Universidad Nacional de Córdoba

Facultad de Cs. Exactas, Físicas y Naturales

Programación Concurrente

Trabajo Práctico Final

Profesores:

- Ing. Luis Orlando Ventre
- Ing. Mauricio Ludemann

Alumnos:

- Emanuel Nicolás Rodríguez 42259187
- Brian Joel Gerard 40989936
- Franco Viotti 42051491

FCEFyN
FACULTAD DE CIENCIAS EXACTAS, FÍSICAS y NATURALES

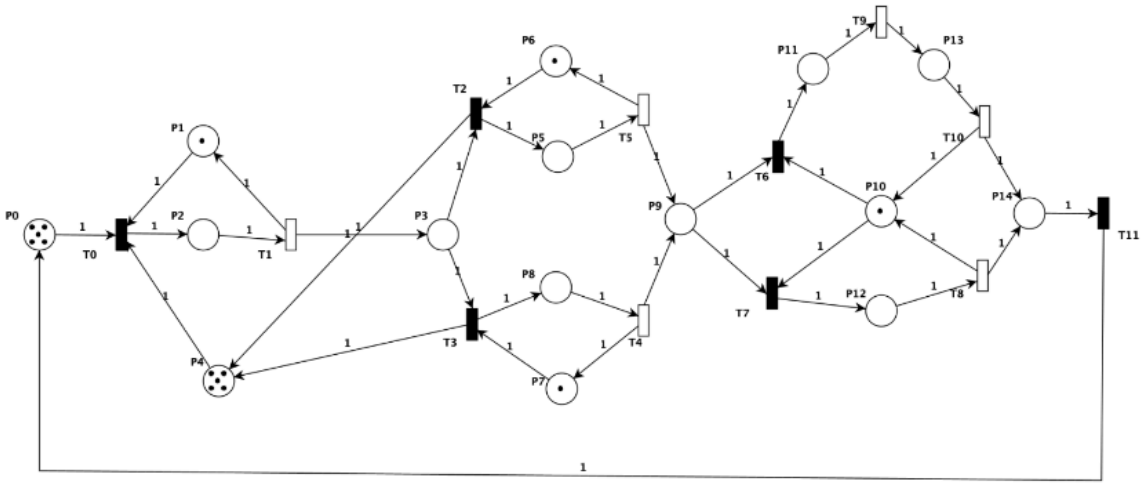
unc

Índice

Introducción

Este informe presenta un análisis detallado, modelado y simulación de una red de Petri aplicada a un sistema de agencia de viajes. El trabajo se llevó a cabo en el marco de la asignatura Programación Concurrente, con el objetivo de estudiar el comportamiento de sistemas concurrentes utilizando redes de Petri, un modelo gráfico y matemático ampliamente empleado en la modelización de sistemas distribuidos. Para este propósito, se utilizó la herramienta Petrinator, que permitió identificar y analizar las propiedades clave de la red de Petri. Posteriormente, se implementó un monitor de concurrencia en Java, siguiendo las consignas del trabajo práctico.

Descripción de la Red de Petri



- Fig. 1.0, Red de Petri

La red de Petri utilizada en este trabajo modela el flujo de clientes y la gestión de reservas en una agencia de viajes. Cada lugar y transición dentro de la red representa diferentes estados y eventos en el sistema. A continuación, se describen las principales plazas y transiciones:

- **P0 (Idle)**: Representa el buffer de entrada de clientes a la agencia.
- **P1, P4, P6, P7, P10 (Recursos Compartidos)**: Plazas que modelan recursos compartidos en el sistema, como agentes de reservas y áreas de gestión.
- **P2 (Ingreso a la Agencia)**: Representa el ingreso de un cliente a la agencia de viajes.
- **P3 (Sala de Espera)**: Lugar donde los clientes esperan antes de ser atendidos por un agente.
- **P5, P8 (Gestión de Reservas)**: Estados del sistema en los cuales se gestionan las reservas de los clientes.
- **P9 (Espera para Aprobación/Rechazo de Reservas)**: Modela la espera de los clientes mientras su reserva es procesada por un agente.
- **P11 (Confirmación de Reserva)**: Plazas que representan la confirmación de una reserva por parte del agente.
- **P12 (Cancelación de Reserva)**: Lugar donde se modela la cancelación de una reserva por parte del cliente o del agente.
- **P13 (Pago de la Reserva)**: Representa el momento en que el cliente realiza el pago de la reserva confirmada.
- **P14 (Salida del Cliente)**: Instancia previa a que el cliente se retire de la agencia.

La estructura de esta red de Petri permite capturar de manera efectiva las interacciones y dependencias entre los diferentes estados y procesos del sistema de la agencia de viajes.

Análisis de propiedades de la red

Se hizo uso de la herramienta Pipe para clasificación de la red, de donde se obtuvo el siguiente resultado:

Tipos de Redes de Petri

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

- Fig. 2.0, Tipos de Redes

Propiedades matemáticas

Bounded	true
Safe	false
Deadlock	false

- Fig. 2.1, Propiedades de la Red

Red Simple y Extendida

La red de Petri es **simple y extendida**, lo que significa que las transiciones reciben entradas de una sola plaza, o bien, en aquellos casos donde reciben más de una, el conjunto de salida de una de estas últimas está contenido o es igual al de las otras. Esta característica asegura una estructura regular en la red, lo que facilita su análisis y garantiza la coherencia en el flujo de tokens a través del sistema.

Limitación

La red de Petri es **limitada**, lo que significa que la cantidad de tokens en cada lugar nunca excederá un valor máximo predefinido. Esta propiedad es fundamental para evitar un crecimiento ilimitado en cualquier parte del sistema, lo que podría llevar a estados de desbordamiento o comportamiento incontrolado. La limitación previene la acumulación excesiva de tokens en cualquier lugar lo que nos asegura un modelo más acorde a la realidad.

Seguridad

A pesar de ser limitada, la red no es **completamente segura**. Esto se debe a que, en algunos estados alcanzables desde la marca inicial, uno o más lugares pueden contener más de un token. Esta condición podría representar la ocurrencia simultánea de eventos que deberían ser mutuamente excluyentes, lo que podría tener implicaciones en la concurrencia y sincronización de eventos dentro del sistema. En el caso analizado, esto puede reflejar situaciones en las que múltiples clientes están esperando ya sea a ser atendidos o a recibir la confirmación/rechazo de su reserva, lo cual es aceptable pero requiere supervisión para evitar sobrecargas.

Ausencia de Deadlock

Un deadlock es una situación en la que ninguna transición del sistema puede ser disparada, provocando que el sistema se quede bloqueado en un estado inactivo. La ausencia de deadlocks asegura que desde cualquier estado alcanzable siempre existe al menos una transición que puede ser disparada, garantizando que el sistema nunca se quedará sin posibilidad de avanzar. Esta propiedad es esencial para mantener la continuidad operativa del sistema. En conclusión, la red de Petri **no presenta deadlock**.

Vivacidad

El hecho que una red de Petri sea viva significa que todas las transiciones tienen la posibilidad de ser disparadas en algún momento del ciclo de vida. Esto garantiza que ninguna parte quedará bloqueada indefinidamente y que todos los procesos podrán completarse eventualmente. En el caso analizado la red de Petri **es viva**. Esta es una propiedad clave para asegurar que este sistema pueda gestionar de manera eficiente todas las reservas, cancelaciones y otros eventos sin interrupciones.

Invariantes de la Red.

El análisis de invariantes es crucial para comprender la conservación y repetitividad dentro de la red de Petri. Las invariantes de una red son propiedades independientes tanto del marcado inicial como de la secuencia de disparos, y pueden asociarse a ciertos subconjuntos de plazas o de transiciones.
Para llevar a cabo este análisis, se ha utilizado la matriz de incidencia, que juega un papel fundamental en la identificación tanto de los invariantes de plaza como de los invariantes de transición.

Matriz de Incidencia

La matriz de incidencia de la red de Petri es una representación matemática que describe cómo las transiciones afectan las plazas en la red. Esta matriz se define como la diferencia entre la **matriz de incidencia de salida** I^+ y la **matriz de incidencia de entrada** I^-

$$I = I^+ - I^-$$

- I^+= Indica cómo las transiciones añaden tokens a las plazas.
- I^-= Indica cómo las transiciones remueven tokens de las plazas.

Las matrices I^+e I^- son también conocidas como *post* y *pre*, respectivamente. Para el caso de la matriz *post* se tiene que cada elemento $I^+(P_i, T_j)$ contiene el peso asociado al arco que va desde T_j hasta P_i, es decir, la cantidad de tokens que se *generan* en la plaza P_i cuando la transición T_j es disparada. El caso contrario ocurre para los elementos de $I^-(P_i, T_j)$, que indica los tokens que se *retiran* en la plaza P_i cuando la transición T_j es disparada.

Las filas de las matrices representan las plazas mientras que las columnas las transiciones.

Invariantes de Plaza

Una invariante de plaza o **p-invariante** es un conjunto de plazas cuya suma de tokens no se modifica con una secuencia de disparos arbitrarios. Estos invariantes son esenciales para analizar la **conservación de recursos** dentro del sistema, asegurando que ciertas cantidades o recursos se mantengan estables a lo largo del tiempo
Para su cálculo se hace uso de la ecuación:

$$I \cdot x = 0$$

siendo I la matriz de incidencia y x un vector característico constituído por las plazas que forman parte de la invariante.

Invariantes de Transición

Una invariante de transición o **t-invariante** es el conjunto de transiciones que deben dispararse para que la red retorne a su estado inicial. Estos invariantes son clave para entender los **ciclos operacionales** en la red, asegurando que el sistema puede regresar a un estado de referencia después de completar un ciclo de operaciones.
Para el cálculo de los vectores que forman parte de las t-invariantes, la ecuación asociada es:

$$I^T \cdot x = 0$$

donde I^T es la transpuesta de la matriz de incidencia y x un vector característico constituído por las transiciones necesarias para que el sistema vuelva a su estado inicial.

Análisis de Matriz de Incidencia

Obtención de las matrices post, pre e incidencia utilizando Petrinator

Forwards incidence matrix I^+

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
P0	0	0	0	0	0	0	0	0	0	0	0	1
P1	0	1	0	0	0	0	0	0	0	0	0	0
P2	1	0	0	0	0	0	0	0	0	0	0	0
P3	0	1	0	0	0	0	0	0	0	0	0	0
P4	0	0	1	1	0	0	0	0	0	0	0	0
P5	0	0	1	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	1	0	0	0	0	0	0
P7	0	0	0	0	1	0	0	0	0	0	0	0
P8	0	0	0	1	0	0	0	0	0	0	0	0
P9	0	0	0	0	1	1	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	1	0	1	0
P11	0	0	0	0	0	0	1	0	0	0	0	0
P12	0	0	0	0	0	0	0	1	0	0	0	0
P13	0	0	0	0	0	0	0	0	0	1	0	0
P14	0	0	0	0	0	0	0	0	1	0	1	0

Backwards incidence matrix I^-

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
P0	1	0	0	0	0	0	0	0	0	0	0	0
P1	1	0	0	0	0	0	0	0	0	0	0	0
P2	0	1	0	0	0	0	0	0	0	0	0	0
P3	0	0	1	1	0	0	0	0	0	0	0	0
P4	1	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	1	0	0	0	0	0	0
P6	0	0	1	0	0	0	0	0	0	0	0	0
P7	0	0	0	1	0	0	0	0	0	0	0	0
P8	0	0	0	0	1	0	0	0	0	0	0	0
P9	0	0	0	0	0	0	1	1	0	0	0	0
P10	0	0	0	0	0	0	1	1	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	1	0	0
P12	0	0	0	0	0	0	0	0	1	0	0	0
P13	0	0	0	0	0	0	0	0	0	0	1	0
P14	0	0	0	0	0	0	0	0	0	0	0	1

- Fig. 3.0, Matriz de incidencia entrada/salida

Combined incidence matrix I

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
P0	-1	0	0	0	0	0	0	0	0	0	0	1
P1	-1	1	0	0	0	0	0	0	0	0	0	0
P2	1	-1	0	0	0	0	0	0	0	0	0	0
P3	0	1	-1	-1	0	0	0	0	0	0	0	0
P4	-1	0	1	1	0	0	0	0	0	0	0	0
P5	0	0	1	0	0	-1	0	0	0	0	0	0
P6	0	0	-1	0	0	1	0	0	0	0	0	0
P7	0	0	0	-1	1	0	0	0	0	0	0	0
P8	0	0	0	1	-1	0	0	0	0	0	0	0
P9	0	0	0	0	1	1	-1	-1	0	0	0	0
P10	0	0	0	0	0	0	-1	-1	1	0	1	0
P11	0	0	0	0	0	0	1	0	0	-1	0	0
P12	0	0	0	0	0	0	0	1	-1	0	0	0
P13	0	0	0	0	0	0	0	0	0	1	-1	0
P14	0	0	0	0	0	0	0	0	1	0	1	-1

- Fig. 3.1, Matriz de incidencia combinada

Análisis de Invariantes

Invariantes de Plaza

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

- Fig. 4.0, Invariantes de plaza

Un uno en una posición indica que esa plaza es parte de la invariante y un cero indica lo contrario.

Ecuaciones de las invariantes de plaza

$$M(P1) + M(P2) = 1$$
$$M(P5) + M(P6) = 1$$
$$M(P7) + M(P8) = 1$$
$$M(P2) + M(P3) + M(P4) = 5$$
$$M(P10) + M(P11) + M(P12) + M(P13) = 1$$
$$M(P0) + M(P2) + M(P3) + M(P5) + M(P8) + M(P9) + M(P11) + M(P12) + M(P13) + M(P14) = 5$$

Invariantes de Transición

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

- Fig. 4.1, Invariantes de transición

Un uno en una posición indica que esa transición es parte de la invariante y un cero indica lo contrario.

Implementación

Tabla de Estados del Sistema

Plaza	Estado
P0	Buffer entrada clientes a la agencia
P1	Recurso de ingreso a la agencia
P2	Ingreso a la agencia
P3	Sala de espera de la agencia
P4	Recursos de ingreso a sala de espera
P5	Estado donde se realiza la reserva
P6	Recurso para realizar reserva
P7	Recurso para realizar reserva
P8	Estado donde se realiza la reserva
P9	Espera de quien aprueba/rechaza reservas
P10	Recurso para confirmación y pago o cancelación
P11	Confirmación de la reserva
P12	Cancelación de la reserva
P13	Pago de la reserva
P14	Instancia previa a la salida del cliente

Tabla de Eventos del Sistema

Transición	Evento
T0	Ingreso de cliente a la agencia
T1	Ingreso de cliente a la sala de espera
T2	Inicio de reserva
T3	Inicio de reserva
T4	Espera de realización de reserva
T5	Espera de realización de reserva
T6	Ingreso a confirmación de la reserva
T7	Cancelación de la reserva
T8	Ingreso al buffer de salida de la agencia
T9	Ingreso a pago de la reserva
T10	Ingreso al buffer de salida de la agencia
T11	Salida del cliente

Determinación de la cantidad de hilos para la ejecución del sistema con el mayor paralelismo posible

El número máximo de hilos que pueden correr en paralelo es igual al número máximo de secuencias de transiciones que pueden ejecutarse concurrentemente sin conflictos. Para determinar la cantidad de hilos necesarios para la ejecución del sistema con el mayor paralelismo posible, es necesario:

1. Obtener los invariantes de transición de la red:

$$IT_1 = \{T0,\ T1,\ T3,\ T4,\ T7,\ T8,\ T11\}$$
$$IT_2 = \{T0,\ T1,\ T3,\ T4,\ T6,\ T9,\ T10,\ T11\}$$
$$IT_3 = \{T0,\ T1,\ T2,\ T5,\ T7,\ T8,\ T11\}$$
$$IT_4 = \{T0,\ T1,\ T2,\ T5,\ T6,\ T9,\ T10,\ T11\}$$

2. Obtener el conjunto de plazas PI para cada IT

$$PI_1 = \{P0,\ P1,\ P2,\ P3, P4,\ P7,\ P8,\ P9,\ P10,\ P12,\ P14\}$$

$PI_2 = \{P0,\ P1,\ P2,\ P3,\ P4,\ P7,\ P8,\ P9,\ P10,\ P11,\ P13,\ P14\}$

$PI_3 = \{P0,\ P1,\ P2,\ P3,\ P4,\ P5,\ P6,\ P9,\ P10,\ P12,\ P14\}$

$PI_4 = \{P0,\ P1,\ P2,\ P3,\ P4,\ P5,\ P6,\ P9,\ P10,\ P11,\ P13,\ P14\}$

3. Obtenemos el conjunto de plazas de acción (PA) de cada IT:

$PA_1 = \{P2,\ P8,\ P12,\ P14\}$

$PA_2 = \{P2,\ P8, P11, P13, P14\}$

$PA_3 = \{P2,\ P5,\ P12,\ P14\}$

$PA_4 = \{P2,\ P5,\ P11,\ P13,\ P14\}$

4. Obtenemos el conjunto de estados MA del conjunto de plazas PA donde

   $PA\ = \{P2,\ P5,\ P8,\ P11,\ P12,\ P13,\ P14\}$

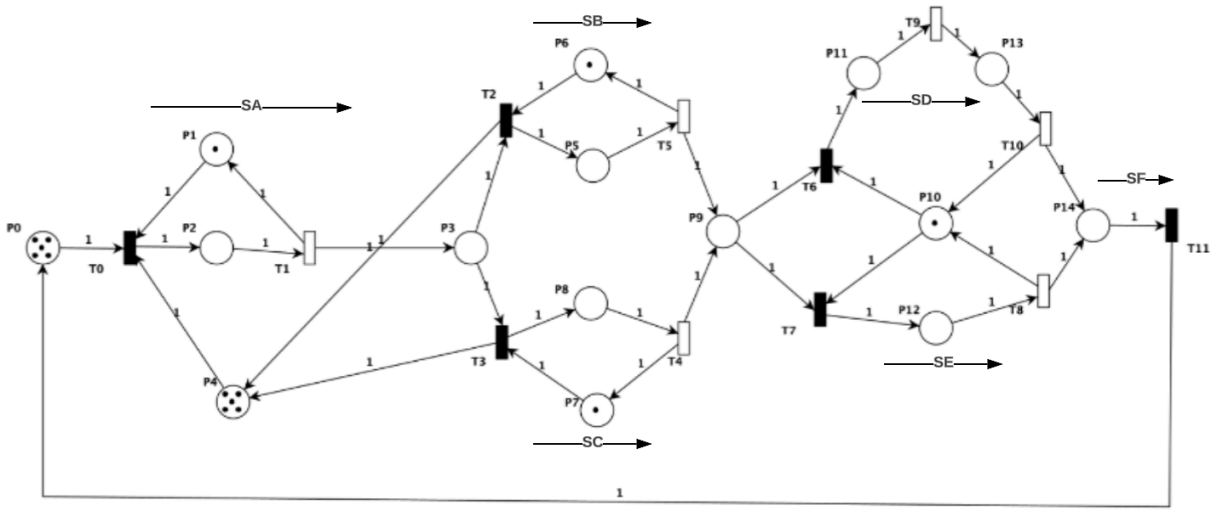
   Luego, se busca el marcado máximo entre todos los marcados posibles con la tabla MA:

P2	P5	P8	P11	P12	P13	P14	SUMA
1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	1
1	1	0	0	0	0	0	2
0	1	1	0	0	0	0	2
1	1	1	0	0	0	0	3
1	0	1	1	0	0	0	3
0	1	1	1	0	0	0	3
1	1	1	0	0	1	0	4
1	1	1	0	0	0	1	4
1	1	0	0	1	0	1	4
1	1	1	0	1	0	1	5

Tabla: la tabla muestra el número de tokens en las plazas pertenecientes al conjunto PA en un momento dado, al que se puede alcanzar mediante el disparo de una serie definida de transiciones.

Con el resultado obtenido de la tabla MA, se puede observar que la cantidad máxima de hilos simultáneos en el sistema será *cinco*.

Se aplica el algoritmo para determinar la responsabilidad de los hilos:



- Fig. 5.0, Segmentos de la rdp

Tenemos 6 segmentos:

$PA = \{P2,\ P5,\ P8,\ P11,\ P12,\ P13,\ P14\}$

$PS_A = \{P2\}$
$PS_B = \{P5\}$
$PS_C = \{P8\}$
$PS_D = \{P11,\ P13\}$
$PS_E = \{P12\}$
$PS_F = \{P14\}$

$MS_A = \{0\} \mid \{1\}$
$MS_B = \{0\} \mid \{1\}$
$MS_C = \{0\} \mid \{1\}$
$MS_D = \{0\} \mid \{1\}$
$MS_E = \{0\} \mid \{1\}$
$MS_F = \{0\} \mid \{1\}$

¿Cuál es la responsabilidad de cada hilo?

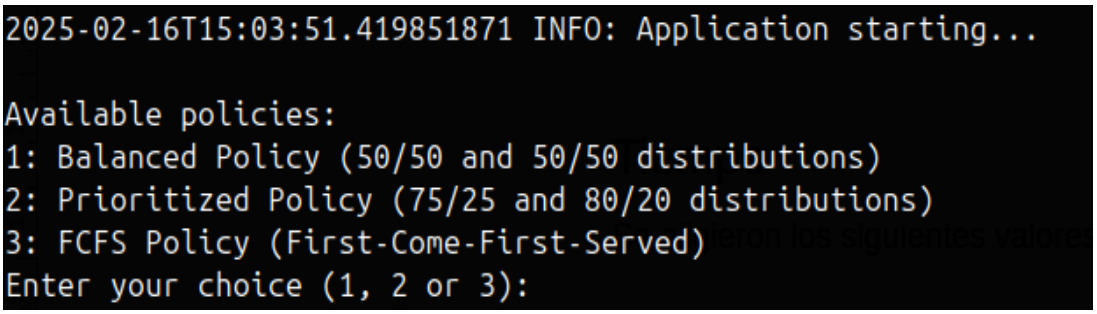
H_A: El hilo A se encarga del proceso de ingreso y espera por parte de los clientes a la agencia.
H_B: El hilo B se encarga del proceso de gestión de las reservas.
H_C: El hilo C se encarga del proceso de gestión de las reservas.
H_D: El hilo D se encarga del proceso de confirmación y pago de las reservas.
H_E: El hilo E se encarga del proceso de rechazo de las reservas.
H_F: El hilo F se encarga del proceso de salida de los clientes de la agencia.

Los hilos D y E se podrían unificar, ya que la plaza P10 tiene un solo recurso y no es posible la ejecución en paralelo de los segmentos D y E. Decidimos dejar un hilo para cada segmento como criterio de diseño, teniendo en cuenta que se podría aumentar la cantidad de recursos en P10 y así permitir la ejecución en paralelo.

Políticas

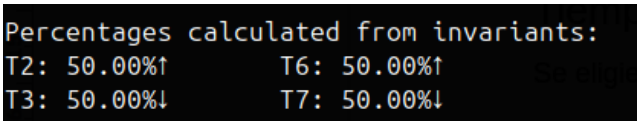
Se desarrollaron tres políticas para poder cambiar el comportamiento del sistema.
Se comprueba el funcionamiento de las mismas utilizando una expresiones regulares y un script de *python*.
Para poder ejecutar el programa con alguna política determinada hay dos opciones:

1. Pasar el número de la política como argumento al ejecutar el programa, por ejemplo "./gradlew run –args=<1/2/3>" donde 1 corresponde a la política balanceada, 2 a la que tiene prioridades y 3 a la FCFS.
2. Correr el programa y seleccionar la opción deseada desde la consola:



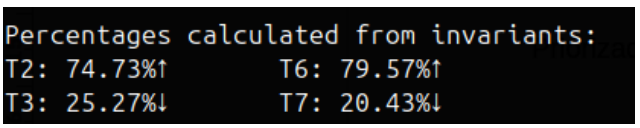
- Fig. 6.0, Políticas disponibles

Resultados de ejecuciones para cada política
Balanceada:



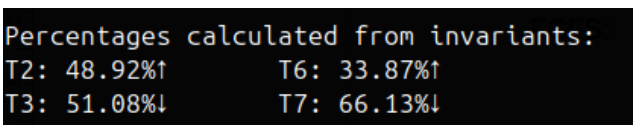
- Fig. 6.1, Política balanceada resultado

Priorizada:



- Fig. 6.2, Política Priorizada resultado

FCFS:



- Fig. 6.3, Política FCFS resultado

Tiempo

Se eligieron los siguientes valores para las transiciones temporales:

$$T1 = 2\ minutos$$
$$T4 = 15\ minutos$$
$$T5 = 15\ minutos$$
$$T8 = 5\ minutos$$
$$T9 = 5\ minutos$$
$$T10 = 15\ minutos$$

En el código se utilizó la escala:

$$1\ milisegundo = 1\ minuto$$

para poder representar el tiempo en minutos sin hacer que la simulación dure demasiado. Para la primera transición se pensó en dos minutos ya que ésta representa el ingreso del cliente a la sala de espera por lo que no debería ser mucho tiempo. Luego para T4 y T5 se pensó en quince minutos asumiendo que ese tiempo sería suficiente para poder realizar la reserva. Para T8 se asume que cinco minutos sería correcto para recibir la cancelación de la misma. Por último cinco minutos para confirmar la reserva y controlar que todo esté bien, y quince para hacer el pago de la misma.

Con un script de *bash* se automatizó la ejecución de diez ejecuciones del programa, alcanzando un total de 1860 invariantes de plaza. Esto con el objetivo de tener valores representativos de comportamiento del sistema. Luego se analizaron los resultados con un script de *python* y expresiones *regex*.

Demora en uno de los agentes que toman las reservas

Se aumentó el tiempo de demora de uno de los agentes que toman las reservas:

$$T4 = 45\ minutos$$

Se ejecutó el programa diez veces usando el script de *bash* y la política FCFS. Los resultados fueron:
La ejecución demoró 41.762 segundos en total.
Se puede observar que una mayor cantidad de clientes pasan por el agente de la plaza P6, aproximadamente tres veces más, lo que coincide con la relación de tiempo de delay entre las transiciones T4 y T5.

```
Percentages calculated from invariants:
T2: 73.76%↑        T6: 39.62%↑
T3: 26.24%↓        T7: 60.38%↓
```

- Fig. 6.4, Política FCFS T4=45

Si se utiliza la política balanceada, se obtiene que:
El programa demora 58.922 segundos en total para terminar su ejecución.
Pero el porcentaje de clientes atendidos por cada agente sigue siendo balanceado. Esto tiene sentido ya que estamos forzando a que la cantidad de clientes por agente sea la misma, pero genera un problema en la ejecución del programa empeorando su performance.

Demora en la última etapa del sistema (cancelación/aceptación y pago)

En una de las pruebas, se aumentó el tiempo en caso de que haya muchas demoras en la recepción de la cancelación y pago de la reserva con los siguientes tiempos:

$$T8 = 30\,minutos$$
$$T9 = 10\,minutos$$
$$T10 = 50\,minutos$$

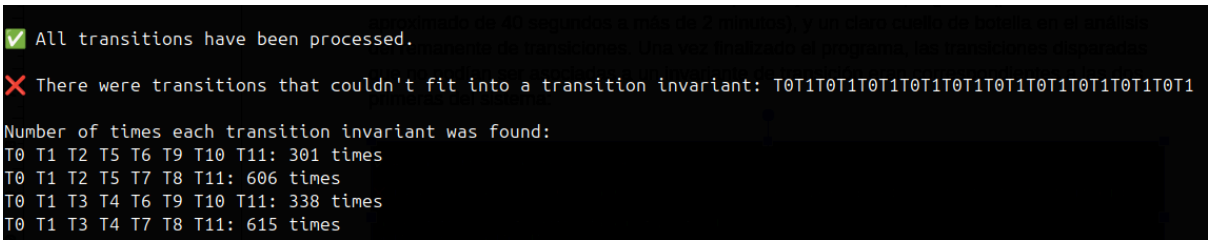
Se observó que en la última etapa en la plaza (P10) ocurrirá un cuello de botella debido a que posee un único recurso compartido entre la cancelación de reserva y la confirmación y posterior pago de la reserva.
Inicialmente y antes de la modificación de los valores temporales, diez ejecuciones del programa utilizando la política FCFS demoraron 46.425 segundos. Con los valores modificados, la duración obtenida es de 104.95 segundos.

Demora en la etapa inicial del sistema

La transición T1 es la encargada de enviar a los clientes a la plaza P3, donde esperan para ser recibidos por algunos de los agentes que tomarán su reserva. Se agregó un aumento en el tiempo de demora del disparo de esta transición:

$$T1 = 60\,minutos$$

En este caso se vio un aumento en el tiempo de ejecución del programa (pasando de un aproximado de 40 segundos a más de 2 minutos), y un claro cuello de botella en el análisis del remanente de transiciones. Una vez finalizado el programa, las transiciones disparadas que no podían ser asociadas a un invariante de transición eran correspondientes a las dos primeras del sistema:



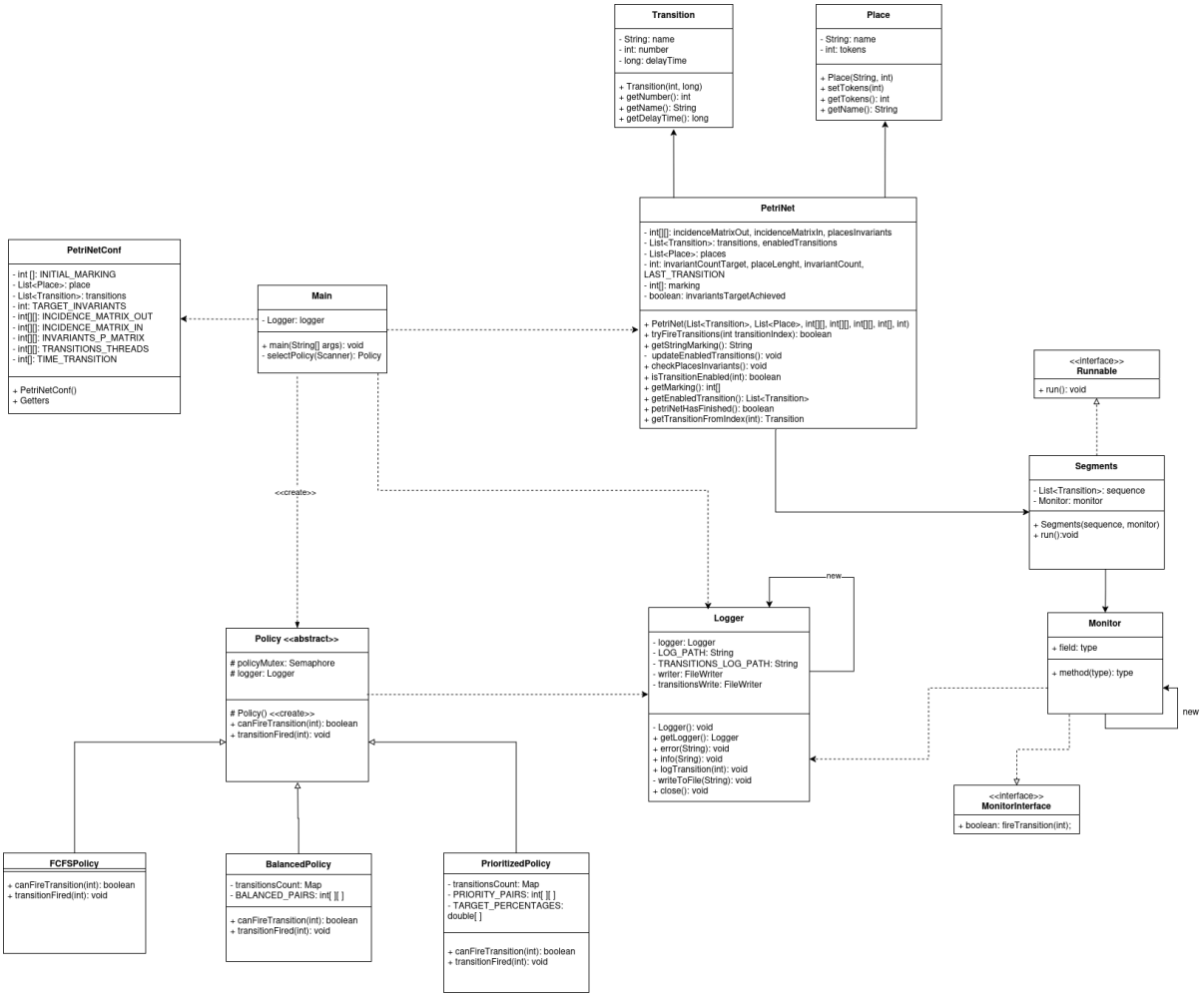
- Fig. 7.0, Resultado de REGEX

La prueba se realizó utilizando la política FCFS.

Desarrollo en Java

[Proyecto github link]

Diagrama de clases



- Fig. 8.0, Diagrama de clases.

El proyecto está compuesto por las siguientes clases:

- **Main**: Inicializa el sistema, permite seleccionar la política a utilizar y maneja la ejecución de los hilos.
- **Place**: Representa las plazas en la red de Petri, conteniendo tokens y su respectivo nombre.
- **Transition**: Modela las transiciones, incluyendo tiempos de retardo para transiciones temporales.
- **PetriNet**: El núcleo del sistema que implementa la lógica de la red de Petri, maneja el disparo de transiciones y verifica invariantes de plaza.
- **Policy (abstracta)**: Define la interfaz base para implementar diferentes políticas de control.

- **BalancedPolicy**: Implementa una política equilibrada 50/50 para ciertos pares de transiciones.
- **PrioritizedPolicy**: Implementa una política priorizada (75/25 y 80/20) para pares específicos.
- **PetriNetConf**: Contiene toda la configuración del sistema, incluyendo matrices de incidencia y marcado inicial.
- **Logger**: Sistema de registro que guarda eventos y errores en archivos de log.
- **Segments**:La clase Segments implementa la interfaz Runnable, permitiendo la ejecución concurrente de diferentes secuencias de transiciones en la red de Petri.

Cada instancia de Segments se encarga de ejecutar una secuencia específica de transiciones asignadas.

La clase recibe en su constructor una lista de transiciones que debe ejecutar y una referencia al Monitor que controla la red. Durante su ejecución, cada hilo Segments intenta disparar sus transiciones asignadas en orden, respetando las restricciones impuestas por el Monitor y las políticas de control.

El ciclo de ejecución continúa hasta que la red de Petri alcanza su objetivo (en este caso, completar 186 invariantes). Durante este proceso, múltiples instancias de Segments pueden estar ejecutándose simultáneamente, cada una manejando su propia secuencia de transiciones, lo que permite aprovechar el paralelismo dentro de la estructura de la red.

- **Monitor**: Su principal responsabilidad es controlar y sincronizar el acceso concurrente a la red, asegurando que las transiciones se disparen de manera segura y consistente. Implementa el patrón Singleton para garantizar que existe una única instancia que controla toda la red. Esto es crucial porque se necesita un punto único de control para mantener la consistencia del sistema.

El Monitor gestiona tres aspectos fundamentales:

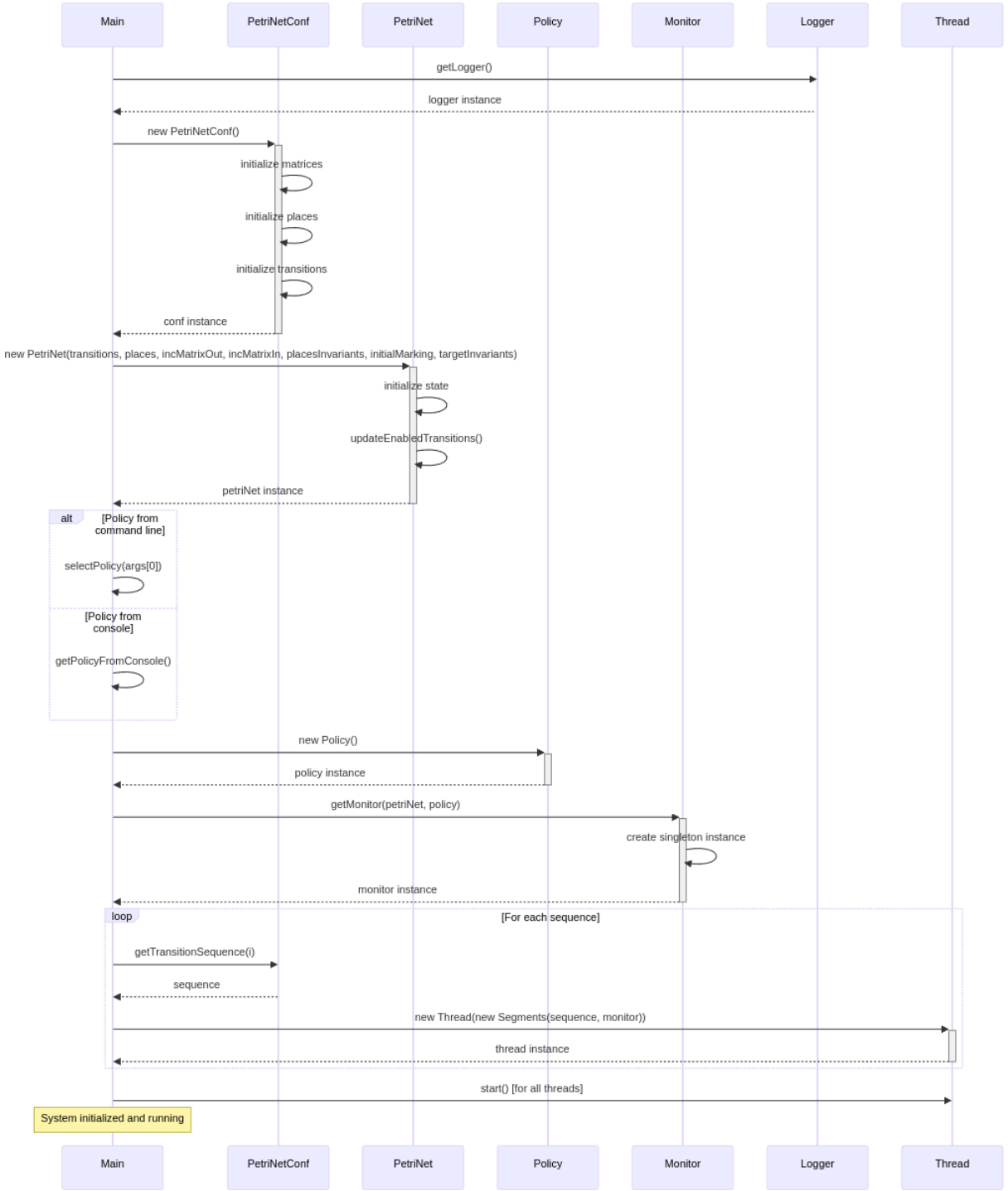
1. La sincronización entre hilos mediante un semáforo, que garantiza el acceso exclusivo a las secciones críticas de la red.

2. La aplicación de políticas de control, consultando si una transición puede dispararse según la política actual y notificando cuando se realiza el disparo. Esto permite implementar diferentes estrategias como la política balanceada o priorizada.

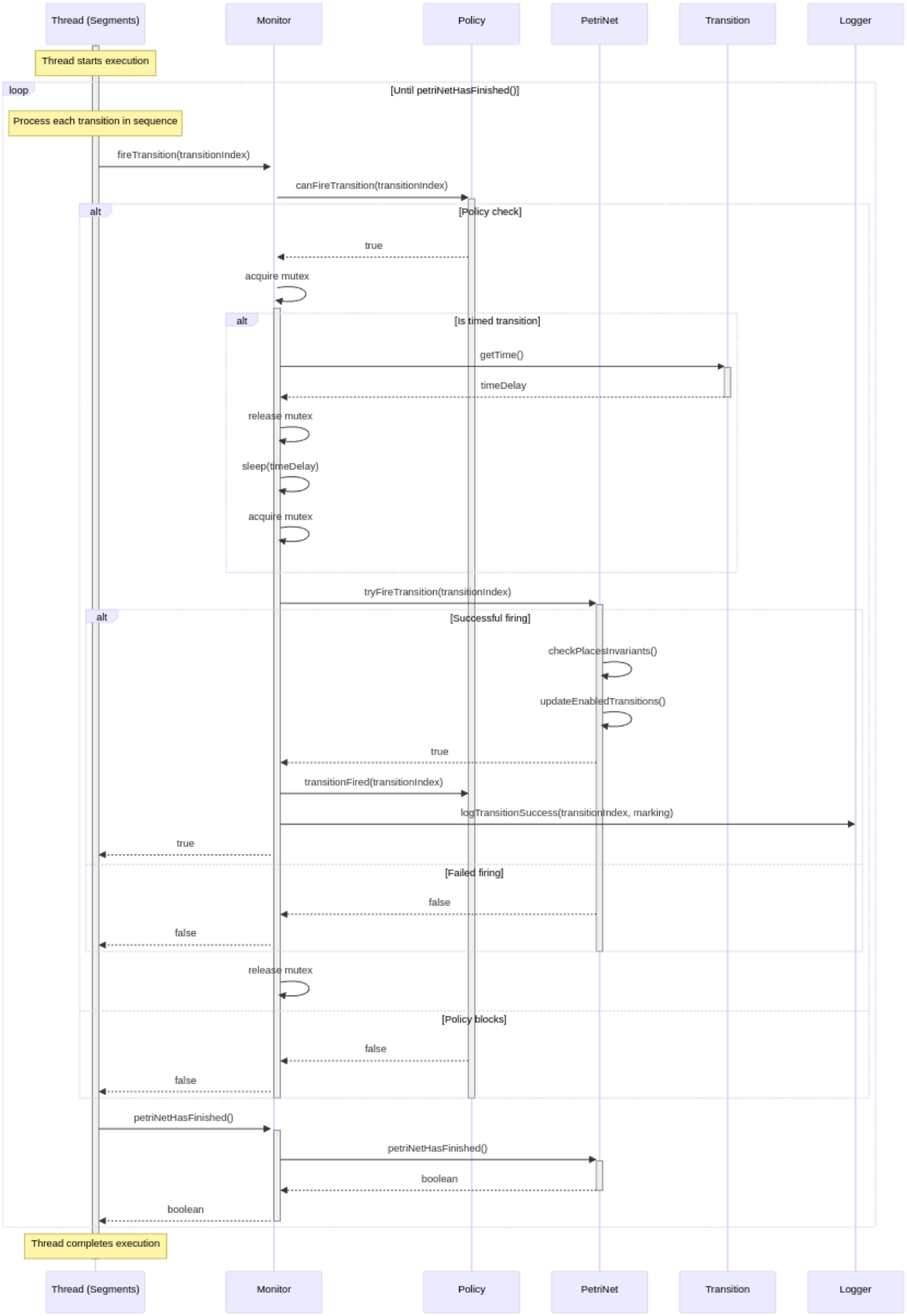
3. El manejo de transiciones temporales, gestionando los tiempos de espera necesarios y asegurando que se respeten las restricciones temporales de cada transición.

El Monitor expone una interfaz simple con un único método para disparar transiciones, pero internamente maneja la sincronización, el control de políticas y el manejo de errores. También se encarga del registro de eventos y errores para facilitar el seguimiento y depuración del sistema.

Diagramas de Secuencia



- Fig. 8.1, Diagrama de secuencia inicialización



- Fig. 8.2, Diagrama de secuencia hilos

Expresiones Regulares (REGEX)

El script *regex.py* analiza los logs generados por la ejecución de la red de Petri. Sus funciones principales son:

1. Leer secuencias de transiciones desde el archivo "*/tmp/transitionsSequence.txt*"

2. Utiliza una expresión regular para identificar 4 invariantes de transición válidos:
   - T0 T1 T2 T5 T6 T9 T10 T11
   - T0 T1 T2 T5 T7 T8 T11
   - T0 T1 T3 T4 T6 T9 T10 T11
   - T0 T1 T3 T4 T7 T8 T11

3. Procesar iterativamente el archivo:
   - Busca coincidencias de invariantes
   - Las cuenta
   - Las elimina del texto
   - Continúa hasta procesar todas las transiciones

4. Calcula estadísticas importantes:
   - Cantidad de veces que aparece cada invariante
   - Porcentajes de uso de las transiciones T2/T3 y T6/T7 (útil para verificar políticas)

5. Al finalizar:
   - Muestra los resultados del análisis