a los hilos, y se configura el pool con 4 hilos. Esta configuración permite ejecutar las tareas (segmentos) de forma concurrente.
4. **Inicio del Scheduler:** El `Monitor` inicia el Scheduler (del paquete `monitor`), el cual se encarga de enviar los segmentos al `PoolManager` cuando tienen transiciones habilitadas. De este modo, el sistema comienza a disparar transiciones según las reglas de la red y la política de resolución de conflictos.
5. **Espera de la condición de invariante:** La clase `Main` se bloquea esperando que se cumpla la condición invariante (en este caso, que la transición de inicio, T0, se dispare 187 veces). Para ello, se utiliza el lock proporcionado por el `Monitor` (mediante `getInvariantLock()`) y se espera a que se notifique el cumplimiento de dicha condición.

6. **Finalización de la simulación:** Una vez alcanzada la condición (186 invariantes completados), se detiene el Scheduler y se cierra inmediatamente el pool de hilos. Seguidamente, se mide y registra el tiempo total de ejecución.

7. **Impresión de resultados:** Se muestran en la salida estándar los tokens finales presentes en cada una de las `Places` y, dependiendo de la política utilizada, se imprimen los contadores asociados a las transiciones relevantes (por ejemplo, los contadores de reservas superiores, inferiores, confirmadas y canceladas). Además, se calculan y muestran algunos invariantes derivados de estos contadores.

8. **Cierre del Logger:** Finalmente, se registra el final de la simulación y se cierra el Logger, liberando los recursos asociados.

La clase `Main` orquesta la puesta en marcha del sistema, integrando la construcción del modelo, la configuración de la concurrencia y la coordinación mediante el Monitor y el Scheduler, y finalmente, la recogida de resultados para su análisis. Esto demuestra la integración completa del modelo teórico de la red de Petri con la implementación práctica en Java, cumpliendo con los requerimientos del proyecto.

14 Análisis de los Resultados

Se realizaron diversas pruebas modificando los tiempos de demora en las transiciones para evaluar el impacto de las políticas **Balanced** y **Priority**. A continuación, se presentan los resultados obtenidos para cada configuración:

14.1 Caso 1: Todas las transiciones con 5 ms

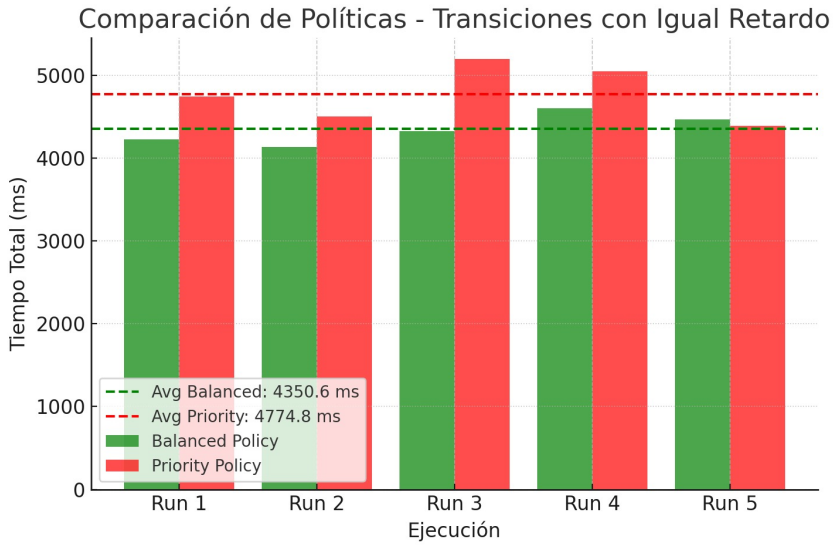


Figure 1: Resultados del Caso 1.

En este caso, todas las transiciones tienen el mismo tiempo de demora (**5 ms**), lo que significa que ninguna transición representa un cuello de botella significativo. Los tiempos de ejecución son similares, aunque la política **Balanced** muestra un tiempo menor

(**4350.6 ms**) en comparación con **Priority** (**4774.8 ms**). Esto se debe a que **Priority** prioriza ciertos segmentos, lo que en este caso no aporta una ventaja significativa.

14.2 Caso 2: Demora en el pago

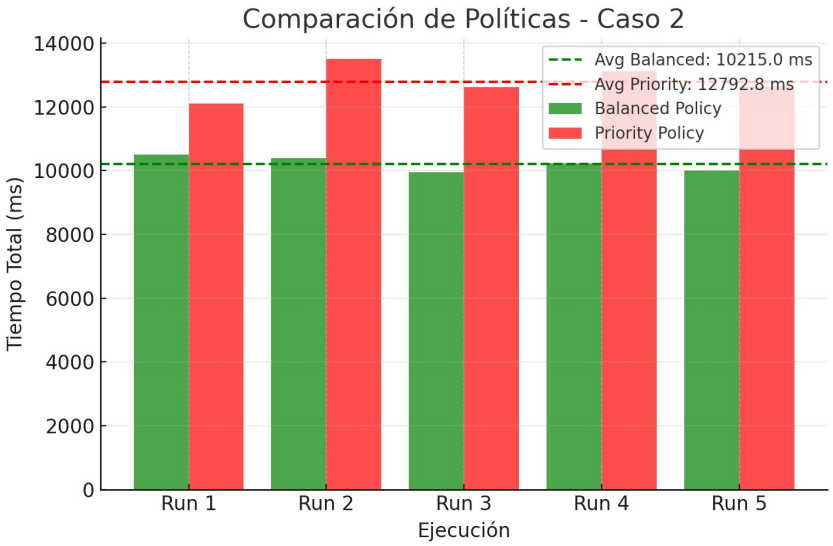


Figure 2: Resultados del Caso 2.

En esta configuración, la transición **Pago** (segmento D) tiene un tiempo de espera mayor. La política **Priority** favorece **Pago** con un **80%** de probabilidad, lo que incrementa la espera general. Como resultado, el tiempo de ejecución de **Priority** (**12792.8 ms**) es mayor que el de **Balanced** (**10215 ms**), ya que la estrategia prioritaria selecciona mayormente un camino más lento.

14.3 Caso 3: Demora en Cancelación y Confirmación de Pago

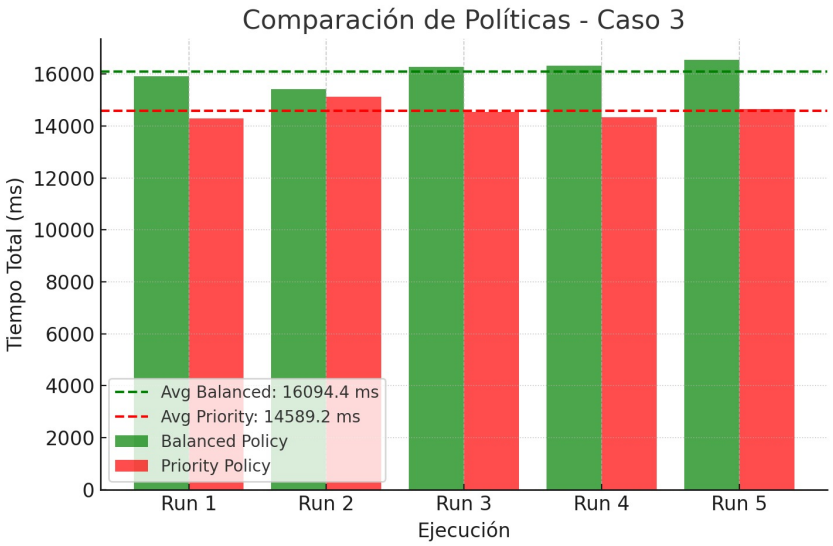


Figure 3: Resultados del Caso 3.

En esta configuración, las transiciones **Cancelación** (segmento E) y **Confirmación de Pago** (segmento F) tienen **50 ms** de demora, mientras que las demás solo **5 ms**. La política **Priority** evita mayormente **Cancelación** (20% de probabilidad) pero no puede evitar **Confirmación de Pago**, ya que es una transición de cierre del sistema.

Dado que **Confirmación de Pago** se encuentra en el segmento final, su impacto afecta a ambas políticas por igual. Sin embargo, al reducir el paso por **Cancelación**, **Priority** logra un mejor tiempo de ejecución (**14589.2 ms**) en comparación con **Balanced** (**16094.4 ms**).

14.4 Caso 4: Demora en Reserva Superior y Confirmación de Pago

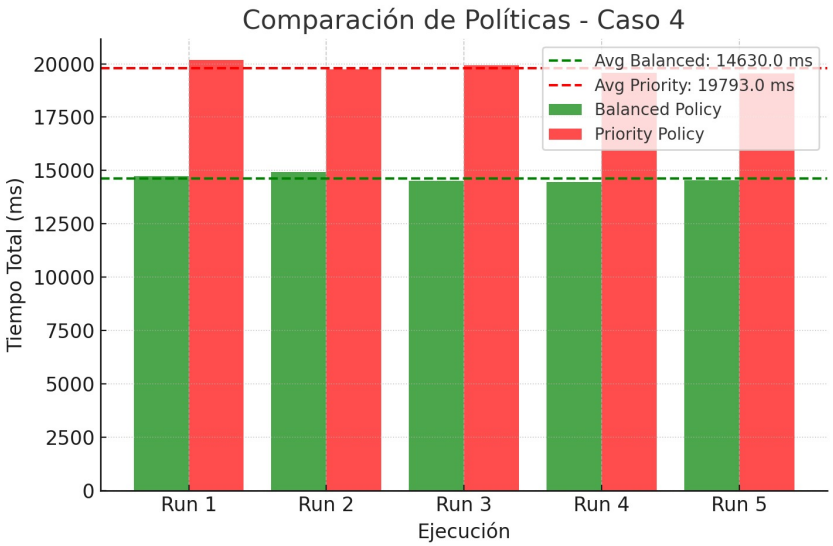


Figure 4: Resultados del Caso 4.

Las transiciones **Reserva Superior** (segmento B) y **Confirmación de Pago** (segmento F) tienen **50 ms** de demora. **Priority** favorece **B** con **75%** de probabilidad, lo que incrementa la espera en este segmento. Como resultado, el tiempo de ejecución con **Priority** (**19793 ms**) es significativamente mayor que con **Balanced** (**14630 ms**), ya que la estrategia prioritaria no puede evitar los segmentos más lentos.

14.5 Caso 5: Demora en Ingreso a Sala de Espera

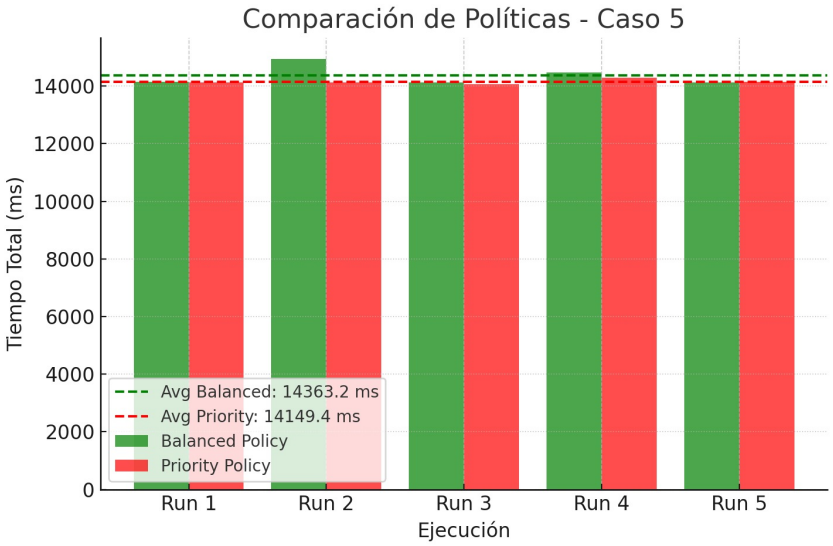


Figure 5: Resultados del Caso 5.

Aquí, la transición **Ingreso a Sala de Espera** (segmento A) tiene **50 ms** de demora, afectando de manera uniforme a todo el sistema. Esto hace que no haya una gran diferencia entre las políticas, con **Balanced** obteniendo **14363.2 ms** y **Priority 14149.4 ms**. Dado que **Ingreso a Sala de Espera** es el primer paso del sistema, ninguna política puede optimizar su impacto.

14.6 Caso 6: Demora en Reserva Inferior y Reserva Superior

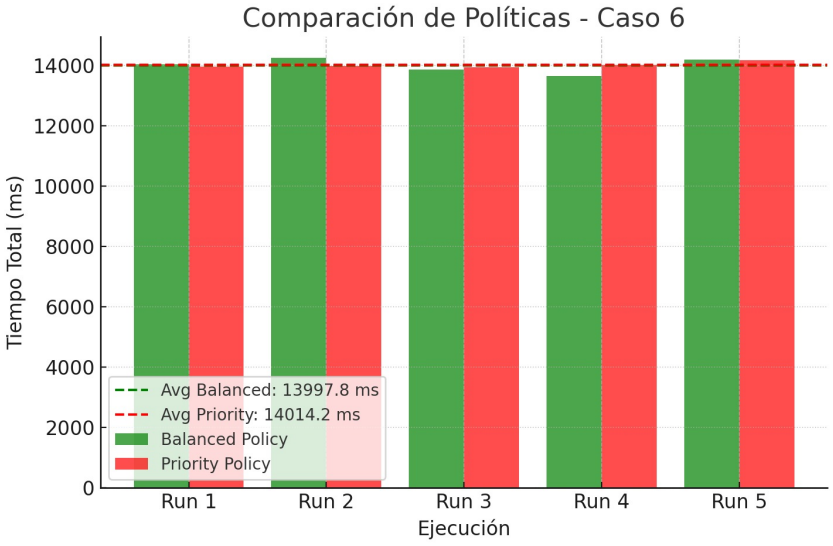


Figure 6: Resultados del Caso 6.

Las transiciones **Reserva Inferior** (segmento B) y **Reserva Superior** (segmento C) tienen **50 ms** de demora. Aunque **Priority** favorece **B** con **75%**, la demora es similar

en ambos caminos, lo que hace que el tiempo de ejecución sea casi igual entre ambas políticas (**Balanced**: **13997.8 ms**, **Priority**: **14014.2 ms**).

14.7 Caso 7: Demora en Reserva Inferior y Cancelación

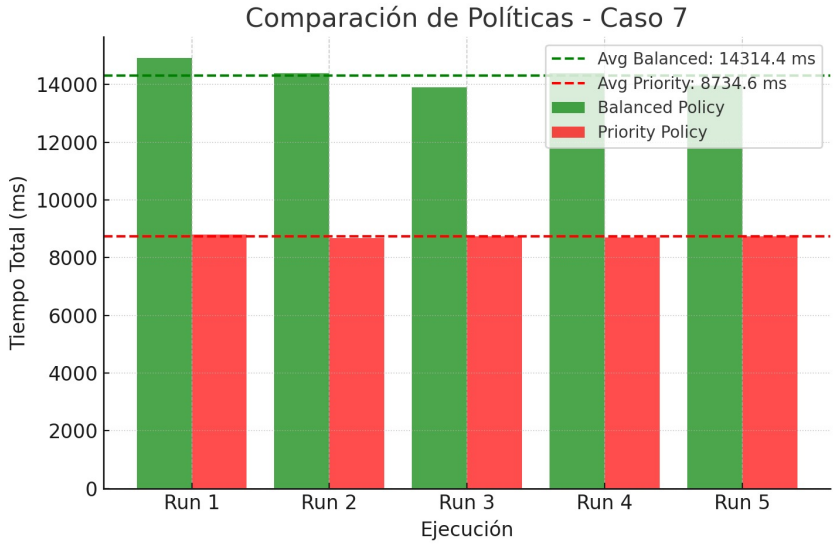


Figure 7: Resultados del Caso 7.

En esta configuración, **Reserva Inferior** (segmento B) y **Cancelación** (segmento E) tienen **50 ms** de demora. **Priority** evita en gran medida **Cancelación**, favoreciendo **B** y **D** en su lugar. Esto hace que **Priority** sea considerablemente más eficiente (**8734.6 ms**) que **Balanced** (**14314.4 ms**), ya que minimiza la cantidad de ejecuciones que atraviesan **Cancelación**, reduciendo el impacto del tiempo de espera.