Programación Concurrente

Trabajo Práctico Final

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

Realizado por:

- *Landaeta, Ezequiel* *41092007* *Ing. en Computación*
- *Olocco, Laureano* *40522648* *Ing. en Computación*

Profesor:

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

Introducción

La programación concurrente representa uno de los paradigmas fundamentales en el desarrollo de software moderno, permitiendo la ejecución simultánea de múltiples procesos y maximizando el aprovechamiento de los recursos computacionales disponibles. En entornos donde se requiere gestionar múltiples tareas interrelacionadas, como en sistemas de reservas o servicios al cliente, la concurrencia se vuelve no solo ventajosa sino necesaria.

El presente trabajo tiene como objetivo principal implementar un sistema de simulación para una agencia de viajes utilizando Redes de Petri como herramienta formal de modelado. Las Redes de Petri, desarrolladas por Carl Adam Petri en la década de 1960, ofrecen un marco matemático y gráfico para la representación de sistemas concurrentes, permitiendo visualizar la interacción entre procesos y analizar propiedades críticas como la ausencia de bloqueos o la conservación de recursos.

Nuestra implementación se centra en el desarrollo de un monitor concurrente en Java que controla la ejecución de una Red de Petri temporizada, representando las operaciones típicas de una agencia de viajes: ingreso de clientes, gestión de reservas por agentes de diferentes niveles, procesamiento de pagos y finalización de transacciones. El sistema contempla aspectos clave como la exclusión mutua en el acceso a recursos compartidos, la sincronización entre procesos y la implementación de diferentes políticas para la resolución de conflictos.

La incorporación de semántica temporal a transiciones específicas dentro de la red permite simular con mayor precisión los tiempos de operación reales, añadiendo una dimensión adicional al análisis del comportamiento del sistema. Esta característica resulta particularmente relevante en contextos de servicios donde los tiempos de respuesta constituyen un factor crítico en la experiencia del usuario.

El trabajo aborda tanto el análisis teórico de la Red de Petri diseñada, verificando propiedades como vivacidad, seguridad y ausencia de bloqueos, como su implementación práctica mediante programación orientada a objetos. Se desarrollaron y compararon dos políticas diferentes para la resolución de conflictos: una política balanceada que distribuye equitativamente la carga entre los procesos, y una política priorizada que favorece ciertos tipos de transiciones según criterios específicos.

Los resultados de múltiples simulaciones, que completaron 186 invariantes por ejecución, permitieron evaluar la equidad y eficiencia de las políticas implementadas, así como verificar el correcto funcionamiento del sistema mediante el análisis de los invariantes de plazas y transiciones.

Este desarrollo no solo demuestra la aplicabilidad de las Redes de Petri como herramienta de modelado para sistemas concurrentes, sino que también ofrece un framework extensible que podría adaptarse a otros contextos donde la gestión de recursos compartidos y la coordinación de procesos resultan fundamentales.

Marco Teórico

Redes de Petri

Las Redes de Petri son una poderosa herramienta matemática y gráfica desarrollada para modelar y analizar sistemas que exhiben comportamientos:

- Concurrentes
- Asíncronos
- Distribuidos
- Paralelos
- Estocásticos

Componentes Fundamentales

Una Red de Petri es un grafo bipartito compuesto por:

1. **Plazas (P)**: Representan estados, condiciones o recursos del sistema. Gráficamente se simbolizan como círculos.
    a. Plaza de acción: Conjunto de plazas asociadas a los invariantes de transición que no corresponden a restricciones del sistema recursos compartidos o idle.
2. **Transiciones (T)**: Representan eventos o acciones que pueden cambiar el estado del sistema. Se simbolizan como rectángulos o barras.
3. **Arcos**: Conectan plazas con transiciones (arcos de entrada) o transiciones con plazas (arcos de salida), estableciendo relaciones causales entre estados y eventos. Cada arco puede tener un peso que indica cuántos tokens se requieren o producen.
4. **Marcado (M)**: Vector que describe la distribución de tokens (representados por puntos) en las plazas. El marcado representa el estado global del sistema en un momento específico.

Dinámica del Sistema

El comportamiento dinámico de una Red de Petri se rige por reglas de disparo de transiciones:

- Una transición está habilitada cuando todas sus plazas de entrada contienen al menos tantos tokens como indica el peso del arco correspondiente.
- Cuando una transición habilitada se dispara, consume tokens de sus plazas de entrada y produce tokens en sus plazas de salida.

Esta notación matemática permite modelar y analizar propiedades importantes como alcanzabilidad, vivacidad, acotación y conflictos en sistemas complejos.

Propiedades Fundamentales de las Redes de Petri

Las redes de Petri permiten analizar propiedades críticas de los sistemas modelados, facilitando la identificación de posibles problemas y garantizando comportamientos deseados. Entre las propiedades más importantes se encuentran:

1. Deadlock (Bloqueo)

Un deadlock ocurre cuando la red alcanza un estado donde ninguna transición puede dispararse, provocando que el sistema quede completamente bloqueado.

- Representa una situación de estancamiento total del sistema.
- Impide cualquier progreso adicional en la ejecución.
- Su detección y prevención son cruciales para sistemas críticos.
- Matemáticamente, corresponde a un marcado donde ninguna transición está sensibilizada.

2. Vivacidad (Liveness)

La vivacidad garantiza la posibilidad de progreso continuo del sistema. Una red es viva cuando:

- Todas las transiciones tienen la posibilidad de dispararse eventualmente desde cualquier marcado alcanzable.
- No existen partes del sistema que queden permanentemente inactivas.
- Asegura que el sistema nunca llegará a un estado de bloqueo permanente.

3. Seguridad (Safeness)

La seguridad establece límites en la acumulación de recursos. Una red es segura cuando:

- Cada plaza contiene como máximo un token en cualquier marcado alcanzable
- Previene la sobrecarga de recursos en el sistema
- Facilita la implementación del sistema en hardware digital (donde los estados binarios son naturales)

4. Otras Propiedades Relevantes

- **Acotación (Boundedness)**: Limita el número máximo de tokens en cualquier plaza

- **Reversibilidad**: Capacidad de volver al estado inicial desde cualquier estado alcanzable
- **Persistencia**: Garantiza que las transiciones habilitadas no se deshabilitarán por el disparo de otras transiciones

Estas propiedades permiten verificar formalmente el comportamiento de sistemas complejos antes de su implementación, reduciendo errores y mejorando la fiabilidad.

Disparos en una red de Petri

En una red de Petri, los disparos representan eventos que desencadenan transiciones de estado dentro del sistema modelado. Cada transición está conectada a lugares mediante arcos dirigidos, y estos lugares contienen *tokens* que simbolizan recursos o condiciones necesarias para que ocurra una transición.

Para que una transición pueda dispararse, debe estar habilitada, lo cual generalmente implica que sus lugares de entrada contengan una cantidad mínima de tokens. Al producirse el disparo, se consumen tokens de esos lugares de entrada y se generan nuevos tokens en los lugares de salida. Este proceso modifica el estado del sistema y permite que el modelo evolucione conforme a las reglas estructurales de la red.

Representación matemática

El comportamiento dinámico de una red de Petri puede analizarse mediante el *grafo de marcado*, una representación gráfica que ilustra los distintos estados (o marcados) del sistema y las transiciones posibles entre ellos. En este grafo, cada nodo representa un marcado, es decir, una configuración específica de tokens en los lugares de la red, mientras que las aristas indican las transiciones que permiten pasar de un estado a otro.

Otra forma de describir el flujo de la red es mediante la *matriz de incidencia*, donde las filas corresponden a los lugares y las columnas a las transiciones. En esta matriz, los valores negativos representan tokens que se eliminan de un lugar durante un disparo, y los valores positivos indican tokens que se añaden.

El efecto de un disparo puede calcularse utilizando la siguiente expresión:
marcado final = marcado actual + (matriz de incidencia × vector de transición).
El vector de transición especifica cuántas veces se dispara cada transición. Si al realizar esta operación algún valor de marcado final resulta negativo, se concluye que la transición no estaba habilitada.

Esta representación matemática permite evaluar condiciones de habilitación y analizar el efecto de los disparos en la red, facilitando así el modelado de sistemas concurrentes y distribuidos de forma precisa y sistemática.

Redes de Petri Temporizadas

Las *Redes de Petri Temporizadas* (Time Petri Nets, TPN) son una extensión de las redes de Petri clásicas que incorporan el tiempo como un parámetro esencial en el comportamiento de las transiciones. Esta ampliación permite modelar y analizar sistemas donde el factor temporal es crítico, como los sistemas en tiempo real, procesos de manufactura, redes de comunicación, entre otros entornos dinámicos que requieren cumplir con restricciones temporales específicas.

Incorporación del tiempo

En las TPN, cada transición puede tener asociado un **intervalo de tiempo** que define cuándo puede o debe dispararse una vez habilitada. Este intervalo está determinado por dos parámetros:

- **Retraso mínimo (α):** Es el tiempo mínimo que debe transcurrir desde que una transición se habilita hasta que puede dispararse.

- **Retraso máximo (β):** Es el tiempo máximo durante el cual una transición puede permanecer habilitada antes de que deba dispararse. Si no se dispara dentro de este plazo, pierde su habilitación.

Reglas de funcionamiento

El comportamiento de las redes de Petri temporizadas se rige por las siguientes reglas:

1. **Habilitación:** Una transición se habilita (o se sensibiliza) cuando todos sus lugares de entrada contienen la cantidad necesaria de tokens.

2. **Espera mínima (α):** Una vez habilitada, la transición debe esperar al menos un tiempo antes de poder dispararse.

3. **Límite máximo (β):** La transición debe dispararse dentro del tiempo β tras haber sido habilitada. Si no se dispara dentro de este intervalo, pierde su habilitación y debe esperar a habilitarse nuevamente.

4. **Disparo:** Al dispararse, la transición consume tokens de los lugares de entrada y genera tokens en los lugares de salida, del mismo modo que en las redes de Petri tradicionales.

Monitores: Mecanismo de Control de Concurrencia

Un **monitor** es un mecanismo de software de alto nivel diseñado para gestionar la concurrencia y coordinar el acceso a recursos compartidos reutilizables. Encapsula tanto los **datos** como los **procedimientos** necesarios para controlar el acceso a un recurso o grupo de recursos compartidos por múltiples hilos de ejecución. En términos simples, un monitor puede entenderse como una instancia de una clase que puede ser utilizada de forma segura por varios hilos concurrentes.

Componentes de un monitor

El código de un monitor se compone de dos partes fundamentales:

- **Algoritmo de manejo del recurso y sincronización:** Define cómo se manipulan los recursos compartidos.

- **Mecanismo de asignación y orden de acceso:** Establece el orden en que los procesos pueden acceder al recurso y cómo se sincronizan entre sí.

Propiedades clave

- **Exclusión mutua:** Todos los métodos definidos dentro de un monitor se ejecutan en exclusión mutua, es decir, solo un hilo puede ejecutar un método del monitor a la vez.

- **Sincronización por variables de condición:** Cada método puede incluir variables de condición bajo las cuales los hilos deben bloquearse hasta que sea seguro continuar. Una vez que las condiciones se cumplen, se notifica a los hilos en espera para que reanuden su ejecución.

Prioridad y equidad

Para evitar la **postergación indefinida** (starvation), el monitor debe garantizar que, los procesos que ya están esperando, tengan prioridad sobre los nuevos que intentan ingresar. De lo contrario, los procesos recién llegados podrían adelantarse continuamente, impidiendo que los procesos en espera accedan al recurso.

Control mediante colas

El control de acceso al monitor se gestiona mediante una **cola asociada**:

- Cuando un hilo está ejecutando un método del monitor, cualquier otro hilo que intente entrar será bloqueado y colocado en la cola.

- Cuando el hilo activo termina su ejecución dentro del monitor, se desbloquea al siguiente hilo en la cola (el que esté al frente).

- Si no hay procesos en espera, el monitor queda libre para ser utilizado por nuevos hilos.

Políticas de concurrencia

Las políticas del monitor definen la **secuencia de acceso** a los recursos compartidos, estableciendo reglas de exclusión mutua y sincronización, basadas en criterios de prioridad, permitiendo así que hilos con tareas más críticas tengan acceso preferente.

Desarrollo

Se desarrolló una implementación en Java de una red de Petri que modela el funcionamiento de una agencia de viajes, utilizando un monitor como mecanismo de control de concurrencia. Esta solución permite gestionar de manera segura el acceso compartido a los recursos del sistema, respetando las restricciones propias del modelo. Además, se incorporaron políticas de resolución de conflictos que afectan el comportamiento de la red, las cuales fueron analizadas en cuanto a su equidad y eficiencia.

Requerimientos del proyecto

Para alcanzar los objetivos propuestos, se definieron los siguientes requerimientos:

- Implementar la red de Petri respetando su estructura y verificando sus propiedades fundamentales.

- Determinar la cantidad de hilos necesarios para lograr la ejecución del sistema con el mayor grado de paralelismo posible.

- Incorporar semántica temporal a las transiciones T1, T4, T5, T8, T9 y T10, considerando restricciones de tiempo mínimo y máximo.

- Desarrollar dos políticas distintas para la resolución de conflictos:

    - Una política balanceada, que distribuya equitativamente la carga entre los procesos.

    - Una política prioritaria, que otorgue preferencia a ciertos procesos según criterios definidos.

- Modelar el sistema mediante programación orientada a objetos, haciendo uso de un monitor de concurrencia para garantizar la exclusión mutua y la sincronización de hilos.

- Ejecutar múltiples simulaciones, completando 186 invariantes por ejecución, con el fin de:

    - Evaluar la equidad de la política implementada respecto del balance de carga sobre los invariantes.

    - Determinar la cantidad de ocurrencias de cada tipo de invariante.

Verificación y análisis

- Mostrar e interpretar los invariantes de plazas y transiciones definidos en la red.

- Verificar el cumplimiento de los invariantes de plazas luego de cada disparo, garantizando la conservación de tokens.

- Analizar un archivo de registro de las transiciones disparadas para comprobar el cumplimiento de los invariantes de transición al finalizar la ejecución.

Análisis de la Red de Petri

Se modeló la red de Petri (Fig. 1) utilizando la herramienta de simulación PIPE (Platform Independent Petri net Editor), con el objetivo de analizar y verificar las propiedades estructurales y de comportamiento del sistema representado.

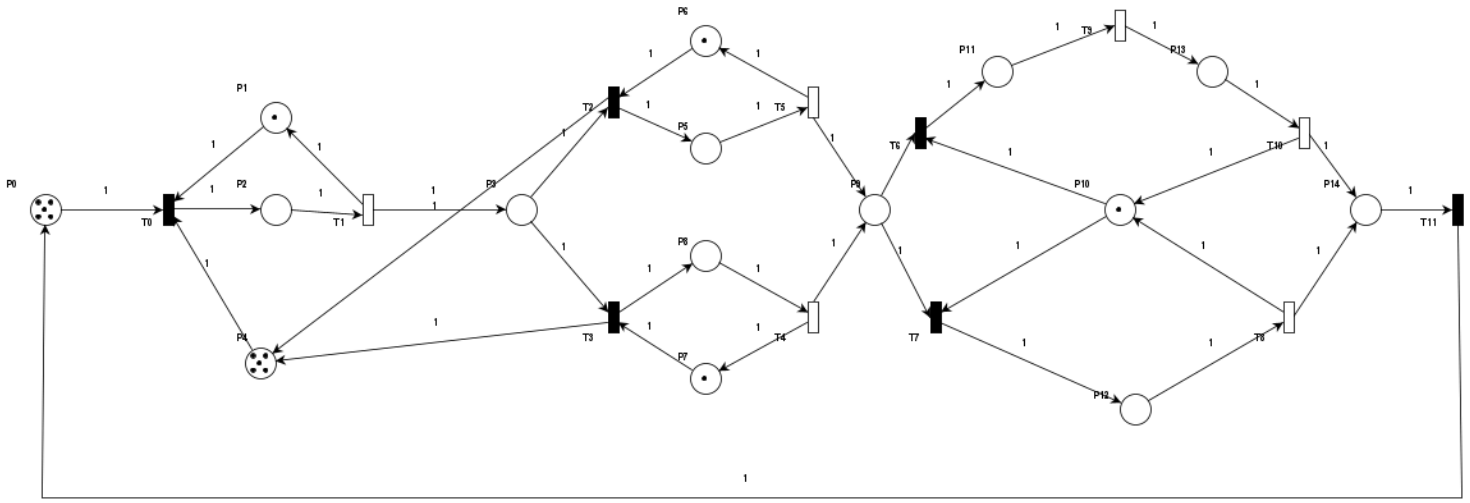


Fig. 1: Red de Petri modelando una agencia de viajes

La utilización de esta herramienta permitió llevar a cabo un análisis detallado de las propiedades de la red. Dicho análisis se centró en las siguientes propiedades fundamentales, previamente abordadas en el marco teórico:

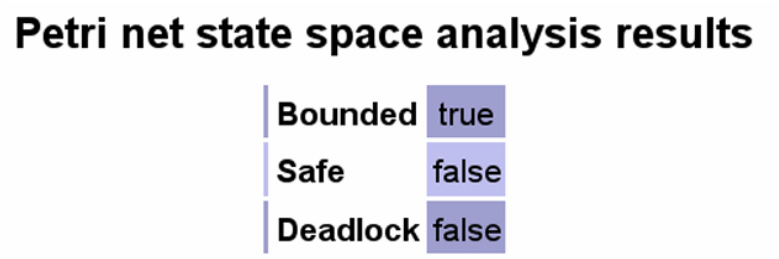


Fig. 2: Propiedades de la red de Petri brindadas por Pipe.

En el caso particular de esta red, se puede observar que no es segura, no presenta situaciones de bloqueo (deadlock) y se encuentra en un estado de vivacidad. Las dos primeras características fueron verificadas mediante el uso de la herramienta PIPE (ver Fig. 2).

La red no es segura debido a la posibilidad de que ciertos lugares contengan más de un token. Por ejemplo, las plazas P0 y P4 pueden acumular múltiples tokens bajo el marcado inicial m_0 = {5,1,0,0,5,0,1,1,0,0,1,0,0,0,0}.

Por otra parte, ninguno de los marcados alcanzables conduce la red a un estado en el que no sea posible disparar alguna transición, lo que indica la ausencia de deadlocks. Asimismo, todas las transiciones conservan la posibilidad de ser disparadas en algún momento del comportamiento del sistema, lo cual demuestra que la red está viva.

Análisis de los invariantes de la Red

Invariantes de transición

Un invariante de transición es un vector compuesto por transiciones que, al ser disparadas en una secuencia específica, devuelven a la red de Petri a un estado equivalente al estado inicial. Esto implica que, luego de ejecutar todas las transiciones que conforman el invariante, el marcado de la red no se ve alterado.

En este caso particular, se identificaron cuatro invariantes de transición (ver Fig. 3), definidos como:

- **IT1** = {T0, T1, T3, T4, T7, T8, T11}

- **IT2** = {T0, T1, T3, T4, T6, T9, T10, T11}
- **IT3** = {T0, T1, T2, T5, T7, T8, T11}
- **IT4** = {T0, T1, T2, T5, T6, T9, T10, T11}

Cada uno de estos invariantes representa un trayecto dentro del sistema modelado, desde el ingreso de una persona a la agencia de viajes, pasando por una serie de acciones, hasta su egreso.

El invariante **IT1** modela el escenario en el cual un cliente realiza una reserva con un agente de nivel inferior y el pago es rechazado, resultando en la cancelación de la reserva y la salida del cliente del sistema.

Por su parte, **IT2** describe una situación similar, pero en este caso el pago es aceptado, y la reserva queda confirmada.

Los invariantes **IT3** e **IT4** representan trayectos equivalentes a **IT1** e **IT2**, respectivamente, pero involucran reservas gestionadas por un agente de nivel superior.

En todos los casos, el token que representa al cliente transita por el sistema realizando las operaciones correspondientes, y retorna finalmente a la plaza inicial. Como resultado, el disparo de las transiciones asociadas a los distintos itinerarios mantiene inalterado el estado global de la red, cumpliendo así con la definición de invariante de transición

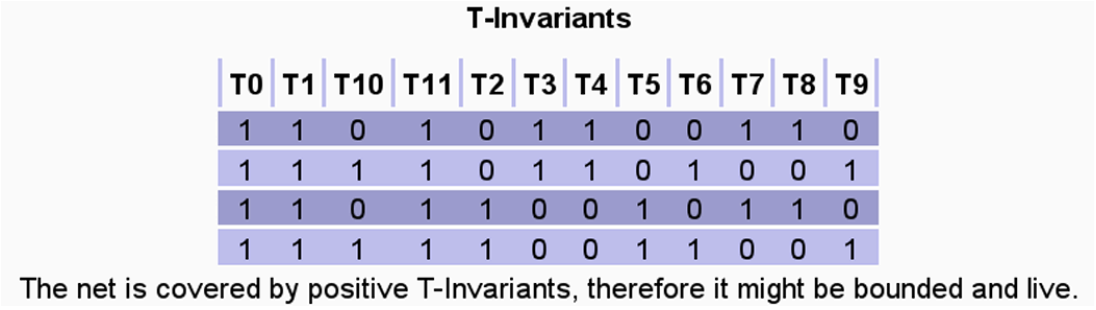
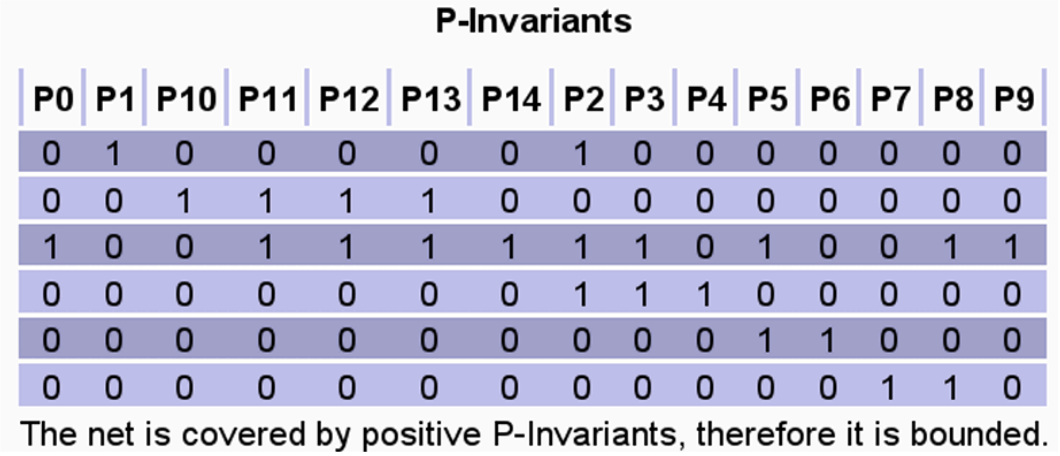


Fig. 3: Invariantes de transición de la RdP.

Invariantes de plaza

Un **invariante de plaza** es un vector que representa una propiedad de conservación de tokens en un conjunto determinado de lugares de la red. Esta propiedad implica que la **suma total de tokens** en dichos lugares se mantiene constante a lo largo de todas las configuraciones posibles que puede alcanzar la red, sin importar el orden o la cantidad de veces que se disparen las transiciones.

En el caso de la red analizada, se identificaron **seis invariantes de plaza** (ver Fig. 4).



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 P-Invariants, therefore it is bounded.

Fig. 4: Invariantes de plaza de la Red de Petri.

Las ecuaciones correspondientes a los **invariantes de plaza** identificados en la red son las siguientes:

- **IP1:** M(P1) + M(P2) = 1
- **IP2:** M(P10) + M(P11) + M(P12) + M(P13) = 1
- **IP3:** M(P0) + M(P11) + M(P12) + M(P13) + M(P14) + M(P2) + M(P3) + M(P5) + M(P8) + M(P9) = 5
- **IP4:** M(P2) + M(P3) + M(P4) = 5
- **IP5:** M(P5) + M(P6) = 1
- **IP6:** M(P7) + M(P8) = 1

En esta red, cada uno de estos invariantes representa un conjunto de lugares en los cuales la cantidad total de tokens permanece constante a lo largo de la ejecución del sistema.

En el caso del **IP1**, las plazas P1 y P2 se relacionan con un recurso compartido. Cuando hay un token en P1, significa que el recurso está disponible y, por lo tanto, no se encuentra en uso (P2 está vacía). Por el contrario, si el token se encuentra en P2, implica que el recurso está en uso y no se encuentra disponible en P1. Esta exclusión mutua da lugar a la ecuación:
IP1 = M(P1) + M(P2) = 1

El **IP2** involucra a la plaza P10, que representa un recurso compartido, y a las plazas P11, P12 y P13, que modelan las distintas etapas del proceso de pago y confirmación o cancelación de la reserva. Cuando el recurso está siendo utilizado en alguna de estas etapas, el token se desplaza desde P10 hacia las plazas asociadas

al proceso en curso. Además, si se confirma la reserva, no se produce su cancelación y viceversa, lo que asegura la validez de la ecuación:
IP2 = M(P10) + M(P11) + M(P12) + M(P13) = 1

El **IP3** abarca plazas que no están asociadas a recursos compartidos y representan a las personas realizando diversas acciones dentro de la agencia. En este caso, la cantidad total de personas en el sistema está limitada a cinco, lo que se refleja en la siguiente relación de conservación:
IP3 = M(P0) + M(P11) + M(P12) + M(P13) + M(P14) + M(P2) + M(P3) + M(P5) + M(P8) + M(P9) = 5

Por su parte, las plazas que conforman el **IP4** están relacionadas con el proceso de entrada de personas a la agencia. Al igual que en el caso anterior, se mantiene un límite de cinco personas ingresando al sistema, lo que se expresa mediante:
IP4 = M(P2) + M(P3) + M(P4) = 5

El **IP5** involucra la plaza P6, que representa un recurso compartido (un agente disponible), y la plaza P5, donde una persona está siendo atendida. Este invariante refleja que un agente solo puede atender a una persona a la vez: si el token está en P6, el agente está libre; si está en P5, el agente se encuentra ocupado. La relación se formaliza como:
IP5 = M(P5) + M(P6) = 1

Finalmente, el **IP6** es análogo al anterior, y también representa una exclusión mutua entre una persona que está por ser atendida y el recurso disponible para su atención. Se expresa como:
IP6 = M(P7) + M(P8) = 1

Definición de estados y eventos del sistema

Para la red de Petri planteada en este caso, los diferentes estados posibles del sistema pueden describirse mediante la siguiente tabla, la cual detalla el significado de cada plaza y su función dentro del modelo. Esta clasificación facilita la interpretación del comportamiento del sistema, así como el análisis del flujo de tokens en los distintos procesos representados.

Plaza	Estado
P0	Buffer de ingreso a la agencia
P1	Lugar para ingresar a la agencia (Recurso compartido)
P2	Persona preparada para entrar
P3	Persona en sala de espera
P4	Recurso compartido
P5	Persona gestionando la reserva
P6	Agente de gestión de reservas (Recurso compartido)
P7	Agente de gestión de reservas (Recurso compartido)
P8	Persona gestionando la reserva
P9	Persona con reserva en espera de agente
P10	Agente de gestión (Recurso compartido)
P11	Persona con pago confirmado
P12	Persona con pago cancelado
P13	Pago de la reserva
P14	Persona lista para retirarse

Tabla 1: Estados del sistema

Por otro lado, los distintos eventos que pueden ocurrir en el sistema fueron detallados en la Tabla 2, donde se especifica su relación con las transiciones de la red y el impacto que tienen sobre el marcado.

Transición	Evento
T0	Ingreso del buffer
T1	Ingreso a la sala de espera
T2	Gestión de la reserva
T3	Gestión de la reserva
T4	Paso a la sala de espera del agente
T5	Paso a la sala de espera del agente
T6	Confirmación de pago
T7	Cancelación del pago
T8	Paso a la salida
T9	Pago de la reserva
T10	Paso a la salida
T11	Salida de la agencia

Tabla 2: Eventos del sistema.

Cantidad máxima de hilos en simultáneo

Para realizar el cálculo de la **máxima cantidad de hilos activos en simultáneo**, se utilizó un algoritmo (ver Referencias [1]). En primer lugar, se identificaron los **invariantes de transición**:

- **IT1** = {T0, T1, T3, T4, T7, T8, T11}
- **IT2** = {T0, T1, T3, T4, T6, T9, T10, T11}
- **IT3** = {T0, T1, T2, T5, T7, T8, T11}
- **IT4** = {T0, T1, T2, T5, T6, T9, T10, T11}

A partir de estos, se determinaron las **plazas involucradas en cada invariante**:

- **PI1** = {P0, P1, P2, P3, P7, P8, P10, P12, P14}
- **PI2** = {P0, P1, P2, P3, P7, P8, P9, P10, P11, P13, P14}
- **PI3** = {P0, P1, P2, P3, P5, P6, P9, P10, P12, P14}
- **PI4** = {P0, P1, P2, P3, P5, P6, P9, P10, P11, P13, P14}

Luego, se extrajeron las **plazas de acción** de cada subconjunto:

- **PA1** = {P2, P3, P8, P9, P12, P14}
- **PA2** = {P2, P3, P8, P9, P11, P13, P14}
- **PA3** = {P2, P3, P5, P9, P12, P14}
- **PA4** = {P2, P3, P5, P9, P11, P13, P14}

Con esta información, se conformó el **conjunto total de plazas de acción**:

- **PA** = {P2, P3, P5, P8, P9, P11, P12, P13, P14}

Luego se calcularon todos los **marcados posibles** para las plazas de acción. De este análisis se obtuvo la **máxima cantidad de hilos en ejecución simultánea** para la red.

Finalmente, se obtiene el conjunto de todos los marcados posibles para todas las plazas de acción (ver Anexo 1), donde la máxima cantidad de hilos activos en

simultáneo **corresponde al mayor valor de las sumas de todas las marcas para cada marcado posible**, siendo en este caso igual a **5**.

Por otra parte, para definir la **cantidad total de hilos a utilizar**, se dividió el sistema en segmentos y se asignó a cada uno de ellos una cantidad de hilos según el siguiente criterio:

- **Caso 1:** Si un invariante de transición presenta un **conflicto** con otro invariante, se asigna **un hilo encargado de ejecutar las transiciones previas al conflicto**, y luego **un hilo adicional por cada invariante** involucrado.

- **Caso 2:** Si un invariante de transición presenta un **join** con otro invariante, después del join deben existir **tantos hilos como tokens simultáneos pueda haber en la plaza**, encargados de las transiciones posteriores, ya que solo existe un único camino posible.

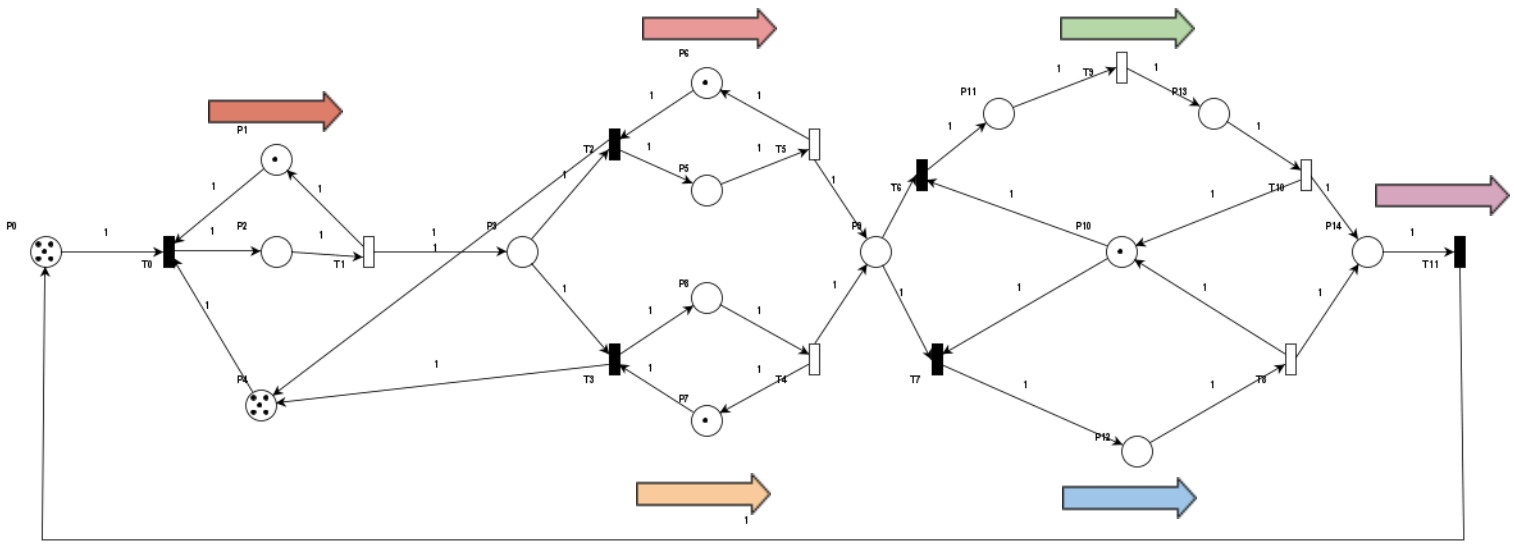


Fig. 5: Hilos por segmentos en la red de Petri

Aplicando este criterio, se definieron **6 segmentos**, con la siguiente cantidad de hilos asignados:

- **S1 = {T0, T1} → 1 hilo**
- **S2 = {T2, T5} → 1 hilo**
- **S3 = {T3, T4} → 1 hilo**
- **S4 = {T6, T9, T10} → 1 hilo**
- **S5 = {T7, T8} → 1 hilo**
- **S6 = {T11} → 1 hilos**

Por lo tanto, **la cantidad total de hilos del sistema es 6**. La red de Petri dividida por secciones puede observarse en la **Figura 5**.

Implementación de la red en Java

Para la implementación del sistema en Java, se hizo uso del paradigma de programación orientada a objetivos, donde se separa a las clases en 6 paquetes.

- **Paquete agencia**: Compuesto por las clases Agencia y Procesos.
- **Paquete logger**: Compuesto por la clase Log.
- **Paquete matriz**: Compuesto por la clase Matriz.
- **Paquete monitor**: Compuesto por las clases Monitor y Política, además de la interfaz MonitorInterface.
- **Paquete rdp**: Compuesto por la clase RedDePetri.
- **Paquete threadfactory**: Compuesto por la clase MyThreadFactory.

A continuación se describirán cada uno de los paquetes con sus respectivas clases y métodos.

Paquete agencia

Clase Agencia

La clase Agencia modela una entidad encargada de registrar e informar disparos de transiciones en un sistema. Lleva un conteo individual de disparos para cada una de las 12 transiciones disponibles y permite verificar si se ha alcanzado un límite total de disparos (186). Además, proporciona métodos sincronizados para incrementar el contador de una transición específica y consultar el estado de disparos acumulados, asegurando un acceso seguro en entornos concurrentes.

Métodos:

- **Agencia()**: Constructor de la clase. Inicializa el arreglo de disparos por transición con ceros y establece el contador de disparos finales.

- **synchronized boolean limiteAlcanzado()**: Verifica si la transición final (índice 11) alcanzó o superó el límite total de disparos permitidos. Retorna true si se superó el límite, false en caso contrario.

- **synchronized void transicionDisparada(int t)**: Incrementa en uno el contador de disparos de la transición especificada por el índice t.

- **int[] getDisparoPorTransicion()**: Retorna el arreglo que contiene el número de disparos realizados por cada transición.

Clase Proceso

La clase Proceso representa un hilo de ejecución concurrente dentro de la simulación, encargado de disparar transiciones en una Red de Petri mediante el uso del Monitor. Cada proceso está asociado a una Agencia, tiene asignadas ciertas transiciones que puede disparar, y su objetivo es colaborar en alcanzar un número máximo de disparos (186 en total). La clase implementa la interfaz Runnable para poder ser ejecutada como un hilo.

Métodos

- **Proceso(Agencia agencia, int[] transiciones, String nombreProceso)**: Constructor del proceso. Inicializa la agencia asociada, copia el arreglo de transiciones permitidas para este proceso, y asigna un nombre identificador.

- **void run()**: Método principal del hilo. Si la transición inicial es 0, ejecuta el método entradaAgencia(), de lo contrario ejecuta realizarAccion(). Maneja posibles interrupciones del hilo.

- **private void entradaAgencia()**: Método que se ejecuta cuando el proceso se encarga de las reservas iniciales (transición 0). Dispara transiciones mientras el total de reservas no alcance el máximo permitido (186). Incrementa el contador de reservas y registra cada disparo en la agencia correspondiente.

- **private void realizarAccion()**: Método ejecutado por procesos que no manejan la transición inicial. Dispara las transiciones asignadas mientras no se haya alcanzado el límite total de disparos, consultando este valor a través de la agencia.

- **public String getNombreProceso()**: Devuelve el nombre identificador del proceso.

Paquete logger

Clase Log

La clase Log representa un registrador centralizado que recolecta información sobre la ejecución de una Red de Petri, incluyendo transiciones disparadas, estadísticas de ejecución y finalización de invariantes. Implementa el patrón Singleton para asegurar que sólo exista una instancia del logger, y la interfaz Runnable para ejecutarse en un hilo separado. Al finalizar, escribe un archivo de texto con todos los datos recolectados durante la simulación

Métodos

- **Log(String fileName)**: Constructor privado del logger. Inicializa el archivo de log donde se almacenará toda la información de ejecución. Solo puede ser llamado internamente por el método getInstancia().

- **static Log getInstancia()**: Devuelve la instancia única del logger. Si aún no ha sido creada, la instancia se inicializa con el nombre de archivo "log.txt".

- **void run()**: Método principal del hilo. Espera a que la ejecución se marque como finalizada (terminado = true) y, al terminar, escribe en el archivo el historial de transiciones disparadas, estadísticas por transición e información de invariantes completados.

- **void setTdisparadas(String t)**: Establece la cadena con la secuencia de transiciones disparadas (por ejemplo: "T0 T1 T3...").

- **void setDisparoPorT(int[] d)**: Asigna el arreglo que contiene la cantidad de veces que se disparó cada transición. El índice del arreglo representa el número de transición.

- **void setInvariantes(int[] i)**: Asigna el arreglo con la cantidad de veces que se completó cada uno de los invariantes definidos en la Red de Petri.

- **void setTerminado(boolean terminado)**: Marca el logger como finalizado. Esto permite que el hilo comience el proceso de escritura de resultados en el archivo de log.

Paquete matriz

Clase Matriz

La clase Matriz es una clase de utilidad que proporciona métodos estáticos para operar con matrices dentro del contexto de una Red de Petri. Incluye funciones para crear matrices de transición, realizar multiplicaciones con matrices de incidencia, y sumar marcados. Su uso es exclusivamente estático, orientado a facilitar operaciones comunes en la simulación.

Métodos

- **private Matriz()**: Constructor privado que impide la creación de instancias de la clase. Lanza una excepción si se intenta instanciarla. Se usa para reforzar su propósito como clase utilitaria.

- **static int[] crearMatrizTransicion(int t, int transiciones)**: Crea una matriz unidimensional de tamaño transiciones, con todos los valores en cero excepto un 1 en la posición t. Lanza una excepción si t es menor que 0 o mayor o igual que transiciones.

- **static int[] multiplicarMatriz(int[][] matrizIncidencia, int[] matrizTransicion)**: Multiplica una matriz de incidencia por una matriz de transición. Devuelve el resultado como un vector. Lanza excepciones si las matrices son nulas o incompatibles en dimensiones.

- **static int[] sumarMatriz(int[] matrizMarcado, int[] matrizM)**: Suma dos vectores unidimensionales de igual longitud, retornando un nuevo vector con los resultados. Lanza excepciones si las matrices son nulas o de diferentes tamaños.

Paquete monitor

Clase Monitor

La clase Monitor implementa el patrón Singleton para coordinar el acceso concurrente a una Red de Petri por parte de múltiples hilos. Su objetivo principal es asegurar la exclusión mutua durante los disparos de transiciones, manejar colas de espera para transiciones no sensibilizadas, y aplicar políticas de desbloqueo eficientes. Incluye mecanismos de sincronización con semáforos y controla la ejecución basada en tiempos para transiciones temporales. Su uso está orientado a entornos multihilo donde varios procesos deben interactuar con una única instancia de Red de Petri de forma segura.

Métodos

- **private Monitor()**: Constructor privado que inicializa semáforos, colas, la Red de Petri y la política de disparo. Solo puede ser invocado internamente por el Singleton. Garantiza que no existan múltiples instancias del monitor.

- **public static Monitor getInstanciaMonitor()**: Devuelve la única instancia del monitor utilizando el patrón Double-Check Locking. Asegura que la instancia sea única incluso en contextos multihilo.

- **public boolean fireTransition(int transicion)**: Intenta disparar una transición. Si no está sensibilizada, bloquea al hilo en la cola correspondiente hasta que sea posible dispararla. Verifica condiciones de fin de simulación y sincroniza el acceso con un mutex.

- **private boolean verificarMarcadoYTiempo(int transicion)**: Verifica si una transición está sensibilizada tanto por marcado como por tiempo (si es temporal). Si el tiempo mínimo no se ha cumplido, espera el tiempo restante.

- **private boolean puedeDispararse(int transicion)**: Devuelve true si la transición no está en espera y cumple condiciones de marcado y tiempo. Combina validaciones previas para evaluar si puede ser disparada.

- **private boolean estaAntesDeVentanaTemporal(int transicion, long tiempoActual)**: Indica si una transición temporal aún no alcanzó su tiempo mínimo de disparo (alfa). Si no es temporal, devuelve false.

- **private void esperarTiempoMinimo(int transicion, long tiempoEspera)**: Suspende el hilo por el tiempo restante necesario para que la transición temporal pueda dispararse. Libera y recupera el mutex apropiadamente.

- **private void adquirirMutex()**: Solicita el semáforo mutex para garantizar exclusión mutua. Si no está disponible, el hilo se bloquea.

- **private void liberarMutex()**: Libera el mutex. Si hay procesos en espera y se completaron los disparos, intenta despertarlos usando liberarHilos() o despertarTransicion().

- **private boolean liberarHilos()**: Recorre las colas de transiciones buscando un hilo en espera para liberarlo. Devuelve false si se liberó un hilo, true si no había hilos esperando.

- **private boolean despertarTransicion()**: Aplica una política para seleccionar una transición que tenga hilos en espera y esté sensibilizada. Si existe, libera un hilo de esa cola.

- **private int[] transicionesEnCola()**: Devuelve un arreglo binario que indica qué transiciones tienen hilos esperando en sus colas.

- **private int[] obtenerTransiciones()**: Devuelve un arreglo binario con las transiciones que están tanto sensibilizadas como con procesos en espera, haciendo una operación lógica AND entre ambas condiciones.

- **private boolean disparosCompletados()**: Devuelve true si se alcanzó el número total de disparos definido, señalando el fin de la simulación o del proceso de ejecución.

Clase Política

La clase Politica define estrategias de selección para determinar qué transición disparar en una Red de Petri cuando múltiples están sensibilizadas. Se configura con una instancia de RedDePetri y ofrece dos políticas predeterminadas: una política balanceada para distribuir equitativamente los disparos, y una política priorizada que favorece ciertos tipos de transiciones según proporciones específicas. Esta clase encapsula la lógica de decisión, permitiendo modularidad y flexibilidad en la simulación.

Métodos

- **public Politica(RedDePetri redDePetri)**: Constructor que recibe una instancia de Red de Petri y obtiene el arreglo de disparos actual. Prepara la política para operar sobre dicha red.

- **public int elegirPolitica(int[] t)**: Método principal que selecciona qué política aplicar de acuerdo al tipo configurado (balanceada o priorizada). Retorna el índice de la transición elegida. Lanza una excepción si la política no está implementada.

- **private int politicaBalanceada(int[] transiciones)**: Implementa una política que elige la transición sensibilizada con menor cantidad de disparos acumulados, favoreciendo el balance entre todas las transiciones. Recorre el arreglo de transiciones y selecciona la que haya sido disparada menos veces.

- **private int politicaProcesamientoPriorizada(int[] transiciones)**: Aplica reglas de prioridad en función de proporciones entre transiciones específicas:
    - Prioriza agentes superiores si su proporción es baja (< 75%)
    - Prioriza confirmaciones de reservas si su proporción es baja (< 80%) Si no se cumplen esas condiciones, busca entre otras transiciones no priorizadas. Retorna el índice de la transición seleccionada o 0 por defecto si ninguna aplica.

Interfaz MonitorInterface

La interfaz MonitorInterface define el contrato que deben seguir los monitores encargados de gestionar el disparo de transiciones en una Red de Petri. Su propósito es abstraer el mecanismo de disparo, permitiendo que distintas implementaciones puedan integrarse de manera intercambiable, facilitando la extensibilidad y pruebas del sistema.

Métodos

- **boolean fireTransition(int transition)**: Método que representa el intento de disparar una transición. Recibe como parámetro el índice de la transición y devuelve true si el disparo se realizó con éxito, o false en caso contrario. Su implementación puede incluir mecanismos de sincronización, espera condicional o verificación de sensibilización.

Paquete rdp

Clase RedDePetri

La clase Matriz representa una clase de utilidad para realizar operaciones matriciales dentro del contexto de una Red de Petri. Hereda de RedDePetri pero no puede ser instanciada, ya que su constructor es privado y lanza una excepción. Su función principal es proporcionar métodos estáticos para construir, multiplicar y sumar matrices que representan transiciones y marcados dentro de la simulación.

Métodos

- **Matriz()**: Constructor privado que impide la creación de instancias de la clase. Lanza una excepción AssertionError al ser invocado. Refuerza el uso exclusivo de la clase como utilidad estática.

- **static int[] crearMatrizTransicion(int t, int transiciones)**: Crea y devuelve un vector de tamaño transiciones con todos sus elementos en cero, excepto la posición t que toma el valor 1. Se utiliza para representar una transición única activada. Lanza una excepción si la posición es inválida.

- **static int[] multiplicarMatriz(int[][] matrizIncidencia, int[] matrizTransicion)**: Realiza el producto entre una matriz de incidencia y una matriz de transición. Retorna un vector resultado con el efecto de disparar una transición sobre el marcado. Valida que las dimensiones sean compatibles y que ninguna de las matrices sea nula.

- **static int[] sumarMatriz(int[] matrizMarcado, int[] matrizM)**: Suma dos vectores que representan marcados en la Red de Petri. Retorna un nuevo vector con la suma componente a componente. Lanza excepciones si alguna matriz es nula o si no tienen la misma longitud.

Paquete threadfactory

Clase MyThreadFactory

La clase MyThreadFactory implementa la interfaz ThreadFactory del paquete java.util.concurrent y representa una implementación del patrón de diseño Factory. Este patrón se utiliza para encapsular la creación de objetos —en este caso, de hilos— de forma controlada y personalizada.

Permite generar hilos con nombres específicos y un contador incremental, lo cual es útil para identificar y gestionar fácilmente los hilos dentro de un sistema concurrente o multihilo.

Métodos

- **Thread newThread(Runnable r)**: Sobrescribe el método de la interfaz ThreadFactory. Crea un nuevo hilo a partir del Runnable recibido, asignándole un nombre compuesto por el formato:

<nombre>_Thread_<contador>.
Después de crear el hilo, incrementa el contador para mantener nombres únicos en futuras invocaciones.

Clase main

La clase Main es el punto de entrada de la aplicación de la agencia. Se encarga de configurar y lanzar múltiples procesos concurrentes que interactúan con una Red de Petri a través de hilos. Cada proceso gestiona transiciones específicas, lo que permite simular una operación distribuida de la red. Además, incorpora un hilo de logging que registra los eventos relevantes de la simulación. La clase también mide el tiempo total de ejecución de la simulación para propósitos de evaluación de rendimiento.

Métodos

- **public static void main(String[] args)**: Método principal de la aplicación. Inicializa la fábrica de hilos, crea la agencia y los procesos asociados, lanza su ejecución en paralelo y calcula el tiempo total de ejecución. También inicia el hilo del logger y espera a que todos los procesos y el logger terminen.

- **private static Proceso[] crearProcesos(Agencia agencia, int[][] segmentos)**: Crea las instancias de los procesos que interactúan con la agencia. Cada proceso recibe un conjunto de transiciones (segmentos) y un identificador único. Retorna un array de procesos configurados.

- **private static void ejecutarProcesos(MyThreadFactory threadFactory, Proceso[] procesos)**: Lanza los procesos en hilos separados utilizando la fábrica de hilos. También inicia un hilo adicional para el logger y espera la finalización de todos los procesos. Una vez que terminan, notifica al logger para que finalice su trabajo y genere el informe.

A continuación puede observarse el diagrama de clases resultante de la implementación descripta

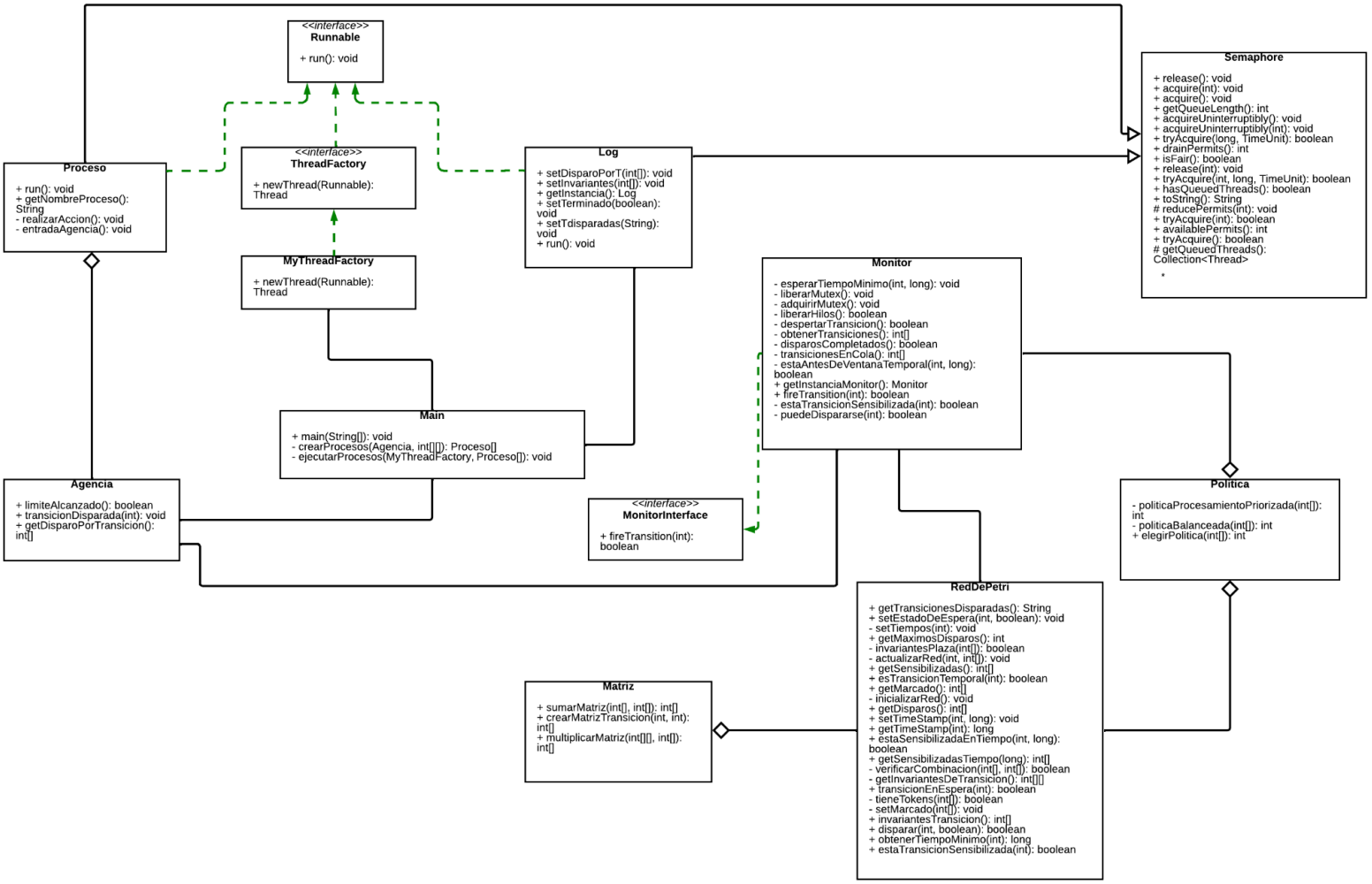


Fig. 6: Diagramas de clases del sistema

Análisis de las políticas implementadas

Política equitativa

En el caso de una política equitativa —donde tanto el agente superior como el inferior atienden la misma cantidad de personas, y se aprueba y cancela un número igual de reservas— se obtuvo el siguiente resultado:

---------- **Transiciones disparadas** ----------
Transición 0 disparada: 186 veces.
Transición 1 disparada: 186 veces.
Transición 2 disparada: 93 veces.
Transición 3 disparada: 93 veces.
Transición 4 disparada: 93 veces.
Transición 5 disparada: 93 veces.
Transición 6 disparada: 93 veces.
Transición 7 disparada: 93 veces.
Transición 8 disparada: 93 veces.
Transición 9 disparada: 93 veces.
Transición 10 disparada: 93 veces.
Transición 11 disparada: 186 veces.

---------- **Invariantes completados** ----------
Invariante 1: [0 1 3 4 7 8 11] completado: 47 veces
Invariante 2: [0 1 3 4 6 9 10 11] completado: 46 veces
Invariante 3: [0 1 2 5 7 8 11] completado: 46 veces
Invariante 4: [0 1 2 5 6 9 10 11] completado: 47 veces

Total de invariantes completados: 186

Política de procesamiento priorizada

En este escenario, se establece una política que prioriza al agente de reservas superior sobre el inferior, asignándole la creación del 75% de las reservas. Además, se confirma el 80% de las reservas generadas, mientras que el 20% restante es cancelado.
Los resultados obtenidos bajo esta política fueron los siguientes:

---------- **Transiciones disparadas** ----------
Transición 0 disparada: 186 veces.
Transición 1 disparada: 186 veces.
Transición 2 disparada: 139 veces.
Transición 3 disparada: 47 veces.
Transición 4 disparada: 47 veces.
Transición 5 disparada: 139 veces.

Transición 6 disparada: 149 veces.
Transición 7 disparada: 37 veces.
Transición 8 disparada: 37 veces.
Transición 9 disparada: 149 veces.
Transición 10 disparada: 149 veces.
Transición 11 disparada: 186 veces.

---------- **Invariantes completados** ----------
Invariante 1: [0 1 3 4 7 8 11] completado: 19 veces
Invariante 2: [0 1 3 4 6 9 10 11] completado: 28 veces
Invariante 3: [0 1 2 5 7 8 11] completado: 18 veces
Invariante 4: [0 1 2 5 6 9 10 11] completado: 121 veces

Total de invariantes completados: 186

Análisis de los resultados obtenidos

En la política equitativa implementada, se observa que todas las transiciones presentan un número similar de disparos, salvo las transiciones 2, 3, 4, 5, 6, 7, 8 y 9, que registran una menor cantidad. Este comportamiento sugiere que tanto el agente superior como el inferior están siendo utilizados de manera equitativa para la creación de reservas, al igual que las acciones de confirmación y cancelación, las cuales también se reparten de forma balanceada. Esta equidad se refleja en los invariantes de transición, los cuales se completan exactamente 186 veces, evidenciando que se alcanzó el límite máximo de disparos definidos.

En contraste, la política de procesamiento priorizado muestra una distribución desigual en los disparos de transición. Las transiciones 2 y 5 —asociadas al agente superior y a la confirmación de reservas, respectivamente— tienen una frecuencia de disparo notablemente mayor en comparación con las transiciones 3, 4, 7 y 8. Esta diferencia evidencia una preferencia por el agente superior al momento de generar reservas, y por la confirmación de las mismas por sobre su cancelación.

Asimismo, los invariantes de transición también reflejan esta priorización: algunos, como el invariante 3, se completan considerablemente más veces que otros. Sin embargo, al igual que en la política equitativa, el total de invariantes completados sigue siendo 186, lo que indica nuevamente que se alcanzó el máximo de disparos permitidos.

Cabe destacar que el sistema implementado garantiza que, independientemente de la política aplicada o del número de ejecuciones, el total de disparos se mantenga constante. Esto se logra gracias a una gestión controlada y estructurada de las transiciones, mediante políticas bien definidas que regulan el acceso y la ejecución de acciones dentro de la red de Petri. Como resultado, se asegura un comportamiento consistente y predecible del sistema a lo largo del tiempo.

Análisis de los tiempos

Ejecución N°	Política 1			Política 2		
	Caso 1	Caso 2	Caso 3	Caso 1	Caso 2	Caso 3
1	1.57s	4.29s	12.98s	1.62s	4.45s	12.80s
2	1.58s	4.26s	12.95s	1.59s	4.49s	12.83s
3	1.63s	4.24s	13.64s	1.62s	4.44s	12.82s
4	1.58s	4.25s	12.96s	1.62s	4.44s	12.81s
5	1.57s	4.23s	12.96s	1.61s	4.47s	12.80s
6	1.62s	4.25s	12.96s	1.59s	4.44s	12.80s
7	1.58s	4.21s	13.15s	1.61s	4.50s	12.83s
8	1.63s	4.25s	12.97s	1.60s	4.48s	12.82s
9	1.59s	4.24s	12.96s	1.59s	4.47s	12.80s
10	1.59s	4.24s	12.96s	1.60s	4.47s	12.79s
11	1.58s	4.25s	12.97s	1.62s	4.47s	12.82s
12	1.57s	4.23s	13.15s	1.61s	4.47s	12.79s
13	1.57s	4.29s	13.64s	1.64s	4.46s	12.81s
14	1.58s	4.29s	12.97s	1.62s	4.47s	12.82s
15	1.58s	4.28s	13.43s	1.62s	4.44s	12.84s
Promedio	**1.58s**	**4.25s**	**13.11s**	**1.61s**	**4.46s**	**12.81s**

Tabla 3: Tiempos de ejecución para cada caso

Configuración rápida

Para la configuración rápida se tiene los siguientes alfas

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
0	3	0	0	5	5	0	0	7	3	3	0

Tabla 4: Tiempos de Alfa para la configuración rápida

Donde, se tienen 4 posibles tiempos de ejecución para realizar una reserva según los diferentes invariantes de transición:

- IT1 = 3ms + 5ms + 7ms = 15ms
- IT2 = 3ms + 5ms + 3ms + 3ms = 14ms
- IT3 = 3ms + 5ms + 7ms = 15ms
- IT4 = 3ms + 5ms + 3ms + 3ms = 14ms

Política balanceada

Para la política balanceada, el cálculo del peor tiempo corresponde a:
15ms*47 + 14ms*46 + 15ms*46 + 14ms*47 = 2604ms = 2.6s

Por otra parte, el mejor tiempo puede calcularse como:
15ms*10 + 14ms*9 + 15ms*9 + 14ms*10 = 551ms

Donde, se considera la ejecución de 5 hilos en simultáneo, cantidad máxima de hilos calculada anteriormente, lo que se calculó el cociente del número total de reservas y la cantidad máxima de reservas en simultáneo y luego se distribuyó en cada invariante equitativamente.

Política priorizada

Para la política priorizada, el cálculo del peor tiempo corresponde a:
15ms*19 + 14ms*28 + 15ms*18 + 14ms*121 = 2641ms = 2.6s

El mejor tiempo puede calcularse como: 15ms*5 + 14ms*6 + 15ms*3 + 14ms*24 = 540ms

Configuración media

En el caso de la configuración media, los tiempos están distribuidos de la siguiente manera:

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
0	10	0	0	15	15	0	0	20	10	12	0

Tabla 5: Tiempos de Alfa para la configuración media

En este caso, los 4 casos diferentes según cada IT se dividen de la siguiente manera:
- IT1 = 10ms + 15ms + 20ms = 35ms
- IT2 = 10ms + 15ms + 10ms + 12ms = 47ms
- IT3 = 10ms + 15ms + 20ms = 35ms
- IT4 = 10ms + 15ms + 10ms + 12ms = 47ms

Política balanceada

Para la política balanceada, el cálculo del peor tiempo corresponde a:
35ms*47 + 47ms*46 + 35ms*46 + 47ms*47 = 7626ms = 7.6s

Por otra parte, el mejor tiempo puede calcularse como:
35ms*10 + 47ms*9 + 35ms*9 + 47ms*10 = 1558ms = 1.5s

Las consideraciones son las mismas que las realizadas para el caso anterior

Política priorizada

Para la política priorizada, el cálculo del peor tiempo corresponde a:
35ms*19 + 47ms*28 + 35ms*18 + 47ms*121 = 8298ms = 8.2s

El mejor tiempo puede calcularse como:
35ms*5 + 47ms*6 + 35ms*3 + 47ms*24 = 1690ms = 1.6s

Configuración lenta

Por lo último se tiene la configuración lenta

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
0	30	0	0	50	50	0	0	70	30	35	0

Tabla 6: Tiempos de Alfa para la configuración lenta

Donde, los 4 IT para este último caso son:
- IT1 = 30ms + 50ms + 70ms = 150ms
- IT2 = 30ms + 50ms + 30ms + 35ms = 145ms
- IT3 = 30ms + 50ms + 70ms = 150ms
- IT4 = 30ms + 50ms + 30ms + 35ms = 145ms

Política balanceada

Para la política balanceada, el cálculo del peor tiempo corresponde a:
150ms*47 + 145ms*46 + 150ms*46 + 145ms*47 = 26970 = 27.5s

Por otra parte, el mejor tiempo puede calcularse como:
150ms*10 + 145ms*9 + 150ms*9 + 145ms*10 = 5510ms = 5.6s

Política priorizada

Para la política priorizada, el cálculo del peor tiempo corresponde a:
150ms*19 + 145ms*28 + 150ms*18 + 145ms*121 = 26410ms = 27.1s

El mejor tiempo puede calcularse como:
150ms*5 + 145ms*6 + 150ms*3 + 145ms*24 = 5400ms = 5.5s

Análisis de tiempos

Configuración		Peor tiempo	Mejor tiempo	Tiempo promedio
Rápida	Política Balanceada	2.6s	0.5s	1.5s
	Política Priorizada	2.6s	0.4s	1.5s
Media	Política Balanceada	7.6s	1.5s	4.5s
	Política Priorizada	8.2s	1.6s	4.9s
Lenta	Política Balanceada	27.5s	5.6s	16.5s
	Política Priorizada	27.1s	5.5s	16.3s

Tabla 7: Resumen de tiempos por política

Como se puede observar no hay una diferencia significativa en los tiempos de ejecución de las diferentes políticas. Esto puede deberse a la naturaleza del sistema analizado.

Análisis de los invariantes de transición mediante expresiones regulares

Una expresión regular constituye una forma concisa y precisa de representar un lenguaje regular, y se emplea para definir patrones de búsqueda dentro de cadenas de texto.
Además, dicha expresión fue utilizada como prueba unitaria para verificar que el comportamiento del programa se mantuviera consistente a lo largo del desarrollo, asegurando que no se introdujeran errores durante las modificaciones.

En este trabajo, la expresión regular correspondiente a los invariantes de transición de la red de Petri asociada al sistema de agencias de viaje puede definirse del siguiente modo (ver Fig. 7).

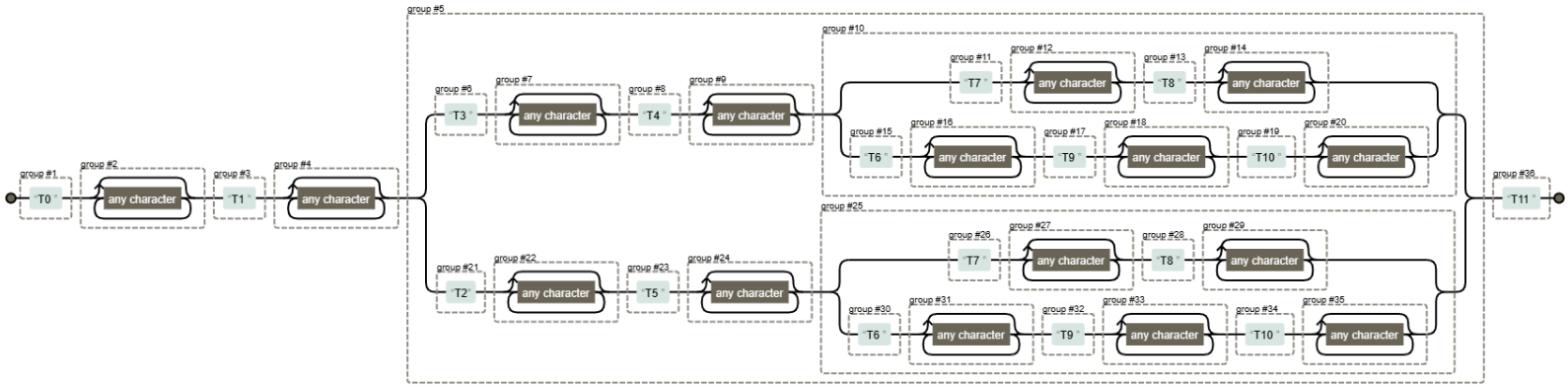


Fig. 7: Representación visual de las expresiones regulares.

Conclusiones

La implementación de una red de Petri para modelar el sistema de una agencia de viajes en Java, utilizando un monitor, ha demostrado ser una solución eficaz para gestionar la concurrencia y garantizar la estabilidad del sistema. Este enfoque permitió superar la indeterminación observada en métodos anteriores, logrando resultados consistentes y libres de errores.

El uso de un monitor centralizó el control de los hilos, facilitando la coordinación de las transiciones concurrentes, lo cual no solo mejoró la eficiencia del sistema, sino también la legibilidad y mantenibilidad del código. Asimismo, la incorporación de una semántica temporal para determinadas transiciones permitió modelar con precisión los aspectos temporales del sistema, asegurando que estas se ejecutaran en los momentos adecuados y respetaran las restricciones necesarias para el correcto funcionamiento de la agencia.

La implementación y comparación de dos políticas distintas de manejo —una balanceada y otra priorizada— evidenció cómo el comportamiento del monitor se ve afectado por la estrategia adoptada. Mientras que la política balanceada distribuye equitativamente la carga entre las transiciones, la priorizada otorga preferencia a ciertos invariantes críticos, permitiendo adaptar el sistema a diferentes requerimientos operativos.

Las ejecuciones múltiples permitieron evaluar el desempeño de ambas políticas y verificar el cumplimiento de los invariantes de plazas y transiciones. Los resultados confirmaron que el sistema funcionaba de manera coherente y predecible, preservando su integridad y reduciendo significativamente la indeterminación y los errores.

Finalmente, la utilización de expresiones regulares como mecanismo de prueba unitaria resultó ser una herramienta eficaz para validar el cumplimiento de los invariantes de transición en la red de Petri implementada, contribuyendo a asegurar la robustez del sistema desarrollado.

Referencias

1. Algoritmos para determinar cantidad y responsabilidad de hilos en sistemas embebidos modelados con Redes de Petri S 3 PR, https://www.researchgate.net/publication/358104149_Algoritmos_para_determinar_cantidad_y_responsabilidad_de_hilos_en_sistemas_embebidos_modelados_con_Redes_de_Petri_S_3_PR. Consultado 03 2025.